Polynomial Circuit Models for Component Matching in High-Level Synthesis

James Smith and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—Design reuse requires engineers to determine whether or not an existing block implements desired functionality. If a common high-level circuit model is used to represent components that are described at multiple levels of abstraction, comparisons between circuit specifications and a library of potential implementations can be performed accurately and quickly. A mechanism is presented for compactly specifying circuit functionality as polynomials at the word level. Polynomials can be used to represent circuits that are described at the bit level or arithmetically. Furthermore, in representing components as polynomials, differences in precision between potential implementations can be detected and quantified. We present a mechanism for constructing polynomial models for combinational and sequential circuits. Furthermore, we derive a means of approximating the functionality of nonpolynomial functions and determining a bound on the error of this approximation. These methods have been implemented in the POLYSYS synthesis tool and used to synthesize a JPEG encode block and infinite impulse response filter from a library of complex elements.**

***Index Terms*—Binary decision diagrams (BDDs), high-level synthesis, polynomials.**

I. Introduction

THE increased complexity of integrated circuits has forced designers to reuse existing circuitry when constructing new systems. The proliferation of reusable blocks has promised opportunities to complete new designs more quickly and with fewer errors. Reuse of existing components requires those components to have suitable characteristics, including area, power consumption, performance, and testing features. However, it is most important that the component implement the functionality required by the system. Searching the space of existing implementations for functional validity is time consuming and fraught with pitfalls, as the suitability of existing blocks is determined by manual methods or verbal descriptions. This search promises to become more complex as the number and need for reusable designs increases [1]. The models and methods presented in this paper enable automation of this search by generating circuit representations that are at a higher level of abstraction than those used in traditional library binding applications.

Component matching is the problem of allocating complex blocks given a system specification. This problem reduces to determining whether or not the functionality of a library element is the same as the functionality of part of a specification. For example, in designing the baseline JPEG encode block of Fig. 1, subblocks are required to perform a discrete cosine transform (DCT), quantization, dc (zero frequency) encoding, and ac (nonzero frequency) encoding. Given a library of existing blocks, a word level representation can be derived from the Boolean equations that describe the functionality of library elements. The Boolean equations that specify an existing block can be derived in a straightforward manner from commonly available component netlists. The JPEG system can then be synthesized by matching the arithmetic specification of each of these functions to the word-level representation of each library element.

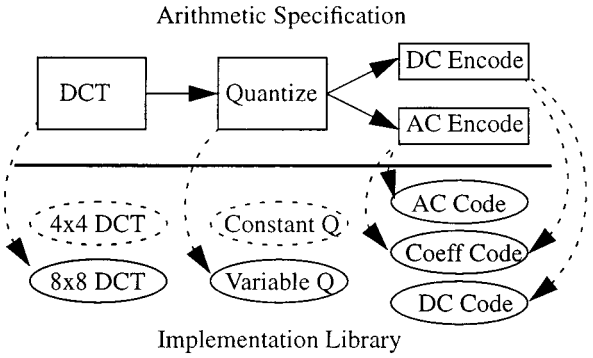


Fig. 1. Mapping JPEG encode onto existing designs.

Component matching is closely related to verifying that a specification and an implementation match exactly, but presents important differences. In matching a component to a specification, it is valuable to detect components that implement functionality that is similar to, but not necessarily the same as, that of the specification. For example, in performing DCT operations, a specification may require computation of the $\cos(x)$. One possible implementation may be a function that does not implement $\cos(x)$ exactly, but instead implements an approximation of the function to preserve area and increase performance. Furthermore, a specification may indicate that computation of $\cos(x)$ can take up to three cycles; however, existing implementations may exist that require only two cycles. Thus, the specification and implementation are similar, but do not match exactly, allowing for tradeoffs in execution time, area, power consumption, precision, and other qualities.

The examples discussed above can be specified very efficiently with polynomial models. For example, $\cos(x)$ can be approximated by

$$1 - x^2/2! + x^4/4! - x^6/6! + \cdots + x^n/n!\,.$$

This article presents methods for developing analogous word-level polynomial models for existing implementations given a

Manuscript received June 1, 1999; revised December 2, 2000.

The authors are with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA (e-mail: jsmith@mdv.com).

Publisher Item Identifier S 1063-8210(01)07657-0.

bit-level description of the implementation. These methods are ideally suited for circuits that implement arithmetic functions and can be applied to combinational and sequential circuits.

This comparison often must be performed between arithmetic and bit-level abstractions of the functionality. Polynomial methods provide a means for generating word-level polynomial representations, given bit-level descriptions of an implementation. In generating a mathematical structure common to both levels of abstraction, allocation of complex components can be performed, closing the semantic gap between specifications such as those generated in MATLAB and implementations, such as those modeled with Boolean logic or hardware design languages (HDLs). This technique is used by the POLYSYS synthesis tool to map arithmetic specifications onto existing designs that are described by Boolean equations.

The techniques presented here are most effective for allocating blocks that are arithmetic intensive, but may contain significant control logic. Common application domains that fit this description include computer graphics and digital signal processing. To illustrate the application of the polynomial methods developed in this article, we map a JPEG encode specification to complex elements and compare the specification of a filter suitable for controlling the velocity of a tape through a tape drive to an existing filter. The arithmetic specification for the JPEG encode block and the IIR filter are derived from MATLAB, while the existing implementation are described by Boolean equations.

II. RELATED WORK

Reusable blocks have traditionally been characterized by verbal or object-oriented descriptions [2], [3] such as "ethernet core" or "rasterizer," combined with component-specific attributes, such as "floating point" or "integer," and waveforms. Precise descriptions of functionality are usually restricted to smaller blocks such as combinational logic gates or simple arithmetic operations (e.g., addition or multiplication). For example, in allocating a JPEG block, current techniques may require that the specification and implementation both be described by the keyword "JPEG." This description is imprecise, however, as potential JPEG implementations may implement different compression schemes, different levels of accuracy, or operations on data sets of different sizes.

Component matching has historically been restricted to matching bit-level circuit specifications to logic gates. Many structures, such as binary decision diagrams (BDDs) [4], are ideal for mapping combinational logic onto a library of gates. The canonicity and ease of composition that BDDs provide make them ideal for matching small combinational circuits. However, for more complex functions, like multiplication, the potentially exponential size of BDD structures makes comparison of BDDs time consuming and memory intensive. When comparisons are sought between functions that are not described at the bit level, BDD structures are not sufficient to represent circuit functionality. Furthermore, BDDs can yield information on whether or not a specification and implementation match exactly, but offer no path for quantifying the degree to which the two differ. That is, two functions that have similar, but not equal, BDD structures may implement drastically different arithmetic functions, while two very different BDDs may implement the same mathematical operation with different degrees of precision.

Binary moment diagrams (BMDs) [5] have been developed to ease the memory and time required to manipulate complex structures by generating word level representations. BMDs have been used to verify the functionality of linear circuits [6] and could be adapted to perform component matching for those circuits. However, BMDs are unsuitable for use in non linear functions because of the resulting exponential complexity. hybrid decision diagrams [7] and multiterminal BDDs [8] suffer from similar restrictions. power hybrid decision diagrams (PHDDs), developed in [9] are well suited to handling the non linearities associated with floating point arithmetic, but can still require, in the worst case, an exponentially large data structure to represent nonlinear functions.

In order to raise the complexity of blocks for which a functional characterization can be generated, algorithms have been developed to reduce the size of circuit representations. This can be achieved by generating data structures that represent an approximation of circuit functionality. For example, in [10], a compact circuit approximation is derived that minimizes the number of input assignments for which the approximation and the actual circuit differ. In contrast, our work generates a compact circuit approximation that minimizes the numerical distance between the functionality of the representation and the actual block. Similarly, the allocation mechanism presented in this paper determines the accuracy of a match by the numerical distance between a specification and a possible implementation.

Minato introduced a method for modeling and manipulating circuits that implement polynomial functions using zero-suppressed BDDs [11]. This structure provides an efficient representation for those circuits for which a polynomial description is specified, but becomes exponentially large if discontinuities exist in the function. The methods we will present here develop a mechanism for deriving the polynomial representation given a Boolean circuit description. In addition, we will present a mechanism for detecting circuit discontinuities and generating compact approximations for highly discontinuous circuits.

Efficient component matching requires data structures that are canonical, constructible in polynomial time, and allow for simple composition. This paper will demonstrate methods for determining polynomial representations for circuits that are described at the bit level. Furthermore, we will prove that a unique minimum-order polynomial representation exists for all circuitry without feedback. In representing hardware as polynomials, blocks can be efficiently compared with one another to determine if they implement the same functionality. In addition, polynomials are easily composable, allowing efficient determination of the functionality of hierarchical or partitioned blocks.

III. POLYNOMIAL REPRESENTATIONS

To map an arithmetic specification to a complex element that is described at the bit level by Boolean logic, a word-level polynomial that encapsulates the element's functionality is derived.

We consider only completely-specified Boolean functions for the sake of simplicity. Generating a word-level polynomial representation for a Boolean function may appear to be an inconsistent problem because Boolean functions are inherently discontinuous. However, a Boolean function $\mathbf{y} = \mathbf{F}(\mathbf{x})$: $B^m \rightarrow B^k$, $B = \{0, 1\}$ are bit vectors of length m and k, respectively, can be treated as a set of coordinates (x, y), where $x, y \in Z$

$$x = \text{Encode}(\mathbf{x}), \quad \mathbf{x} = \text{Decode}(x),$$
$$y = \text{Encode}(\mathbf{y}), \quad \mathbf{y} = \text{Decode}(y).$$

Thus, "encode" is an integer interpretation of a Boolean vector, such as two's complement or sign magnitude and "decode" is the inverse transformation. For the sake of simplicity, this paper will focus on those components based on two's complement arithmetic. The following encoding examples will be referred to in succeeding sections:

$$\begin{aligned} 0 &= \text{Encode}(\mathbf{0}), & \mathbf{0} &= \text{Decode}(0) = 00\cdots00 \\ 1 &= \text{Encode}(\mathbf{1}), & \mathbf{1} &= \text{Decode}(1) = 00\cdots01 \\ -1 &= \text{Encode}(-\mathbf{1}), & -\mathbf{1} &= \text{Decode}(-1) = 11\cdots11 \\ -2 &= \text{Encode}(-\mathbf{2}), & -\mathbf{2} &= \text{Decode}(-2) = 11\cdots10. \end{aligned}$$

A minimum-order polynomial can be determined that fits the set of coordinates (x, y). If the order of this polynomial is known to be n, then $n+1$ coordinates can be extracted from the function and a set of $n+1$ equations and variables (the coefficients of the polynomial) can be constructed and solved. Thus, the problem of generating a word level polynomial representation for a Boolean function reduces to determining the order of the polynomial.

A. Existence and Uniqueness

The following theorem is the basis for determining the polynomial representation of circuits described at the bit level. This theorem, derived from the binomial distribution from traditional calculus, is proven for integers and used to prove the existence and uniqueness of polynomial representations of Boolean functions.

Theorem 3.1: Given a polynomial function $F(x)$ of order n, where $x \in Z$, the function $F(x+1) - F(x)$ is of order exactly $n-1$.

Proof: Let

$$F(x) = \sum_{i=0}^{n} c_i \cdot x^i$$
$$F(x+1) - F(x) = \sum_{i=0}^{n} c_i \cdot (x+1)^i - c_i \cdot x^i.$$

Each term of order i in $F(x)$ contributes a polynomial of order exactly $i-1$ to $F(x+1) - F(x)$

$$c_i(x+1)^i - c_i x^i = c_i \cdot \left(\sum_{j=0}^{i} \binom{i}{j} \cdot x^j - x^i \right)$$
$$= c_i \left(\sum_{j=0}^{i-1} \binom{i}{j} \cdot x^j \right).$$

Thus, when $F(x+1) - F(x)$ is computed, the polynomial term $c_n x^n$ of $F(x)$ contributes a polynomial of order exactly $n-1$ and is the only term to do so. Therefore, $F(x+1) - F(x)$ is of order exactly $n-1$. □

Although this paper will focus on integer encodings of Boolean vectors, note that Theorem 3.1 holds for any domain of x in which addition, subtraction, and multiplication are defined and the associative, distributive, and identity properties hold (e.g., $x \in R$). Furthermore, the theorem is independent of the details of the encoding of $\mathbf{x} \in B^m$ (e.g., two's complement, sign magnitude, fixed point, floating point). To illustrate Theorem 3.1 for $x \in Z$, note that if $F(x) = x^3$, then $F(x+1) - F(x) = x^3 + 3x^2 + 3x + 1 - x^3 = 3x^2 + 3x + 1$. From Theorem 3.1, a useful corollary can be derived.

Corollary 3.1.1: For all $x, m \in Z$, the following set of row vectors is linearly independent:

$$A = \begin{bmatrix} (x)^m & (x)^{m-1} & \cdots & x^0 \\ (x+1)^m & (x+1)^{m-1} & \cdots & (x+1)^0 \\ \cdots & \cdots & \cdots & \cdots \\ (x+m)^m & (x+m)^{m-1} & \cdots & (x+m)^0 \end{bmatrix}.$$

Proof: The set of row vectors can be reduced by multiplying it by nonsingular matrices. The matrices shown in the following computation are nonsingular (the determinant of each is 1), as shown in the equation at the bottom of the next page. The rows of matrix B are linearly independent. Therefore, the original set of vectors A are linearly independent. □

Example 3.1.1: To illustrate Corollary 3.1.1, notice that for $x = 0$ and $m = 3$

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 8 & 4 & 2 & 1 \\ 27 & 9 & 3 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 7 & 3 & 1 & 0 \\ 19 & 5 & 1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 6 & 2 & 0 & 0 \\ 12 & 2 & 0 & 0 \end{bmatrix}$$
$$\Rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 6 & 2 & 0 & 0 \\ 6 & 0 & 0 & 0 \end{bmatrix}.$$

Thus, the initial set of vectors is linearly independent.

The following theorems establish the existence of polynomial representations for combinational univariate functions and the uniqueness of the minimum-order polynomial representation.

Theorem 3.2 (Existence): Let $\mathbf{x} \in B^m$, $\mathbf{y} \in B^k$, and $x, y \in Z$ be the integers corresponding to $\mathbf{x}$, $\mathbf{y}$. Given a Boolean function $\mathbf{y} = \mathbf{F}(\mathbf{x})$: $B^m \rightarrow B^k$, there exists a polynomial $y = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_0$, where $n < 2^m$, that defines the corresponding function F: $x \rightarrow y$.

