Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 194574, 15 pages
http://dx.doi.org/10.1155/2014/194574

Research Article

Groebner Bases Based Verification Solution for SystemVerilog Concurrent Assertions

Ning Zhou,[1,2] Xinyan Gao,[3] Jinzhao Wu,[1,4] Jianchao Wei,[3] and Dakui Li[3]

[1] *School of Computer and Information Technology, Beijing Jiaotong University, Beijing 10044, China*
[2] *School of Electronic and Information Engineering, Lanzhou Jiaotong University, Lanzhou 730070, China*
[3] *G & S Labs, School of Software of Dalian University of Technology, Dalian 116620, China*
[4] *Guangxi Key Laboratory of Hybrid Computation and IC Design Analysis, Guangxi University for Nationalities, Nanning 530006, China*

Correspondence should be addressed to Jinzhao Wu; jzwu205@gmail.com

Received 13 February 2014; Accepted 7 April 2014; Published 11 June 2014

Academic Editor: Xiaoyu Song

We introduce an approach exploiting the power of polynomial ring algebra to perform SystemVerilog assertion verification over digital circuit systems. This method is based on Groebner bases theory and sequential properties checking. We define a constrained subset of SVAs so that an efficient polynomial modeling mechanism for both circuit descriptions and assertions can be applied. We present an algorithm framework based on the algebraic representations using Groebner bases for concurrent SVAs checking. Case studies show that computer algebra can provide canonical symbolic representations for both assertions and circuit designs and can act as a novel solver engine from the viewpoint of symbolic computation.

1. Introduction

SystemVerilog [1, 2] is the most important unified Hardware Description and Verification Language (HDVL) and provides a major set of extensions from the Verilog language with the added benefit of supporting object orientated constructs and assertions feature. SystemVerilog provides special language constructs and assertions, to specify and verify design behavior. An assertion is a statement that a specific condition, or sequence of conditions, in a design is true. In the industry of integrated circuits design, Assertions-based verification (ABV) using SystemVerilog Assertions (SVAs) is now changing the traditional design process. With the help of SVAs, it is easy to formally characterize the design requirements at various levels of abstraction, guide the verification task, and simplify the design of the testbench. Assertions essentially become active design comments, and one important methodology treats them exactly like active design comments.

Moreover, assertions can be attached directly to RTL, working in cycle-precise domain, or they can operate in transactional domain with the help of monitors data extractors. Then, these modular checkers minimize or even eliminate the need of model checkers, which consist of HDL modules to provide the verification.

An important benefit of assertions is the ease of specifying functional coverage. Simulation tools compute the functional coverage, as defined by the assertions, which add a greater level of assurance that the testbench invoked the desired functions.

More recently, the key EDA tool vendors have researched new verification methodologies and languages, which implement functional code coverage, assertion-based coverage. and constrained random techniques inside a complete verification environment.

In [3], a subset of SystemVerilog assertions is defined to apply induction-based bounded model checking (BMC) to this subset of SVAs within acceptable run times and moderate memory requirements. As is well known, the conventional simulation for assertion checking is the well-understood and most commonly used technique, but only feasible for very small scale systems and cannot provide exhaustive checking. While symbolic simulation proposed by Darringer [4] as early as 1979 can provide exhaustive checking by covering

many conditions with a single simulation sequence, but it could not handle large circuits due to exponential symbolic expressions. Earlier work in applications of symbolic manipulation and algebraic computation has gained significant extensions and improvements. In [5], a technique framework on Groebner bases demonstrated that computer algebraic geometry method can be used to perform symbolic model checking using an encoding of Boolean sets as the common zeros of sets of polynomials. In [6], a similar technique to framework based Wu's Method has been further extended to bit level symbolic model checking. In [7], an improved framework of multivalued model checking via Groebner bases method was proposed, which is based on a canonical polynomial representation of the multivalued logics.

In our previous work [8], we proposed a method using Groebner bases to perform SEREs assertion verification for synchronous digital circuit systems. Then we introduced a verification solution based on Wu's method towards SystemVerilog assertion checking [9]. This paper aims to verify whether a SVA property holds or not on the traces produced after several cycles running over a given synchronous sequential circuit. It is the follow-up work of [9]. Groebner Bases and Wu's method are the most important methods of computer algebra. Wu's method of characteristic set is a powerful theorem proving technique, and Groebner Bases allows many important properties of the ideal and the associated algebraic variety to be deduced easily. So Groebner Bases method has a better theoretical guide, and the algorithm based on Wu's method is more efficient.

