Toward a Super Duper Hardware Tactic

Mark Aagaard Miriam Leeser Phil Windley

May 7, 1993

EE-CEG-93-4

School of Electrical Engineering
Cornell University
Ithaca, NY

Submitted to the HOL User's Group Meeting 1993

Toward a Super Duper Hardware Tactic

Mark Aagaard Miriam Leeser
School of Electrical Engineering
Cornell University
Ithaca, NY

Phil Windley
Laboratory for Applied Logic
Department of Computer Science
Brigham Young University
Provo, UT

Abstract

We present techniques for automating many of the tedious aspects of hardware verification in a higher order logic theorem proving environment. We employ two complementary approaches. The first involves intelligent tactics which incorporate many of the smaller steps currently applied by the user. The second uses hardware combinators to partially automate inductive proofs for iterated hardware structures. We envision a system that captures most of this reasoning in one tactic, `SuperDuperHWTac`. Ideally, users would use this tactic on a goal for proving that a hardware component meets its specification, and get back a proof documented at a level they would have written by hand. This paper presents preliminary work toward `SuperDuperHWTac` in both the HOL and Nuprl proof development systems.

1 Introduction

Higher order logic makes specifying hardware designs natural. Unfortunately, it also makes verification tedious. If verification engineers adopt a specific style for doing hardware proofs, they find they do similar steps over and over again. This paper describes support for a specific style of hardware verification. The goal is to provide the user with intelligent tactics which automate those steps that are common to proofs done in the given style, and raise the level at which the user interacts with a tactic oriented theorem prover.

The style of hardware verification we support is one that is very widely used in higher order logic theorem proving. The user describes a hardware component as a relation over inputs and outputs. The component is described by a specification of desired behavior and an implementation in terms of the interconnection of simpler hardware modules. The steps required to verify such a component are:

1. Set the goal to be verified as `implementation` $\Longrightarrow$ `specification`
2. Expand definitions in the goal
3. Simplify and prove the goal using domain specific and general knowledge.

We provide specific support for bit vectors, Boolean reasoning, and regular structures described with *hardware combinators*.

We view the ideal higher order theorem proving environment as follows. The user defines the goal to be proved and applies one tactic at the top of the proof. We will call this ideal tactic `SuperDuperHWTac`. `SuperDuperHWTac` should have the following behavior:

- It finds a proof in reasonable time.
- The report provided of the resulting proof is close to what the user would have written by hand.
- Only those paths taken by the proof system that succeeded are reported.

Obviously we are a long way from obtaining such a tactic. This paper reports on preliminary work on writing tactics for hardware proofs that move in the general direction of `SuperDuperHWTac`. That work has focused on two complementary approaches. The first is implementing smart tactics that incorporate much of the repetitive reasoning found in hardware proofs. The second is to provide hardware combinators to provide automated support for a wide variety of iterated hardware structures.

The most similar work to the work in smart tactics is that done using the Faust and Mephisto packages [KKS91, SKK91]. They attempt to automate the process of hardware verification by simplifying proofs into subgoals that can be automatically verified using a tool outside the theorem proving environment. Our approach is to use tactics that apply many simpler tactics in order to find a proof of the goal within the theorem proving environment. Hanna [HDL90] has done similar work in hardware combinators. Our work provides iteration schemes and automated support for a wider variety of hardware structures and specifications.

The goals of our approach are:

- to enable the user to interact with the theorem prover at a natural level,
- to keep the proving process within the context of the theorem prover, and
- to generate a proof which provides documentation of the proof process.

The work described in this paper has been done in both the Nuprl and the HOL theorem proving environments. In our experience, these concepts apply to both systems, and will aid users of both environments. This paper is written from a Nuprl perspective. HOL versions of the tactics are given in the Appendix. In the next section we give a brief overview of the HOL and Nuprl systems, and explain our conventions for the paper. Next we describe our smart tactics and show how they can be used on some examples. Then we show how to use hardware combinators to simplify inductive proofs. We close with a discussion of issues and future directions.

2 Background

This work was done in both the HOL and Nuprl proof development systems. These systems have a great deal in common. Both are tactic-oriented proof systems using a higher-order logic and are descendents of the LCF project at Edinburgh [GMW79]. Similar to predicate logic in allowing quantification over variables, higher-order logic also allows quantification over predicates and functions thereby permitting more general systems to be described.

Tactics are functions that map a goal into a set of simpler subgoals. By repeatedly applying tactics, a complex goal can be broken down into subgoals simple enough that they can be solved using primitive rules. Both systems provide large collections of tactics to support goal directed proof and proof management systems that keeps track of the state of an interactive proof session. A metalanguage, ML, for programming and extending the theorem prover allows the user to extend the system. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes these systems extremely flexible.

HOL, developed at the University of Cambridge [CGM86, Gor86], is based on Church's theory of simple types [Chu40]. The simple type theory in HOL only allows type variables to be universally quantified at the outermost level. Furthermore, HOL does not directly support the use of subtypes or dependent types. HOL does, however, provide parsing and pretty printer support for restricted quantifiers which allow HOL specifications to succinctly express many of the same notions that are expressed by subtypes and dependent types. In addition, Melham has done preliminary work on adding quantification over type variables to HOL[Mel92].

Nuprl, developed at Cornell University [C^+86, Lee92], is based on a sequent version of Martin-Löf's constructive type theory [ML82]. Nuprl uses a rich and expressive type theory with dependent types which allow for a natural specification style. Because of the rich type theory, type checking is in general undecidable. In practice, type checking which does not involve complicated uses of dependent types can almost always be handled automatically by tactics available in Nuprl. We do not exploit the constructive nature of Nuprl's logic in hardware verification.

