Formal Verification of the Sobel Image Processing Chip*

Paliath Narendran
General Electric Company
Corporate Research and Development
Schenectady, N.Y. 12345

Jonathan Stillman
State University of New York at Albany
Albany, N.Y. 12222

Abstract

We describe an approach to hardware verification in the context of our recent success in formally verifying the description of an image processing chip currently under development at Research Triangle Institute. We demonstrate that our approach, which uses an implementation of an equational approach to theorem proving developed by Kapur and Narendran, can be a viable alternative to simulation. In particular, we are able to take advantage of the "recursive" nature of many circuits, such as n-bit adders, and our techniques allow verification of sequential circuits. To the best of our knowledge this is the first time a complex sequential circuit which was not designed with formal verification specifically in mind has been verified. Finally, we describe the discovery of several design errors in the circuit description, detected during the verification attempt (the actual verification could only take place once these errors were removed), and discuss directions that future work will take. A significantly more detailed description of this work can be found in [NaSt 88].

1 Introduction

Formal verification of hardware involves using theorem-proving techniques to verify that a statement describing (not necessarily completely) the behavior of a circuit is a logical consequence of the structural description of the circuit, i.e., proving that the structure of the circuit forces it to behave as stated. There has been a great deal of recent interest in formal verification as an alternative to exhaustive simulation, since simulation is often not a practical approach for large circuits. Related work can be found in [Ba 84,Go 85], among others. It is hoped that formal verification tools can ultimately be developed which will be useful in many of the situations where exhaustive simulation is impractical.

Our approach utilizes an equational approach to theorem-proving that was developed by Kapur and Narendran, as implemented in the Rewrite Rule Laboratory (RRL) developed under NSF grant No. CCR-8408461. Both the structural and behavioral definitions are specified in a first-order predicate calculus with equality. The approach is refutational in nature: the statements describing the structure are assumed to hold, the behavior is assumed not to hold, and an attempt is made to reach a contradiction. Previous work which is closely related is described in [NaSt 87] and, for combinational circuits, in [ChPC 87]. In addition to the circuit described herein, we have verified a number of the leaf cells of a CMOS bit-serial compiler currently under development at GE CRD. This work will be described in an upcoming report.

Our recent work on formal hardware verification has been part of an effort to develop a designer's workstation in which the designer uses graphical tools to build circuits hierarchically; the resulting information can be used both to generate VHDL programs and to generate the structural information for formal verification of the design. A more complete description of the workstation (IVW) and the current level of integration of verification tools into it can be found in [NaSt 87]. To the best of our knowledge, this is the first time that formal verification tools have been integrated into a design environment.

The Sobel chip has approximately 10,000 transistors, and implements the Sobel edge detection algorithm [VaBK 87,Pr 78], which is commonly used in the image segmentation phase of military image processing systems. The structure of the design was completely specified in VHDL. We were not provided with a formal behavioral description of the chip, but we were able to extract the necessary information from [VaBK 87]. The overall effort (extraction and verification) took about two man-months.

2 Theoretical Background

The basis of our approach is the Kapur-Narendran method for theorem-proving for first-order predicate calculus. In theory, any complete proof method for

*work partially supported by Contract F33615-85-C-1862, AFWAL, Wright-Patterson Air Force Base, Ohio.