Checking SVAs is computationally very complex in general while for practical purposes a subset is sufficient. In this work we

(1) define a constrained subset of SVAs;

(2) perform algebraization of SVA operators for the constrained subset;

(3) do translation of SVAs into polynomial set representations;

(4) provide a symbolic computation based algebraic algorithm for SVAs verification.

Our approach can handle safety properties that argue over a bounded number of time steps. Local variables in SVAs can be handled as symbolic constant without any temporal information. Nevertheless, liveness properties and infinite sequences in SVAs are excluded.

2. Preliminaries

In this section, to keep the paper self-contained, we will give the basics of SystemVerilog and algebraic symbolic computation used throughout this paper.

2.1. SystemVerilog Preliminary. SystemVerilog is an IEEE-approved (IEEE 1800-2005) [1] hardware description language. It provides superior capabilities for system architecture, design, and verification.

SystemVerilog has combined many of the best features of both VHDL and Verilog.

Therefore, on the one hand, VHDL users will recognize many of the SystemVerilog constructs, such as enumerated types, records, and multidimensional arrays. On the other hand, Verilog users can reuse existing designs; SystemVerilog is a superset of Verilog so no modification of existing Verilog code is required.

Generally, the SystemVerilog language provides three important benefits over Verilog.

(1) *Explicit design intent*—SystemVerilog introduces several constructs that allow designers to explicitly state what type of logic should be generated.

(2) *Conciseness of expressions*—SystemVerilog includes commands that allow the users to specify design behavior more concisely than previously possible.

(3) *High level design abstraction*—The SystemVerilog interface construct facilitates intermodule communication.

Especially, SystemVerilog provides special language constructs and assertions, to verify design behavior. An assertion is a statement that a specific condition, or sequence of conditions, in a design is true. If the condition or sequence is not true, the assertion statement will generate an error message.

Additionally, one important capability in SystemVerilog is the ability to define assertions outside of Verilog modules and then bind them to a specific module or module instance. This feature allows test engineers to add assertions to existing Verilog models, without having to change the model in any way. One of the goals of SystemVerilog assertions is to provide a common semantic meaning for assertions so that they can be used to drive various design and verification tools.

In SystemVerilog, there are two types of assertions.

(1) Immediate Assertions. Immediate assertions follow simulation event semantics for their execution and are executed like a statement in a procedural block. Immediate assertions are primarily intended to be used with simulation and evaluate using simulation event-based semantics.

(2) Concurrent Assertions. Concurrent assertions are based on clock semantics and use sampled values of variables. All timing glitches (real or artificial due to delay modeling and transient behavior within the simulator) are abstracted away. Concurrent assertions can be used in always block or initial block as a statement, or a module as a concurrent block, or an interface block as a concurrent block, or a program block as a concurrent block.

An example of a property using sequence and formal argument is shown below.

property *test* [(*req*, *c*, *ack*)];

@(*posedge clk*)
req |− > *c* ##1 *ack*;

endproperty [: *test*].

This property states that signal *req* and then signal *c* become high and signal *ack* will be high in the next cycle.

As illustrated in this example, a concurrent assertion property in SystemVerilog will never be evaluated by itself except when it is invoked by a verification statement. Therefore, the statement: **assert property** $test(a, b, c)$ will cause the checker to perform assertion checking.

Basically, the verification statement in SVA has three forms described as follows:

(i) **assert** to specify the property as a checker to ensure that the property holds for the design;

(ii) **assume** to specify the property as an assumption for the environment. Simulators check that the property holds, while formal tools use the information to generate input stimulus. The purpose of the assume statement is to allow properties to be considered as assumptions for formal analysis as well as for dynamic simulation tools;

(iii) **cover** to monitor the property evaluation for coverage.

When a property is assumed, the tools constrain the environment so that the property holds. In simulation, asserted and assumed properties are continuously verified to ensure that the design or the testbench never violate them.

In some tools, the assumptions on the environment can be used as sequential constraints on the DUT (device under test) inputs in constrained-random simulation.