Nuprl's primitive types include `integer, atom`, and `void`. Type constructors include `list`, `union`, `function`, `product`, `quotient` and `subset`. Integer induction and list induction are built in. Types are stratified in an unbounded hierarchy of universes. $\mathtt{U}_1$ is the first universe and contains all small types, including integers, lists, pairs, disjoint union, function space, equality (e. g. `a=b in int`), and first order propositions. $\mathtt{U}_i$ and all elements of $\mathtt{U}_i$ are in $\mathtt{U}_{i+1}$. This concept of universes allows the Nuprl user to quantify over types in a very natural manner.

Propositions in HOL are Boolean; this is not the case in Nuprl. Nuprl's logic is intuitionistic and is based on a *propositions-as-types* correspondence: a proposition is true if and only if the type associated with that proposition is inhabited. In other words, a proposition `P` is a type whose elements are proofs of `P`. We have not found these features to be a hindrance in developing proofs.

In this paper we discuss the implementation of powerful tactics and hardware combinators in both the HOL and Nuprl systems. While the details of the implementations vary, the same tactics were implemented without much difficulty in both systems. Making use of the tactics and functions provided is similar for a user of either system. The rest of the paper is given from a Nuprl perspective, and using Nuprl terminology. The appendix shows some of the implementations in HOL.

A proof may be thought of as a tree. Associated with each node of the tree is a sequent and, if the node is not a leaf, a proof rule. A *sequent* is a number of hypotheses and a goal, and can be written in the form:

$$\mathtt{x}_1\mathtt{:H}_1\mathtt{,}\ \mathtt{x}_2\mathtt{:H}_2\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_n\mathtt{:H}_n \vdash \mathtt{P}$$

A sequent is true if the goal follows from the hypotheses.

We found that in addition to some terminology differences between the two systems, the styles of verification varied as well. In particular, in HOL the hypothesis list may grow as a proof progresses, but existing hypotheses remain static. In Nuprl it is quite common to modify the hypotheses. Usually when a hypothesis is modified, the altered hypothesis becomes the last hypothesis in the list. To control the length of the list it is common to `Thin` hypotheses. As a result, the number of hypotheses and their ordering may change a fair amount as the proof progresses. By adopting a specific proof style, we are able to maintain proofs that are not overly sensitive to perturbations in the hypotheses.

3 Towards More Intelligent Tactics

In this section we describe the existing tactics we use and the new tactics we have written. Our approach is to provide the user with a small set of tactics that take advantage of the particular structure shared by most combinational and sequential circuits and their implementations. Hardware proofs should largely consist of the repeated application of these tactics. Most of the tactics by themselves are very general, but we have combined them in ways which are tailored for doing hardware verification. These smart tactics are:

`RewriteWith convns`	Rewrites sequent using the rewrite rules named in `convns` .
`ChainAndSubst`	Performs forward chaining, backward chaining and and equality reasoning.
`BoolCaseSplit`	Finds an appropriate Boolean term to do case analysis on, then does the case analysis and calls `RewriteWith bool_lemmas` to simplify the resulting goals.
`SpecializeAuto`	Finds a hypothesis of the form $\forall$`t.P` then looks for an unambiguous instantiation for `t`. If one is found, then the instantiation is performed, otherwise the tactic fails.
`SuperHWTac`	Sets up hardware proofs.

These tactics are discussed in more detail below. Ideas for improving them further are also presented. In the next section we give examples of how these tactics are used in hardware proofs. Our eventual goal is an even more general tactic, which we have dubbed `SuperDuperHWTac`, comprised of these tactics and others that can be used to make substantial progress in hardware proofs, hopefully completely verifying many simple circuits. We discuss some initial experiments we have done with this smarter tactic in Section 3.5.

3.1 Standard Tactics and Tacticals

Nuprl has an `Autotactic` which is run after most proof steps. In our smart tactics we use `Autotactic` repeatedly. `Autotactic` does the following:

1. It completely solves simple goals. These include most membership goals, goals that exactly match a hypothesis, simple contradictions, and simple arithmetic equalities and inequalities. The arithmetic goals are solved using the decision procedures `Arith` and SUP-INF [Sho77].
2. It breaks up some complex goals. For example, goals of the form $\mathtt{A} \implies \mathtt{B}$ are simplified with `A` added to the hypothesis list; the resulting goal is `B`. For goals of the form $\forall$`x.P`, `x` is brought into the hypothesis list, and `P` is left as the goal.
3. It simplifies hypotheses. The hypothesis $\mathtt{A} \wedge \mathtt{B}$ is rewritten as two separate hypotheses, `A` and `B`.
4. It performs β reduction on hypotheses and the conclusion.
5. It takes hypotheses of the form $\exists$`x.A` and finds a witness for the existential quantifier.

With Nuprl, as in HOL, we make extensive use of rewriting. The tactic we use most often for rewriting is `RewriteWith`; it is very similar to `REWRITE_TAC` in HOL. Like `REWRITE_TAC`, `RewriteWith` takes a list of rules and repeatedly applies them until no more progress is made. The most significant difference between the two is that `RewriteWith` applies the rewrite rules to the conclusion and hypotheses and `REWRITE_TAC` applies them only to the conclusion.