Proof: If $\mathbf{x} \in B^m$, then there are 2^m possible values that x can take on $\{0, 1, \ldots, 2^m - 1\}$ and 2^m corresponding values that y can take on $\{F(0), F(1), \ldots, F(2m-1)\}$. The solution to the set of linear equations ($\mu = 2^m - 1$)

$$\begin{bmatrix} (0)^\mu & (0)^{\mu-1} & \cdots & 1 \\ (1)^\mu & (1)^{\mu-1} & \cdots & (1)^0 \\ \cdots & \cdots & \cdots & \cdots \\ (\mu)^\mu & (\mu)^{\mu-1} & \cdots & (\mu)^0 \end{bmatrix} \bullet \begin{bmatrix} c_\mu \\ c_{\mu-1} \\ \cdots \\ c_0 \end{bmatrix} = \begin{bmatrix} F(0) \\ F(1) \\ \cdots \\ F(\mu) \end{bmatrix}$$

exists if no row of the matrix

$$\begin{bmatrix} (0)^{\mu} & (0)^{\mu-1} & \cdots & 1 \\ (1)^{\mu} & (1)^{\mu-1} & \cdots & (1)^{0} \\ \cdots & \cdots & \cdots & \cdots \\ (\mu)^{\mu} & (\mu)^{\mu-1} & \cdots & (\mu)^{0} \end{bmatrix}$$

is a linear combination of the others. We know this is true from Corollary 3.1.1. Note that the dimension of *y* does not affect the polynomial representation of $\mathbf{F}$. □

Theorem 3.3 (Uniqueness): The minimum-order polynomial representation of a Boolean function $\mathbf{y} = \mathbf{F}(\mathbf{x})$: $B^m \rightarrow B^k$ is unique.

Proof: Assume there exist two minimum-order polynomial representations for $\mathbf{F}(\mathbf{x})$, where $x, y \in Z$ are the integers corresponding to $\mathbf{x}$, $\mathbf{y}$

$$y = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$$
$$y = b_n x^n + b_{n-1}x^{n-1} + \cdots + b_0$$

$\Rightarrow$ there are two possible solutions to the set of linear equations

$$\begin{bmatrix} (0)^{n} & (0)^{n-1} & \cdots & 1 \\ (1)^{n} & (1)^{n-1} & \cdots & (1)^{0} \\ \cdots & \cdots & \cdots & \cdots \\ (n)^{n} & (n)^{n-1} & \cdots & (n)^{0} \end{bmatrix} \bullet \begin{bmatrix} c_n \\ c_{n-1} \\ \cdots \\ c_0 \end{bmatrix} = \begin{bmatrix} \text{Encode}(\mathbf{F}(00\cdots00)) \\ \text{Encode}(\mathbf{F}(00\cdots01)) \\ \cdots \\ \text{Encode}(\mathbf{F}(\text{Decode}(n))) \end{bmatrix}.$$

$\Rightarrow$ there exists a row in the matrix

$$\begin{bmatrix} (0)^{n} & (0)^{n-1} & \cdots & 1 \\ (1)^{n} & (1)^{n-1} & \cdots & (1)^{0} \\ \cdots & \cdots & \cdots & \cdots \\ (n)^{n} & (n)^{n-1} & \cdots & (n)^{0} \end{bmatrix}$$

that is a linear combination of the others. But from Corollary 3.1.1 we know that this is not possible $\Rightarrow\Leftarrow$. Therefore, the minimum-order polynomial is unique. □

Example 3.1.2: An example of the application of Theorems 3.2 and 3.3 is the following set of Boolean equations (input width $m = 2$ and output width $k = 5$) that model an existing circuit

$$F_0(x) = x_0$$
$$F_1(x) = x_1 \cdot x_0$$
$$F_2(x) = 0$$
$$F_3(x) = x_1$$
$$F_4(x) = x_1 \cdot x_0.$$

$y = x_3$ is the unique minimum-order polynomial ($n = 3$) that represents this circuit and would match a specification that requires the computation of the third power of x.

B. Polynomial Computation

In the previous section, we have proven that any combinational circuit can be uniquely represented by a minimum-order polynomial. Once the order of this polynomial is determined, then the coefficients of the polynomial can be calculated by examining a finite number of circuit outputs. Thus, the problem of determining a canonical polynomial representation for a circuit can be reduced to finding the order of the polynomial that represents that circuit.

To begin deriving a method for determining the order of a Boolean function, remember from Theorem 3.2 that a polynomial representation $F(x)$, where $x \in Z$, always exists for a Boolean function $\mathbf{y} = \mathbf{F}(\mathbf{x})$: $B^m \rightarrow B^k$. Furthermore, from Theorem 3.1, we might deduce that the order of $\mathbf{F}(\mathbf{x})$ will be reduced by exactly one by computing $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$. Therefore, the order of $F(x)$ could be determined exactly by recursively performing $\mathbf{F}(\mathbf{x}) = \mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$ until this difference is identically zero for all values of $\mathbf{x}$. In the algorithm discussed here, two's complement arithmetic is employed to compute this difference. The number of iterations required to set $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x}) = \mathbf{0}$ is the order of the unique, minimum-order polynomial $F(x)$ that represents the circuit.

In computing the order of a Boolean function, we assume that each output bit (y_0, y_1, …, y_{k-1}) of the function $\mathbf{y} = \mathbf{F}(\mathbf{x})$ is represented as a Binary Decision Diagram. While this does present an exponentially sized data structure for some functions, we will show a heuristic in Section IX that reduces this

---

$$B = \begin{bmatrix} 1 & 0 & 0 & .. & 0 & 0 \\ -1 & 1 & 0 & .. & 0 & 0 \\ 0 & -1 & 1 & .. & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & .. & 0 & 0 \\ 0 & 1 & 0 & .. & 0 & 0 \\ 0 & -1 & 1 & .. & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & -1 & 1 \end{bmatrix} \cdot\cdot \begin{bmatrix} 1 & 0 & 0 & .. & 0 & 0 \\ 0 & 1 & 0 & .. & 0 & 0 \\ 0 & 0 & 1 & .. & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & -1 & 1 \end{bmatrix}$$
$$\cdot \begin{bmatrix} (x)^{m} & (x)^{m-1} & .. & x^{0} \\ (x+1)^{m} & (x+1)^{m-1} & .. & (x+1)^{0} \\ .. & .. & .. & .. \\ (x+m)^{m} & (x+m)^{m-1} & .. & (x+m)^{0} \end{bmatrix}$$
$$= \begin{bmatrix} (x)^{m} & & (x)^{m-1} & \cdots & x^{0} \\ (x+1)^{m} - (x)^{m} & & (x+1)^{m-1} - (x)^{m-1} & \cdots & 0 \\ \cdots & & \cdots & \cdots & \cdots \\ (x+m)^{m} - \left(\binom{m}{1} \cdot (x+m-1)^{m}\right) + \cdots + (-1)^{m} & & 0 & \cdots & 0 \end{bmatrix}.$$

data structure to linear complexity with respect to the number of input bits. In Sections III-B1—B4, we derive in detail the steps required to compute $\mathbf{F}(\mathbf{x}+\mathbf{1})-\mathbf{F}(\mathbf{x})$ and determine if $\mathbf{F}(\mathbf{x}+\mathbf{1})-\mathbf{F}(\mathbf{x})=\mathbf{0}$. These sections provide the rationale for the order computation algorithm shown in Fig. 3.

1) Determining $\mathbf{F}(\mathbf{x}+\mathbf{1})$*:* The first step in computing $\mathbf{F}(\mathbf{x}+\mathbf{1})-\mathbf{F}(\mathbf{x})$ is to determine $\mathbf{F}(\mathbf{x}+\mathbf{1})$. This can be performed in polynomial time by replacing each bit $\{x_i\colon i=1, 2, \ldots, m-1\}$ of $\mathbf{x}$ with $(x_i \oplus x_{i-1}\cdot x_{i-2}\cdot \cdots \cdot x_0)$ and x_0 by x_0' in the BDD of $\mathbf{F}(\mathbf{x})$.

2) Determining $-\mathbf{F}(\mathbf{x})$*:* The next step in computing $\mathbf{F}(\mathbf{x}+\mathbf{1})-\mathbf{F}(\mathbf{x})$ is determining $-\mathbf{F}(\mathbf{x})$. Using two's complement arithmetic, this could be performed by inverting each output bit $F_i(\mathbf{x})$ of $\mathbf{F}(\mathbf{x})$ and adding one [$-\mathbf{F}(\mathbf{x})=\mathbf{F}'(\mathbf{x})+\mathbf{1}$, where $\mathbf{F}'(\mathbf{x})$ is the bitwise complement of $\mathbf{F}(\mathbf{x})$ and $\mathbf{1}$ is the vector $00\cdots01$]. Computation of $\mathbf{F}'(\mathbf{x})$ is simple as it only requires inverting each leaf of each BDD that represents the output $F_i(\mathbf{x})$. However, if we make the assumptions that $\mathbf{F}(\mathbf{x})$ is an m-bit function, $\mathbf{x}$ is an m-bit word, and the BDD of $F_i(\mathbf{x})$ has at least m nodes, computing $\mathbf{F}'(\mathbf{x})+\mathbf{1}$ is of complexity $O(m^4)$ due to the propagation of the carry [carry computation requires $m(m+1)/2$ logic operations each of which is of complexity m^2].

To reduce the complexity the negation, we transform the problem of recursively computing $\mathbf{F}(\mathbf{x})=\mathbf{F}(\mathbf{x}+\mathbf{1})-\mathbf{F}(\mathbf{x})$ until $\mathbf{F}(\mathbf{x})=\mathbf{0}$ to the problem of recursively computing $\mathbf{F}(\mathbf{x})=\mathbf{F}(\mathbf{x}+\mathbf{1})+\mathbf{F}'(\mathbf{x})$ until $\mathbf{F}(\mathbf{x})=-\mathbf{1}$. This is the equivalent of computing $F(x+1)-F(x)-1$ in two's complement encoding. This computation reduces the order of $\mathbf{F}(\mathbf{x})$ by one on each iteration, but avoids the complexity introduced by incrementation. This is possible because, on successive computations of $F(x+1)-F(x)-1$, the subtraction of one does not accumulate

1st iteration:

$$F(x)=F(x+1)-F(x)-1$$

2nd iteration:

$$\begin{aligned}F(x)&=(F(x+2)-F(x+1)-1)\\&\quad-(F(x+1)-F(x)-1)-1\\&=(F(x+2)-F(x+1))-(F(x+1)-F(x))-1.\end{aligned}$$

Thus, instead of computing $\mathbf{F}(\mathbf{x}+\mathbf{1})-\mathbf{F}(\mathbf{x})$ to reduce the order of $\mathbf{F}(\mathbf{x})$ by one, we compute $\mathbf{F}(\mathbf{x}+\mathbf{1})+\mathbf{F}'(\mathbf{x})$, which is a computationally simpler way to reduce the order of $\mathbf{F}(\mathbf{x})$ by one.

3) Performing $\mathbf{F}(\mathbf{x}+\mathbf{1})+\mathbf{F}'(\mathbf{x})$*:* Once $\mathbf{F}(\mathbf{x}+\mathbf{1})$ and $\mathbf{F}'(\mathbf{x})$ have been determined, the two functions are summed to produce the new reduced order $\mathbf{F}(\mathbf{x})$. If this summation is performed in ripple carry fashion, the number of logic operations required is exponentially complex with respect to word length, due to the propagation of the carry. This is a result of the fact that for the ith bit, the carry computation requires 3^i logic operations (note that complexity can be reduced by factoring the equation for ripple carry addition). To eliminate the additional complexity associated with ripple carry addition, a carry–save addition can be performed. Let us define

$$\begin{aligned}\mathbf{F_{sum}}(\mathbf{x})&=\mathbf{F}(\mathbf{x}+\mathbf{1})\oplus\mathbf{F}'(\mathbf{x})\\\mathbf{F_{carry}}(\mathbf{x})&=\mathbf{F}(\mathbf{x}+\mathbf{1})\cdot\mathbf{F}'(\mathbf{x})\end{aligned}$$

where $\oplus$ and $\cdot$ are applied bitwise. Thus, $\mathbf{F}(\mathbf{x})$ is uniquely specified as

$$\mathbf{F}(\mathbf{x})=\mathbf{F_{sum}}(\mathbf{x})+(\mathbf{F_{carry}}(\mathbf{x})\ll 1).$$

Note that there are now two terms that must be complemented when recursively computing $\mathbf{F}(\mathbf{x})=\mathbf{F}(\mathbf{x}+\mathbf{1})+\mathbf{F}'(\mathbf{x})$. These terms are $\mathbf{F_{sum}}(\mathbf{x})$ and $\mathbf{F_{carry}}(\mathbf{x})\ll 1$. Complementing both terms requires, according to two's complement arithmetic, a bit-wise inversion and an increment of each term. As in Section III-B2, in order to avoid these increments and their associated carry operations, order reduction can be performed by recursively computing

$$\begin{aligned}\mathbf{F}(\mathbf{x})&=\mathbf{F_{sum}}(\mathbf{x}+\mathbf{1})+(\mathbf{F_{carry}}(\mathbf{x}+\mathbf{1})\ll 1)\\&\quad+\mathbf{F}'_{\mathbf{sum}}(\mathbf{x})+(\mathbf{F}'_{\mathbf{carry}}(\mathbf{x})\ll 1)\end{aligned}$$

until $\mathbf{F}(\mathbf{x})=-\mathbf{2}$. The condition for terminating recursion has changed to $\mathbf{F}(\mathbf{x})=-\mathbf{2}$ because the equivalent computation in two's complement arithmetic is

$$\begin{aligned}&F_{sum}(x+1)+(F_{carry}(x+1)\ll 1)\\&\quad-(F_{sum}(x)+(F_{carry}(x)\ll 1))-2\\&=F(x+1)-F(x)-2.\end{aligned}$$

Since $\mathbf{F}(\mathbf{x}+\mathbf{1})$ and $\mathbf{F}'(\mathbf{x})$ are specified as the summation of a sum and carry term, their summation can be performed in two steps, as if two carry–save additions (Fig. 2) were executed.