These proofs are usually subject to other properties that describe the assumed behavior of the environment using assume property statements.

2.2. Groebner Bases Preliminary. Firstly, we will recall some of the key notions of Groebner bases theory and symbolic computation. More detailed and elementary introduction to this subject can be available in books, such as those by Little et al. [10] or those by Becker and Weispfenning [11].

We begin by listing some general facts and establishing notations.

Let k be an algebraically closed field, and let $k[x_1, \ldots, x_n]$ be the polynomial ring in variables $x_1, x_2, \ldots, x_n$ with coefficient in k, under addition and multiplication of polynomial. The basic structure of polynomial rings is given in terms of subsets called ideals which is closed under addition and closed under multiplication by any element of the ring.

Here, let $I \subseteq k[x_1, \ldots, x_n]$ be an ideal. As we all know, the following theorem holds.

Theorem 1 (Hilbert basis theorem). *Every ideal* $I \subset k[x_1, \ldots, x_n]$ *has a finite generating set. That is,* $I = \langle g_1, \ldots, g_t \rangle$ *for some* $g_1, \ldots, g_t \in I$.

Then, by the Hilbert basis theorem, there exist finitely many polynomials $f_1, \ldots, f_m$ such that $I = \langle f_1, \ldots, f_m \rangle$. A polynomial $f \subseteq k[x_1, \ldots, x_n]$ defines a map $f : k^n \to k$ via evaluation $(a_1, \ldots, a_n) \mapsto f(a_1, \ldots, a_n)$.

The set $V(I) := a \in k^n \mid \forall f \in I : f(a) = 0 \subseteq k^n$ is called the variety associated with I.

If $V_1 = V(I_1)$ and $V_2 = V(I_2)$ are the varieties defined by ideals I_1 and I_2, then we have $V_1 \cap V_2 = V(\langle I_1, I_2 \rangle)$ and $V_1 \cup V_2 = V(I_1 \times I_2)$, where $I_1 \times I_2 = \langle f_1 f_2 \mid f_1 \in I_1, f_2 \in I_2 \rangle$. If $I_1 = \langle f_1, \ldots, f_r \rangle$ and $I_2 = \langle h_1, \ldots, h_s \rangle$, then $I_1 \times I_2 = \langle f_i \times g_j \mid 1 \le i \le r,\ 1 \le j \le s \rangle$.

Any set of points in k^n can be regarded as the variety of some ideal. Note that there will be more than one ideal defining a given variety. For example, the ideals $\langle x_0 \rangle$ and $\langle x_0, x_1 x_0 - 1 \rangle$ both define the variety $V(x_0)$.

In order to perform verification, we need to be able to determine when two ideals represent the same set of points. That is to say, we need a canonical representation for any ideal. Groebner bases can be used for this purpose.

An essential ingredient for defining Groebner bases is a monomial ordering on a polynomial ring $k[x_1, \ldots, x_n]$, which allows us to pick out a leading term for any polynomial.

Definition 2 (monomial ordering). A monomial ordering on $k[x_1, \ldots, x_n]$ is any relation $\prec$ on $Z^n_{\ge 0}$, or equivalently, any relation on the set of monomials $x^\alpha, \alpha \in Z^n_{\ge 0}$, satisfying

(i) $\prec$ is a total (or linear) ordering on $Z^n_{\ge 0}$;

(ii) $\prec$ is a well-ordering. This means that every nonempty subset of $Z^n_{\ge 0}$ has a smallest element under $\prec$;

(iii) for all $\gamma \in Z^n_{\ge 0}$, $\alpha \prec \beta \Rightarrow \alpha + \gamma \prec \beta + \gamma$.

Examples of monomial ordering include lexicographic order, graded lexicographic order, and graded reverse lexicographic order.

Definition 3 (lexicographic order). Let $\alpha = (\alpha_1, \ldots, \alpha_n)$ and $\beta = (\beta_1, \ldots, \beta_n) \in Z^n_{\ge 0}$. We say $\alpha \prec_{\text{lex}} \beta$ if, in the vector difference $\alpha - \beta \in Z^n$, the leftmost nonzero entry is positive. We will write $x^\alpha \prec_{\text{lex}} x^\beta$ if $\alpha \prec_{\text{lex}} \beta$.