Our smart tactics are built up from simpler tactics using tacticals. Both HOL and Nuprl have `ORELSE`, `THEN`, `THENL`, `Repeat`, `Try` and `Progress` (named `CHANGED_TAC` in HOL) tacticals which exhibit essentially the same behavior. In addition, Nuprl provides `ORTHEN` and `Complete` which can easily be implemented in HOL. `t1 ORTHEN t2` runs tactic `t1` then runs tactic `t2`. Both tactics are always tried whether or not the first one fails. `Complete tac` succeeds if `tac` can completely prove the goal it is applied to. Otherwise it fails, and leaves the current goal unaltered.

3.2 Chaining and Equality Reasoning

Much of the work in the hardware proofs requires reasoning with equalities and chaining through implications. We use the tactic `ChainAndSubst` to perform these tasks. This tactic is currently implemented using rewriting tactics available in HOL and Nuprl. We commonly rewrite using the following types of formulas:

`A = B`	Replace occurrences of `A` with `B`
`P` $\Longrightarrow$ `Q`	If `P` can be proved trivially then add `Q` to the hypothesis list.
`P` $\Longrightarrow$ `A = B`	If `P` can be proved trivially then replace occurrences of `A` with `B`

We have many ideas for improving `ChainAndSubst`. We are considering using forward chaining, back chaining and congruence closure rather than relying solely on rewriting. Forward chaining takes a hypothesis of the form `A` $\Longrightarrow$ `B` and tries to automatically prove `A`. If we can prove `A`, then `B` is added to the hypothesis list. Backward chaining takes hypotheses of the form `A` $\Longrightarrow$ `B` and tries to match `B` against the goal. If the match succeeds, then `A` is the new goal. In general, this matching process may be complex. We also intend to investigate rewriting expressions to normal forms.

3.3 Boolean Reasoning

Boolean reasoning arises frequently in hardware verification. We provide two primary techniques to deal with this: Boolean simplification and Boolean induction (case analysis).

We perform Boolean simplification by repeatedly applying a set of rewrite rules. `RewriteWith` takes a list of rewrite rules to apply and repeatedly applies these rules to the hypotheses and goal. We have developed a collection of lemmas about Boolean logic which we use with `RewriteWith`. The Boolean rules we are currently using include DeMorgan's laws, simplifying Boolean expressions with constants, and eliminating double negations. This set of rules is guaranteed to terminate. These Boolean lemmas are our first attempt at automated rewriting. This is a well studied area, with known pitfalls, including termination and efficiency. We address these issues in Section 6.

`BoolCaseSplitOn` is Nuprl's version of Boolean case analysis. First, it takes a Boolean term `t` and brings all hypotheses that contain `t` into the conclusion as antecedents of an implication. Then it generates two subgoals, one for the goal with all occurrences of `t` replaced by `T`, and the other with `t` replaced by `F`. This always generates subgoals of Boolean expressions with constants in them. `BoolCaseSplitOn` is usually followed by `RewriteWith bool_lemmas`, which is used to simplify the subgoals.

Picking the term to do induction on is a very difficult problem to solve in general, but we have discovered an effective heuristic for picking Boolean terms to do case analysis on. We do automated case analysis only on Boolean terms (`b`) which occur in `if b then t else e`. `BoolCaseSplit` looks for terms in this form. If such a term is found, then it performs case analysis on `b` and then does Boolean simplification using the Boolean lemmas.

3.4 Specializing Hypotheses

We have defined a tactic that is general purpose, but is especially useful for sequential hardware verification. Sequential modules are almost always specified in the form $\forall$`t. spec(t)`. We frequently want to eliminate the universal quantifier and replace it with an appropriate instantiation. In general this is a very hard problem, because we cannot always determine a unique instantiation for the variable. We have developed a heuristic which is quick and has worked quite well so far.

Our first step in specializing hypotheses is `ChainAndSubst`, which takes care of hypotheses when they are of one the forms listed in Section 3.2. There will be some times when the hypothesis is not in one of these forms. In hardware, if the specification is not of a form which is amenable for use as a rewrite rule, then it is quite often of the form `if b then t1=t2 else e1=e2`. In this case, rather than try to match the entire body of the hypothesis with terms in the sequent, we try matching each of `b`, `t1`, `t2`, `e1`, and `e2` in the sequent. If all of these matchings produce the same unique instantiation for `t`, then we perform the specialization.

If all of these techniques fail, we try matching the entire body of the hypotheses with the terms in the sequent. As always, if we find a unique instantiation for `t`, then we instantiate the hypothesis with that value. We are working on extending these techniques to handle multi-bit case statements as well as other forms of hypotheses that occur frequently in hardware verification.

3.5 Putting It All Together

`SuperHWTac` is the top level tactic we use to start our hardware proofs. It is implemented as shown in Definition 1. When beginning a proof we usually have some set of definitions which we wish to unfold so as to reveal implementations of modules. `RewriteWith def_names` looks for occurrences of definitions in the list `def_names` modules and unfolds them. The names of all hardware modules in the library are kept on a list bound to the ML variable `hw_names`. `RewriteWith hw_names` repeatedly unfolds hardware names on that list, until it makes no further progress.

In addition to `SuperHWTac`, we have begun to experiment with `SuperDuperHWTac` (Definition 2) which incorporates all of the techniques that we have discussed in this section.

