Verifying a Logic Synthesis Tool in Nuprl: A Case Study in Software Verification

Mark Aagaard, Miriam Leeser*

School of Electrical Engineering, Cornell University, Ithaca NY 14853, USA

Abstract. We have proved a logic synthesis tool with the Nuprl proof development system. The logic synthesis tool, *Pbs*, implements the weak division algorithm, and is part of the *Bedroc* hardware synthesis system. Our goal was to develop a proven and usable implementation of a hardware synthesis tool. *Pbs* consists of approximately 1000 lines of code implemented in a functional subset of Standard ML. The program was verified by embedding this subset of SML in Nuprl and then verifying the correctness of the implementation of *Pbs* in Nuprl. In the process of doing the proof we learned many lessons which can be applied to efforts in verifying functional software. In particular, we were able to safely perform several optimizations to the program. In addition, we have invested effort into verifying software which will be used many times, rather than verifying the output of that software each time the program is used. The work required to verify hardware design tools and other similar software is worthwhile because the results of the proofs will be used many times.

1 Introduction

This paper describes our experiences in using the Nuprl proof development system to verify the correctness of a logic synthesis tool. The lessons that we have learned are applicable to researchers using theorem proving based methods to verify functional programs.

We have implemented and proved *Pbs*: Proven Boolean Simplification. *Pbs* is based on Brayton and McMullen's weak division algorithm for logic synthesis [BM82]. The implementation of *Pbs* required approximately 1000 lines of Standard ML code. The proof of *Pbs* consists of a formal description of the properties to be proved, a formal semantics for the implementation language (a functional subset of Standard ML), and a mechanized formal proof showing that the implementation satisfies the properties claimed by the weak division algorithm. The proof was done in the Nuprl proof development system [C+86], and involved emulating a subset of SML in Nuprl, verifying the implementation of *Pbs* in Nuprl and showing that the semantics for the subset of SML that we

* Mark Aagaard is supported by a fellowship from Digital Equipment Corporation. This research was supported in part by the National Science Foundation under Award No. MIP-9100516.

emulated in Nuprl are equivalent to those defined for SML. Although the development of the proof required a significant amount of time, the results are used over and over again. Thus, the expenditure was well worth the effort.

The weak division algorithm was first described in a paper by Brayton and McMullen in 1982 [BM82]. It is currently used in several CAD tools, including *Mis*, which is part of the Berkeley Synthesis System [BR+87]. Our work is based upon the definitions, algorithms, and proof outlines presented in these articles. In some cases we have clarified previous definitions and algorithms, and in many instances we have developed formal proofs from the informal outlines presented earlier. The aim of this work is to prove an *implementation* of the weak division algorithm. Because of this, we reason about the algorithm at a much more specific and lower level than that of earlier efforts.

Theorem proving based formal methods have used mathematics to model and reason about a wide variety of different subjects. Originally, most theorem proving based work in digital hardware was done by proving the correctness of an implementation after it was designed [CGM86, Hun86]. This methodology suffers from the fact that such a *post hoc* verification process is invariably time-consuming and labor intensive. Many researchers are proving hardware design tools correct and investigating synthesis by proven transformations. For example, Martin [Mar90] uses proved correct transformations to synthesize delay insensitive circuits, Chin [Chi90] uses verified design procedures to synthesize array multipliers. More recently, McFarland [McF91] found several errors in the System Architect's Workbench [TDW+88] while proving their transformations correct. *Pbs* is the only work being done in applying formal methods to logic synthesis.

Our implementation of *Pbs* is described in Sect. 2. We outline our proof techniques in Sect. 3. Sect. 4 analyzes the results of the verification of *Pbs*. More detailed descriptions of the algorithms used in *Pbs* can be found elsewhere [AL91, Aag92].

2 Implementation of PBS

Pbs implements the weak division algorithm, which is a global approach to Boolean simplification. This means that the algorithm works with an entire system of Boolean equations at once. In contrast, local optimization techniques examine and optimize individual or small sets of equations independently.