Definition 4 (Groebner basis). Fix a monomial order. A finite subset $G = \{g_1, \ldots, g_t\}$ of an ideal I is said to be a Groebner basis (or standard basis) if $\langle LT(g_1), \ldots, LT(g_t) \rangle = \langle LT(I) \rangle$.

Equivalently, but more informally, a set $\{g_1, \ldots, g_t\} \subset I$ is a Groebner basis of I if and only if the leading term of any element of I is divisible by one of the $LT(g_i)$.

In [12], Buchberger provided an algorithm for constructing a Groebner basis for a given ideal. This algorithm can also be used to determine whether a polynomial belongs to a given ideal.

A *reduced Groebner basis* G is a Groebner basis where the leading coefficients of polynomials in G are all 1 and no monomial of an element of G lies in the ideal generated by the leading terms of other elements of $G : \forall g \in G$ and no monomial of g is in $\langle LT(G - \{g\}) \rangle$.

The important result is that, for a fixed monomial ordering, any nonzero ideal has a unique reduced Groebner basis. The algorithm for finding a Groebner basis can easily be extended to output its reduced Groebner basis. Thus we will have a canonical symbolic representation for any ideal.

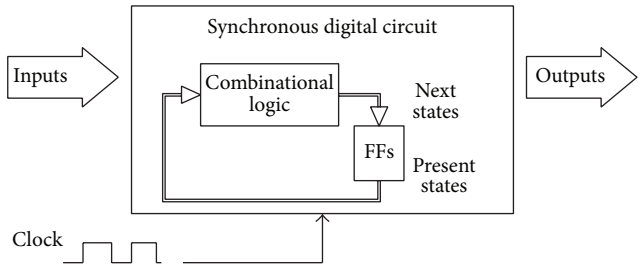

Figure 1: Synchronous digital circuits model.

Theorem 5 (the elimination theorem). *Let $I \subset k[x_1, \ldots, x_n]$ be an ideal and let G be a Groebner basis of I with respect to lexicographic order where $x_1 \succ x_2 \succ \cdots \succ x_n$. Then, for every $0 \leq l \leq n$, the set*

$$G_l = G \cap k\left[x_{l+1}, \ldots, x_n\right] \tag{1}$$

is a Groebner basis of the lth elimination ideal I_l.

Theorem 6. *Let G be a Groebner basis for an ideal $I \subset k[x_1, \ldots, x_n]$ and let $f \in k[x_1, \ldots, x_n]$. Then $f \in I$ if and only if the remainder on division of f by G is zero, denoted by $remd(f, G) = 0$.*

The property given in Theorem 6 can also be taken as the definition of a Groebner basis. Then we will get an efficient algorithm for solving the ideal membership problem. Assuming that we know a Groebner basis G for the ideal in question, we only need to compute the remainder with respect to G to determine whether $f \in I$.

In this paper, we will then use the ideal or any basis for the ideal as an efficient way of algebraization of system to be verified.

3. System Modeling with Polynomial

3.1. Circuit Representation Model. In this section, we will sketch the underlying digital system model for simulation used in our work.

Most modern circuit design is carried out within the synchronous model, which simplifies reasoning about behavior at the cost of strict requirements on clocking.

As shown in Figure 1, a classical synchronous design is comprised of combinatorial logic and blocks of registers with a global clock. In a synchronous digital system, the clock signal is often regarded as simple control signal and used to define a time reference for the movement of data within that system. Combinational logic performs all the logical functions in the circuit and it typically consists of logic gates. Registers usually synchronize the circuit's operation to the edges of the clock signal, and are the only elements which have memory properties.

The basic circuit model we used can be abstracted as the following model.

Definition 7 (synchronous circuit model). A synchronous circuit model structure is a tuple:

$$\mathbf{C} = \{\mathbf{clk}, \mathbf{L}, \mathbf{A}, \mathbf{Mux}, \mathbf{FFs}, \mathbf{I}, \mathbf{O}\}, \text{ where}$$

(i) **clk** is a global synchronous clock signal,
(ii) **L** is a set of logical operation units,
(iii) **A** is a set of arithmetic operation units,
(iv) **Mux** is a set of multiplex units,
(v) **FFs** is a set of sequential units,
(vi) **I** is a set of primary input signals,
(vii) **O** is a set of primary output signals.