With these transformations, the order of $\mathbf{F}(\mathbf{x})$ is successively being reduced by one by recursively computing $\mathbf{F}(\mathbf{x})=\mathbf{F}(\mathbf{x}+\mathbf{1})+\mathbf{F}'(\mathbf{x})$. This computation is of polynomial complexity with respect to the size of the BDD representation of $\mathbf{F}(\mathbf{x})$.

4) Checking if $\mathbf{F}(\mathbf{x})=-\mathbf{2}$*:* Using a two's complement encoding, the following transformations can be used to determine if the recursively computed $\mathbf{F}(\mathbf{x})=-\mathbf{2}$, without performing a ripple carry addition

$$\begin{aligned}\mathbf{F}(\mathbf{x})&=-\mathbf{2}\\&\Leftrightarrow\mathbf{F_{sum}}(\mathbf{x})+(\mathbf{F_{carry}}(\mathbf{x})\ll 1)=-\mathbf{2}\\&\Leftrightarrow\mathbf{F_{sum}}(\mathbf{x})+(\mathbf{F_{carry}}(\mathbf{x})\ll 1)+\mathbf{1}=-\mathbf{1}.\end{aligned}$$

To avoid performing the ripple carry addition, a two-stage carry–save increment is performed at the end of each recursive step

$$\mathbf{F_{sum}}(\mathbf{x})+(\mathbf{F_{carry}}(\mathbf{x})\ll 1)+\mathbf{1}=\mathbf{S_{test}}+\mathbf{C_{test}}$$

by performing the following logic operations ($i=1, 2, \ldots, k-1$)

$$\begin{aligned}S_{test_0}(\mathbf{x})&=F'_{sum_0}(\mathbf{x})\\C_{test_0}(\mathbf{x})&=F_{sum_0}(\mathbf{x})\\S_{text_i}(\mathbf{x})&=\left(F_{sum_i}(\mathbf{x})\oplus F'_{carry_{i-1}}(\mathbf{x})\right)\end{aligned}$$

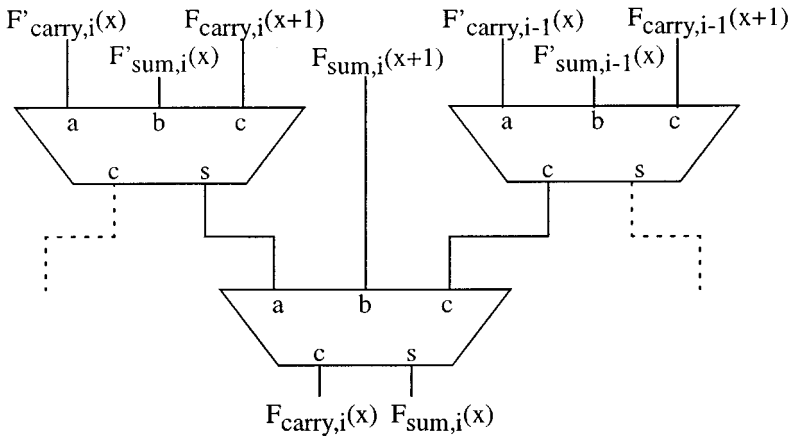


Fig. 2. Physical visualization of the two stage carry–save addition for computation of $\mathbf{F}(\mathbf{x}+\mathbf{1})+\mathbf{F}'(\mathbf{x})$.

$$C_{test_i}(\mathbf{x}) = \begin{aligned} &\oplus \left(F_{sum_{i-1}}(\mathbf{x}) + F_{carry_{i-2}}(\mathbf{x})\right) \\ &\left(F_{sum_i}(\mathbf{x}) \oplus F'_{carry_{i-1}}(\mathbf{x})\right) \\ &\cdot \left(F_{sum_{i-1}}(\mathbf{x}) + F_{carry_{i-2}}(\mathbf{x})\right). \end{aligned}$$

Each bit of the resulting sum ($\mathbf{S_{test}}$) is checked for tautology and each bit of the resulting carry ($\mathbf{C_{test}}$) is checked whether it is tautologically zero. We refer to this test as the *tautology check* and it is necessary and sufficient to guarantee $\mathbf{F_{sum}}(\mathbf{x}) + (\mathbf{F_{carry}}(\mathbf{x}) \ll 1) + \mathbf{1} = \mathbf{-1}$ as proven in Theorem 3.4. As a result, the ripple carry computation does not need to be performed.

Theorem 3.4: Given three Boolean vectors $\mathbf{G_{sum}}, \mathbf{G_{carry}}, \mathbf{G} \in B^k$, where $\mathbf{G} = \mathbf{G_{sum}} + (\mathbf{G_{carry}} \ll 1)$, then $\mathbf{G} = \mathbf{-1}$ iff $G_{sum_0} = 1$, $G_{sum_i} \oplus G_{carry_{i-1}} = 1$ and $G_{sum_i} \cdot G_{carry_{i-1}} = 0$ for all $i = 1, 2, \ldots, k-1$.

Proof: Forward implication (by induction)

Base Case:

$$\mathbf{G} = \mathbf{G_{sum}} + (\mathbf{G_{carry}} \ll 1) \Rightarrow G_0 = G_{sum_0}$$

and

$$\begin{aligned} G_1 &= G_{sum_1} \oplus G_{carry_0} \\ \mathbf{G} &= \mathbf{-1} \Rightarrow G_0 = 1 \Rightarrow G_{sum_0} = 1 \\ \mathbf{G} &= \mathbf{-1} \Rightarrow G_1 = 1 \Rightarrow G_{sum_1} \oplus G_{carry_0} = 1 \end{aligned}$$

and

$$G_{sum_1} \cdot G_{carry_0} = 0$$

Assume:

$$G_{sum_j} \oplus G_{carry_{j-1}} = 1$$

and

$$G_{sum_j} \cdot G_{carry_{j-1}} = 0 \qquad \text{for all } j \le i.$$

Inductive step:

$$G_{j+1} = 1$$

and

$$\begin{aligned} G_{sum_j} \cdot G_{carry_{j-1}} &= 0 \qquad \text{for all } j \le i \\ \Rightarrow G_{sum_{j+1}} \oplus G_{carry_j} &= 1 \end{aligned}$$

and

$$G_{sum_{j+1}} \cdot G_{carry_j} = 0.$$

Reverse implication

$$\begin{aligned} G_{sum_0} = 1 &\Rightarrow G_0 = 1. \\ G_{sum_i} \oplus G_{carry_{i-1}} &= 1, \\ G_{sum_i} \cdot G_{carry_{i-1}} &= 0 \qquad \text{for all } i \Rightarrow G_i = 1. \\ &\Rightarrow \mathbf{G} = \mathbf{-1}. \end{aligned}$$

□

The following assignments allow Theorem 3.4 to be used to perform the tautology check:

$$\begin{aligned} G_{sum_i}(\mathbf{x}) &= \left(F_{sum_i}(\mathbf{x}) \oplus F'_{carry_{i-1}}(\mathbf{x})\right) \\ G_{carry_i}(\mathbf{x}) &= \left(F_{sum_{i-1}}(\mathbf{x}) + F_{carry_{i-2}}(\mathbf{x})\right) \\ S_{test_i}(\mathbf{x}) &= G_{sum_i}(\mathbf{x}) \oplus G_{carry_{i-1}}(\mathbf{x}) \\ C_{test_i}(\mathbf{x}) &= G_{sum_i}(\mathbf{x}) \cdot G_{carry_{i-1}}(\mathbf{x}). \end{aligned}$$

In summary, $\mathbf{F}(\mathbf{x}) = \mathbf{-2}$ if and only if $S_{test_i}(\mathbf{x}) = 1$ and $C_{test_i}(\mathbf{x}) = 0$ for all $i = 0, 1, \ldots, k-1$.

5) Bounding Function: A function $y = F(x)$: $Z \to Z$ has a corresponding Boolean function $\mathbf{y} = \mathbf{F}(\mathbf{x})$: $B^m \to B^k$, $\mathbf{x} =$ Decode(x), and $\mathbf{y} =$ Decode(y), defined only over the domain $[0, 2^m-1]$. This is important to consider when performing order computations because $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$ actually corresponds to $F(0) - F(2^m-1)$ if $\mathbf{x} = \mathbf{-1}$ (e.g., $11\cdots11$). In performing order computations, this may result in $F(x)$ appearing to be non polynomial over the domain $[-\infty, \infty]$ even if $\mathbf{F}(\mathbf{x})$ does have a polynomial representation over the range of possible values for $\mathbf{x}$ (Fig. 3). Thus, in executing order computations, it is necessary to determine a bounding function that specifies which values do not need to be considered when performing tautology checks.

Definition 3.1: Given a function $\mathbf{F}(\mathbf{x})$, where $\mathbf{x} \in B^m$, the bounding function $B(\mathbf{x})$ on the nth order iteration is

$$B(\mathbf{x}) = \sum_{i=2^m-n}^{2^m-1} \qquad (\mathbf{x} = \text{Decode}(i)).$$

In words, this is the sum of the Boolean vectors whose corresponding integer values are greater than $2^m - n$. For example,

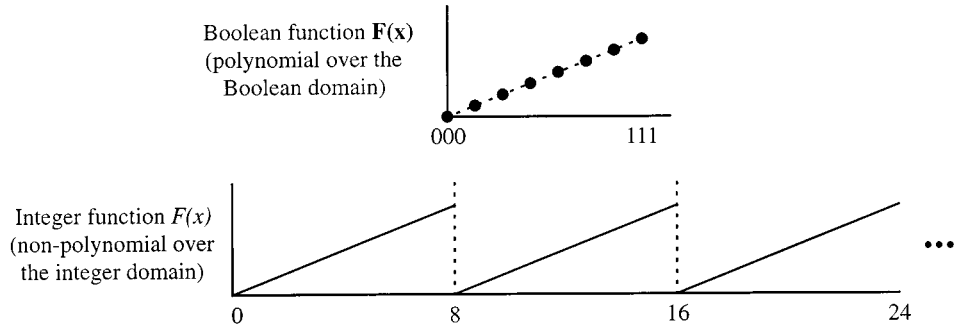


Fig. 3. A Boolean function that is polynomial that appears to be nonpolynomial in the integer domain.

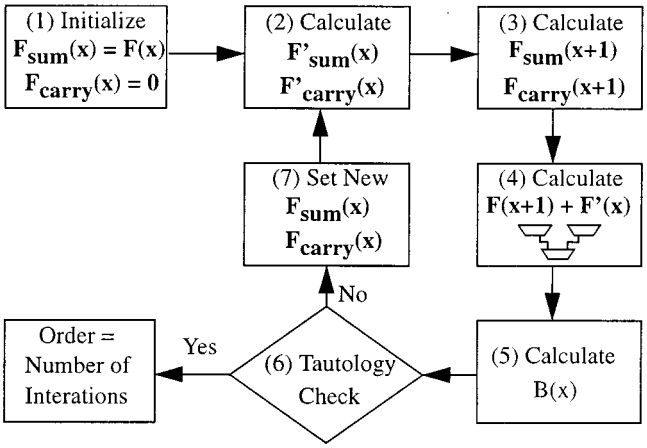


Fig. 4. Algorithm for computing the order of a Boolean function $\mathbf{F}(\mathbf{x})$.

after one recursion of order reduction with respect to an m bit vector $\mathbf{x}$, the bounding function would be $B = x_{m-1} \cdot x_{m-2} \cdot \cdots \cdot x_0$. After two iterations, the bounding function would be $B = x_{m-1} \cdot x_{m-2} \cdot \cdots \cdot x_0 + x_{m-1} \cdot x_{m-2} \cdot \cdots \cdot x_0'$.

If the input is out of range when incremented, i.e., $\mathbf{x} = 11 \cdots 11$, then the resulting $\mathbf{F(x+1) - F(x)}$ is immaterial, since the input pattern can not be applied. Thus, $\mathbf{F(x+1) + F'(x) = -1}$ requires that if $\mathbf{S_{test}}$ is not a tautology, the bounding function must be true. Similarly, if $\mathbf{C_{test}}$ is not tautologically zero, the bounding function must be true if $\mathbf{F(x+1) + F'(x) = -1}$. The tautology check requires that

$$(S_{test_i}(\mathbf{x}) + B(\mathbf{x})) \cdot \left(C'_{test_i}(\mathbf{x}) + B(\mathbf{x})\right) = 1 \quad \text{for all } i = 0, 1, \ldots, k-1.$$

Example 3.2.1: If, after two order computations, $S_{test_0}(\mathbf{x}) = (x_{m-1} \cdot x_{m-2} \cdot \cdots \cdot x_0)'$ and all other bits of $\mathbf{S_{test}}$ and $\mathbf{C'_{test}}$ are a tautology, then $S_{test_0}(\mathbf{x}) + B = (x_{m-1} \cdot x_{m-2} \cdot \cdots \cdot x_0)' + x_{m-1} \cdot x_{m-2} \cdot \cdots \cdot x_0 + x_{m-1} \cdot x_{m-2} \cdot \cdots \cdot x_0' = 1$ and the bit satisfies the tautology check. Thus, within the interval $x = [0, 2^m - 1]$, the original Boolean function $\mathbf{F(x)}$ is of order 2.

6) The Complete Algorithm: The complete algorithm for computing the order of a Boolean function $\mathbf{F(x)}$, given its BDD representation, is shown in Fig. 4.

Step 1) Initialize the function $\mathbf{F_{sum}(x)}$ to $\mathbf{F(x)}$ and the function $\mathbf{F_{carry}(x)}$ to $\mathbf{0}$, an operation of linear complexity with respect to the size of the BDD representation of $\mathbf{F(x)}$.

Step 2) Compute $\mathbf{F'(x)}$ by complementing $\mathbf{F_{sum}(x)}$ and $\mathbf{F_{carry}(x)}$, an operation of constant complexity with respect to BDD size.

Step 3) Compute the function $\mathbf{F(x+1)}$ by replacing $\mathbf{x}$ with $\mathbf{x+1}$ in the functions $\mathbf{F_{sum}(x)}$ and $\mathbf{F_{carry}(x)}$, an operation of quadratic complexity with respect to BDD size.