The weak division algorithm seeks to decrease circuit area by removing redundant combinational logic. A sub-circuit contains redundant logic if it implements precisely the same function as another. Weak division removes redundant logic by finding common subexpressions among the *divisors* of different functions. The common subexpressions are replaced by new intermediate variables. This results in the duplicated logic being implemented only once, thereby reducing the area of the circuit.

For example, (1) contains two functions, one defining the variable p and one defining the variable q.

$$p = (a \wedge b \wedge c) \vee (a \wedge b \wedge d) \vee (a \wedge b \wedge e)$$
$$q = (g \wedge c) \vee (g \wedge d) \vee h \quad (1)$$

There is one common subexpression, $(c \vee d)$, among the divisors of p and q. We can substitute a new variable z into the equations in place of $(c \vee d)$. Next, we can substitute a new variable (x) for the term $(a \wedge b)$, which appears twice in the expression for p. This results in the set of equations shown below.

$$p = x \wedge z \vee x \wedge e$$
$$q = (g \wedge z) \vee h$$
$$z = c \vee d$$
$$x = a \wedge b \quad (2)$$

These substitutions have reduced the size of the circuit from twelve two input gate equivalents to seven, because the factors $(c \vee d)$ and $(a \wedge b)$ are now only implemented once. The substitutions increased the delay through the circuit from two gate delays to three, because the signals z and x added an additional layer of logic to the circuit. We have found that as the size of circuits increase the reduction in area increases significantly, but the additional delay converges rapidly: we are able to achieve reductions in area of 88% for circuits with more than three thousand gates, but yet add only nine additional layers of logic.

Our goals were for *Pbs* to be a proven *and* usable implementation of the weak division algorithm. In order to meet these two goals, we decided to embed a functional subset of the Standard ML (SML) programming language in the Nuprl proof development system. This allows the code for *Pbs* to be reasoned about in Nuprl and compiled and run using an SML compiler. Thus there is a very high degree of confidence that the SML implementation has the same behavior as the Nuprl implementation.

Standard ML is a very high level programming language and is based upon a formal definition which prescribes the precise semantics of the language [RM90]. SML is primarily a higher order functional language, but it does support some non-functional features, such as sequential operations, references, and exception handling. SML is strongly typed and polymorphic, thus it closely parallels much of the Nuprl type system.

Nuprl [C+86] is a mechanical proof development system based upon Martin-Löf's constructive type theory. In Nuprl, the user begins by entering a theorem to be proved. The theorem represents the *goal* of a proof. The user applies *tactics* which manipulate the goal, usually by breaking it down into a set of subgoals. This process of using tactics to break goals down into subgoals creates a structure known as a proof tree. In order to successfully complete a proof, the subgoals should become increasingly simple. Eventually an individual sub-goal will be simple enough that it matches one of the primitive rules in the Nuprl logic. When all of the leaves of the proof tree have been shown to be true, the proof is completed and the original theorem is proved.

Nuprl contains a set of primitive operations which are the basis for its computation system. Many SML instructions are very similar to these primitive operations. By limiting *Pbs* to use only these instructions, we were able to emulate

a subset of SML in Nuprl. The primitive operations upon which we based our subset include integer arithmetic, list recursion, integer equality, string equality, and pairing. The principal features which we did not include in our subset (because of the difficulty of implementing them in terms of the Nuprl primitives) are: references, exceptions, sequentiality, explicit recursion, pattern matching, real numbers, modules, streams, and records.

We now demonstrate this method by showing the definition of our primitive list recursion function in Nuprl and SML. Nuprl provides a primitive list recursion operation (*list_ind*), while SML does explicit recursion (the name of the function appears within the body of the function). Equation (3) shows the semantics for *list_ind* in Nuprl by describing its behavior on an empty list and on a non-empty list.

$$\begin{aligned} &\mathit{list\_ind}(\mathit{nil};\ \mathit{nil\_val};\ h,t,\mathit{rest}.\ f(h)(t)(\mathit{rest})) &&= \mathit{nil\_val} \\ &\mathit{list\_ind}(\mathit{hd}::\mathit{tl};\ \mathit{nil\_val};\ h,t,\mathit{rest}.\ f(h)(t)(\mathit{rest})) &&= \\ &\quad f(\mathit{hd})(\mathit{tl})(\mathit{list\_ind}(\mathit{tl};\mathit{nil\_val};\ h,t,\mathit{rest}.\ f(h)(t)(\mathit{rest})) && \end{aligned} \tag{3}$$

