Equivalence Checking of Datapaths Based on Canonical Arithmetic Expressions

Zheng Zhou Wayne Burleson
Department of Electronic & Computer Engineering
University of Massachusetts at Amherst, MA 01003
{zhou,burleson}@ecs.umass.edu

Abstract— **Numerous formal verification systems have been proposed and developed for Finite Sate Machine based control units (notably SMV[19] as well as others). However, most research on the equivalence checking of datapaths is still confined to the bit-level. Formal verification of arithmetic expressions and synthesized datapaths, especially considering finite word-length computation, has not been addressed. Thus formal verification techniques have been prohibited from more extensive applications in numerical and Digital Signal Processing.**

In this paper a formal system, called Conditional Term Rewriting on Attribute Syntax Trees (ConTRAST) is developed and demonstrated for verifying the equivalence between two differently synthesized datapaths. This result arises from a sophisticated integration of *attribute grammars*, which provide expressive data structures for syntactic and semantic information about designed datapaths, and *term rewriting systems*, which transform functionally equivalent datapaths into the same canonical form. The equivalence relation is defined as a congruence closure in the rewriting system, which can be generated from arbitrary axioms, such as associativity, commutativity, etc. in a certain algebraic system. Furthermore, the effect of finite word-lengths and their associated arithmetic precision are also considered in the definition of equivalence classes. As a particular application of ConTRAST, a formal verification system is designed to check equivalence under precision constraints. The results of initial DSP synthesis experiments are displayed, where two differently implemented IIR filters in direct II and cascaded architectures are automatically compared under given precision constraints.

Keywords— **Design Verification, High-Level Synthesis and System-Level Design Aids**

I. Introduction

The architectural model at register transfer level is represented in terms of control units (or FSMs) and datapaths, which consist of a set of components, such as ALUs, multiplexors, latches and shifters. The datapath is specified by a block of arithmetic expressions in the behavioral domain. Because of the close relationship between language constructs and synthesis algorithms, semantically equivalent descriptions that differ syntactically may result in different designs. So the behaviors of a datapath can be optimized by applying a set of transformation rules on its corresponding arithmetic expressions [12] [20]. An arithmetic expression can be represented in many different forms. For example, even just applying a *single commutativity axiom* of the form $x+y = y+x$ on the expression $a_1+a_2+\cdots+a_{n-1}$, we have 2^{n-1} different forms.

The wide variety of descriptions of a datapath demand that EDA tool designers provide a canonical intermediate representation for preserving the original behavior of the input HDL specification, while allowing the addition of synthesis results through various refinements, bindings, optimizations and mappings. Equivalence-checking is an essential problem in the development of such a canonical representation, and conversely, canonical forms are directly applied to verify equivalence.

In computational theory, a fundamental problem is to decide if two expressions are equivalent, and its computability varies with its actual definition [11]. Its related problems have been widely discussed, such as *Simple Word Problem* [17], *Common Subexpression Problems* [10], and *Confluence of Rewrite Systems* [9].

In digital computation, equivalence between two datapaths is not completely identical to the equivalence between two arithmetic expressions, since real numbers may not be exactly expressed in a limited wordlength such that basic algebraic properties do not exist. In formal hardware verification, however, the equivalence-checking problem has not been solved, even neglecting the issue of finite wordlength in datapaths. Related previous works are reviewed below.

1. **at the lower level**: equivalence is decided by enumerating all possible values in a canonical representation, which is derived from some particular expansion, such as Shannon expansion-based Binary Decision Diagrams (BDDs) [4], Multi Terminal BDDs (MTBDDs) [8] and Algebraic Decision Diagrams (ADDs) [3], Reed-Muller expansion-based Function Decision Diagrams (FDDs) [15], monomial expansion-based Binary Moment Diagrams (BMDs) [5], and linear expression-based Edge-valued BDDs (EVBDDs) [18]. Except BMDs, the space complexity of these approaches is exponential even for representing simple arithmetic expressions, such as $a_1 \times b_1$. Moreover, the effect of finite word-length on numerical quantization and arithmetic