In the following, we will discuss polynomial representation of a given circuit model. Firstly, let us retrospect the classical circuit representation model.

Traditionally, Binary decision diagrams (BDD) [13], the first generation decision diagrams technique, designed by Akers in 1978, acts as an efficient digital circuit representation (or Boolean functions). Recently, the next generation of decision diagrams, that is, word-level decision diagrams (WLDD) [14], has considerably widened its expressiveness for datapath operations due to processing of data in word-level format.

Generally, ROBDD or WLDD is mapped into hardware description languages according to the following scheme:

$$Function\ (circuit) \Longleftrightarrow Model\ in\ ROBDD \mid WLDD\ form. \tag{2}$$

Though the dimension of a decision diagram is exponentially bounded by the number of variables, decision diagrams based canonical data structure is very useful in many well-known verification methods.

From the viewpoint of abstract symbolic computation, these decision diagrams based on representation for circuit systems are not suitable any more. Thus, we will adopt an alternative representation form based on polynomial sets instead of decision diagrams according to the following scheme:

$$Function\ (circuit) \Longleftrightarrow Zero\ Set\ \left(Polynomial\ Set\right). \tag{3}$$

As is well known, given a monomial order, there is precisely one polynomial representation of a function.

For convenience, we introduce the following symbols for algebraic representations.

(1) For any symbolic (circuit, unit, signal, sequence, property, etc.) f, its algebraic representation form is denoted by $[\![f]\!]$.

(2) If a running cycle t is given, its algebraic representation form can be denoted by $[\![f]\!]_{[t]}$.

(3) Furthermore, if a time range $[m \cdots n]$ is specified, its algebraic representation form can then be denoted by $[\![f]\!]_{[t]}^{k}$ or $[\![f]\!]_{[t]}^{[m \cdots n]}$.

Here, t denotes the current time and $k = (n - m)$ denotes time steps.

Table 1: Polynomial model for arithmetic operation.

Arithmetic operation	Polynomial representation
$y = a + b$	$[\![+]\!] = (y - a - b)$
$y = a - b$	$[\![-]\!] = (y - a + b)$
$y = a * b$	$[\![*]\!] = (y - a * b)$
$y = a/b$	$[\![/]\!] = (y * b - a)$

Table 2: Polynomial model for logic operation.

Arithmetic operation	Polynomial representation
$y = \text{NOT } x$	$[\![\text{NOT}]\!] = (1 - x - y)$
$y = x_1 \text{ AND } x_2$	$[\![\text{AND}]\!] = (x_1 * x_2 - y)$
$y = x_1 \text{ OR } x_2$	$[\![\text{OR}]\!] = (x_1 + x_2 - x_1 * x_2 - y)$

Cycle-based symbolic simulation will be performed on the system model for verification. Intuitively, cycle-based symbolic simulation is a hybrid approach in the sense that the values that are propagated through the network can be either symbolic expressions or constant values. It assumes that there exists one unified clock signal in the circuit and all inputs of the systems remain unchanged while evaluating their values in each simulation cycle. The results of simulation report only the final values of the output signals or states in the current simulation cycle.

The detailed simulation process can be described as follows. Firstly, cycle-based symbolic simulation is initialized by setting the state of the circuit to the initial vector. Each of the primary input signals will be assigned a distinct symbolic or a constant value. Then, at the end of a simulation step, the expressions representing the next-state functions generally undergo a parametric transformation based optimization. After transformation, the newly generated functions are used as present state for the next state of simulation.

3.2. Arithmetic and Logic Unit Modeling. In this paper, we only focus on arithmetic unit for calculating fixed-point operations. For any arithmetic unit, integer arithmetic operations (addition, subtraction, multiplication, and division) can be constructed by the polynomials in Table 1.

The basic logic operations, like "**AND**", "**OR**", and "**NOT**" can be modeled by the following forms. Their corresponding polynomial representations [15] are specified as in Table 2.

Furthermore, we can extend the above rule to other common logic operators. For example,

$$y = x_1 \oplus x_2 \,(\text{or } y = x_1 \,\mathbf{XOR}\, x_2) \Longrightarrow [\![\oplus]\!] = \left(y - \left(x_1 + x_2 - x_1 * x_2\right) * \left(1 - x_1 * x_2\right)\right). \tag{4}$$

For all bit level variables x_i $(0 \le i \le n)$, a limitation $\langle \{x_i * x_i - x_i\} \rangle$ should be added.