first-order logic would suffice; in practice, the Kapur-Narendran method as implemented in RRL seems to perform better than other automated theorem-proving methods that we have had access to. It is particularly easy to introduce first-order equational theories using this method; for details, we refer the reader to [KaNa 85]. We briefly describe the method, and give a simple example of how it may be applied to hardware verification; for a detailed description of the Kapur-Narendran method, one should consult [KaNa 85]. The method is closely related to methods for term-rewriting; the interested reader is referred to [Bu 83] for an overview of this topic, and to [HuOp 80,KnBe 70] for more detailed accounts. Basically, the Kapur-Narendran method involves taking a set of first-order formulae, performing any necessary Skolemization (see [ChLe 74]) to remove existential quantifiers, then translating the resulting formulae into an equivalent set of polynomials over a boolean ring whose operators are "exclusive-or" and "and" (sometimes denoted by $*$ and $+$, respectively), in which the atoms appearing in the original formulae make up the indeterminates of the polynomial equations. The original set of formulae is satisfiable if and only if the resulting system of polynomials has a solution. It is possible that a contradiction is found in the translation process itself; if not, the polynomials are oriented into rewrite rules and a modified version of the Knuth-Bendix completion procedure is performed on them using the method of critical pairs to compute new rules. One can view the set of polynomials as a basis of an ideal, and the completion procedure as computing the Gröbner basis of the original basis (see [Buc 85,MiYa 86] for details). Such a Gröbner basis has the property that every polynomial in the ideal can be rewritten to 0 using the polynomials in the basis for simplification. It can be shown that a set of first-order formulae is unsatisfiable if and only if the corresponding Gröbner basis contains 1 (where 1 stands for true and 0 stands for false). One should note, however, that if the formulae are simultaneously satisfiable, the Gröbner basis may be infinite, i.e., the completion procedure may not terminate.

To insure termination of rewriting, a partial well-founded ordering can be introduced on monomials; such an ordering is easily extended to polynomials by considering them to be multisets of monomials.

In order to orient the polynomials into rewrite rules, each polynomial can be broken up into two parts as follows: for a polynomial P, let *HD(P)* be the set of maximal monomials (using the chosen partial ordering) in P; let *TL(P)* be $P - HD(P)$. With the polynomial P we associate the rule $HD(P) \rightarrow TL(P)$. A polynomial Q can be rewritten using the rule associated with P if and only if there is a monomial m and a substitution θ such that $m * \theta(HD(P)) \subseteq Q$. When this occurs we obtain the polynomial Q' by replacing the occurrence of $m * \theta(HD(P))$ in Q with $m * \theta(TL(P))$, and say that $Q \rightarrow Q'$ by the rule associated with P. The above defines a *rewrite relation* $\rightarrow$ induced by a set of rules, and we denote the reflexive and transitive closure of $\rightarrow$ by $\stackrel{*}{\rightarrow}$. In addition to the rules obtained from the polynomials, rules are also introduced for idempotence of "and" (for every predicate $P(x_1, \ldots x_n)$, introduce the rule $P(x_1, \ldots x_n) * P(x_1, \ldots x_n) \rightarrow P(x_1, \ldots x_n)$) and nilpotence of "exclusive-or" (add the rule $1 + 1 \rightarrow 0$).

New rules are generated by computing *critical pairs*. This is done by finding, for each pair of rules, a minimal polynomial which can be rewritten by both rules. The critical pair is the pair of polynomials p_1, p_2 resulting from rewriting the minimal superposition using each of the two rules involved. The polynomial $p_1 - p_2$ associated with a critical pair is called an *S-polynomial*. If the polynomial $p_1 - p_2$ can be rewritten to 0 the critical pair is said to be *trivial*, and no new rule is introduced; if not, the resulting polynomial is oriented into a rule and added to the rule set. In general, computing superpositions is somewhat involved; in the interest of brevity, we only illustrate the case when $HD(p_1)$ and $HD(p_2)$ are both monomials. The general case is fully explored in [KaNa 85]. Let $HD(p_1) \rightarrow TL(p_1)$ and $HD(p_2) \rightarrow TL(p_2)$ be two (not necessarily distinct) rules, where $HD(p_i)$ can be partitioned into monomials $g_i f_i, i = 1, 2$, and there exists a most general unifier σ which unifies g_1 and g_2. Let $\sigma(g_1) = \sigma(g_2) = g$. Then the polynomial $p = \sigma(f_1)g\sigma(f_2) = \sigma(HD(p_1))\sigma(f_2) = \sigma(HD(p_2))\sigma(f_1)$ is a superposition of p_1 and p_2. The resulting *S-polynomial* is $\sigma(TL(p_1))\sigma(f_2) - \sigma(TL(p_2))\sigma(f_1)$.