32nd ACM/IEEE Design Automation Conference ®

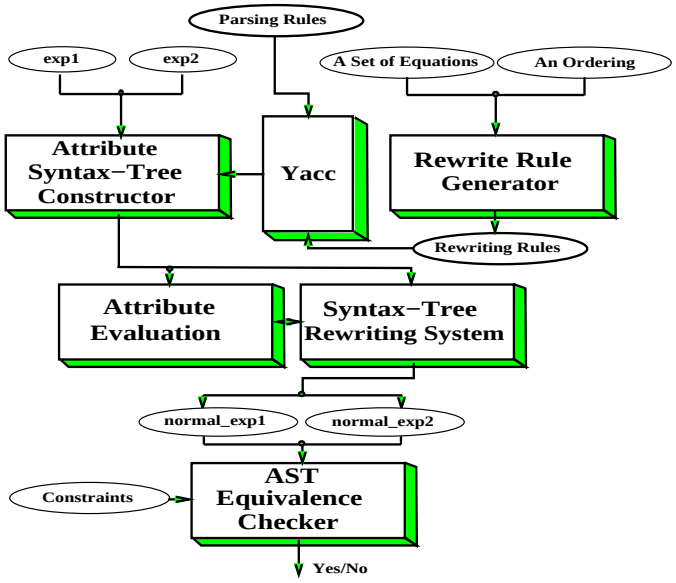

Fig. 1. Overview of ConTRAST

operations is ignored.

2. **at the higher level**: an architectural implementation is described in the form of Control-Dataflow graphs (CDFGs)[12]. The syntactic variances on arithmetic expressions can be minimized by using the most parallel form, such as a pure dataflow graph. Chaiyakul's Assignment Decision Diagrams were based on this idea [7].

To find an efficient method for representing and manipulating arithmetic expressions, we move out of the scope of these enumeration-based or graph isomorphism-based approaches to explore a symbolic representation based on attribute grammars [16] and term rewriting systems [9]. Thus a new formal approach, called *Conditional Term Rewriting on Attribute Syntax Trees* (**ConTRAST**), has been developed[22]. Its architecture is shown in Fig. 1. Its basic idea and main feature will be discussed in this paper.

In ConTRAST, two powerful computation approaches: attribute grammars and term rewriting systems, are integrated in a sophisticated way:

1. an *attribute grammar* is used to represent arithmetic expressions, the attributes can be used as a repository for storing semantic information during compilation and design information of corresponding datapaths in synthesis.
2. a *term rewriting system* is designed to translate semantically equivalent expressions that differ syntactically into the same canonical form. The equivalence relation is defined as a congruence closure in the rewriting system, which can be generated from the application-related axioms and partial orderings.

To easily embed ConTRAST with other presently used design and verification tools, specific symbolic processing facilities, such as LISP or unification functions, etc. are NOT used in its term rewriting system. Instead, the only facilities used are common UNIX utilities: *Lex* and *Yacc*.

1. Make a *Yacc* specification file, which defines a Context Free Grammar (CFG) of arithmetic expressions, and specifies how the attributes are calculated;
2. Use the LALR parser generated from *Yacc* to fulfill the unification and substitution tasks, thus determining which rewrite rules should be called.

The paper is organized as follows. In section 2 we define the equivalence on arithmetic expressions, exhibit different definitions and the computability of their corresponding decision problems, and decide a proper equivalence relation for specifying equivalent datapaths in real designs. In section 3 attributed syntax trees are introduced to describe expressions or datapaths and their particular attributes and attribute evaluation rules are discussed. Then section 4 shows how to construct a conditional-term rewriting system on attributed syntax trees, and provides essential theorems that ConTRAST is terminating and confluent, which ensure that any rewriting will terminate in a canonical form. In section 5, we demonstrate the power of this formal system on a simple but novel example – automatically checking the functional equivalence of two differently implemented IIR filters under given precision constraints based on canonical arithmetical expressions. Finally, a simple summary about this representation is presented, and possible integration with SMV is suggested.