Step 4) Reduce the order of $\mathbf{F(x)}$ by exactly one by computing the sum $\mathbf{F(x+1) + F'(x)}$. This computation is performed by adding the results of Steps 2) and 3) with a two-stage carry–save addition, producing a new $\mathbf{F_{sum}(x)}$ and $\mathbf{F_{carry}(x)}$. This step is of quadratic complexity with respect to BDD size.

Step 5) Compute the bounding function $B(\mathbf{x})$ that restricts the domain over which the sum $\mathbf{F(x+1) + F'(x)}$ is evaluated, an operation that is of constant complexity relative to BDD size.

Step 6) Check the sum $\mathbf{F(x+1) + F'(x)}$ to see if each output bit is a tautology within the bounds specified by $B(\mathbf{x})$, an operation of constant complexity with respect to BDD size. If the tautology check is unsuccessful.

Step 7) Set $\mathbf{F_{sum}(x)}$ and $\mathbf{F_{carry}(x)}$ to the result of Step 5) and initiates a new recursion, an operation of linear complexity with respect to BDD size. Otherwise, the order of the minimum-order polynomial representa-

tion is one less than the number of recursive computations that were performed.

Example 3.2.2: Consider the function $\mathbf{y} = \mathbf{F}(\mathbf{x})$, where $\mathbf{x} \in B^2$ and $\mathbf{y} \in B^5$, that implements $F(x) = x^2$. Initializing the sum $\mathbf{s}$ to $\mathbf{F}(\mathbf{x})$ and the carry $\mathbf{c}$ to zero yields the following input vectors:

$$\begin{array}{ll} s_0 = x_0 & c_0 = 0 \\ s_1 = 0 & c_1 = 0 \\ s_2 = x_0' \cdot x_1 & c_2 = 0 \\ s_3 = x_0 \cdot x_1 & c_3 = 0 \\ s_4 = 0 & c_4 = 0. \end{array}$$

The following steps are followed to determine the order of these input vectors.

1) $\mathbf{F}(\mathbf{x}+\mathbf{1})$:

$$\begin{array}{ll} s_0 = x_0' & c_0 = 0 \\ s_1 = 0 & c_1 = 0 \\ s_2 = x_0 \cdot (x_1 \oplus x_0) & c_2 = 0 \\ s_3 = x_0' \cdot (x_1 \oplus x_0) & c_3 = 0 \\ s_4 = 0 & c_4 = 0. \end{array}$$

2) $\mathbf{F}'(\mathbf{x})$:

$$\begin{array}{ll} s_0 = x_0' & c_0 = 1 \\ s_1 = 1 & c_1 = 1 \\ s_2 = x_0 + x_1' & c_2 = 1 \\ s_3 = x_0' + x_1' & c_3 = 1 \\ s_4 = 1 & c_4 = 1. \end{array}$$

3) $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$ (1st iteration)

$$\begin{array}{ll} s_0 = 1 & c_0 = 0 \\ s_1 = x_0' & c_1 = 1 \\ s_2 = x_1 \oplus x_0 & c_2 = x_1' \oplus x_0 \\ s_3 = x_0 + x_1 & c_3 = x_1' \\ s_4 = x_0' \cdot x_1 & c_4 = 1. \end{array}$$

4) Tautology Check

$$s_0 = 0 \qquad c_0 = 0 \text{ fails.}$$

5) $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$ (2nd iteration)

$$\begin{array}{ll} s_0 = 0 & c_0 = 1 \\ s_1 = 1 & c_1 = 0 \\ s_2 = 1 & c_2 = 0 \\ s_3 = x_0' + x_1' & c_3 = x_0 \\ s_4 = x_0' \oplus x_1 & c_4 = x_0' \cdot x_1. \end{array}$$

6) Tautology Check

$$\begin{array}{ll} s_0 = 1 & c_0 = 0 \\ s_1 = 0 & c_1 = 0 \text{ fails.} \end{array}$$

7) $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$ (3rd iteration)

$$\begin{array}{ll} s_0 = 0 & c_0 = 1 \\ s_1 = 0 & c_1 = 0 \\ s_2 = 1 & c_2 = 1 \\ s_3 = x_1 & c_3 = x_1' \\ s_4 = x_0 \cdot x_1' & c_4 = x_0 + x_1. \end{array}$$

8) Tautology Check

$$\begin{array}{ll} s_0 = 1 & c_0 = 0 \\ s_1 = 1 & c_1 = 0 \\ s_2 = 1 & c_2 = 0 \\ s_3 = 1 & c_3 = 0 \\ s_4 = 1 & c_4 = 0. \end{array}$$

Three iterations reduce $\mathbf{F}(\mathbf{x})$ to zero for all $\mathbf{x}$. Thus, $\mathbf{F}(\mathbf{x})$ is of order 2.

Each step within the order computation algorithm is of polynomial complexity with respect to the number of nodes in the BDD representation of $\mathbf{F}(\mathbf{x})$. However, the minimum-order polynomial representation may be of exponential order with respect to the number of bits in the input word $\mathbf{x}$. Thus, the number of recursions that are performed may be exponential. Sections IV and VII detail partitioning and approximation algorithms for efficiently generating polynomial representations for those circuits whose representations would otherwise be of exponential order.

Once the order of the function has been determined to be n, $\mathbf{F}(\mathbf{x})$ is evaluated at $\mathbf{x} = 00\cdots00$, $\mathbf{x} = 00\cdots01, \ldots, \mathbf{x} =$ Decode(n). Solving the following set of linear equations for $c_0, c_1, \ldots, c_n$ yields the polynomial representation of the Boolean function

$$\begin{bmatrix} (0)^n & (0)^{n-1} & \cdots & 1 \\ (1)^n & (1)^{n-1} & \cdots & (1)^0 \\ \cdots & \cdots & \cdots & \cdots \\ (n)^n & (n)^{n-1} & \cdots & (n)^0 \end{bmatrix} \bullet \begin{bmatrix} c_n \\ c_{n-1} \\ \cdots \\ c_0 \end{bmatrix} = \begin{bmatrix} \text{Encode}(\mathbf{F}(00\cdots00)) \\ \text{Encode}(\mathbf{F}(00\cdots01)) \\ \cdots \\ \text{Encode}(\mathbf{F}(\text{Decode}(n))) \end{bmatrix}.$$

C. Extension to Multivariable Functions

The techniques described above consider only univariable functions. However, multivariable polynomials exhibit the same features that allow order computation to be performed recursively; that is, $\mathbf{F}(\mathbf{x}, \mathbf{y}) = \mathbf{F}(\mathbf{x}+\mathbf{1}, \mathbf{y}) + \mathbf{F}'(\mathbf{x}, \mathbf{y})$ recursively reduces the order of $\mathbf{F}(\mathbf{x}, \mathbf{y})$ with respect to $\mathbf{x}$ by one on each iteration if $\mathbf{y}$ is held constant. Thus, the order of $\mathbf{F}(\mathbf{x}, \mathbf{y})$ can be determined with respect to $\mathbf{x}$ and with respect $\mathbf{y}$. However, the unique, minimum-order polynomial computation requires solving a set of $n_x n_y$ simultaneous linear equations, where n_x is the order with respect to $\mathbf{x}$ and n_y is the order with respect to $\mathbf{y}$.

IV. Representation of Functions Containing Branches

To this point, the methods we have described allow computation of a polynomial representation for combinational circuits. As proven in Theorem 3.2, polynomial representations exist for all combinational circuits. For those circuits that implement arithmetic functions such as those generated by composing addition and multiplication operations, this representation is of very low order (e.g., one term to represent multiplication, two terms to represent addition). Consider, however, models of combinational circuits that contain branches, i.e., discontinuities.

For such circuits, polynomial representations, if computed using only the techniques described above, are usually of exponential order with respect to input word size. This is because a branch in the Boolean domain usually describes a set of coordinates in the integer domain that can only be fit to an exponentially-large polynomial. However, a high-order polynomial representation is an indicator that a branch exists within a circuit. This indicator can be used to partition circuit inputs into domains in which polynomial representations of low complexity exist. The boundaries of these domains are termed *discontinuities*.

Example 4.1: Consider the JPEG Coefficient Encoder **coefficient** $= \mathbf{F}(\mathbf{q})$, with a 16-bit input and 4-bit output, which selects an output based on the range of the quantized input values

$$\begin{aligned}&\text{if } (\mathbf{q} == 0\,000\,000\,000\,000\,000)\ \mathbf{coefficient} = 0000;\\ &\text{else if } (\mathbf{q} < 00\,000\,000\,000\,000\,010)\ \mathbf{coefficient} = 0001;\\ &\text{else if } (\mathbf{q} < 00\,000\,000\,000\,000\,100)\ \mathbf{coefficient} = 0010;\\ &\cdots\\ &\text{else } \mathbf{coefficient} = 1111.\end{aligned}$$

The encoder is performing an operation within each branch that is represented by polynomials of order zero. However, using the order computation methods described above, the discontinuities at the integer values $q = 2^i$ cause the overall circuit to have a polynomial representation of order 2^{16}.

To prevent an exponential number of order computation recursions from being performed on functions that contain branches, we use a heuristic based on a *discontinuity threshold*. Once the number of iterations has reached this threshold, the function is assumed to contain branches. The threshold is determined heuristically and enables efficient detection of discontinuities. Discontinuity detection, in turn, allows order computation to be performed on each branch of the circuit model.

Given a function $\mathbf{y} = \mathbf{F}(\mathbf{x})$: $B^m \rightarrow B^k$, with order greater than the discontinuity threshold, discontinuities can be detected by performing order computation on $\mathbf{F}(\mathbf{x})$ for the case $x_{m-1} = 0$ and the case $x_{m-1} = 1$. If the orders for each computation are different, and below the discontinuity threshold, a discontinuity has been detected and exists between $\mathbf{x} = 01\cdots11$ and $\mathbf{x} = 10\cdots00$. If the order of $\mathbf{F}(\mathbf{x})$, for $x_{m-1} = 0$ or $x_{m-1} = 1$, is still above the threshold, then a discontinuity exists within the corresponding domain. Within that domain, an order computation is then performed on $\mathbf{F}(\mathbf{x})$ for the case $x_{m-2} = 0$ and the case $x_{m-2} = 1$. Domain partitioning continues until the discontinuity is detected.

Similar to performing a binary search, detection of a single discontinuity is of linear complexity with respect to the number of input bits, not considering the complexity of the order computation.

Example 4.2: Consider the function $\mathbf{y} = \mathbf{F}(\mathbf{x})$, where $\mathbf{x} \in B^4$, that is implemented by the following Verilog code:

$$\begin{aligned}&\text{if } (\mathbf{x} > 4'b1011)\\ &\text{then } \mathbf{y} = \mathbf{x}^*\mathbf{x}^*\mathbf{x};\\ &\text{else } \mathbf{y} = \mathbf{x}^*\mathbf{x}.\end{aligned}$$

If we proceed blindly, computing the order of $\mathbf{F}(\mathbf{x})$ will generate an order of 2^4 because of the discontinuity at $\mathbf{x} = 1011$. However, if we start with an initial discontinuity threshold of four, then after four order iterations, the uppermost bit of x will be set to zero, then one, and the order computations will be performed for each case. The order computation for $x_3 = 0$ will result in an order of two. The order computation for $x_3 = 1$ will again reach the fourth iteration without passing the tautology check. The second most significant bit is set to zero, then one, and the order computation is performed again. Then order computation for $x_3x_2 = 11$ will result in an order of 3 and the computation for $x_3x_2 = 10$ will result in an order of two. Since both computations converged, but converged to different values, there is a discontinuity on the interval boundary. Thus, over the integer interval [0, 11] an order of two is determined and over the integer interval [12, 15] an order of three is determined.

Every discontinuity detected introduces a new polynomial into the description of a component. If the number of discontinuities is large, the polynomial representation of a component will also become large. Such cases can be handled by implementing a heuristic based on a *domain threshold*. If the number of discontinuities is greater than this threshold, then the functionality of the component may be approximated by the polynomial representation. The approximation technique is described in Section VII.

V. Synchronous Acyclic Circuits

From Theorem 3.2, we have established that a polynomial representation, $y = F(x)$, exists for all combinational circuits. This is due to the fact that combinational circuits specify a finite number of input/output pairs ($\mathbf{x}$, $\mathbf{y}$) with corresponding integer values (x, y) that can be treated as coordinates to which a polynomial can be fit. Synchronous circuits pose an additional problem because circuit outputs are not only a function of the current inputs but also previous inputs. Thus, the polynomial representation of a synchronous circuit contains terms that are dependent on previous input values: $y = F(x, x@1, x@2, \ldots, x@p)$. The symbol $x@i$ indicates the value of x that is delayed by i cycles.

A. Determining Combinational Equivalents

A polynomial representation for synchronous acyclic circuits can be computed by computing the polynomial representation for the equivalent combinational circuit with delayed input values. Consider a synchronous circuit represented by a synchronous logic network, i.e., a directed acyclic graph whose vertices represent combinational logic functions, whose edges represent function dependencies, and whose edge weights represent synchronous delays introduced by registers. The sequential depth of the network, p, is the weight of the longest path. A synchronous logic network can be transformed, as shown in Fig. 5, into a combinational function of delayed input variables with delay less than or equal to p.