The completion procedure simply involves repeatedly computing critical pairs until either no more rules can be superposed or the rule $1 \rightarrow 0$ is generated, signaling a contradiction (since we are using a refutational approach, this means that the original formula is valid). If the procedure halts without finding a contradiction, the polynomials in the resulting Gröbner basis can be used to find a model for the original formula; remember, however, that there is no guarantee that the procedure will terminate in all cases.

Rather than go into more technical detail, we present the following example to help illustrate the method.

3 An Example

We demonstrate a proof that the Muller C-element (see figure below) indeed functions as a consensus mechanism; in the description of the circuit the variable **x**

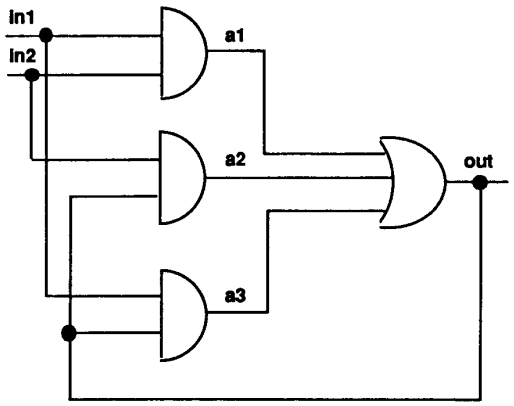

Figure 1

denotes an arbitrary point in time and the function `f` denotes the next point in time. For simplicity, in this example we assume that logical gates take 1 time unit to propagate a signal. Thus the formulae describing the structure are

1. `a1(f(x)) equ (in1(x) and in2(x))`
2. `a2(f(x)) equ (in1(x) and out(x))`
3. `a3(f(x)) equ (in2(x) and out(x))`
4. `out(f(x)) equ (a1(x) or a2(x) or a3(x))`
5. `not( all x ((in1(x) equ in2(x)) ⇒ (out(f(f(x))) equ in1(x)) and ((in1(x) xor in2(x)) ⇒ (out(f(f(x))) equ out(x)))))`

The proof proceeds as follows:
Input 1 produces the rule:
`[1] (in1(x) and in2(x)) → a1(f(x))`
Input 2 produces the rule:
`[2] (in1(x) and out(x)) → a2(f(x))`
Input 3 produces the rule:
`[3] (in2(x) and out(x)) → a3(f(x))`
Input 4 produces the rule:
`[4] (a1(x) and a2(x) and a3(x)) →`
`(a1(x) xor a2(x) xor a3(x) xor out(f(x)) xor (a1(x) and a2(x)) xor (a1(x) and a3(x)) xor (a2(x) and a3(x)))`
Input 5 reduced by Rules [1], [2], [3], produces the rule:
`[5] out(f(f(s1))) →`
`(true xor a1(f(s1)) xor a2(f(s1)) xor a3(f(s1)))`

Rule [3] superposed itself[1] , reduced by Rule [3], produces the rule:
`[6] (a3(f(x)) and in2(x)) → a3(f(x))`
Rule [3] superposed itself, reduced by Rule [3], produces the rule:
`[7] (a3(f(x)) and out(x)) → a3(f(x))`
Rule [1] superposed with Rule [3], produces the rule:
`[8] (a1(f(x)) and out(x)) → (a3(f(x)) and in1(x))`
Rule [2] superposed itself, reduced by Rule [2], produces the rule:
`[9] (a2(f(x)) and in1(x)) → a2(f(x))`
Rule [8] superposed with Rule [2], reduced by Rule [9], produces the rule:
`[10] (a2(f(x)) and a3(f(x))) →`
`(a1(f(x)) and a2(f(x)))`
Rule [7] superposed with Rule [2], produces the rule:
`[11] (a3(f(x)) and in1(x)) →`
`(a1(f(x)) and a2(f(x)))`
which simplifies [8] to:
`[8] (a1(f(x)) and out(x)) → (a1(f(x)) and a2(f(x)))`
Rule [6] superposed with Rule [1], reduced by Rule [11], produces the rule:
`[12] (a1(f(x)) and a3(f(x))) →`
`(a1(f(x)) and a2(f(x)))`
Rule [11] superposed itself, reduced by Rules [11], [4], [12], [10], produces the rule:
`[13] out(f(f(x))) →`
`(a1(f(x)) xor a2(f(x)) xor a3(f(x)))`
Rule [13] deleted Rule [5], produces the rule:
`[14] true → false`