II. Equivalence Relations and Their Computability

Similar to word problems[17] in abstract algebra, the general problem for checking equivalence between arithmetic expressions is unsolvable. So a proper definition should be determined first based on its computability.

If defined as *semantic equivalence*, i.e., two expressions have equivalent mathematical meaning, the complexity of equivalence checking is very sensitive to the arithmetic system to be used[14]. For example, $(N, +, \times)$ is undecidable, but $(R, +, \times)$ is decidable. So automatically deciding semantic equivalence between two arithmetic expressions is not feasible in practice.

For *syntactic equivalence*, i.e., two expressions can be matched as two strings, the complexity is $\mathcal{O}(n^2)$[1], but such equivalence is not powerful enough to cover the design space generated by synthesis transformations.

If the equivalence relation is defined as a *congruence closure on a set of equations*, i.e., an equality $t_1 = t_2$ logically follows from a set of equations $\mathcal{E} = \{s_{11} = s_{12}, s_{21} = s_{22}, ..., s_{k1} = s_{kk}\}$, its complexity is very sensitive to the definition of operators [6]. For example, associative $\times$ is undecidable, but commutative $\times$ is EXPSPACE-complete.

By adding a (strict) partial order on the set of constants and operation symbols in equations, we have a *congruence*

closure in a rewriting system, i.e., an equality $t_1 = t_2$ logically follows a set of rules: $\mathcal{R} = \{s_1 \rightarrow s_2, ..., s_l \rightarrow s_m\}$. Since the reduction order $\succ$ provides strong guidance to the deduction mechanism and drastically limits the search space of equivalent consequences that need to be computed, many unsolvable problems above become decidable. Moreover, term rewriting systems have been developed and used in many AI systems. So we conclude that the equivalence between two arithmetic expressions should be defined and checked based on this definition.

A. Attribute syntax trees(ASTs) for arithmetic expressions

To represent possibly very large size equivalence classes of arithmetic expressions, and to store complicated design information on each construct of expressions, we select attribute grammars as data structures for arithmetic expressions such that the attributes can be used for storing semantic information during compilation and design information of corresponding datapaths in synthesis. For example, based on an *unambiguous* CFG in [2], an arithmetic expression can be partially represented as a syntax tree, and precision information can be attached to each node in the syntax tree, thus resulting in an AST. As an example, $-(a*b*c)+b*c$ and its ASTs are given in Fig. 2.

An attribute can represent anything the user wants, and can be passed up and down to derive other properties. In ConTRAST, all attributes are simply classified based on their possible usages:

1. *Basic attributes* are the information needed to distinguish expressions in a symbolic representation such that algebraic equivalence can be determined. For example, v.`Id` is a unique identifier for a node v; v.`Signature` denotes the equivalence classes of v, and v.`Sym` represents the token appearing at this node, etc.
2. *Auxiliary attributes* are application-related information such as precision, latency, and cost, etc. These are used for high-level synthesis and verification. For example, to verify arithmetic transformations in digital filter design under given precision constraints, the attributes: v.`WLength`, v.`FLength` and v.`Err`, are attached to each node to represent the wordlength, fractional length and roundoff errors.

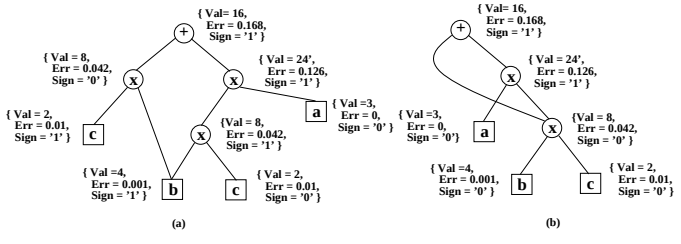

Fig. 2. Attribute syntax trees(ASTs) of $-(a*b*c)+b*c$, where `Err` is an attribute for storing accumulated roundoff error. (a). an uncanonical form: $((-c)*(-b))+((-b)*c)*a$, (b). the canonical form under the lexicographic path ordering: $((b*c)+(-(a*(b*c))))$

The evaluation procedures of attributes or values on each node are specified in the form of subroutines, which depend on which attribute is manipulated, and what direction is followed. For example, to determine the equivalence class of an expression at node v, at first we need to invoke a rewriting system to transform it into the normal form, then use the following procedure (TABLE I) on the basic attribute v.`signature`.