Following the approach outlined above, we wrote a function (*recurse*) in SML (Equation (4)) which has the same behavior as the list recursion primitive in Nuprl.

$$\begin{aligned} &\mathit{fun}\ \ \mathit{recurse}\ \ f\ \ \mathit{nil\_val}\ \ \mathit{nil} &&= \mathit{nil\_val} \\ &\ \ |\ \ \mathit{recurse}\ \ f\ \ \mathit{nil\_val}\ \ (\mathit{hd}::\mathit{tl}) &&= f\ (\mathit{hd})\ (\mathit{tl})\ (\mathit{recurse}\ f\ \mathit{nil\_val}\ \mathit{tl}) \end{aligned} \tag{4}$$

Equation (5) shows the definition of *recurse* in Nuprl. All other recursive functions in *Pbs* are written in terms of *recurse*. By using this methodology we have isolated the functions that are dependent upon primitives in Nuprl down to a very small number of low level functions.

$$\begin{aligned} &\mathit{recurse}\ f\ \mathit{nil\_val}\ \mathit{a\_list} = \\ &\quad \mathit{list\_ind}(\mathit{a\_list};\ \mathit{nil\_val};\ h,t,\mathit{rest}.\ f(h)(t)(\mathit{rest})) \end{aligned} \tag{5}$$

To complete the process, we proved Thms 1 and 2, which show that the Nuprl definition of *recurse* has the same behavior as the SML function. Using this methodology we defined and verified each of the constructs in the subset of SML that we emulated. Having defined the function *recurse*, we can now use it in our implementation of other functions and can use Thms 1 and 2 to prove theorems describing the behavior of functions built upon *recurse*. Using this methodology, the SML code for a function is identical to the Nuprl object representing the function.

The only informal link in the connection between Nuprl and SML arises because Nuprl uses lazy evaluation and SML uses eager evaluation. In reality, this does not pose a problem for us, because the subset of SML that we are using is purely functional and all of the functions are guaranteed to terminate. (We are able to prove termination because the only recursion done in *Pbs* is list recursion using Nuprl's list induction primitive, which always terminates.) Purely functional programs with guaranteed termination will exhibit identical

Thm 1 *Recurse - base case*

$\vdash \forall f, nil\_val.$
$\quad recurse\ f\ nil\_val\ nil = nil\_val$

Thm 2 *Recurse - inductive case*

$\vdash \forall f, hd, tl, nil\_val.$
$\quad recurse\ f\ nil\_val\ (hd::tl) = f(hd)(tl)(recurse\ f\ nil\_val\ tl)$

behavior in eager and lazy evaluation environments. Thus, for the subset that we are using, programs will have identical behavior in Nuprl and in an SML compiler. Ongoing research at Cornell includes work aimed at creating a type theoretic semantics for SML within Nuprl. Once this has been done, programs will be able to be verified without relying on informal arguments to show the correspondence between the Nuprl and SML semantics.

3 Verification of PBS

The proof of *Pbs* shows two things. First, the output circuits generated by *Pbs* are functionally equivalent to the input circuits. Second, all output circuits satisfy the minimality property claimed by the weak division algorithm. Informally, a circuit with this property is completely irredundant - that is, there is no duplicated logic in the circuit. Others have shown that circuits which satisfy this minimality property are completely single stuck-at fault testable [HJKM89].

In doing the proof of *Pbs* we began with a specification of the overall algorithm and our implementation in Standard ML, which we had tested on a number of sample circuits. Our approach was to write several theorems describing the behavior of each function in *Pbs* and then to prove that the code used to implement the function satisfied the theorems that we had written. In general we worked in a bottom up fashion. We began with very simple functions, such as adding an element to a list, and testing if an element is a member of a list. After proving that these function had their intended behavior, we were able to move up a level in the hierarchy, and prove theorems describing the behaviors of more complicated functions.