This completes the proof, which was derived automatically using RRL; since the negation of the behavior in conjunction with the structure results in a contradiction, it follows that the behavior is correct (note that it may not completely describe the circuit's behavior; we only know that the stated behavior is forced by the circuit).

4 The Circuit Description Model

The VHDL description of the Sobel circuit included as primitives registers, multiplexors, and gates; we also modelled these elements as primitives in our description.

Since the behavior of the Sobel circuit is largely numerical, we needed to develop a model which allowed us to relate bit-strings and the integer values they represented. In addition, we needed to take advantage of the

[1] A rule can superpose itself using the idempotence property

recursive nature of such circuits as ripple-carry adders, etc.

In order to facilitate the recursive specification of circuits we chose to model bit strings as infinite arrays of bits, of which we can see an n-bit slice at any given time. Three critical functions are:

- *apply(bitstring,x)* which returns the *xth* bit of *bitstring*
- *intval(bitstring,x,y)* which returns the integer value of the segment of length $y + 1$ of *bitstring* starting at the *xth* position, and
- *intv(boolean)* which returns the integer associated with a truth value.

The *intval* function is defined recursively on the length value y. The necessary axioms are included in the specification.

In addition, a number of axioms and lemmas were introduced to handle such concepts as *integer relations, Boolean representation, two's complement,* and *absolute value.* Any lemmas were proved separately before they were used. These are discussed in more detail in [NaSt 88].

We demonstrate a typical input to the theorem prover with a very simple example, a circuit with n input lines which outputs *true* if and only if each of the inputs is *false*, i.e. the circuit tests whether of not the integer value of the input is zero. The input follows:

```
; axioms for integers
0 + x = x
x + 0 = x
x + succ(y) = succ(x + y)
succ(x) + y = succ(x + y)
twice(x) = x + x
not(eq(succ(x),x))
; axioms for boolean representation.
; e0 stands for 'false', e1 for 'true'.
not(eq(e0,e1))
bar(e0) = e1
bar(e1) = e0
; axioms defining the integer
; value of a bit string
intv(e0) = 0
intv(e1) = succ(0)
intval(x,y,0) = intv(apply(x,y))
intval(x,y,succ(z)) = intval(x,y,0) +
twice(intval(x,succ(y),z))
eq(apply(x,y),e0) xor eq(apply(x,y),e1)
; definition of the predicate bszero
; when a single bit is considered
bszero(x,y,0) equ eq(e0,apply(x,y))
; negation of theorem (basis)
not(all x all y (bszero(x,y,0) equ
(eq(intval(x,y,0),0))))
; recursive definition of bszero
bszero(x,y,succ(z)) equ (eq(e0,apply(x,y))
and bszero(x,succ(y),z))
; inductive hypothesis (used
; once the basis is verified)
bszero(x,y,k) equ (eq(intval(x,y,k),0))
; inductive step (negation)
not(all x all y (bszero(x,y,succ(k)) equ
(eq(intval(x,y,succ(k)),0))))
```

Thus the proof that the predicate *bszero* is true if and only if each of the n input bits are *false* proceeds by verifying the basis case, then, using an inductive hypothesis, verifying the property for an arbitrary $n > 0$. In general, both the predicates defining circuits and the properties of the circuits are much more complex; the proofs of these more complex circuits currently require a fair amount of user guidance.