Definition 1 SuperHWTac – initial tactic for hardware proofs

```
let SuperHWTac =
      Autotactic
         THEN RewriteWith hw_defs
         THEN ChainAndSubst
         THEN Autotactic
```

Definition 2 SuperDuperHWTac – experimental hardware tactic

```
let SuperDuperHWTac =
      Autotactic
         ORTHEN RewriteWith hw_defs
         ORTHEN (Repeat (ChainAndSubst
                         ORTHEN Autotactic
                         ORTHEN SpecializeAuto
                         ORTHEN BoolCaseSplit
                         ORTHEN RewriteWith bool_lemmas
                        )
                )
```

4 Verifying Hardware with Smart Tactics

In this section we illustrate the use of our smart tactics with the proof of several hardware components including a selector and an n-bit counter. All proofs are done at the gate level. We use the standard HOL methodology of describing circuits as relations over inputs and outputs. The top level goal to be proved for each component is of the form:

$\texttt{implementation} \Longrightarrow \texttt{specification}$

4.1 A Boolean Selector

This example [HB90] was chosen to illustrate the use of Boolean rewriting. The top level description is `out = if c then a else b`, where `a`, `b` and `c` all have type `bool`. The implementation shown in Figure 1 consists of three `nand` gates and one inverter.

The main goal of the proof is shown in Definition 3.

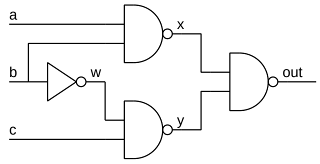

Figure 1: Schematic of bool selector

Definition 3 `Bool select specification`

```
∀a, b, c, out : Bool.
   bool_select_imp(a,b,c,out) ⟹
      out = if a then b else c
```

The only version of this proof was done with smart tactics. The entire proof is:

```
SuperHWTac Then BoolCaseSplit
```

`BoolCaseSplit` does case analysis on `a`, then repeatedly applies Boolean rewrites to simplify the goal.

4.2 An n-bit Counter

The `n`-bit counter is a more interesting example since it computes an arithmetic function, and uses `n`-bit vectors and sequential hardware.

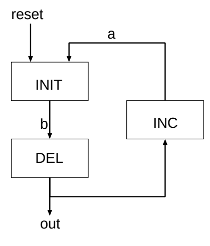

Figure 2: Implementation of counter

The proof of the counter was originally done before the smart tactics were written. A picture of the proof tree is shown in Figure 3. The proof required fifteen steps. Not only are there a lot of steps where the user must interact with the proof system, but each step requires a fair amount of detail. For example, the user needs to explicitly tell the prover which hardware

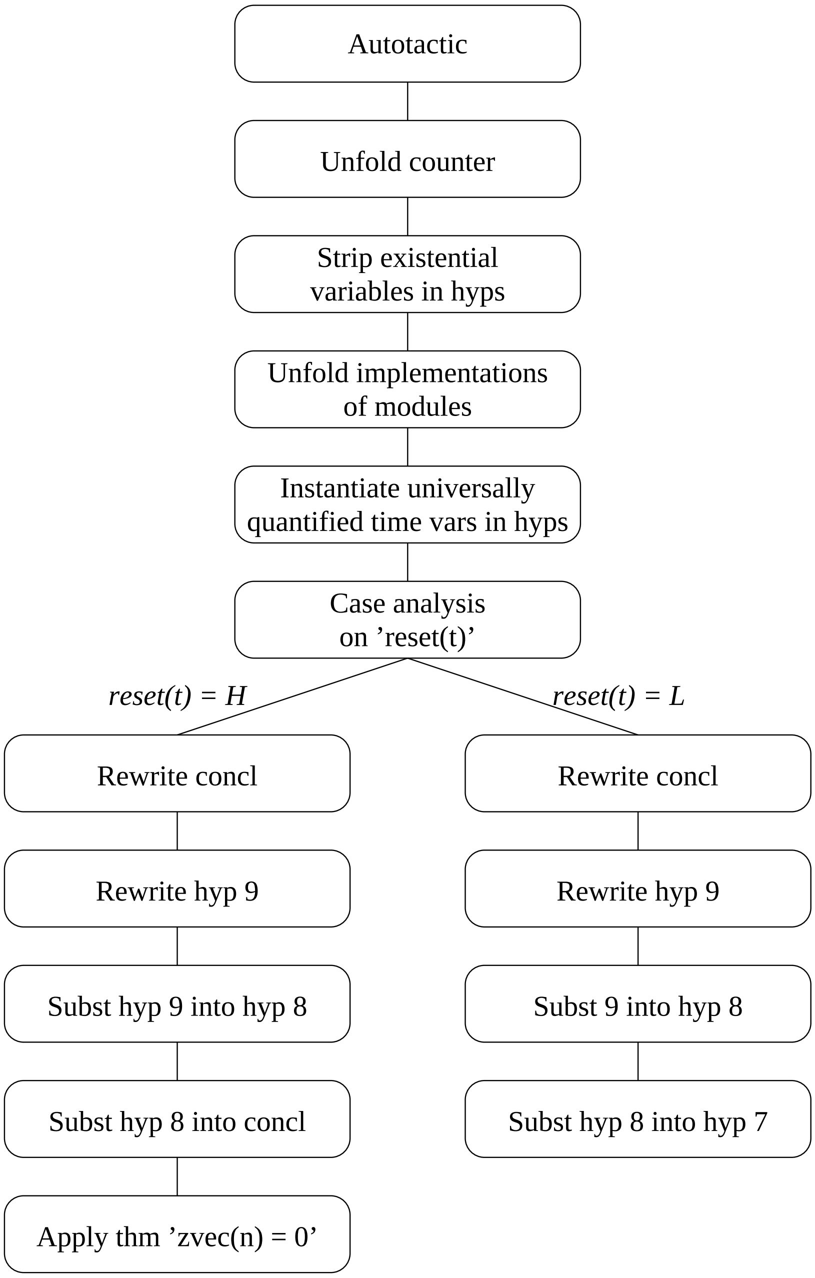

Figure 3: Proof tree for counter

component to unfold, which hypothesis to rewrite the conclusion with, what rewrite rule to apply, and so on at each of these steps.