There are two basic categories of theorems in the proof of *Pbs*. The first is theorems which describe abstract properties of functions. The second, and more common, category is theorems which describe the behavior of functions at a level which is very close to the actual implementation. The first category of theorems includes the theorem that the output of *Pbs* satisfies the correctness criteria for the weak division algorithm. Theorems in the second category usually describe how a function behaves for certain inputs. For example, the function for dividing Boolean expressions is partially characterized by a theorem which

states that dividing an empty expression by any expression produces an empty expression.

For the first category of theorems, we did not find any specific methodology which was applicable to all proofs. For the second category of theorems, we found a technique which was used for these theorems throughout the proof of *Pbs*. This technique consists of four steps: list induction, unfolding definitions, rewriting and application of previously proven lemmas. As an example of these techniques, we describe the proof of a theorem describing membership in a list (Theorem 3). This is a trivial example; it is included here because it illustrates the techniques which we used throughout the verification of *Pbs*.

Thm 3 *Membership in a non-empty list*

$\vdash \forall A,\ eq\_fn,\ tl,\ hd,\ a.$
$\quad mem\ eq\_fn\ a\ (hd::tl) \iff$
$\quad (eq\_fn\ a\ hd) \vee (mem\ eq\_fn\ a\ tl)$

As an alternative to the approach taken here, we could have defined the membership function in such a way that Nuprl could have completed the proof of this function automatically. This approach would have been similar to that of proof systems which are capable of automatically verifying many inductively defined functions [BM88]. We could have done this by writing the function directly in terms of Nuprl's primitive list induction operator, which was described in Sect. 2. In Nuprl, there are several disadvantages to choosing this alternative. Most importantly, it would prevent our implementation of *Pbs* in Nuprl from being the same as our implementation in SML. Secondly, verifying more complicated functions in this alternative style would be more difficult than in the style which we used. By using the function *recurse* as the only primitive function for recursion, we were able to hide the implementation details of recursion and thereby prevent our proofs from becoming cluttered with low level details.

The complete proof of the theorem describing the membership function is shown in Figure 1 and is discussed in the following paragraphs. In the proof, only the conclusion and the rule for each step are shown. The hypotheses contain variable declarations and are not modified in the proof. The rules, which appear after "**BY**", are the only text other than the initial goal that the user types in.

Because lists are so pervasive in *Pbs*, most of the functions in *Pbs* are defined in terms of list recursion. This also means that most proofs rely on list induction. Thm 3 shows the inductive case for membership in a list, which says that an element is a member of a list if and only if it is equal to the head of the list or it is a member of the tail of the list. Another theorem (which is not shown), describes the base case for this function. The theorem for the base case says that an empty list does not have any members.

One of the first steps of each proof is to unfold the definition of the function

$\vdash$ $(mem\ eq\_fn\ a\ (hd::tl))$
$\iff$
$(eq\_fn\ a\ hd)\ \vee\ (mem\ eq\_fn\ a\ tl)$
BY $(RewriteConcl\ (NthC\ 1\ (UnfoldC\ `mem`))...)$

$\vdash$ $(reduce\ (fn\ hd\ =>\ fn\ result\ =>$
$(eq\_fn\ a\ hd)\ orelse\ result)\ false\ (hd::tl))$
$\iff$
$(eq\_fn\ a\ hd)\ \vee\ (mem\ eq\_fn\ a\ tl)$
BY $(RewriteConcl\ reduce\_ht\_convn...)$

$\vdash$ $(eq\_fn\ a\ hd)\ orelse$
$reduce\ (fn\ hd\ =>\ fn\ result\ =>$
$(eq\_fn(a))(hd)\ orelse\ result)\ false\ tl)$
$\iff$
$(eq\_fn\ a\ hd)\ \vee\ (mem\ eq\_fn\ a\ tl)$
BY $(RewriteConcl\ mem\_fold\_convn...)$