5 The Sobel Chip

5.1 An Overview

The chip under consideration was designed at Research Triangle Institute (RTI) partly as a test case for the VHDL design methodology. It has approximately 10,000 transistors, and implements the Sobel edge detection algorithm [VaBK 87,Pr 78], which is commonly used in the image segmentation phase of military image processing systems. The chip can be broken down into two subsections (see Figure 2). The first is a windowing section which filters the image with four 3×3 windows, generating four values which measure the differences in intensity along the vertical, horizontal, left diagonal, and right diagonal directions. These values are passed to the second section of the chip, a magnitude and direction processor, which combines the window processor outputs to compute both gradient magnitude and gradient direction.

The window processors consist of a network of adders, subtractors, and registers; while their size was fixed at 12 bits, the verification was done inductively and (except for overflow conditions) the algorithm was shown to be correct for arbitrary widths. The verification was hierarchical, first confirming that the adders indeed added, etc. then showing that under the assumption that the components worked, the implementation of the algorithm was correct.

The magnitude and direction processor consists of adders, comparators, multiplexors, and registers. Basically, this phase consists of computing the absolute values of the four inputs, comparing these to find the maximum and its orthogonal value, then using these

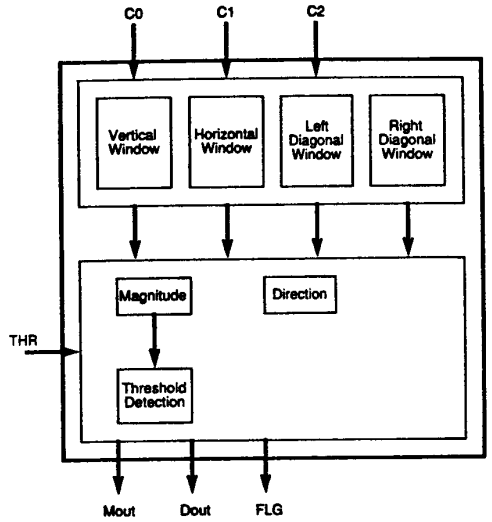

Figure 2

two values to compute the gradient direction and gradient magnitude.

The structural specification of the chip was presented to us in the form of VHDL code; although we feel that the extraction of first-order statements from VHDL can be largely automated, the translation was done by hand in this test case.

As mentioned before, we were not provided with a formal behavioral description of the chip, but were able to get the necessary information from the high-level statements provided in the document [VaBK 87] cited above.

5.2 Verification of the Circuit

The window processor was decomposed into its 4 windows, each of which were verified separately. The basic operations involved in the windows were addition, subtraction and doubling of bit-strings represented in two's complement form. The adders and subtractors were relatively straightforward to verify inductively since the ripple-carry paradigm was used. Some complexity was introduced, however, by the fact that in general the ripple-carrying technique does not work correctly for two's complement numbers (the sign bit is not guaranteed to be correct). The designers of the circuit compensated for this problem by making sure that the absolute values of the summands were always small enough relative to the size of the adders so that the adders worked correctly. Thus, we had to verify the following property:

> if the absolute values of two n-bit integers in two's-complement form are less than $2^{(n-1)}$, then adders (as well as subtractors) that use the ripple-carrying technique work correctly.

As a result of this, we had to keep track of the absolute values of numbers at each stage as they passed through the windows. A detailed account of this is presented in [NaSt 88].