TABLE I
PROCEDURE CLASSIFYING

Step 1	*if* ((v.`Lo` == NIL) && (v.`Hi` == NIL)) v.`signature` = $h_1(v.\texttt{sym})$; *else* v.`signature` = $h_2(v.\texttt{sym}, s_1, s_2)$; /* calculating the signature of v from v.`sym` and v's subexpressions */
Step 2	*if* (v.`signature` == NIL) *create_new_class*(v); *else find_old_class*(v);
Step 3	Return the signature of v.

To analyze the worst-case effect of finite wordlength, we have a procedure (in TABLE II) to calculate v.`Val` and v.`Err`.

TABLE II
PROCEDURE ROUNDING

Step 1	*if* (v.`Val` < 2) $L = v$.`WLength`; *else* $L = v$.`FLength`;
Step 2	$Temp = v.\texttt{Val} \times 2^{L-1}$;
Step 3	$ITemp = floor(Temp)$; /* the largest integer not greater than v.`Val` */
Step 4	$RND = \|Temp - ITemp\|$;
Step 5	*if* ($RND >= 0.5$) $ITemp = ITemp + 1$;
Step 6	$v.\texttt{Val} = (ITemp/2^{L-1})$; $v.\texttt{Error} = MAX\{v.\texttt{Error}, RND\}$;

III. TERM REWRITING ON ASTS

To achieve a canonical representation of ASTs, we use a novel conditional term rewriting technique with LR parsing algorithms: *if a string of grammar symbols, e.g. A_i+B_i on a stack can be reduced to an equivalence class represented by A_{i+1}, and attributes on A_i, B_i and $+$ satisfy the condition of a rewrite rule, then $A_i + B_i$ is replaced with the normal form* [22].

In general, a rewrite system $(E, \succ)$ can be generated from a set of equations E and an ordering function $\succ$ given by users. In the current implementation of ConTRAST, the equations are specified as axioms in an algebraic field: *Commutative law*, *Associative law*, *Distributive law*, *Identity law*, and *Inverse law*. The ordering function is *lexicographic path ordering (lpo)*[9], which is based on well-founded orderings of vocabularies.

To ensure that any rewriting will terminate in a normal (or canonical) form, the rewriting system must have two essential properties: *termination*, i.e., there are no infinite derivations; and *confluence*, i.e., any two equivalent terms can be reduced to the same term. The problem of deciding confluence is decidable for a finite and terminating rewrite system. A well-known sufficient condition is based on the patterns of rewrite rules: *left-linear*, i.e., no variable occurs more than once on any LHS of rules; and *non-overlapping*, i.e., no substitution can make two rules to have a unified LHS. Based on properties of *lpo* and our defined unambiguous CFG, we have the following results, which have been proved in [22]:

Theorem 1: ConTRAST is terminating on the set of ASTs.

and

Theorem 2: CoNTRAST is left-linear and nonoverlapping, so it is confluent.

Thus all equivalent arithmetic expressions defined by axioms can be transformed into a *canonical* expression, so the rewriting system in ConTRAST provides a solid base for developing formal verification systems. Compared with other canonical representations such as BDDs,ADDs,etc., ConTRAST uses an essentially different methodology, i.e., applying term rewriting on attribute grammars, instead of implicitly enumerating all values with a graph. Such differences are summarized in TABLE III. A more detailed discussion is given in[23].