3.3. Branch Unit Modeling. Basically, multiway branch is an important control structure in digital system. It provides a set of condition bits, bi $(0 \le i \le B)$, a set of target identifiers, $(0, \ldots, T-1)$, and a mapping from condition bit values to target identifiers. This mapping takes the form of a condition tree.

For any binary signal x, its value should be limited to $\{1, 0\}$ by adding $\{x * x - x\}$

$$y = Mux\left(x_0, x_1, \ldots, x_n, s\right), \quad i = s \longrightarrow y = x_i, \quad (0 \le i \le n) \Longrightarrow [\![Mux]\!] = \left\langle \left\{ y - \sum_{i=1}^{n-1} \left(\prod_{j \in \{0,1,\ldots,n-1\} \setminus \{i\}} \left(\frac{(s-j)}{(i-j)} \right) \right) * x_i \right\} \right\rangle, \tag{5}$$

with $\prod_{i=0}^{n-1}(s-i) = 0$.

3.4. Sequential Unit Modeling. Each flip-flop (FF) in the circuit can be modeled as a multiplexer. We have the following proposition to state this model.

Proposition 8. *For a D flip-flop (D' is the next state), with an enable signal c, its equivalent combinational formal is $y' = Mux(D, D', s) : i = s \rightarrow y' = x_i$, $(0 \le i < 2, x_0 = D, x_1 = D')$, whose polynomial algebraic model can be described as* $[\![FFs]\!] =$

$$\left\langle \left(y' - D\right) * \left(c' - 1\right), \left(y' - D'\right) * c, \left(y' - D\right) * \left(y' - D'\right) \right\rangle \tag{6}$$

or

$$\left\langle y' - D * \left(c' - 1\right) - D' * c' \right\rangle. \tag{7}$$

Proof. Let D be the current state and let y' denote the next state of the flip-flop. When the signal c' value is 0, y' has the same value as D so that the FF maintains its present state; when the signal c' value is 1, y' takes a new value from the D' input (where, D' denotes the new value next state of the FF). Therefore, we have the 2-value multiway branch model and its polynomial set representation for FF. □

Proposition 9. *Let D be an FF model, (D' is the next state), without enable signal, then its equivalent combinational formal polynomial algebraic model can be described as:* $(y' - D)$.

Proof. Straightforward. □

3.5. Sequential Unrolling. Generally, for a sequential circuit **C**, one time frame of a sequential circuit is viewed as a combinational circuit in which each flip-flop will be converted into two corresponding signals: a pseudo primary input (PPI) and a pseudo primary output (PPO).

Symbolical simulation of a sequential circuit for n cycles can be regarded as unrolling the circuit n times. The unrolled circuit is still a pure combinational circuit, and the ith copy of the circuit represents the circuit at cycle i. Thus, the unrolled circuit contains all the symbolic results from the n cycles.

3.6. Indexed Polynomial Set Representation. To illustrate the sequential modeling for a given cycle number clearly, we define an *indexed polynomial set representation* for the ith cycle.

Let $xi_{[l]}$ $(0 \le i \le r)$ denote the input signals for the lth clock, $mi_{[l]}$ $(0 \le i \le s)$ the intermediate signals, and $yi_{[l]}$ $(0 \le i \le t)$ the output signals. We then have the following time frame expansion model for the sequential circuit:

$$FM = \left\{ \bigcup_{i=0}^{n} FM\,[i] \right\}, \tag{8}$$

where $FM[i] = \mathbf{C}(x1_{[i]}, \ldots, m1_{[i]}, \ldots, m1_{[i]}, \ldots, x1_{[i+1]}, \ldots, m1_{[i+1]}, \ldots, y1_{[i+1]}, \ldots)$ denote the ith time frame model.

Time frame expansion is achieved by connecting the PPIs (e.g., $x1_{[i+1]}$ from $FM[i+1]$) of the time frame to the corresponding PPOs ($x1_{[i+1]}$ from $FM[i]$) of the previous time frame.

4. Sequence Depth Calculation

In this subsection, we will discuss the important feature of SVA, time range, and its signal constraint unrolled model.