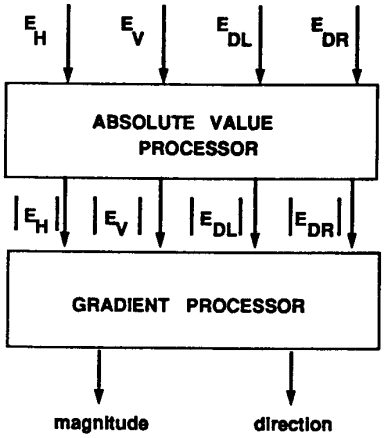

Figure 3

For our purposes it was convenient to further decompose the magnitude-and -direction processor into two components (see Figure 3), one which computes absolute values, (the "absolute value processor") and one which computes the gradient (the "gradient processor"). Verification of the absolute value processor was conceptually straightforward, since all that had to be done was to verify that the circuit computing the absolute value of a two's complement number worked correctly. The gradient processor receives four integers in the form of their absolute values and the corresponding sign bits and uses these to compute the magnitude and direction of the intensity gradient. We formalize these steps as follows. We take "angle" as a basic entity (or concept); the four basic angles that we deal with are horizontal, vertical, left-diagonal and right-diagonal, and these are represented by the constant symbols `ach1`, `acv1`, `adl1`, and `adr1`. The functions `aval`, `sign`, `ortho`, `code1`, and `code2` operate on angles, or, in other words, have angles as arguments, where

- **aval** stands for the absolute value of the 12-bit integer output by the window processor corresponding to the angle,
- **sign** stands for the sign (i.e. the 12th bit) of the 12-bit integer output by the window processor corresponding to the angle,
- **ortho** stands for the direction orthogonal (perpendicular) to the given direction, and
- **code1** and **code2** stand for the first and second bits in the 2-bit codes associated with the four angles.

Of these, we have given an extra argument for all functions except **ortho**, signifying that their values vary with time (though we really don't need these for **code1** and **code2**).

The function **maxdir** stands for "maximum direction" and returns the angle that has the higher absolute value associated with it in accordance with the following axioms:

```
(ge(aval(x,z),aval(y,z)) =>
              eq(maxdir(x,y,z),x))
(not(ge(aval(x,z),aval(y,z))) =>
              eq(maxdir(x,y,z),y))
```

(Note the asymmetry involved: if the absolute values are equal, then the first argument is returned.)

We can now prove that the absolute value of the maximum direction is the maximum of the two absolute values input, i.e.,

```
aval(maxdir(x,y,z),z) ==
    dmax(aval(x,z),aval(y,z)),
```

where the function **dmax** returns the maximum of two integers.

The circuit was broken down into several smaller components. A typical subcircuit is the comparator-and-mux combination that computes the maximum and minimum directions of two angles. The circuit consists of a comparator and two muxes that multiplex two 11-bit numbers in accordance with (but differently) the flag raised by the comparator. We specify these as follows:

```
compare(aval(x,z),aval(y,z),u)
mux(aval(y,z),aval(x,z),vmax,u)
mux(aval(x,z),aval(y,z),vmin,u)
```

Now, with the additional assumption that

```
y = ortho(x)
```

we can prove that

```
vmin = aval(ortho(maxdir(x,y,z)),z)
```

and

```
vmax = aval(maxdir(x,y,z),z).
```

6 Error Detection

In the process of verifying the design of the Sobel chip, several errors were detected. The two errors that we detected that the design team was unaware of involved the reversal of multiplexor input lines in the magnitude and direction processor. In addition, the proof process pointed out an error in a comparator circuit which had already been caught and remedied by the design team, but which had not been corrected in the version we were working with. While the multiplexor errors weren't semantically deep, neither were they obvious. In fact, they were not detected until a point was reached in the proof attempt where a simple, consistent statement contrary to what we knew to be true was derived. We conjecture that the most common design errors will be of this form (with the design being basically correct, but having some simple errors which can easily go undetected), especially when a hierarchical approach to design is used. (A case study of an error that we detected is given in [NaSt 88].)