The expressive power can be reflected by the space complexity for word-level operations such as x, $x+y$, $x \times y$, x^2 and c^x. Since ConTRAST is based on ASTs other than decision diagrams, and the results of those operations are defined by evaluation procedures on attributes, instead of implicitly enumerating all values with a graph, its space complexity is *linear*. But the size of attributes on a node is definitely larger than other approaches.

The time complexity for ensuring canonical forms is primarily determined by the complexity of *Apply* algorithms, which were provided for operations: $+$ and $\times$ on BDDs, EVBDDs, *BMDs, and ConTRAST. The time complexity of ConTRAST is dependent on the ordering on a set of terms. For the present *lpo*, it is $\mathcal{O}((|G_1| \cdot |G_2|)^3)$, where $|G|$ denotes the size of an AST. Unlike decision diagrams in BDDs, EVBDDs, and *BMDs, the size of an AST is not larger than the size of the given expressions.

IV. Example: equivalence-checking on two filter designs

In [21], a fourth-order digital filter $H(z)$ is defined as follows.

$$H(z) = \frac{0.001836(1+z^{-1})^4}{(1-1.499z^{-1}+0.8482z^{-2})(1-1.5548z^{-1}+0.6493z^{-2})}$$

Using algebraic transformations, numerous implementations are possible, however,they differ in terms of numerical properties due to finite wordlength effects. Two implementations are constructed: (1) the *direct II architecture* (Fig. 3(a)) and the *cascaded architecture* (Fig. 3(b)). The equivalence between these two implementations can be checked under different precision constraints. In Fig. 4 even if all variables and coefficients are represented using the IEEE double-precision standard, two contradictory results arise when the precision constraints are 0.001 and 0.00001.

This example looks somewhat simplistic, but it shows the potential applications of ConTRAST on higher level DSP descriptions and formal verification of finite wordlength effects. As more sophisticated methods of precision analysis and wordlength specifications are introduced, more interior properties of different synthesis strategies will be verified through ConTRAST, thus more complicated examples can be displayed, such as in [24].

V. Conclusion and Future Work

In this paper we introduced a formal verification approach, ConTRAST, for checking functional equivalence of differently designed datapaths based on a canonical representation for arithmetic expressions. The internal data structures are attributed syntax trees(ASTs), which provide expressive data structures for syntactic and semantic information about designed datapaths. The canonical representations and attribute evaluations are automatically performed in a particularly designed membership conditional term rewriting system, which has been built from a LR parser with *Lex* and *Yacc*, without using particular symbolic processing facilities such as LISP, or unification functions.

Based on ConTRAST, a DSP design verification tool called Fixed-Point Verifier (**FPV**) has been developed[24]. Similar to present DSP hardware design tools, FPV allows users to describe filters in the form of arithmetic expressions and to specify arbitrary fixed-point wordlengths on various signals. However, unlike simulation-based verification methods like Cadence/Alta's Fixed Point Optimizer and Mentor's DSPstation, FPV can automatically perform correctness-checking and equivalence-checking for a given filter design under the effect of finite word length.

The power of ConTRAST is based on the following features:

1. The canonical representations for equivalent arithmetic expressions are defined in terms of congruence closure in a rewriting system, thus providing an efficient internal representation for huge equivalence classes, and making formal verification on numerical and DSP architectures feasible.
2. The equivalence on arithmetic expressions can be redefined according to application conditions and constraints, thus allowing formal verification to be applied for practical engineering problems, which arise in synthesis, such as equivalence under precision constraints.
3. The rewriting ordering in ConTRAST can be changed