$\vdash$ $(eq\_fn\ a\ hd)\ orelse\ (mem\ eq\_fn\ a\ tl)$
$\iff$
$(eq\_fn\ a\ hd)\ \vee\ (mem\ eq\_fn\ a\ tl)$
BY $(RewriteConcl\ orelse\_x\_x\_convn...)$

Fig. 1. Proof of Theorem 3

being described. In Nuprl "unfolding" means to replace an instantiation of a function with the code used to implement the function. It is analogous to the compiler optimization of in-line expansion. The purpose of unfolding definitions is to reveal the implementation of functions. When this has been done, rewrite rules or lemmas describing lower level functions can be used in the proof. In the first step of the proof the definition of the function *mem* is unfolded to reveal that it is implemented in terms of *reduce*. The function *reduce* (Equation (6)) is a higher order recursive function which is defined using the function *recurse* (Equation (4)).

```
fun reduce f nil_val a_list =
    let
        fun f2 hd tl result = f hd result
    in
        recurse f2 nil_val a_list
    end
```
(6)

As illustrated here, unfolding is really just one type of rewriting that can be

performed. Nuprl has a very powerful rewriting package, which is used to replace one term with another term, where the two terms are related by some property. This property does not have to be equality, it may be any relation which the user has proved to be reflexive and transitive. The rewrite package supports rewriting terms in hypotheses as well as in the conclusion. Rewrite rules may be constructed from previously proven lemmas, hypotheses in the current proof, or direct computation. Lemmas may be used to construct *conditional* rewrite rules, that is, a rule which only holds under certain conditions. We made extensive use of these features throughout the proof of *Pbs*.

In the proof of Thm 3, four different rewrite rules are used. In the first and third steps, direct computation rules are used to fold and unfold the instantiation of *mem*. The rewrite rules used in steps two and four are derived from lemmas that were proved about the functions *reduce* and *orelse*.

Although not a proof technique, Autotactic is a very important tactic which was used throughout the proof of *Pbs*. Autotactic is comprised of a collection of tactics which can be used to handle many of the minor details involved in using mechanical proof systems. These proof systems offer a high degree of confidence in the correctness of the theorems proved with their use, but the tradeoff is that the user is exposed to a great many details that are usually ignored in paper proofs. Common uses of Autotactic include automatically introducing universally quantified variables that appear in conclusions and proving type checking goals. In each step of the proof, Autotactic was used after applying the rewrite rule. Using Nuprl's display forms, Autotactic is represented by the (...) in the proof steps.

By adopting the proof style described here, we are able to write concise theorems describing complex functions. An example of this appears in Thm 4, which describes the behavior of the quotient function for dividing an expression by a cube (2). Theorem 4 says that a cube (*co*) is a member of the quotient of an expression (*ei1*) and another cube (*ci2*) if and only if there is a cube (*ci1*) in *ei1* such that *ci2* is a subset of *ci1* and *co* is equal to *ci2* deleted from *ci1*. This theorem was proved in a total of fourteen steps, which included seven rewrites and three lemma applications (Lemma application is one of the four primary techniques used throughout *Pbs*, but was not demonstrated in the proof of the membership function).

4 Discussion

This section describes reasons for verifying software, lessons that we learned about theorem proving techniques for software verification, an analysis of the amount of time required to verify *Pbs* with Nuprl, and some directions for future research.

The verification of *Pbs* was valuable because we found several errors while formalizing the proof, we were able to safely perform several optimizations to the code, and we gained a much deeper understanding of the algorithm. In the process of verifying *Pbs*, several obscure errors in the implementation and

Thm 4 *Membership in a quotient*

```
⊢∀ei1:Expr_t.
   ∀ci2,co:Cube_t.
      is_valid_expr e1 ⇒
         tr(MEM_ce(co)(QUOT_ec(ei1)(ci2))) ⟺
         ∃ ci1:Cube_t.
            tr(MEM_ce(ci1)(ei1)) &
            tr(IN_cc ci2 ci1) &
            tr(EQc co (DEL_cc(ci1)(ci2)))
```

```
fun QUOT_ec ei1 ci2 =
   let
      fun f c_hd result =
         if IN_cc ci2 c_hd
         then (DEL_cc c_hd ci2)::result
         else result
   in
      Cs2E(reduce f NIL_e (E2Cs ei1))
   end
```

Fig. 2. Function for quotient of an expression and a cube

formal description of *Pbs* were found. The nature of the errors was such that they would most likely manifest themselves only in rare occurrences in large systems of equations, exactly the times when they would be least likely to be detected. These errors are described elsewhere [Aag92].