Given a synchronous network with depth p, the equivalent combinational function is $\mathbf{F}(\mathbf{x}, \mathbf{x@1}, \mathbf{x@2}, \ldots, \mathbf{x@p})$. Note that p is finite due to the restriction that the circuit does not have feedback. A polynomial representation for $\mathbf{F}(\mathbf{x})$ can now

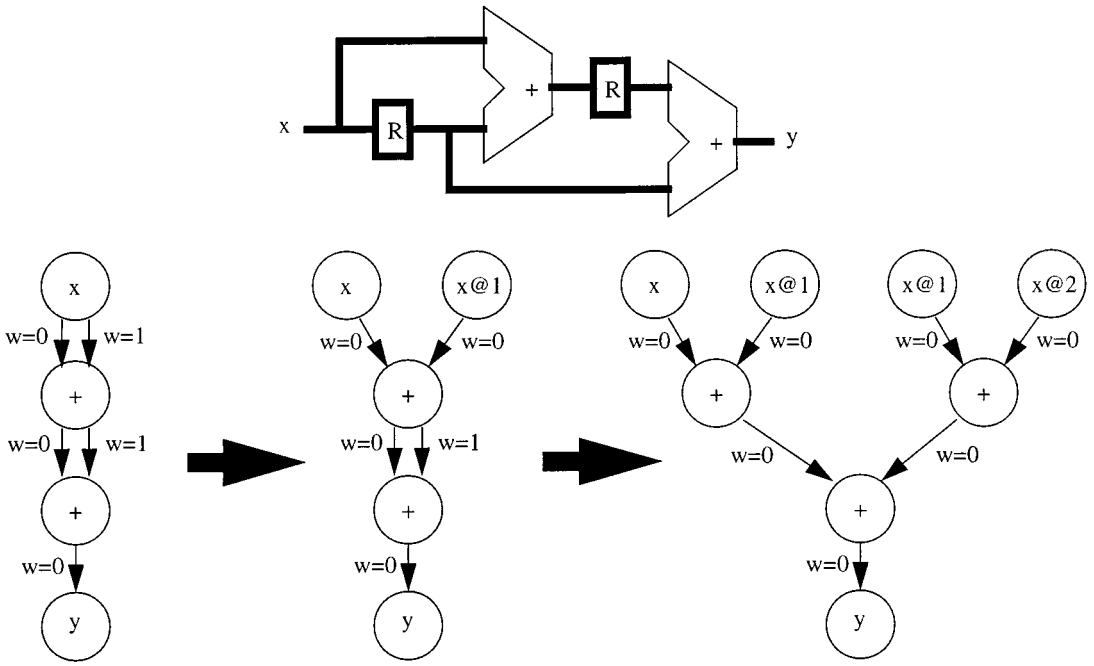


Fig. 5. Transformation of a sequential adder into a combinational circuit.

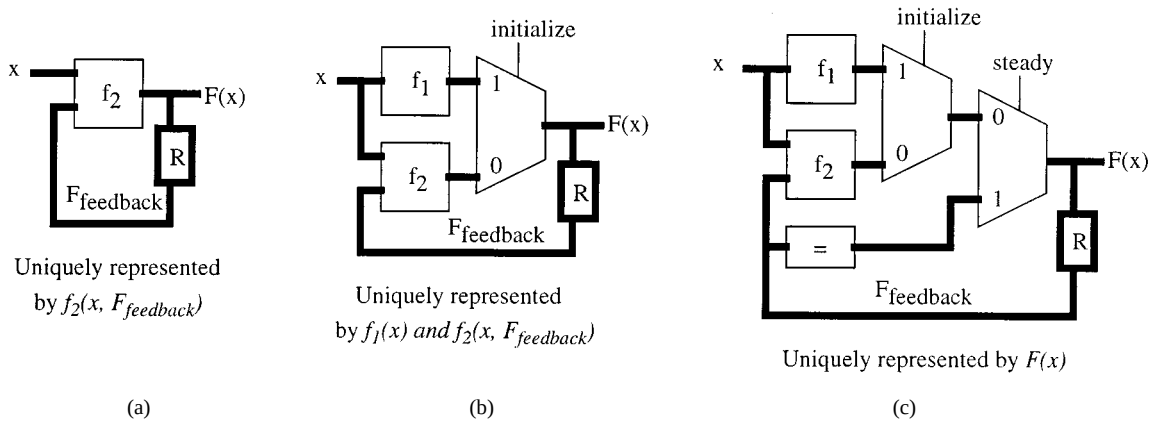


Fig. 6. Synchronous cyclic circuit models: (a) with only a transient feedback branch; (b) with a transient and an initialization branch; and (c) with a transient, initialization, and steady state branch.

be determined from $\mathbf{F}(\mathbf{x}, \mathbf{x@1}, \mathbf{x@2}, \ldots, \mathbf{x@p})$. The order of $\mathbf{F}(\mathbf{x}, \mathbf{x@1}, \mathbf{x@2}, \ldots, \mathbf{x@p})$ is determined with respect to each $\mathbf{x@j}$ for $0 \leq j \leq p$ as independent variables and the coefficients of the polynomial representation are determined. In the example of Fig. 5, this would result in the polynomial representation $F(x) = x + 2x@1 + x@2$.

VI. Synchronous Cyclic Circuits

The method for determining polynomial representations for sequential acyclic circuits relied on the acyclic nature of the circuit to guarantee that a finite number of time-shifted inputs were required. However, by breaking the feedback path of a cyclic circuit $\mathbf{F}(\mathbf{x})$, the previous techniques can be used to derive the order of the cyclic circuit. This is achieved by introducing an input $\mathbf{F_{feedback}}$, and determining the order of $\mathbf{F}(\mathbf{x}, \mathbf{F_{feedback}})$ with respect to $\mathbf{x}$ and $\mathbf{F_{feedback}}$.

A synchronous cyclic circuit can be modeled as a Mealy/Moore finite state machine (FSM) that may or may not have an initial state. For example, a rasterizer is a synchronous cyclic circuit with an initial state and an infinite impulse response filter is a synchronous cyclic circuit with no initial state. For the sake of this analysis, we consider three different topologies of synchronous cyclic circuits: 1) an FSM with no initial state; 2) an FSM with an initial state that does not reach a steady state; and 3) an FSM with an initial state that reaches a steady state after a finite number of cycles. As shown in Fig. 6, we can represent each of these topologies as a function $\mathbf{F}(\mathbf{x})$ that may have up to three branches: a branch corresponding to an initialization state $(\mathbf{f_1}(\mathbf{x}))$, a branch corresponding to the transient states $(\mathbf{f_2}(\mathbf{x}, \mathbf{F_{feedback}}))$, and a branch corresponding to a steady state (labeled $=$). The techniques described in Section IV enable automatic detection of each of these branches. However, this is beyond the scope of this article. The succeeding discussion assumes that the

TABLE I
POLYNOMIAL REPRESENTATION FOR THE ARBITER OF EXAMPLE 6.1

Domain	Polynomial
initialize = 1	*F(initialize, $F_{feedback}$) = 4*
initialize = 0 AND *1 < $F_{feedback}$*	*F(initialize, $F_{feedback}$) = 2*
initialize = 0 AND *1 < $F_{feedback}$ < 4*	*F(initialize, $F_{feedback}$) = 1*
initialize = 0 AND *$F_{feedback}$ < 2*	*F(initialize, $F_{feedback}$) = 4*

presence of each of these branches has been detected and the polynomial representation has been determined.

Using the techniques described previously, we can compute a polynomial representation for each branch. An initialization branch has a polynomial representation that contains no terms with the variable F_{feedback}. A steady-state branch has the polynomial representation $F(x, F_{\text{feedback}}) = F_{\text{feedback}}$. If the function contains no initialization branch or no steady state branch [topology 1) or 2)], then no polynomial representation $F(x)$ exists. However, the circuit is uniquely represented by the polynomial $F(x, F_{\text{feedback}})$. In the case of topology 1), $F(x, F_{\text{feedback}})$ is simply $f_2(x, F_{\text{feedback}})$. In the case of topology 2), $F(x, F_{\text{feedback}})$ is comprised of two domains (corresponding to $initialize = 1$ and $initialize = 0$ in Fig. 6), and is $f_1(x)$ within the first domain and $f_2(x, F_{\text{feedback}})$ within the second domain. Example 6.1 illustrates computation of a polynomial representation for FSM with topology 2).

Example 6.1: Consider the finite state machine with a one bit input (initialize) and a three bit output $\mathbf{F}(\text{initialize}) = \{\text{enableA, enableB, enableC}\}$ that provides round-robin access to memory for three clients. Breaking the feedback loops yields the function $\mathbf{F}(\text{initialize}, \mathbf{F_{feedback}})$. Performing order computation results in the detection of four branches, each of which is order zero (i.e., constant). For example, in the branch that is executed under the condition initialize $= 1$, the output $\mathbf{F}(\text{initialize}, \mathbf{F_{feedback}}) = \{\text{enableA, enableB, enableC}\} = 100$. Thus, the polynomial representation for this branch is $F(initialize, F_{\text{feedback}}) = 4$. Coefficient computation for each branch yields the following order zero polynomial representations for $F(initialize, F_{\text{feedback}})$ as shown in Table I. An initialization branch exists, but no steady-state branch exists, thus $F(initialize, F_{\text{feedback}})$ uniquely represents the finite state machine (although other finite state machines exist that perform the same operation with different state encodings).

The remainder of this analysis focuses on circuits for which $F(x, F_{\text{feedback}})$ is not a unique representation, i.e., those circuits that contain both an initialization state and steady state [topology 3)].

A. Order Computation With Feedback

Assume function $\mathbf{y} = \mathbf{F}(\mathbf{x})$: $B^m \rightarrow B^k$ implements three branches, one initialization branch $(\mathbf{f_1}(\mathbf{x}))$, one steady state branch, and one transient feedback branch $(\mathbf{f_2}(\mathbf{x}, \mathbf{F_{feedback}}))$. We assume that a signal controls the number of iterations through the transient feedback path. We can then evaluate the circuit based on the number of iterations of the transient feedback branch.

The order of $\mathbf{f_1}(\mathbf{x})$ with respect to $\mathbf{x}$, referred to as n_{x1}, can be determined using the techniques presented in earlier sections. As a result, the polynomial representation of this branch, $f_1(x)$, can be determined. Furthermore, if $\mathbf{y} = \mathbf{F_{feedback}}(\mathbf{x})$ is treated as an input to $\mathbf{f_2}(\mathbf{x}, \mathbf{y})$, then the order of $\mathbf{f_2}(\mathbf{x}, \mathbf{y})$ with respect to $\mathbf{x}$, referred to as n_{x2}, and with respect to $\mathbf{y}$, referred to as n_y, can also be determined. As a result, the polynomial representation of this branch $f_2(x, y)$ can be determined. After initialization, the order of $\mathbf{F}(\mathbf{x})$ is n_{x1} and after the first iteration of the nonsteady-state feedback branch, the order of $\mathbf{F}(\mathbf{x})$ is less than $(n_y n_{x1} + n_{x2})$ and greater than $n_y n_{x1}$. In general, if the order of $\mathbf{F}(\mathbf{x})$ is n_t after t iterations, then the order of $\mathbf{F}(\mathbf{x})$, after one more iteration of the nonsteady-state feedback branch is less than $n_y n_t + n_2$ and greater than $n_y n_t$. Thus, the upper bound on the order of $\mathbf{F}(\mathbf{x})$ after t iterations is:

$$n_y^t \cdot n_{x1} + \sum_{i=0}^{t-1} n_y^i \cdot n_{x2}.$$

To determine the order of $\mathbf{F}(\mathbf{x})$ there are three cases that follow, which need to be considered:

1) t is known;
2) t is not known, $n_y = 1$, and there is no xy term in $f_2(x, y)$;
3) t is not known, and [$n_u \neq 1$ or there is an xy term in $f_2(x, y)$].

In *case* 1), the order of $\mathbf{F}(\mathbf{x})$ can be bounded according to the equation above. In *case* 2), since there is no xy term in $f_2(x, y)$, the order of $\mathbf{F}(\mathbf{x})$ does not increase on successive iterations and is simply the greater of n_{x1} and n_{x2}. For both of these cases, since the order of $\mathbf{F}(\mathbf{x})$ is bounded, a polynomial representation exists for $\mathbf{F}(\mathbf{x})$. If the upper bound on the order is n_u, this representation can be determined by extracting $n_u + 1$ points from the circuit to create the system of linear equations that determine the polynomial coefficients. In *case* 3), the order of $\mathbf{F}(\mathbf{x})$ is dependent on t and is therefore unbounded and has no polynomial representation. However, like the cyclic circuits with no initialization or steady state branch, the polynomial representation $F(x, F_{\text{feedback}})$ uniquely specifies the functionality of the circuit, and can be used to perform matching as shown in Section VIII-C.

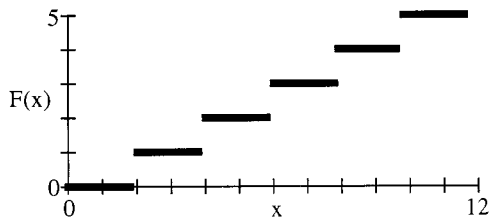
Example 6.1.1: Consider a Boolean circuit $\mathbf{F}(\mathbf{x}, \mathbf{y})$with inputs $\mathbf{x}$, $\mathbf{y}$, output $\mathbf{z}$, that performs multiplication through iterative addition by executing the following initialization branch and feedback branches:

initial begin	always @(z or x or d) begin
$\mathbf{z = x}$;	if $(\mathbf{d})\mathbf{z = z + x}$
$\mathbf{d = y}$;	if $(\mathbf{d})\mathbf{d = d - 1}$
end	end.

Breaking the feedback loops introduces variables $\mathbf{z_{feedback}}$ and $\mathbf{d_{feedback}}$ and results in computation of the set of polynomials shown in Table II. Since the feedback polynomial $z = z_{\text{feedback}} + x$ is of order one with respect to z_{feedback} and contains no xz_{feedback} term, case 2) is satisfied and the

TABLE II
POLYNOMIAL REPRESENTATION FOR THE LIBRARY ELEMENT F3 IN THE JPEG ENCODE EXAMPLE

initialize = 1	initialize = 0 and d ≠ 0	initialize = 0 and d = 0
$z = x$	$z = z_{feedback} + x$	$z = z_{feedback}$
$d = y$	$d = d_{feedback} - 1$	$d = d_{feedback}$



Domain	Polynomial
[0, 1]	F(x) = 0
[2, 3]	F(x) = 1
[4, 5]	F(x) = 2
[6, 7]	F(x) = 3
[8, 9]	F(x) = 4
...	...

Fig. 7. Subdomains generated by the function $F(x) = x \gg 1$.

order of $\mathbf{F}(\mathbf{x}, \mathbf{y})$ with respect to $\mathbf{x}$ is the greater of n_{x1} and n_{x2}, both of which are one. Since the feedback polynomial $d = d_{\text{feedback}} - 1$ is of order one with respect to d_{feedback} and contains no yd_{feedback} term, case 2) is also satisfied for this polynomial and the order of $\mathbf{F}(\mathbf{x}, \mathbf{y})$ with respect to $\mathbf{y}$ is the greater of n_{y1} and n_{y2}, which are one and zero respectively. Thus, $\mathbf{F}(\mathbf{x}, \mathbf{y})$ is of order one with respect to both inputs (i.e., $n_x = 1$ and $n_y = 1$), requiring $(n_x+1)(n_y+1) = 4$ points to be extracted from the circuit. The points $(x, y, z) = \{(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1)\}$ can be extracted, yielding the following system of equations:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \bullet \begin{bmatrix} c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

The solution to the system of equations yields the polynomial representation $F(x, y) = xy$.