TABLE III
FEATURES OF BDDs, FDDs, ADDs, EVBDDs, BMDs AND CONTRAST

	BDD	*FDD*	*ADD*	*EVBDD*	**BMD*	*ConTRAST*
Terminals	0, 1	0, 1	distinguished elements	0	integers, or reals	*id*, *num*
Non-terminals	$f_{\overline{x_i}}$, f_{x_i}	f_{x_i}, $\frac{\partial f}{\partial x_i}$	$f_{\overline{x_i}}$, f_{x_i}	subexps in linear exps	$f_{\overline{x_i}}$, f_{x_i}	subexps in any exps
Weights	0, 1	0, 1	0, 1	integers	integers, or reals	arbitrary attributes
Function f Described†	$B^n \rightarrow B$	$B^n \rightarrow B$	$B^n \rightarrow D$	$B^n \rightarrow W$	$B^n \rightarrow W$	$D^n \rightarrow D$
Decomposition	Shannon expansion	Reed-Muller expansion	Shannon expansion	linear expression	monomial expansion	CFG
Apply operations	Yes	No	No	Yes	Yes	Yes
Application Levels	bit-level	bit-level	semi-bit-level	bit-level& word-level	bit-level& word-level	high-level

† B denotes a Boolean space. W denotes a word space, a subspace of integers or real numbers that could be represented in a word of size n. D denotes an abstract space.

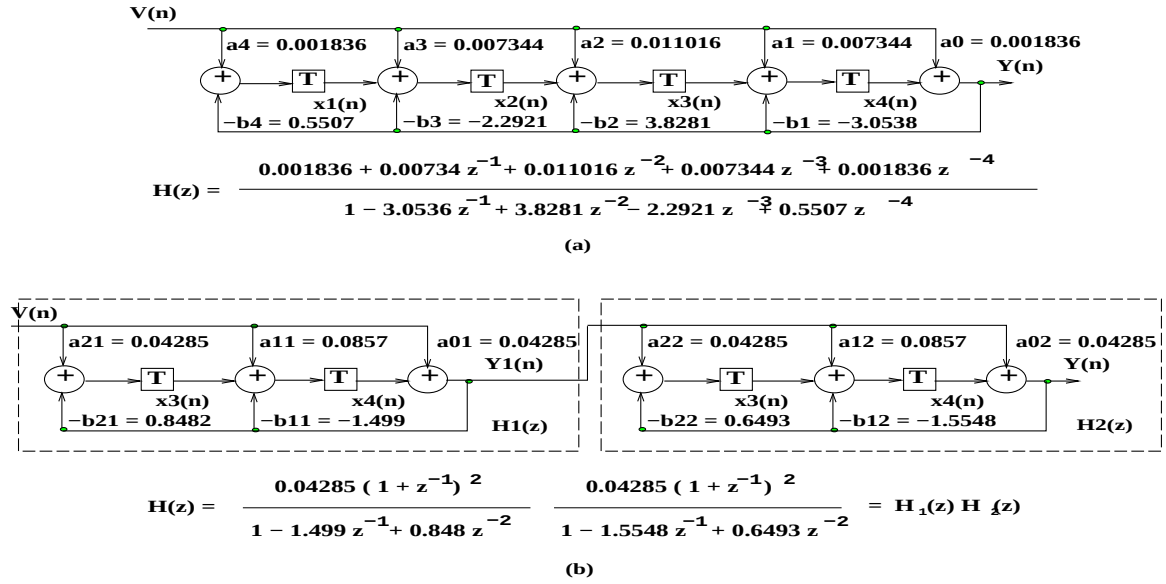

Fig. 3. Fourth-order digital filter: (a) direct II architecture, (b) cascaded architecture

from the present lexicographic path ordering (*lpo*) into other design information-related orderings, thus the rewriting procedure here can possibly be converted into an optimization procedure, in which the normal form becomes the optimal form.

This research explored a novel combination of two powerful computation techniques: attribute grammars in compiler design and term rewriting systems in theorem proving. Moreover, this paper enlightened a new methodology in formal hardware verification, and created an extensive application space. To achieve widespread use, proper combinations of ConTRAST with other formal methods, such as ADDs[3], SMV[19] and SFG-tracing[13], require more comprehensive theoretical research. Some further extensions on ConTRAST being pursued at UMASS include:

1. **Integration of ADDs and ConTRAST**: Since ADDs can be regarded as a hierarchical representation of $if-then-else$ conditions, the symbolic control table of a control unit can be efficiently and canonically expressed in the form of ADDs. By representing

Fig. 4. Experimental results using ConTRAST on two digital filter implementations with different precision constraints

control units in ADDs, and representing datapaths in ConTRAST, a canonical representation is possible for higher level descriptions.