The proof of the counter done with smart tactics is shown in Figure 4. The user entered the main goal, and the lines that begin with `BY`. The entire proof consists of six tactics applied by the user. The user sets up the proof using `SuperHWTac`. The next step is to strip the universal quantifiers for all the time variables (`t`). Next she does case analysis on `reset(t)`. `ChainAndSubst` gets the case where `reset(t)` is false. In the case where `reset(t)` is true, the user must apply `zvec_lemma`. Not only have we reduced the number of rules the user needs to apply, but the level of the rules is more natural and in most cases requires some intelligence.

The implementation of `SuperDuperHWTac` shown in (Definition 2) is capable of completely proving the counter circuit, except for the application of `zvec_lemma`. Also, by using only the tactics used to implement `SuperDuperHWTac` we are able to verify the parity checking circuit shown in Definition 4 and Theorem 1 [SKK91]. `SuperDuperHWTac` is not yet capable of doing the complete the proof of the parity circuit by itself yet. We are continuing to improve `SuperDuperHWTac`. By experimenting with a wider variety of example circuits we believe that we will be able to find a tactic which come very close to solving many simple hardware circuits.

5 Induction and Hardware Combinators

In doing hardware proofs, we have discovered that the user follows the same reasoning repeatedly. The aim of the smart tactics is to capture this reasoning. The tactics we have developed up to this point work well for combinational circuits and sequential circuits whose proofs do not depend on induction; *i.e.* their structures are not defined in terms of iterators. The Faust and Mephisto systems are aimed at this same set of proofs. In their view, induction requires intelligence on the part of the user [SKK91].

We have found that proofs of iterative hardware structures also contain a lot of repetitive reasoning. As with the smart tactics, we would like to capture this reasoning once and for all, and leave users with only those parts of the proof that are unique to the specific structure they are proving. We do this by using *hardware combinators* to capture frequently used iteration schemes. These schemes include one and two dimensional arrays, arrays with a rippled signal, rectangles, triangles, and trees. For each of these structures we provide induction schemes which have been formally derived for each structure. The user can prove properties about an iterated hardware module by making use of the induction schemes provided.

```
⊢? ∀n:N. ∀reset:signal.  ∀out:sig_n.
       counter(n)(reset,out) ⟹
          ∀t:N. if reset(t)
                then vec2num(2,n)(out(t+1))=0
                else vec2num(2,n)(out(t+1))=(vec2num(2,n)(out(t))+1) mod 2^{n+1}
BY   SuperHWTac
     ____________________________

 1. ∀t:N. if reset(t) then b(t)=zvec(n) else b(t)=a(t)
 2. ∀t:N. out(t+1)=b(t)
 3. ∀t:N. vec2num(2,n)(a(t))=(vec2num(2,n)(out(t))+1) mod 2^{n+1}
 ⊢? if reset(t) then vec2num(2,n)(out(t+1))=0
     else vec2num(2,n)(out(t+1))=(vec2num(2,n)(out(t))+1) mod 2^{n+1}
BY   SpecializeAuto
     ____________________________

 1. vec2num(2,n)(a(t))=(vec2num(2,n)(out(t))+1) mod 2^{n+1}
 2. out(t+1)=b(t)
 3. if reset(t) then b(t)=zvec(n) else b(t)=a(t)
 ⊢? if reset(t) then vec2num(2,n)(out(t+1))=0
     else vec2num(2,n)(out(t+1))=(vec2num(2,n)(out(t))+1) mod 2^{n+1}
BY   BoolCaseSplit
     ____________________________

 3. b(t)=zvec(n)
 ⊢? vec2num(2,n)(out(t+1))=0
BY   ChainAndSubst
     __________

 ⊢? vec2num(2,n)(zvec(n))=0
BY   Lemma ‘zvec_lemma‘
     __________

 3. b(t)=a(t)
 ⊢? vec2num(2,n)(out(t+1))=
      (vec2num(2,n)(out(t))+1) mod 2^{n+1}
BY   ChainAndSubst
     __________
```

Figure 4: Proof of counter using SuperHWTac

Definition 4 `Implementation of parity circuit`

```
parity(in)(out) =
   ∃l1, l2, l3, l4, l5 : (bool)signal
      inv(l2)(l1)  ∧
      mux(in)(l1)(l2)(l3)  ∧
      const_one(l4)  ∧
      delay(l4)(l5)  ∧
      mux(l5)(l3)(l4)(out)  ∧
      delay(out)(l2)
```

Theorem 1 `Specification of parity circuit`

```
⊢ ∀in, out :  (bool)signal
     parity(in)(out) ⟹
        ∀t : time
           if t = 0
           then out(t) = T
           else out(t+1) = (in(t+1) = ¬out(t))
```

5.1 Hardware Combinators

Many circuits are built from regular structures, such as arrays and trees. Figures 5 and 6 show two useful hardware combinators for building arrays of components. The first combinator, `array`, replicates a module `M` and connects one copy to each of the signals in two buses. The combinator `arrayRipple` can be used to build circuits using a ripple-carry structure. Each of these structure combinator captures a particular iteration scheme, independent of the internal structure of `cell` and of types of the buses. Luk and Page [LP91] have used similar techniques to describe a wide variety of parameterized hardware.