If a program or optimization is not completely understood, performing the optimization on the code may introduce bugs into the program. For most of the operations in the weak division algorithm there is both a Boolean and an algebraic function which may be used. The algebraic functions are much faster, but return the correct result only under certain conditions. In the original implementation of *Pbs*, only Boolean operations were used. This sacrificed speed for increased assurance that the code was correct. In doing the proof in Nuprl, several instances were discovered where the correctness conditions for the algebraic operations could be guaranteed. When these occurrences were found, Boolean operations were replaced by algebraic operations. This increased the speed of the code and also simplified the proof, because the lemmas describing the algebraic operations were simpler than those describing the Boolean ones.

In the process of verifying *Pbs*, we discovered several guidelines which are useful when writing code which will be verified or when reasoning about a pro-

gram. Beginning with a mathematically defined, very high level language greatly eases the process of proving a program. In addition there are certain programming techniques and styles which can significantly increase the ability to reason about a program.

- All functions should be very short
- Code should be written in an extremely modular style
- Higher order functions should be used wherever possible

These guidelines may seem to be very obvious, but their importance can not be over emphasized. Ideally, each function performs only a single operation and the behavior can be summarized in one or two lemmas. By writing code in a very modular style, a single function and its corresponding lemmas may be used many times. When just writing code, it may seem easier to simply duplicate a piece of code if it is extremely short and is only used a few times. But, when proving code correct, not only must the code be duplicated, but the proofs describing the code must also be duplicated. Along these same lines, the use of higher order functions to handle such tasks as recursion is a much better approach than to try to do explicit recursion.

When we began the verification effort, we quickly learned that each lemma should only bridge two adjacent levels of abstraction. That is, only one function should be unfolded in each proof. This means that each proof is only dependent upon the implementation of a single function. Following this guideline helps ensure that lemmas are as general as possible, which makes them more useful, and requires that fewer total lemmas be written.

Although we learned most of these guidelines while in the process of working on the proof of *Pbs*, a few were not recognized until we had completed the proof and were able to analyze our work as a whole. A technique which did not occur to us was to try to generalize the reasoning to general mathematical principles. The operations in *Pbs* can be described as an algebra. There is a large body of existing knowledge about algebras, which we could have used. Instead, we proved special theorems for each function. Had we shown that the operators in *Pbs* were an algebra, we could have used general theorems about algebras to do the more complicated and abstract reasoning in *Pbs*.

An important tool which we could have made use of, but did not, was the ability to execute our code as we were verifying it. When we developed *Pbs* we did substantial amounts of debugging using informal techniques before beginning to formally verify the code. But we made a number of changes to the implementation as we developed the proof (some were minor bug fixes, others were done to make the proof easier or optimize the code). We did not try running the code with any of these modifications, instead we relied solely upon our proof for debugging these changes.

Looking back on this decision, it is now apparent that it would have been more efficient to do some informal debugging of the modified code, before we spent the time to do the formal verification. The primary reason for this is that testing code on a few test cases can be an extremely fast method to gain some

measure of confidence that the code behaves as desired. Also, with complicated specifications, there may be some doubt as to whether the specification actually describes the intended behavior of the program. For these reasons, such techniques as executable specifications can be very useful.

The implementation of weak-division consists of approximately one thousand lines of code. In the process of implementing *Pbs* and doing the proof in Nuprl there was a large learning curve and several new tools were written to make the proof easier. We estimate that if we were to do it over again, it would take approximately one month to implement the code and an additional two months to complete a formal proof on paper. We believe that using the knowledge gained and tools written, it would take a total of four months to do the proof in Nuprl all over again. Thus, doing the proof in Nuprl would take approximately twice as long as doing the proof on paper.