VII. APPROXIMATIONS

Polynomial representations are an efficient way to encapsulate the functionality of arithmetic circuits. Furthermore, circuits that implement nonarithmetic operations can be modeled efficiently by determining subdomains over which the circuit implements functionality that has a low-order polynomial representation, as shown in Section IV. However, this representation becomes very complex when the number of subdomains is large. For example, circuits that approximate arithmetic functions frequently generate many subdomains.

Example 7.1: Consider a circuit that implements $\mathbf{F}(\mathbf{x}) = (\mathbf{x} \gg 1)$, where $\mathbf{x}$ is an m bit word, requires 2^{m-1} subdomains (Fig. 7) in its polynomial representation. Rather than represent $\mathbf{F}(\mathbf{x})$ as a list of subdomains of $\mathbf{x}$ and corresponding polynomials $F(x)$ that describe $\mathbf{F}(\mathbf{x})$ exactly over those subdomains, it is much more efficient to represent $\mathbf{F}(\mathbf{x})$ as the polynomial $x/2$ and specify the maximum error between the continuous function $x/2$ and the exact polynomial representation $F(x)$.

Given a Boolean function $\mathbf{y} = \mathbf{F}(\mathbf{x})$: $B^m \rightarrow B^k$, with corresponding integer values (x, y), an approximate polynomial representation $y_{\text{approx}} = F_{\text{approx}}(x)$ can be determined. The approximate polynomial representation is determined such that $|F(x) - F_{\text{approx}}(x)| \leq \Delta$ for all x, where Δ is a given accuracy. Approximation allows a low-order polynomial representation to be generated for a Boolean function that would otherwise have a polynomial representation of high order. Sections VII-A and VII-B derive in detail the approximate polynomial representation $F_{\text{approx}}(x)$ and the tolerance Δ within which the approximation is accurate.

A. Computing Approximations

As proven in Theorem 3.1, the order of a function is reduced by one by computing the difference $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$. The algorithms to this point have relied on the resulting fact that if the order of $\mathbf{F}(\mathbf{x})$ is n, then recursively performing this difference $n+1$ times will reduce the function to zero. Now we relax the requirement that $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$ be exactly zero. If performing this difference n times results in a function that is not zero, but is numerically close to zero, then the polynomial representation $F(x)$ of $\mathbf{F}(\mathbf{x})$ can be approximated well by a polynomial of degree n.

To translate this to approximating a Boolean function $\mathbf{F}(\mathbf{x})$ with a polynomial, again consider the function $\mathbf{y} = \mathbf{F}(\mathbf{x})$: $B^m \rightarrow B^k$. If the most significant q bits of $\mathbf{y}$ are one, then for the two's complement integer encoding of $y \in Z$, $y = \text{Encode}(\mathbf{y})$, the inequality $-2^{k-q} < y$ holds. Similarly, if the most significant q bits of $\mathbf{y} - \text{Decode}(2^{k-q})$ (performed using two complement arithmetic) are 1, then the inequality $y < 2^{k-q}$ holds. As, a result, if $\mathbf{F}^-$ is defined to be $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$ and $\mathbf{F}^+$ is defined to be $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x}) - \text{Decode}(2^{k-q})$, then the following statement holds: if the upper k bits of the bit wise or of $\mathbf{F}^-$ and $\mathbf{F}^+$ are one, then $-2^{k-q} < F(x+1) - F(x) < 2^{k-q}$. The bound on $F(x+1) - F(x)$, allows us to derive an approximation of $F(x)$

$$\begin{aligned} &\text{Let } \mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x}) = \mathbf{G}(\mathbf{x}) \\ &\text{Given } \mathbf{F}(\mathbf{0}) \\ &\Rightarrow \mathbf{F}(\mathbf{1}) = \mathbf{G}(\mathbf{0}) + \mathbf{F}(\mathbf{0}) \\ &\Rightarrow \mathbf{F}(\mathbf{2}) = \mathbf{G}(\mathbf{1}) + \mathbf{F}(\mathbf{1}) = \mathbf{G}(\mathbf{1}) + \mathbf{G}(\mathbf{0}) + \mathbf{F}(\mathbf{0}) \\ &\cdots \\ &\Rightarrow \mathbf{F}(\mathbf{x}) = \left(\sum_{i=0}^{x-1} \mathbf{G}(\mathbf{i})\right) + \mathbf{F}(\mathbf{0}). \end{aligned}$$

If $\text{Encode}(\mathbf{G}(\mathbf{i}))$ is small [e.g., $-2^{k-q} < \text{Encode}(\mathbf{G}(\mathbf{i})) < 2^{k-q}$, for suitable q], the polynomial representation of $(\sum_{i=0}^{x-1} \mathbf{G}(\mathbf{i}))$ is well approximated by the line $x \cdot (\text{Encode } \mathbf{F}(11\cdots11) - \mathbf{F}(\mathbf{0})/2^m)$.

$F(x)$ can then be approximated by

$$F_{\text{approx}}(x) \approx x \cdot (\text{Encode}(\mathbf{F}(11\cdots11) - \mathbf{F}(\mathbf{0}))/2^m) + \text{Encode}(\mathbf{F}(\mathbf{0})).$$

Example 7.2.1: Consider the 8-bit function $\mathbf{y} = \mathbf{F}(\mathbf{x})$ where $\mathbf{x} \in B^8$ and $\mathbf{y} \in B^8$

$$\begin{aligned} y_0 &= x_1; & y_4 &= x_5; \\ y_1 &= x_2; & y_5 &= x_6; \\ y_2 &= x_3; & y_6 &= x_7; \\ y_3 &= x_4; & y_7 &= 0. \end{aligned}$$

This circuit could be partitioned into 64 subdomains and represented exactly with 64 order zero polynomials (similar to Fig. 7). However, the first difference iteration reveals that the upper seven bits of $\mathbf{F}(\mathbf{x}+\mathbf{1}) + \mathbf{F}'(\mathbf{x})$ are one, yielding the bound $-1 < F(x+1) - F(x) < 1$. Therefore, $F(x)$ can be approximated by the first-order polynomial $F_{\text{approx}}(x) = x(\text{Encode}(\mathbf{F}(11\cdots11) - \mathbf{F}(\mathbf{0})))/2^8 = 0.498x$.

B. Computing Approximation Error for the Linear Approximation

In this section, we will compute a bound on the accuracy Δ of a linear approximation to the polynomial representation $F(x)$. The difference between $F(x)$ and $F_{\text{approx}}(x)$, termed $\Delta(x)$ is

$$\Delta(x) = \sum_{i=0}^{x-1} (\text{Encode}(\mathbf{G}(\mathbf{i})) - \text{Encode}(\mathbf{F}(11\cdots11) - \mathbf{F}(\mathbf{0}))/2^m).$$

Since $-2^{k-q} < \text{Encode}(\mathbf{G}(\mathbf{i})) < 2^{k-q}$, it requires only $k-q$ bits to represent $\mathbf{G}(\mathbf{i})$. Assuming $k \approx q$ for a good approximation, computation of $(\text{Encode}(\mathbf{G}(\mathbf{i})) - \text{Encode}(\mathbf{F}(11\cdots11) - \mathbf{F}(\mathbf{0}))/2^m)$ need only be performed only over a short word length ($k-q$ bits). Since the Encode operation is distributive [i.e., $\text{Encode}(A) + \text{Encode}(B) = \text{Encode}(A+B)$], the following equivalence holds:

$$\begin{aligned} &2^m(\text{Encode}(\mathbf{G}(\mathbf{i})) - \text{Encode}(\mathbf{F}(11\cdots11) - \mathbf{F}(\mathbf{0}))/2^m) \\ &\quad = \text{Encode}((\mathbf{G}(\mathbf{i}) \ll m) - (\mathbf{F}(11\cdots11) - \mathbf{F}(\mathbf{0}))). \end{aligned}$$

Defining $\delta(\mathbf{i}) = (\mathbf{G}(\mathbf{i}) \ll m) - (\mathbf{F}(11\cdots11) - \mathbf{F}(\mathbf{0}))$ yields:

$$\Delta(x) \cdot 2^m = \sum_{i=0}^{x-1} \text{Encode}(\delta(\mathbf{i})).$$

Replacing $\delta(\mathbf{i})$ by the sum of its bits $\delta_j(\mathbf{i})$ [i.e., Encode$\delta(\mathbf{i}) = \sum_{j=0}^{m+k-q-1} 2^j \cdot \delta_j(\mathbf{i})$] yields

$$\Delta(x) = \sum_{i=0}^{x-1} \left(\sum_{j=0}^{m+k-q-1} 2^{j-m} \cdot (\delta_j(\mathbf{i})) \right).$$

An upper bound on $\Delta(x)$ can then be determined from each bit $\delta_j^+(\mathbf{i})$ of the positive values of $\delta(\mathbf{i})$

$$\Delta(x) < \sum_{i=0}^{x-1} \left(\sum_{j=0}^{m+k-q-1} 2^{j-m} \cdot (\delta_j^+(\mathbf{i}) \neq 0) \right).$$

Similarly, a lower bound can be determined from each bit $\delta_j^-(\mathbf{i})$ of the negative values of $\delta(\mathbf{i})$

$$\Delta(x) > \sum_{i=0}^{x-1} \left(\sum_{j=0}^{m+k-q-1} 2^{j-m} \cdot (\delta_j^-(\mathbf{i}) \neq 0) \right).$$

Following two's complement arithmetic, if the most significant bit of $\delta(\mathbf{i})$ is zero, then $\delta(\mathbf{i})$ is positive and, if the most significant bit of $\delta(\mathbf{i})$ is one, then $\delta(\mathbf{i})$ is negative. Since the most significant bit of $\delta(\mathbf{i})$ is $\delta_{m+k-q-1}(\mathbf{i})$, the bits $\delta_j^+(\mathbf{i})$ and $\delta_j^-(\mathbf{i})$ can be determined by computing the positive and negative cofactor of $\delta_j(\mathbf{i})$ with respect to $\delta_{m+k-q-1}(\mathbf{i})$.

Computing $\Delta(x)$ for all 2^m values of x is prohibitively complex due to the size of the domain and the fact that $\Delta(x)$ is a summation of x values. To circumvent this summation and determine a bound on $\Delta(x)$, the maximum values for the following are determined:

$\Delta(x+1) - \Delta(x)$ where bit x_0 of $\mathbf{x}$ is 0
$\Delta(x+2) - \Delta(x)$ where bits x_0, x_1 of $\mathbf{x}$ are 0
...
$\Delta(x+2^{m-1}) - \Delta(x)$ where bits $x_0, x_1, \ldots, x_{m-1}$ of $\mathbf{x}$ are 0.

The maximum value of the computation $\Delta(x+2^j) - \Delta(x)$, where bits $x_0, x_1, \ldots, x_j$ of $\mathbf{x}$ are zero, yields the maximum error contributed by bit j of the input. Thus, the sum of the maximum values of each of the above equations provides the maximum error contributed by all bits, which is a bound on the error of the approximation. Thus, a bound on the accuracy of the linear approximation is

$$\begin{aligned} \Delta < &[\Delta(x+1) - \Delta(x)] + [\Delta(x+2) - \Delta(x)] + \cdots \\ &+ [\Delta(x+2^{m-1}) - \Delta(x)]. \end{aligned}$$

As shown in Example 7.2.2, values of $\Delta(x)$ can be reached by summing a subset of the above equations.

Example 7.2.2:

$$\begin{aligned} \Delta(0) &= 0 \qquad \Delta(1) = [\Delta(0+1) - \Delta(0)] \\ \Delta(2) &= [\Delta(0+2) - \Delta(0)] \\ \Delta(3) &= [\Delta(1+2) - \Delta(1)] + [\Delta(0+1) - \Delta(0)] \\ \Delta(7) &= [\Delta(6+1) - \Delta(6)] + [\Delta(4+2) - \Delta(4)] \\ &\quad + [\Delta(0+4) - \Delta(0)]. \end{aligned}$$

Example 7.2.3: For the approximation computed in Example 7.2.1, the resulting $\delta(\mathbf{i})$ is:

$$\begin{aligned} \delta_0 &= 1; & \delta_5 &= 0; \\ \delta_1 &= 0; & \delta_6 &= 0; \\ \delta_2 &= 0; & \delta_7 &= 1; \\ \delta_3 &= 0; & \delta_8 &= i_0'; \\ \delta_4 &= 0; & \delta_9 &= i_0'. \end{aligned}$$

The error contributed by $\Delta(x+1) - \Delta(x)$ when $x_0 = 0$ is $\text{Encode}(\delta(\mathbf{x}))/2^8$. This is always negative because the most significant bit of $\delta(\mathbf{x})$ is one when $x_0 = 0$: $\text{Encode}(\delta(\mathbf{x}))/2^8 = -127/2^8 = -0.5$ units. The error contributed by $\Delta(x+2) - \Delta(x)$, when $x_1 x_0 = 00$, is $\text{Encode}(\delta(\mathbf{x}+\mathbf{1}) + \delta(\mathbf{x}))$. This is always positive because the most significant bit of $\delta(\mathbf{x}+\mathbf{1}) + \delta(\mathbf{x})$ is zero when $x_0 = 0$: $\text{Encode}(\delta(\mathbf{x}+\mathbf{1}) + \delta(\mathbf{x})) = (129 - 127)/2^8 = 0.008$ units. Similarly, other differences $\Delta(x+2^i) - \Delta(x)$ contribute only positive error. Other differences $\Delta(x+2^i) - \Delta(x)$ contribute a total of 0.5 units of positive error, resulting in the error bound: $-0.5 < \Delta < 0.5$. Thus, the circuit implements the polynomial $F(x) = 0.498x$ within 0.5

units. This approximate representation is far less complex than the 64 polynomials that would be required to represent the circuit exactly.