Our use of dependent types (in Nuprl) and restricted quantifiers (in HOL) enforces the restriction that the buses passed to the combinators must be of width `n` and that the types of the elements of each bus match the ports of `cell`. The advantage of using dependent types is that they allow us to keep track of these properties as part of type theory, as opposed to maintaining propositions in the logic. HOL supports restricted quantifiers which allow many of these properties to be stated as propositions without cluttering up theorems and proofs. These features allow us to define hardware combinators in a way that is general and yet safe.

The function `arrayRipple` takes the type of the carry signal (`CTy`); a positive integer (`n`) specifying the number of copies of the module to connect together; a cell (`cell`), which is the

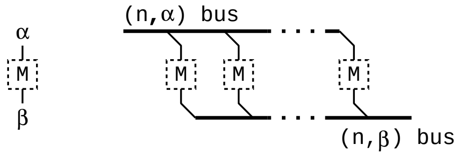

Figure 5: `array`

`n:N+` $\rightarrow$
  $((\alpha\times\beta)\rightarrow$`prop`$)\rightarrow$
    $(((\texttt{n},\alpha)\texttt{bus}\times(\texttt{n},\beta)\texttt{bus})\rightarrow\texttt{prop})$

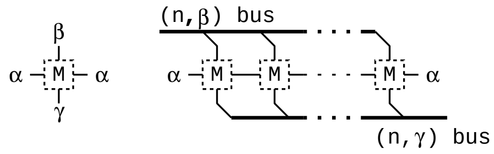

Figure 6: `arrayRipple`

`n:N+` $\rightarrow$
  $((\alpha\times\beta\times\gamma\times\alpha)\rightarrow\texttt{prop})\rightarrow$
    $((\alpha\times(\texttt{n},\beta)\texttt{bus}\times(\texttt{n},\gamma)\texttt{bus}\times\alpha)\rightarrow\texttt{prop})$

Hardware Combinators

module to replicate; a carry in signal (c_i); an input bus (`T`); an output bus (`B`); and a carry out signal (c_o). The implementation shown here defines `arrayRipple` as a base case and an inductive case. In the actual implementation we make use of a higher order function to do the recursion. The base case (`arrayRipple CTy 1 cell` (c_i, `A`, `B`, c_o)) produces one instance of its argument `cell`, connected to elements 0 of the `A` and `B` buses and to c_i and c_o. The inductive case (`arrayRipple CTy n cell` (c_i, `A`, `B`, c_o)) produces an instance of `cell` connected to elements `n-1` of `A` and `B`, and to the carry out from an `n-1` wide array of cells.

Definition 5 `Ripple carry hardware combinator`

```
arrayRipple CTy 1 cell (cᵢ, A, B, cₒ) = cell (cᵢ, A⟨0⟩, B⟨0⟩, cₒ)
arrayRipple CTy n cell (cᵢ, A, B, cₒ) =
   ∃cₓ : CTy
      cell (cₓ, A⟨n-1⟩, B⟨n-1⟩, cₒ)  ∧
      arrayRipple CTy (n-1) cell (cᵢ, A⟨0..n-2⟩, B⟨0..n-2⟩, cₓ)
```

We can use `arrayRipple` and a single bit full adder cell to build an `n`-bit ripple carry adder, as shown in Definition 6. The function `zip` performs an element-wise pairing of the elements of two input buses. That is, the `i`'th element of `zip A B` is a pair consisting of the `i`'th element of `A` and the `i`'th element of `B`.

5.2 Verification with Hardware Combinators

We make use of the hardware combinators to organize the process of verifying hardware designs. Because the ripple carry structure is defined recursively, verifying a ripple carry

Definition 6 `Implementation of ripple carry adder`

```
λ n (c_i, A, B) (Sum, c_o) .
   arrayRipple (bool, n) fullAdder (c_i, (zip A B)) (Sum, c_o)
```

Definition 7 `Specification of ripple carry adder`

```
λ n (c_i, A, B) (Sum, c_o).
   vec2num(2,n+1)(cat c_o Sum) =
   vec2num(2,n)(A) + vec2num(2,n)(B) + bool2num(c_i)
```

module is done by performing induction on the width of the module. By showing that an implementation satisfies a specification for the base case of a module with width one and then showing that inductive case also holds, the user may prove that the circuit is correct for all widths. The use of the induction lemma structures a proof by breaking down a sequent into two subgoals, one for the base case and one for the inductive case. The induction lemma also captures all of the reasoning required to show that the induction scheme is correct for the hardware combinator. So, for the ripple carry adder, applying the induction theorem would leave the user with two goals, one to show that a single full adder implements a one bit ripple carry adder and one to show that if an n-bit ripple carry adder is correct, then connecting a full adder to the ripple carry adder correctly implements an n+1 bit ripple carry adder.

Many times, iterated circuits are used to perform arithmetic functions where each cell in the array performs the same operation on a specific digit of the input vector. For a ripple carry array, Hanna [HDL90] has defined the notions of *factorizable* (Definition 9) and *proper* (Definition 8). If a specification can be shown to be factorizable and proper, then it can be implemented by iterating a single cell in a ripple carry fashion (called `fold` in Hanna's work). The factorization theorem (Theorem 2) says that if a specification (`Spec`) is proper and factorizable, and if an implementation (`cell`) satisfies `Spec` for numbers in the range {`0..m`}, then `l` copies of `cell` connected together in a ripple carry fashion will satisfy `Spec` for numbers in the range $\{0..m^l\}$.