One of the lessons learned in the process of doing the proof is that there is a potential for automating several aspects of the proof process for software verification. This area has not yet been fully explored, so it is difficult to say exactly how much of the proof could be automated, but we estimate that it is feasible to reduce the time to do the proof in Nuprl from four months to three months. (Compared with two months for doing the formal proof on paper.) We are currently investigating some of these ideas. Ideally, doing the proof in a mechanical theorem prover would not take any longer than doing the formal proof on paper, but that time has not yet arrived.

5 Conclusion

A piece of software is verified in order to have higher confidence that it does what it is meant to do. It is very unlikely that the day will come when all software is formally verified, so it is important to decide how much verification should be done for a given piece of software. Often, there are informal proofs that describe an algorithm at an abstract level, but there may be a great disparity between the level of detail of the proof and the implementation. It is in this process of going from an abstract description of an algorithm to the concrete implementation that many errors are introduced. An informal proof that the algorithm is correct is a good first step toward a correct implementation, but unless formal verification is done at the level of implementation, there can not be a high level of assurance that the final implementation is correct. The tradeoff for this increased assurance is that there are a great many details that the proof must take into account. One of the most important advantages of mechanical proof systems is their ability to manage proof efforts and potentially automate a significant portion of this process.

Such was the case with *Pbs*. Brayton and McMullen had provided informal arguments that the weak division algorithm was correct, but by using Nuprl we were able to formally prove *Pbs* at the implementation level. The extra effort of performing a formal proof definitely *was* worthwhile. The proof provided us with greater understanding of the algorithm and allowed us to take advantage

of optimizations to the program. The code will be used many, many times, thus amortizing the initial work. Finally, people without any knowledge of formal methods can easily use *Pbs* and there is no need for *post hoc* verification of the circuits generated by *Pbs*.

6 Acknowledgements

We would like to thank Robert Constable and Jim Caldwell for reading an earlier draft of this paper and Paul Jackson, who implemented the rewrite package, for his advice and assistance in using Nuprl. In addition we would like to thank Chet Murthy and Doug Howe, who were always willing to help and gave us a number of useful suggestions.

References

[Aag92] Mark Aagaard. A verified system for logic synthesis. Master's thesis, Department of Electrical Engineering, Cornell University, January 1992.

[AL91] Mark Aagaard and Miriam Leeser. The implementation and proof of a boolean simplification system. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits, Oxford 1990.* Springer-Verlag, 1991.

[BM82] R. K. Brayton and C. McMullen. Decomposition and factorization of boolean expressions. In *International Symposium on Circuits and Systems*, 1982.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1988. Volume 23 of Perspectives in Computing.

[BR+87] R. K. Brayton, R. Rudell, et al. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6), 1987.

[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.

[Chi90] Shiu-Kai Chin. Combining engineering vigor with mathematical rigor. In *Proceedings of the MSI Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects.* Springer Verlag, 1990. LNCS 408.

[HJKM89] G. Hachtel, R. Jacoby, K. Keutzer, and C. Morrison. On the relationship between area optimization and multifault testability of multilevel logic. In *International Conference on Computer Aided Design*, pages 316–319. ACM/IEEE, 1989.

[Hun86] W. A. Hunt, Jr. *FM8501: A Verified Microprocessor.* PhD thesis, Institute for Computing Science, The University of Texas at Austin, 1986.

[Mar90] A. J. Martin. The design of a delay-insensitive microprocessor: An example of circuit synthesis by program transformation. In *Proceedings of the MSI Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects.* Springer Verlag, 1990. LNCS 408.

[McF91] Michael C. McFarland. A practical application of verification to high-level synthesis. In *International Workshop on Formal Methods in VLSI Design.* ACM, 1991.

[RM90] R. Harper R. Milner, M. Tofte. *The Definition of Standard ML.* The MIT Press, 1990.

[TDW+88] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn. The system architect's workbench. In 25^{th} *Design Automation Conference*, pages 337–343. ACM/IEEE, 1988.

This article was processed using the LaTeX macro package with LLNCS style