C. Nonlinear Approximations

A function $\mathbf{F}(\mathbf{x})$ may implement a nonlinear operation [e.g., $\mathbf{F}(\mathbf{x}) = (\mathbf{x}^2 \gg 1)$] that is well approximated by a nonlinear polynomial representation [e.g., $F(x) = x^2/2$]. In this case, the first iteration of $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$ may not satisfy the condition $-2^{k-q} < F(x+1) - F(x) < 2^{k-q}$. If a suitable bound is found for the nth iteration of $\mathbf{F}(\mathbf{x}+\mathbf{1}) - \mathbf{F}(\mathbf{x})$, termed $\mathbf{G_n}(\mathbf{x})$, instead of the first iteration, then a nonlinear approximation for $F(x)$ can be computed, using Δ, from Newton's forward difference interpolating formula

$$\begin{aligned} F_{\text{approx}}(x) &= \text{Encode}(\mathbf{F}(\mathbf{0})) + \binom{x}{1} \cdot \text{Encode}(\mathbf{G_0}(\mathbf{0})) \\ &\quad + \binom{x}{2} \cdot \text{Encode}(\mathbf{G_1}(\mathbf{0})) + \cdots \\ &\quad + \binom{x}{n} \cdot \text{Encode}(\mathbf{G_n}(\mathbf{0})) + \cdots. \end{aligned}$$

VIII. Matching

Consider a circuit specification $S(x)$ that defines the functionality of a circuit. Given a library of existing components, where each component is described by a Boolean function $\mathbf{F}(\mathbf{x})$, polynomial representations provide a means for quantifying the difference between the specification $S(x)$ and a potential implementation $\mathbf{F}(\mathbf{x})$. This can be achieved by computing the polynomial $\varepsilon(x) = S(x) - F(x) + \Delta$, where $F(x)$ is the polynomial representation of $\mathbf{F}(\mathbf{x})$ within an accuracy of Δ, and using traditional numerical methods to find the maximum value of $\varepsilon(x)$. In quantifying the maximum error ε of an implementation and guaranteeing that ε is within a given tolerance, system traits such as performance, power and area can be optimized by selecting faster or smaller designs that implement less accurate arithmetic.

Example 8.1: Consider the specification for an 8-bit 3×3 sharpening filter used for processing grayscale images

$$\begin{aligned} &S(x[0,0], x[0,1], x[0,2], x[1,0], x[1,1] \\ &\quad x[1,2], x[2,0], x[2,1], x[2,2]) \\ &\quad = (-x[0,0] - x[0,1] - x[0,2] - x[1,0] + 8x[1,1] \\ &\qquad - x[1,2] - x[2,0] - x[2,1] - x[2,2])/9. \end{aligned}$$

Consider an implementation $\mathbf{F}(\mathbf{x})$ with the following approximate polynomial representation:

$$\begin{aligned} &F(x, x@1, x@2, x@3, x@4, x@5, x@6, x@7, x@8) \\ &\quad \approx (-x - x@1 - x@2 - x@3 + 8x@4 - x@5 \\ &\qquad - x@6 - x@7 - x@8)/8 \\ &\quad \Delta = 0.875. \end{aligned}$$

For $0 < x < 2^8$, $\varepsilon(x) < 29$ grayscale units. This implementation yields a sharpening filter that yields an image that is of similar quality to that specified, but likely smaller and faster than an exact implementation.

A. Transcendental Specifications

A means of approximating a specification for transcendental functions can be derived from the results of Taylor series approximation. Given a specification $S(x)$, with Taylor series $S_{\text{approx}}(x) = 1 + (dS(0)/dx)x/1! + (d^2S(0)/dx^2)x^2/2! + \cdots + (d^nS(0)/dx^n)x^n/n!$, the difference between $S_{\text{approx}}(x)$ and $S(x)$ is $\varepsilon(x) = (d^{n+1}F(c)/dx^{n+1})x^{n+1}/(n+1)!$ where $0 < c < x$. Thus, if the error in a Taylor series approximation to a function can be bounded, then the difference between an implementation that matches that approximation and the specification can be bounded.

Example 8.1.1: An implementation that is determined to be of order four and yields the polynomial representation $F(x) = 1 - x^2/4 + x^4/24$ matches the cosine function used in DCT with an error $\varepsilon < 0.0083$ over the interval [0, 1].

B. Composition

The ease with which polynomials can be composed, using traditional algebraic manipulations, can allow seemingly inappropriate implementations to be combined to fulfill a specification.

Example 8.2.1: The Boolean function $\mathbf{F}(\mathbf{x})$ with polynomial representation $F(x) = x^2$ may appear to be a completely inappropriate match for the polynomial specification of $\cos(x)$ derived in Example 8.2. However, if an adder $\mathbf{F_{sum}}(\mathbf{x}, \mathbf{y})(F_{sum}(x) = x + y)$, negation element $\mathbf{F_{neg}}(\mathbf{x})(F_{neg}(x) = -x)$, and shifter $\mathbf{F_{shift}}(\mathbf{x}, \mathbf{y})(F_{shift}(x, y) = x/2^y)$ exist in the implementation library, $\mathbf{F}(\mathbf{x})$ can be allocated and composed with the adder to approximate the $\cos(x)$

$$\begin{aligned} &\mathbf{F_{sum}}(\mathbf{1}, \mathbf{F_{sum}}(\mathbf{F_{neg}}(\mathbf{F_{shift}}(\mathbf{F}(\mathbf{x}), \mathbf{1})) \\ &\qquad \mathbf{F_{shift}}(\mathbf{F}(\mathbf{F}(\mathbf{x})), \mathbf{5}))). \end{aligned}$$

The polynomial representation that results from this composition is $F(x) = 1 - x^2/4 + x^4/32$ and matches the specification derived in Example 8.2 within 1.3%.

C. Cyclic Circuits

As discussed in Section VI, when the order of a circuit with feedback can be bounded, a polynomial representation for that circuit can be determined exactly and the matching techniques described above can be used. Given a specification with bounded order n_s and a cyclic component $\mathbf{F}(\mathbf{x})$ with unbounded order, the inequality

$$n_y^t \cdot n_{x1} \le n_s \le n_y^t \cdot n_{x1} + \sum_{i=0}^{t-1} n_y^i \cdot n_{x2}$$

can be solved for t [where n_{x1} is the order of the initialization branch $\mathbf{f_1}(\mathbf{x})$, n_{x2} is the order of feedback branch $\mathbf{f_2}(\mathbf{x}, \mathbf{F_{feedback}})$ with respect to $\mathbf{x}$, n_y is the order of the feedback branch with respect to the feedback input, and t is the number of times the feedback branch is executed]. The

solution to this inequality provides the bounds on t within which $F(x)$ can have the same order as $S(x)$ and, therefore, possibly implement $S(x)$.

If $S(x)$ has unbounded order, then $S(x)$ is implemented by $\mathbf{F}(\mathbf{x})$ if and only if the specification of the initialization branch of $S(x)$, $s_1(x) = f_1(x)$ and the specification of the feedback branch of $S(x)$, $s_2(x, S_{\text{feedback}}(x)) = f_2(x, F_{\text{feedback}}(x))$. Thus, if a function $\mathbf{F}(\mathbf{x})$ does not have a bounded order and, therefore, no polynomial representation, it can still be compared to a specification $S(x)$ by comparing the initialization and feedback polynomials of $S(x)$ and $F(x)$. An example of this is shown in Section X-B.

IX. Complexity Issues

The order computation techniques described above are of quadratic complexity with respect to the size of the BDD representation of $\mathbf{F}(\mathbf{x})$ and output word length. Solving the set of linear equations for polynomial coefficients is of cubic complexity with respect to the order of the polynomial and we assume this order is small (less than the discontinuity threshold). However, the underlying BDD data structure can be of exponential complexity for common functions. Thus, reducing the complexity of polynomial computation requires reducing the complexity of the order computation, which, in turn, requires reduction of the complexity of the BDD.

Assume a function $\mathbf{F}(\mathbf{x})$ has an BDD with 2^m intermediate nodes, where $\mathbf{x}$ is an m bit word. If $\mathbf{x}$ is partitioned into two words $(x_{m-1}x_{m-2}\cdots x_{m/2}00\cdots 0)$ and $(00\cdots 0x_{m/2-1}x_{m/2-2}\cdots x_0)$, the BDDs that describe each partition will require no more than two sets of $2^{m/2}$ intermediate nodes. Similarly, partitioning $\mathbf{x}$ into C words will result in a worst-case total node count of $T = C2^{m/C}$. Minimizing T with respect to m yields

$$\begin{aligned} dT/dC &= 2^{m/C} - (m/C)2^{m/C} \cdot \log_{10} 2 \\ &= (2^{m/C} \cdot (1 - (m/C) \cdot \log_{10} 2)) \\ &\Rightarrow C = m \cdot \log_{10} 2. \end{aligned}$$

Partitioning $\mathbf{x}$ into words of length $(1/\log_{10} 2) \approx 4$ will minimize BDD complexity. This will result in overall BDD complexity of $(m/4) \cdot 2^4 = 4m$. Furthermore, such a partitioning will guarantee that the order of the polynomial representation for a component is less 2^4. For those circuits implementing functions of order greater than 2^4, a polynomial representation will be determined through domain partitioning and approximation, as explained in Sections IV and VII. In practice, very few circuits implement functions of order greater than 2^4.

X. Applications

To illustrate the application of polynomial methods, two applications are synthesized. A JPEG Encode block is first synthesized to demonstrate order computation and discontinuity detection. An IIR filter is then mapped to an existing filter to demonstrate synthesis with synchronous library elements and approximation.

(1) DCT $\quad DCT = \sum_{i=0}^{7}\sum_{j=0}^{7} x(i,j)$

(2) Quantize $\quad Q = DCT/128 - DC_{previous}$

(3) Coefficient Code $\quad C = \log_2 Q$

(4) DC Code

C	BaseCode	Length
0	010	3
1	011	4
2	100	5
3	00	5
4	101	7
5	110	8
6	1110	10
7	11110	12
8	111110	14
9	1111110	16
10	11111110	18
11	111111110	20

Fig. 8. Arithmetic specification of the blocks for the dc path of JPEG encode [inputs: $x(i, j)$; output: dc].

```
Component 1
assign F1 = x1 + x2 + x3 + ... + x64;

Component 2
assign F2 = x1 - x2;

Component 3
always @ (x1) begin
        if (x1[11]) begin
                F3 = {9'b11111110, x1[10:0]};
        end elsif x1[10] begin
                F3 = {8'b1111110, x1[9:0]};
        end elsif x1[0] begin
                F3 = {3'b011, x1[0]};
        ...
        end else begin
                F3 = 3'b010;
end
```

Fig. 9. Verilog implementations synthesized to produce library elements F1, F2, and F3.

A. JPEG Encode Application

Generating polynomial descriptions allows a specification and implementation to be compared by computing the numerical difference between the polynomials. Consider the dc path for the JPEG encode system described in Fig. 1 and specified in more detail in Fig. 8. The inputs $x(i, j)$ describe grayscale values for an 8×8 pixel block and output dc represents the encoded dc value for that pixel block. Specifications for four system blocks are described: 1) DCT; 2) quantize; 3) coefficient coding; and 4) dc coding. Three library elements were generated by synthesizing the Verilog code shown in Fig. 9. Polynomial representations were computed from the resulting netlists.

TABLE III
POLYNOMIAL REPRESENTATION FOR THE LIBRARY ELEMENT F3 IN THE JPEG ENCODE EXAMPLE

Domain	DC Polynomial
x1 = 0	*F3(x1) = 2 + x1*
0<x1<2	*F3(x1) = 6 + x1*
1<x1<4	*F3(x1) = 16 + x1*
3<x1<8	*F3(x1) = x1*
7<x1<16	*F3(x1) = 80+ x1*
15<x1<32	*F3(x1) =192 + x1*
31<x1<64	*F3(x1) = 896 + x1*
63<x1<128	*F3(x1) = 3840 + x1*
127<x1<256	*F3(x1) = 15872 + x1*
255<x1<512	*F3(x1) = 64512 + x1*
511<x1<1024	*F3(x1) = 26e4 + x1*
1023<x1<2048	*F3(x1) = 1e6 + x1*

The first component requires that an order computation be performed for each input. The order of element $\mathbf{F1(x1, x2, \ldots, x64)}$ with respect to each input is determined to be one and, after coefficient computation, the polynomial representation is

$$F1(x1, x2, \ldots, x64) = x1 + x2 + \cdots + x64.$$

The order of element $\mathbf{F2(x1, x2)}$ block is similarly determined to be one with respect to $\mathbf{x1}$ and $\mathbf{x2}$ and the resulting polynomial representation is

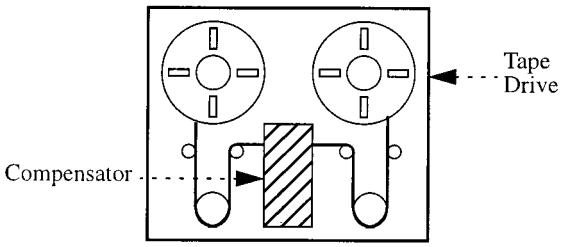
$$F2(x1, x2) = x1 - x2.$$

Order computation for element $\mathbf{F3(x1)}$ yields an order greater than the discontinuity threshold of four. As a result, the upper bits of the inputs to each block are successively set to zero and one, as described in Section IV, and the following partitions and corresponding polynomial representations are determined as shown in Table III.

Performing a numerical comparison between the specification for DCT and $F1(x1, x2, \ldots, x64)$, the specification for quantization and $F2(x1, x2)$ and the specification for coding and $F3(x1)$ reveals an exact match for each ($\varepsilon = 0$). Thus, the specification can be implemented by composing the complex components that exist in the library.