In SVA, for each sequence the earliest time step for the evaluation and the latest time step should be determined firstly. The sequence is then unrolled based on above information. Finally, the unrolled sequence will be performed using algebraization process.

In [3], a time range calculating algorithm is provided. Here, we will introduce some related definition and special handling for our purpose.

The following is the syntax definition for time range.

4.1. Time Range

Definition 10 (time range syntax). The syntax of "time range" can be described as follows.

cycle_delay_const_range_expression ::=

constant_expression : *constant_expression*

| *constant_expression* : $.

Note that *constant_expression* is computed at compile time and must result in an integer value and can only be 0 or greater.

In this paper, we only focus on constant time range case. Thus, its form can be simplified as:

(1) $a\#\#[m:n]b$ $(m, n \in \mathbf{N}$ and $n \ge m \ge 0)$

(2) $S_1\#\#[m:n]S_2$ $(m, n \in \mathbf{N}$ and $n \ge m \ge 0)$.

Here, a, b are signals and S_1, S_2 are sequences.

Assume the starting time is cycle t, then we have: the sequence (1) will start $(n - m + 1)$ sequences of evaluation which are

$a \#\# mb$,

$a \#\# (m+1)b$,

$\ldots$,

$a\#\#(n-m+1)b$, respectively.

Their corresponding algebraic forms are

$[\![a\#\#mb]\!]_{[t]} = [\![a_t \wedge b_{t+1}]\!]$,

$[\![a\#\#(m+1)b]\!]_{[t]} = [\![a_t \wedge b_{t+2}]\!]$,

$\ldots$,

$[\![a\#\#(n-m+1)b]\!]_{[t]} = [\![a_t \wedge b_{t+n-m+1}]\!]$.

Then we have the equivalent form of above representation set as

$$[\![(a_t \wedge b_{t+m}) \vee \cdots \vee (a_t \wedge b_{t+n})]\!]. \tag{9}$$

4.2. Sequential Depth Calculation. The time range of a sequential is a time interval during which an operation or a terminal of a sequence has to be considered and is denoted by a closed bounded set of positive integers:

$$T = [l \cdots h] = \{x \mid l \le x \le h\} \quad (\text{here}, x, l, h \in \mathbf{N}). \tag{10}$$

Furthermore, the maximum of two intervals T_1 and T_2 is defined by $\max(T_1, T_2) = [\max(l_1, l_2) \cdots \max(h_1, h_2)]$.

In the same manner, the sum of two time ranges of T_1 and T_2 is defined as

$$T_1 + T_2 = [(l_1 + l_2) \cdots (h_1 + h_2)]. \tag{11}$$

Definition 11 (maximum sequential depth). The maximum sequential depth of a SVA expression F or a sequence, written as $dep(F)$, is defined recursively.

(i) $dep(a) = [1 \cdots 1]$, if a is a signal;

(ii) $dep(\neg a) = [1 \cdots 1]$, if a is a signal;

(iii) $dep(a\#\#b) = dep(a) + dep(b)$, if F_1, F_2 are sequences of SVA;

(iv) $dep(F_1\#\#[m:n]F_2) = dep(F_1) + dep(F_2) + [m \cdots n]$, if F_1, F_2 are sequences of SVA;

(v) $dep(F_1$ **and** $F_2) = \max(dep(F_1), dep(F_2))$, if F_1, F_2 are sequences of SVA;

(vi) $dep(F_1$ **or** $F_2) = \max(dep(F_1), dep(F_2))$, if F_1, F_2 are sequences of SVA;

(vii) $dep(F_1$ **intersect** $F_2) = dep(F_1) + dep(F_2) - 1$, if F_1, F_2 are sequences of SVA;

(viii) $dep(F[n]) = n * dep(F)$, if F is a sequence of SVA.