7 Summary and Research Directions

In verifying the Sobel chip, we have developed a useful approach to hardware verification, especially in that we were able to detect several design errors before fabrication of the chip. The tools we used are at the stage of development where they can be used only by those conversant with theorem proving techniques, however, and a great deal of work needs to be done before they can be made accessible to a larger community. We feel that we can substantially reduce the amount of user interaction in many cases by using semantic information to help reduce the size of the search space. In addition, better proof-management tools need to be developed for helping a user keep track of what has been verified, what hypotheses were introduced, etc. We mentioned earlier that the proof process may not halt in all cases; since sequential circuits are finite-state machines we should be able to know when a description is incorrect in a finite (but not necessarily reasonable with any known method) amount of time. Heuristics for helping a user determine what might be wrong with an incorrect description need to be developed before the techniques described herein will have widespread applicability.

References

[Ba 84] Barrow, H., "VERIFY: A Program for Proving Correctness of Digital Hardware Designs," *Artificial Intelligence*, 24, pp.437-491, 1984.

[Buc 85] Buchberger, B., "Gröbner: An algorithmic method in polynomial ideal theory," in *Multidimensional Systems Theory* (N.K. Bose, ed.), D. Reidel, 1985.

[Bu 83] Bundy, A., *The Computer Modelling of Human Reasoning*, Academic Press, New York, 1983.

[ChPC 87] Chandrasekhar, M., Privitera, J., Conradt, K., "Application of term rewriting techniques to hardware design verification," in *Proceedings of the 24th Design Automation Conference*, Miami Beach, FL, 1987.

[ChLe 74] Chang, C., Lee, R., *"Symbolic Logic and Mechanical Theorem Proving,"* Academic Press, New York, 1974.

[Go 85] Gordon, M., "Why Higher-Order Logic is a good formalism for specifying and verifying hardware," Technical Report 77, Computer Laboratory, University of Cambridge, Cambridge, U.K., 1986.

[HsDe 83] Hsiang, J., and Dershowitz, N., "Rewrite methods for clausal and non-clausal theorem proving," in *Proceedings of the 10th EATCS Intl. Colloq. on Automata, Languages, and Programming*, Barcelona, Spain, 1983.

[Hu 85] Hunt, W., "FM8501: A Verified Microprocessor," Technical Report 47, Institute for Computing Science, Univ. of Texas at Austin, Austin TX, 1985.

[HuOp 80] Huet, G., and Oppen, D., "Equations and rewrite rules: a survey," in *Formal Languages: Perspectives and Open Problems* (R. Book, ed.), Academic Press, New York, 1980.

[KaNa 85] Kapur, D., and Narendran, P., "An equational approach to theorem proving in first-order predicate calculus," in *Proceedings of the 9th Intl. Joint Conference on Artificial Intelligence*, Los Angeles, California, 1985.

[KaSZ 86] Kapur, D., Sivakumar, G., and Zhang, H., "RRL: a rewrite rule laboratory," in *Proceedings of the 8th Conference on Automated Deduction*, Oxford, U.K., 1986.

[KnBe 70] Knuth, D., and Bendix, P., "Simple word problems in universal algebras," in *Computational Problems in Abstract Algebra* (J. Leech, ed.), Pergamon Press, Oxford, 1970, pp. 263-297.

[MiYa 86] Mishra, B., and Yap, C., "Notes on Gröbner Bases," Technical Report # 257, New York University Courant Institute of Mathematical Sciences, New York, N.Y., 1986.

[NaSt 87] Narendran, P., and Stillman, J., "Hardware verification in the Interactive VHDL Workstation," in *VLSI Specification, Verification, and Synthesis*, (G. Birtwistle, P.A. Subrahmanyam, eds.), Kluwer Academic Publishers, Boston, 1988, pp. 235-255.

[NaSt 88] Narendran, P., and Stillman, J., "Formal Verification of the Sobel Image Processing Chip," unpublished report, available from the authors upon request.

[Pr 78] Pratt, W., *"Digital Image Processing,"* John Wiley & Sons, Inc. 1978.

[VaBK 87] Vasanthavada, N., Baker, R., Kanopoulos, N., "A monolithic image edge detection filter," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1987.