B. IIR Filter Application

Many embedded applications require digital filters to control mechanical operations. Common examples include altitude control systems for satellites, yaw dampers in airplanes, and fuel injection controllers in automobiles. We will apply polynomial methods to determine an existing filter from a library of filters suitable for reuse in a tape drive controller (Fig. 10). The velocity of the tape with the tape drive is controlled by a voltage applied to the reel motor. This voltage is a function of past velocities and, therefore, past voltages, as well as the displacement



Transfer Function:

$$H(z) = \frac{.094 - .28z^{-1} + .19z^{-2} + .19z^{-3} - .28z^{-4} + .094\,z^{-5}}{1 - 5z^{-1} + 10z^{-2} - 10z^{-3} + 5z^{-4} - z^{-5}}$$

Fig. 10. Digital filter used as a compensator for controlling the move of a tape through a tape drive.

```
input: x                        output: F
x_q = REG(x)                    F_q = REG(F)
x_qq = REG(x_q)                 F_qq = REG(F_q)
x_qqq = REG(x_qq)               F_qqq = REG(F_qq)
x_qqqq = REG(x_qqq)             F_qqqq = REG(F_qqq)
x_qqqqq = REG(x_qqqq)           F_qqqqq = REG(F_qqqq)
H1 = 160F_q - 320F _qq + 320F_qqq
H2 = - 160F_qqqq + 32F_qqqqq
H3 = x - 3x_q + 2x_qq + 2x_qqq
H4 = 3x_qqqq + x_qqqqq
H = H1 + H2 + H3 + H4
F = H>>5
```

Fig. 11. Circuit description for library element to be compared to tape controller specification.

required to position the tape properly. An existing circuit implementation within the library of filters is shown in Fig. 11, with combinational blocks already described by polynomials. The challenge is to determine if the circuit can be allocated to implement the following specification, generated from MATLAB:

$$\begin{aligned} S(x) = {} & 5S(x@1) - 10S(x@2) + 10S(x@3) - 5S(x@4) \\ & + S(x@5) + 0.093\,75x - 0.281\,25(x@1) + 0.1875(x@2) \\ & + 0.1875(x@3) - 0.281\,25(x@4) + 0.093\,75(x@5). \end{aligned}$$

The first step in generating a polynomial representation for the circuit described in Fig. 11 is to break the feedback paths. This results in $\mathbf{F_{feedback}}$ replacing $\mathbf{F}$ in the list of equations and being added to the list of inputs. The next step in generating a polynomial representation requires generation of the equivalent combinational circuit. Progressing down directed acyclic graph that represents $\mathbf{F(x, F_{feedback})}$, the first rooted subgraph represents the assignment $\mathbf{x\_q = REG(x)}$. This subgraph is duplicated, generating an additional circuit input $\mathbf{x@1}$, and the original subgraph is removed. Subsequently, the rooted subgraph ending with $\mathbf{x\_qq}$ is duplicated, generating an additional circuit input $\mathbf{x@2}$ and the original subgraph corresponding to the assignment to $\mathbf{x\_qq}$ is removed. Continuing this process, the equivalent combinational circuit is generated, resulting in a circuit with the following inputs: $\{\mathbf{x}, \mathbf{x@1}, \ldots, \mathbf{x@5}, \mathbf{F_{feedback}}, \mathbf{F_{feedback}@1}, \ldots, \mathbf{F_{feedback}@5}\}$. The nodes in the original graph that represented assignments to each of $\{\mathbf{x\_q}, \ldots, \mathbf{x\_qqqqq}, \mathbf{F\_q}, \ldots, \mathbf{F\_qqqqq}\}$

were removed as they have been replaced by $\{\mathbf{x@1}, \ldots, \mathbf{x@5}, \mathbf{F_{feedback}@1}, \ldots, \mathbf{F_{feedback}@5}\}$. The complete set of resulting equations is

$$\begin{aligned}
\mathbf{H1} &= \mathbf{160F_{feedback}@1 - 320F_{feedback}@2} \\
&\quad + \mathbf{320F_{feedback}@3} \\
\mathbf{H2} &= \mathbf{-160F_{feedback}@4 + 32F_{feedback}@5} \\
\mathbf{H3} &= \mathbf{x - 3x@1 + 2x@2 + 2x@3} \\
\mathbf{H4} &= \mathbf{3x@4 + x@5} \\
\mathbf{H} &= \mathbf{H1 + H2 + H3 + H4} \\
\mathbf{F} &= \mathbf{H} \gg 5.
\end{aligned}$$

At this point, the circuit description has no feedback paths and no registers.

Order computation with respect to each of $\{\mathbf{F_{feedback}}, \mathbf{F_{feedback}@1}, \ldots, \mathbf{F_{feedback}@5}\}$ results in an order of one for each input. However, the order of the circuit with respect to each of $\{\mathbf{x}, \mathbf{x@1}, \ldots, \mathbf{x@5}\}$ is very large, indicating that a representation of an approximation of this circuit will be more efficient. Computation of $\mathbf{F(x + 1, x@1, \ldots) - F(x, x@1, \ldots)}$ reveals that $-1 < F(x + 1, x@1, \ldots) - F(x, x@1, \ldots) < 1$. A similar result is determined for $\mathbf{x@1}, \ldots, \mathbf{x@5}$. Thus, the term that each of $\{x, x@1, \ldots, x@5\}$ contributes to the polynomial representation of the circuit can be represented by an approximation of order one, of the form $x(\text{Encode}(\mathbf{F}(11.11) - \mathbf{F(0)}))/2^n$. Following the error quantification steps outlined in Section VII, the bound on the error contributed by approximating each term of the polynomial that contains one of $\{x, x@1, \ldots, x@5\}$ is $-0.968 < \Delta < 0.968$. After performing coefficient computation, the following polynomial representation for the circuit is determined

$$\begin{aligned}
F(x) = {} & 5F_{\text{feedback}}(x@1) - 10F_{\text{feedback}}(x@2) \\
& + 10F_{\text{feedback}}(x@3) - 5F_{\text{feedback}}(x@4) \\
& + F_{\text{feedback}}(x@5) + 0.093\,749x - 0.281\,246(x@1) \\
& + 0.187\,49(x@2) + 0.187\,49(x@3) - 0.281\,246(x@4) \\
& + 0.093\,749(x@5).
\end{aligned}$$

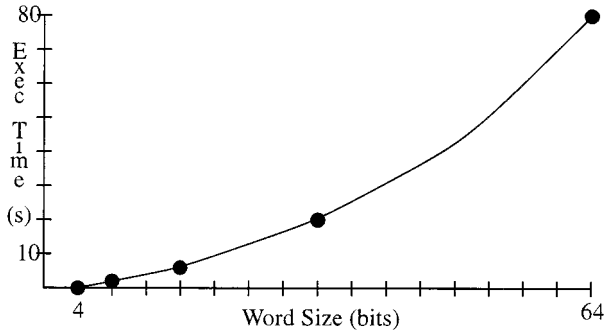
After closing the loop by setting $\mathbf{F_{feedback}} = \mathbf{F}$, the specification $S(x)$ and implementation $\mathbf{F(x)}$ can be compared by comparing their representative polynomials. The coefficients of $S(x)$ and $F(x)$ do not match exactly due to the approximation of $F(x)$, but are the same within 10^{-4}. Thus, the existing component can be allocated to implement the specification if the circuit tolerance of 10^{-4} is acceptable.

XI. Experimental Results

To quantify the performance of order computation, a combinational multiplier with input lengths ranging from 4 to 64 bits, was constructed out of combinational 4-bit multipliers, and the polynomial representation determined. Multiplier logic was synthesized from Verilog to construct the Boolean equations that implement the Synopsys DesignWare multiplier. These equations were then ported to the Cal-2.0 BDD package which was used to perform BDD operations. Experiments were performed on a 200 MHz R4400 Indy Workstation with 256 MB of memory.

Word Sizes	Logic Ops	Exec. Time
4	2003207	0.41s
8	8012236	1.34
16	32050480	4.76
32	128197824	19.31
64	512783104	79.30

(a)



(b)

Accumulator Stages	Number of Registers	Exec. Time
1	16	7.76s
2	32	25.84
3	48	79.47
4	64	177.50
5	80	326.19

(c)

Word Sizes	Exec. Time
4	0.01s
8	0.05
16	0.19
32	1.11
64	9.14
128	76.80

(d)

Circuit Function	Approx. Error
x/8	0.87
x/4	0.75
x/2	0.50
3x/4	1.75
7x/8	1.87

(e)

Fig. 12. (a) Execution time required to determine $F(x, y) = xy$ is of linear complexity with respect to x and y. (b) Graph of execution times in Fig. 10(a). (c) Execution time required for register removal on 16 bit accumulators. (d) Execution time for determining an approximation to the function $x/2$. (e) Accuracy of approximation for several 16-bit functions.

The time required to determine the order of this circuit is shown in Fig. 12(a) and, for the 64-bit multiplier, the order was computed in under 80 s. Note that by using the complexity reduction methods from Section IX, order computation was performed on successive 4-bit chunks of each input word. This yielded a maximum BDD size of 61 nodes which fit completely in the 16 KB cache.

As expected, execution time varied with the square of the size of the input word. This is due to the function $\mathbf{F(x, y)}$ being of

order one with respect to each input and having two inputs. Note that a similar computation for a function with polynomial representation $F(x) = x + K$ would have been of linear complexity with respect to the size of $\mathbf{x}$ and a more complex function such as that with polynomial representation $F(x) = x^2y^2$ would have varied with the fourth power of the size of the input word.

To quantify the performance of polynomial methods for synchronous circuits, experiments were conducted, to gauge the relationship between the execution time required to generate equivalent combinational circuits and the number of registers [Fig. 12(c)]. The circuits on which this was performed were 16-bit accumulators with between one and five register stages [i.e., $F(x) = x + x@1$, $F(x) = x + x@1 + x@2$, ..., $F(x) = x + x@1 + \cdots + x@5$]. Execution time varied quadratically with the number of registers. Note that the register removal tool is written in Perl and the execution times in Fig. 12(c) can be reduced greatly using compiled code.

Further experiments were conducted to determine the execution time of circuit approximation relative to input bit width. Polynomial approximations were computed for the circuit that implements the function $\mathbf{y} = (\mathbf{x} \gg 1)$ for input bit widths ranging from 4 to 128 bits [Fig. 12(d)]. While of high order complexity, approximations completed quickly, even for the widest datapaths. The accuracy of circuit approximation was determined for several circuits of bit width 16 [Fig. 12(e)], all of which resulted in an error of less than two units over the integer range $[0, 2^{16}-1]$. These experiments were performed with compiled code.

XII. Summary

In performing high-level synthesis with complex components, automating component matching requires a means for quickly determining whether an existing block performs the function outlined in the specification. Current methods for completing this task become prohibitively memory intensive or time consuming for circuits that implement complex functions. We have demonstrated an algorithm for performing component matching with complex library elements by constructing word-level polynomial representations for combinational and sequential circuits.

Circuit specifications can be efficiently matched to existing implementations by generating the unique minimum-order polynomial functions for the specification and the implementation and comparing those polynomials. These functions can be generated with quadratic complexity with respect to the number of input bits to each function. Discontinuities in the specification or implementation can be detected, allowing polynomial representations to be computed for intervals between discontinuities. For sequential circuits, the equivalent combinational circuit can be derived, from which a polynomial representation can be computed. Furthermore, an approximate polynomial representation can be derived for those circuits that contain many discontinuities and the error of that approximation can be quantified. Applications of these techniques were demonstrated in mapping the specification of a JPEG Encode block and an IIR filter to existing complex blocks.

Using polynomial representations, differences between a specification and implementation can be quantified, allowing tradeoffs between precision and speed. In addition, the ease with which polynomials can be composed can allow such differences to be compensated for by combining multiple existing blocks or constructing logic around a single block.

The methods presented in this paper are well suited to matching blocks that have compact arithmetic representations, such as those found in DSP, computer graphics, and ALUs. Furthermore, these methods provide a means for separating control operations, such a branches, from arithmetic operations and detecting blocks that contain many discontinuities such as controllers, based on the order of the polynomial representation.

References

[1] G. Martin, "Design methodologies for system level IP," in *Proc. Conf. Design Automation and Test Eur.*, Paris, France, Feb. 1998, pp. 286–289.

[2] C. Barna and W. Rosenstiel, "Object oriented reuse methodology for VHDL," in *Proc. Conf. Design Automation and Test Eur.*, Munich, Germany, Mar. 1999, pp. 689–693.

[3] R. Seepold, "IP and reuse," in *Proc. Conf. Design Automation and Test Eur.*, Munich, Germany, Mar. 1999, p. 725.

[4] R. Bryant, "Graph based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.

[5] R. Bryant and Y. A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Proc. 32nd ACM/IEEE Design Automation Conf.*, San Francisco, CA, June 1995, pp. 535–541.

[6] Y. A. Chen and R. Bryant, "ACV: An arithmetic circuit verifier," in *Proc. ACM/IEEE Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1996, pp. 361–365.

[7] E. M. Clarke, M. Fujita, and X. Zhao, "Hybrid decision diagrams," in *Proc. ACM/IEEE Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1995, pp. 159–163.

[8] E. M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transformations for large Boolean functions with applications to technology mapping," in *Proc. 30th ACM/IEEE Design Automation Conf.*, Dallas, TX, June 1993, pp. 54–60.

[9] Y. A. Chen and R. Bryant, "*PHDD: An efficient graph representation for floating point circuit verification," in *Proc. ACM/IEEE Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1997, pp. 2–7.

[10] K. Ravi, K. McMillan, T. Shiple, and F. Somenzi, "Approximation and decomposition of binary decision diagrams," in *Proc. 35th ACM/IEEE Design Automation Conf.*, San Francisco, CA, June 1998, pp. 445–450.

[11] S. Minato, "Implicit manipulation of polynomials using zero-suppressed BDDs," in *Proce. Eur. Design and Test Conf.*, Paris, France, Mar. 1996, pp. 449–454.

James Smith, photograph and biography not available at the time of publication.

Giovanni De Micheli (S'79–M'82–SM'89–F'94) is a Professor of Electrical Engineering, and by courtesy, of Computer Science at Stanford University, Stanford, CA. His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on synthesis, system-level design, hardware/software codesign, and low-power design. He is the author of *Synthesis and Optimization of Digital Circuits* (New York: McGraw-Hill, 1994) and coauthor of four other books. He was also Co-Director of the NATO Advanced Study Institutes on Hardware/Software Co-Design, held in Tremezzo, Italy, 1995, and on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, 1986.

Dr. De Micheli received a Presidential Young Investigator award in 1988. He received the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS Best Paper Award and two Best Paper Awards at the Design Automation Conference, in 1983 and in 1993. He received the IEEE/CAS Golden Jubilee Medal for his outstanding contribution to the IEEE/CAS Society in 2000. He was Vice President (for publications) of the IEEE CAS Society from 1999 to 2000. He is the Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS. He was the Program Chair and General Chair of the Design Automation Conference (DAC) in 1996–1997 and 2000, respectively. He was also Program and General Chair of International Conference on Computer Design (ICCD) in 1988 and 1989, respectively.