2. **Integration of SMV and ConTRAST**: It is well known that SMV is very efficient for verifying temporal properties at the bit level. Since ConTRAST can solve equivalence-checking at the word level, a suitable integration of these two approaches will allow the user to verify a complicated module at the word level, i.e., using SMV to verify control units, and ConTRAST to verify datapaths. A timely example would be the Pentium FPU.

In addition, possible extensions of ConTRAST in high level synthesis will be explored by replacing the present *lpo* with cost-minimized scheduling, power-minimized objective functions, or fault-secure scheduling.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley,Reading,Mass., 1974.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. Technical report, University of Colorado, 1993.

[4] R. E. Bryant. Graph-based algorithms for boolean functions. *IEEE Trans. on Computers*, C-35(8):677–691, 1986.

[5] R. E. Bryant and Y.A. Chen. Verification of arithmetic functions with binary moment diagrams. Technical Report CMU-CS-94-160, Carnegie Mellon University, 1994.

[6] E. Cardoza, R. Lipton, and A.R. Meyer. Exponential space complete problems for petri nets and communtative semigroups. In *ACM Symp. on Theory of Computing*, 1976.

[7] V. Chaiyakul, D. Gajski, and L. Ramachandran. High-level transformations for minimizing syntactic variances. In *30th Design Automation Conference*, pages 413–418, 1993.

[8] E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *30th ACM/IEEE Design Automation Conference*, pages 54–60, 1993.

[9] N. Dershowitz. Rewrite systems. In *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier Science Publishers B.V., 1990.

[10] P.J. Downey, R. Sethi, and R.E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.

[11] J. Ferrante and C.W. Rackoff. *The Computational Complexity of Logical Theories*, volume 718 of *Lecture Notes in Mathematics*. Springer-Verlag, 1979.

[12] D.D. Gajski, N.D. Dutt, A.C. Wu, and S.Y. Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[13] M. Genoe, L. Claesen, E. Verlind, F. Proesmans, and H. De Man. Illustration of the sfg-tracing multi-level behavioral verification methodolgy, by the correctness proof of a high to low level synthesis application in CATHEDRAL-II. In *Proc. ICCD-91*, pages 338–341, 1991.

[14] N. Immerman. Decision problems for first-order logical languages. Personal Mail, 1993.

[15] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *European Design Automation Conference*, pages 43–47, 1992.

[16] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–163, 1968.

[17] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In ed. J. Leech, editor, *Computational Problems in Abstract Algebras*, pages 263–297. Pergamon Press, Oxford, England, 1970.

[18] Y.T. Lai and S. Sastry. Edge-valued binary decision diagrams for hierarchical verification. In *29th ACM/IEEE Design Automation Conference*, pages 608–613, 1992.

[19] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[20] M. Potkonjak and J. Rabaey. Optimizing resource utilization using transformations. *IEEE Trans. on CAD*, 13(3):277–292, 1994.

[21] F.J. Taylor. *Digital Filter Design Handbook.* Marcel Dekker,Inc., 1983.

[22] Z. Zhou and W. Burleson. A canonical representation of algebraic expressions in high-level synthesis. Technical Report TR-94-CSE-9, ECE Dept., U. of Mass. at Amherst, 1994.

[23] Z. Zhou and W. Burleson. Selecting canonical representations for formal verification of arithmetic computations. Technical Report TR-94-CSE-10, ECE Dept., U. of Mass. at Amherst, 1994.

[24] Z. Zhou and W. Burleson. Formal verification of arithmetic expressions with applications in dsp synthesis. Submitted to Conference on Computer-Aided Verification (CAV'95), 1995.