Applying the factorization theorem in a proof reduces a goal from showing that a ripple carry array of cells satisfies an arithmetic specification to three simpler subgoals. The first two goals are to show that the relation is factorizable and proper. The third goal is to show that a single cell implements the base case of the specification. For the ripple carry adder this means proving that a full adder implements a single bit slice of base two addition.

Given a goal of the form shown in Figure 7, applying the factorization theorem will leave us with the subgoals shown in Figure 8.

Definition 8 Proper

$$\texttt{proper(CTy, Spec)} = \forall \texttt{c} : \texttt{CTy}.\ \ \texttt{Spec(1)(c,0,0)(0,c)}$$

Definition 9 Factorizable

```
factorizable(CTy,Spec) =
   ∀m_0, m_1 : N+
      ∀a_0, b_0, s_0 : {0..m_0}
         ∀a_1, b_1, s_1 : {0..m_1}
            ∀c_i, c_o : CTy
               let a = a_1*m_0 + a_0
                   b = b_1*m_0 + b_0
                   s = s_1*m_0 + s_0
               in
                  (∃c_x : CTy
                     Spec(m_0)(c_i,a_0,b_0)(s_0,c_x)  ∧
                     Spec(m_1)(c_x,a_1,b_1)(s_1,c_o) )
                  ⟹
                     Spec(m_0*m_1)(c_i,a,b)(s,c_o)
               end
```

Theorem 2 Factorization theorem

```
⊢ ∀CTy : U_i
     ∀m : N+ .
        ∀cell : CTy×({0..m}×{0..m}) → CTy×{0..m} → prop
           ∀Spec : m:N+ → (CTy×{0..m}×{0..m}) → ({0..m}×CTy) → prop
              ∀l : N+ .
                 ∀A, B, Sum : (l, {0..m}) vector
                    ∀c_i, c_o : CTy
                       proper(CTy,Spec) ⟹
                          factorizable(CTy,Spec) ⟹
                             arrayRipple(CTy,l)(cell)(c_i,zip(A,B))(Sum,c_o) ⟹
                             Spec(m^l)(c_i, vec2num(m,l)(A), vec2num(m,l)(B))
                                    (vec2num(m,l)(Sum), c_o)
```

```
–  arrayRipple(bool,l)(fullAdder)(ci,A,B)(Sum,co)
⊢? adderSpec(m^l)(ci, vec2num(m,l)(A), vec2num(m,l)(B))
        (vec2num(m,l)(Sum), co)
```

Figure 7: Goal before application of factorization theorem

```
⊢? proper(bool,adderSpec)
⊢? factorizable(bool,adderSpec)
⊢? ∀a,b,sum:bool.  ∀ci, co:bool.
        fullAdder(ci,(a,b))(sum,co) ⟹
           adderSpec(1)(ci, vec2num(2,1)(a)), vec2num(2,1)(b)))
                       (vec2num(2,1)(sum)), co)
```

Figure 8: Subgoals after application of factorization theorem

While our induction lemma works for all properties of a cell iterated using `arrayRipple`, the factorization theorem only works for a specific class of arithmetic relations. Because it is more restrictive, the factorization theorem is able to encapsulate more reasoning about arrayed circuits. The relevance of the factorization theorem to our work is that it is another example of the power that can be gained by making a few restrictions on the style of hardware design and verification.

We do not want to automatically apply the factorization theorem every time we are verifying a circuit built in a ripple-carry style, because the specification may not be factorizable and proper. One solution would be to look for theorems in the library which show that the specification is factorizable and proper. If these theorems exist, then we proceed to apply the factorization theorem. If we cannot find the appropriate theorems for factorization, then we will probably want to apply the induction theorem for the combinator. The general principle here is that we apply the most specific theorem that is applicable. This will ensure that we get the maximum power out of the automated tactics without proceeding down a path that the user would not choose.

6 Discussion

In this section, we discuss issues that have arisen in the process of implementing tactics and combinators. These include issues in automatic rewriting, documentation provided by the proof system, and what happens when our tactics fail to produce the expected results.

6.1 Automatic Rewriting

One approach we use is to rewrite with lists of lemmas automatically. Currently only `RWBoolLemmas` uses this approach but we can envision other applications of the same technique. For example, we may wish to rewrite with lemmas useful for bit vector reasoning or with arithmetic lemmas concerning division and exponentiation. Our main concerns with automatic rewriting are efficiency and termination.

Both HOL and Nuprl use discrimination nets to efficiently match a rule against an expression. We wish to efficiently match sets of rules against sets of expressions. For example, for a Boolean expression without negations, we don't need to try rewriting with the negation rules. This is an active area of research; we hope to learn from others' experiences. .

Another issue in automated rewriting is termination. In addition to providing an efficient rewriting package, we must be able to guarantee that it will always terminate. Currently, we use only a few rules and the rules themselves are simple enough so this is not a problem. However, there is nothing to stop a user from adding rules that will change this. One way to guarantee termination is to always rewrite toward a normal form. This may cause the expressions to become quite large, however.

The best approach to automated rewriting within limited domains (such as Boolean or integer expressions) may be to use decision procedures. Faust [KKS91] is a decision procedure for the restricted sequent calculus in HOL. There are decision procedures for Presburger formulas in both HOL and Nuprl [Sho77]. Nuprl also has a decision procedure for simple arithmetic reasoning. The decision procedures for HOL can generate the HOL proof required; unfortunately this safe mode may take as long as several hours.

6.2 Documentation

In our ideal system `SuperDuperHWTac` proves all hardware proofs completely, or leaves the user with only a few subgoals to prove. One disadvantage of this is that the user no longer gains any deeper understanding of the circuit being verified as a result of doing the proof. Instead the user must place more trust in the correctness of the proof system. In Nuprl we can run a tactic like `SuperDuperHWTac` and get back a proof tree which explicitly shows all the primitive rules run to prove the goal. Such proofs contain so much detail that users very rarely wade through them. The end result is only marginally better than providing no information at all.

We envision a system where the user gets back a proof at the level she would have written for a hand proof. The proof should show the steps which made progress toward solving the goal and the rewrites that succeeded, preferably by formula rather than by name. We plan to implement such a documentation mode in the future. This mode would provide documentation of a proof that is distinct from the interaction the user has with the proof system at the time of running the proof. Currently in Nuprl and HOL the interaction at run time and the documentation are one and the same.

We have some ideas for implementing this. We could mark each tactic with a flag which indicates whether or not that tactic is to be provided as part of the documentation. All of the smart tactics described in this paper would not be part of the documentation. All of the rewrite rules and lemmas would be. We believe that at the time of tactic writing we can tell (in most cases) what level the user wants to see. For example, with the tactic `RewriteWith bool_lemmas`, the user wants to see the formulae of the rewrites that succeeded. In addition, we would annotate tactics with the message reported to the user. For `BoolCaseSplit`, the user wants to see "By case analysis on term `b`", etc.

6.3 Debugging Proofs and Tactics

So far we have assumed that `SuperDuperHWTac` does exactly what we want; that is, it always makes meaningful progress and only leaves the user with a few subgoals to prove interactively. The tactic will at least simplify the initial goal, so it cannot completely "fail". There are two possible behaviors the user may need to deal with. These roughly correspond to the case where the tactic does not behave as expected, and the case where the goal the user is trying to prove is incorrect.

The first case occurs when `SuperDuperHWTac` does not make as much progress as the user would have expected. In the worst case, the tactic gets no further than `Autotactic` would have done. The other extreme is that `SuperDuperHWTac` generates subgoals that the user either cannot prove, or does not understand why they were produced. In either case, the user needs sufficient documentation to see what `SuperDuperHWTac` actually accomplished. In the second case, seeing the rules that succeeded will inform the user how the proof got to the current state. In the first case, the user may also wish to see which tactics failed in order to get a handle on where things are going wrong. More experience with `SuperDuperHWTac` will help us give suitable information to the user in both cases. We may also wish to provide a debugging mode where users can single step through the proof at a suitable level of detail.

7 Conclusions

We have discussed ways of capturing much of the tedious reasoning in hardware proofs. We do this through two techniques. The first is the use of smart tactics to capture low level reasoning that frequently occurs in combinational and sequential hardware verification. The second is to use hardware combinators to capture repetitive reasoning found in iterative hardware structures. We expect that the combination of these two techniques as well as the use of parameterized modules [AL92] will provide a powerful methodology for hardware verification.

The work described so far is in its early stages. We plan to use our approach on a wide variety of hardware modules. In particular we intend to use this approach to develop a library of circuits for floating point arithmetic. As this research develops, we expect some of the details described here to change. The general approach of giving the user more powerful tools to more easily do hardware verification will remain.

Acknowledgements

Mark Aagaard is supported by a fellowship from Digital Equipment Corporation. Miriam Leeser is supported in part by an NSF Young Investigator Award CCR-9257280. Phil Windley is supported in part by the National Science Foundation under Research Initiation Grant MIP-9109618. Windley did most of his work while he was at the University of Idaho.

References

[AL92] M. D. Aagaard and M. E. Leeser. A methodology for reusable hardware proofs. In L. Claesen and M. Gordon, editors, *HOL Theorem Proving System and its Applications*. North-Holland, September 1992. To appear.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[CGM86] A. J. Camillieri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. North Holland, September 1986.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78. Springer Verlag, 1979. Lecture Notes in Computer Science.

[Gor86] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. North-Holland, 1986.

[HB90] Warren Jr. Hunt and Bishop Brock. The verification of a bit-slice ALU. In M. E. Leeser and G. M. Brown, editors, *Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, pages 282–306. Springer Verlag, 1990. LNCS 408.

[HDL90] F. K. Hanna, N. Daeche, and M. Longley. Specification and verification using dependent types. *IEEE Transactions on Software Engineering*, 16(9):949–964, September 1990.

[KKS91] R. Kumar, T. Kropf, and K Schneider. Integrating a first-order automatic prover in the HOL environment. In M. Archer and M. Gordon, editors, *International Workshop on the HOL Theorem Proving System and its Applications*, pages 170–176. IEEE Press, 1991.

[Lee92] M. E. 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*. Prentice-Hall International Series on Computer Science, 1992.

[LP91] W. Luk and I. Page. Parametrising designs for Field-Programmable Gate Arrays. In W. Moore and W. Luk, editors, *FPGAs*, pages 284 – 295. Abingdon EE&CS Books, 1991.

[Mel92] T.F. Melham. HOL logic extended with quantification over type variables. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*. Elsevier Science Publishers, B.V., 1992.

[ML82] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy*, pages 153–175. North-Holland, 1982.

[Sho77] Robert E. Shostak. On the SUP–INF method for proving presburger formulas. *Journal of Automated Reasoning*, 24(4):529–543, Oct 1977.

[SKK91] K. Schneider, R. Kumar, and T. Kropf. Structuring hardware proofs: First steps toward automation a higher order environment. In *International Conference on VLSI*, 1991.