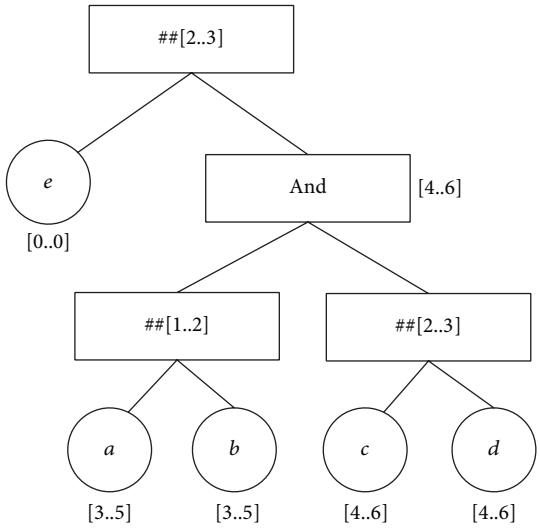

Figure 2: Sequence depth parsing tree.

For example, the following sequence is used to illustrate how to calculate the sequence depth.

sequence

$$e\,\#\#[2\cdots 3]\,((a\#\#[1\cdots 2]b)\ \textbf{and}\ (c\,\#\#[1\cdots 2]\,d))$$

endsequence.

Figure 2 shows the binary parsing tree for calculating sequence depth of this sequence. The box denotes operator and cycle denotes signal node in the syntax tree.

Firstly, consider the top layer of the parsing tree and the sequence $\#\#[2\cdots 3]$. Here, because the first operand e is a terminal of the sequence, thus this operand becomes relevant at time step 0 only ($T_{1st} = [0\cdots 0]$). Since the interval has to be considered, the second operand (the subtree of "and") matches 2 or 3 time steps later. Therefore, calculating is recursively performed with $T = [0\cdots 0] + [2\cdots 3] = [2\cdots 3]$ for the second operand.

Similarly, we can have the sequence depth of the whole sequence and all its subsequences as denoted in the figure.

5. SVA to Polynomial Set Translation

As mentioned previously, SystemVerilog assertions are an integral component of SystemVerilog and provide two kinds of assertions: immediate assertions and concurrent assertions.

In this section, we only discuss concurrent assertions and their temporal layer representation model.

Concurrent assertions express functional design intent and can be used to express assumed input behavior, expected output behavior, and forbidden behavior. That is, assertions define properties that the design must meet. Many properties can be expressed strictly from variables available in the design, while properties are often constructed out of sequential behaviors.

Thus, we will firstly discuss the basic algebraization process for the sequential behavior model.

5.1. Algebraization Process. The properties written in SVA will be unrolled and checked against the design for bounded time steps in our method. Note that only a constrained subset of SVA can be supported by our method (unspecified upper bound time range and first-match operator are excluded).

Firstly, we translate the properties described by the constrained subset of SVA into flat sequences according to the semantics of each supported operator.

As mentioned in [16], the total set of SVA is divided into 4 subgroups, namely, simple sequence expression (SSE), interval sequence expression (ISE), complex sequence expression (CSE), and unbounded sequence expression (USE). Here, in our method, these groups can only be partly supported.

Therefore, we define the following sequence expressions by adding further conditions.

(i) Constrained simple sequence expression (CSSE) is formed by Boolean expression, **repeat** operator, and **cycle delay** operator.

(ii) Constrained interval sequence expression (CISE) is a super set of CSSE formed by extra time range operators.

(iii) Constrained complex sequence expression is a super set of CSSE and CISE containing operators **or** and **intersection**.

Secondly, the unrolled flat sequences will be added to temporal constraints to form proportional formulas with logical connectives ($\vee$, $\wedge$, and $\neg$).

Finally, the resulted proportional formulas will be translated into equivalent polynomial set.

Then, the verification problem is reduced to proving zero set inclusion relationship which can be resolved by Groebner bases approaches.

5.2. Boolean Layer Modeling. The Boolean layer of SVA forms an underlying basis for the whole assertion architecture which consists of Boolean expressions that hold or do not hold at a given cycle.

In this paper, we distinguish between signal logic values and truth logic values. That is, for a truth logic statement about a given property, its truth can be evaluated to *true* or *false*. But for a signal, when primary inputs are symbolic values, its signal logic value may not be evaluated as *high* or *low*.

Therefore, we have the following definition for signal logic.

Definition 12 (signal logic). In digital circuit systems, signal logic (**SL**, for short) is defined as:

(i) if a signal x is active-high (H, for short), then its signal value is defined as 1;

(ii) if a signal x is active-low (L, for short), then its signal value is defined as 0.

Definition 13 (symbolic constant). *A symbolic constant* is a rigid Boolean variable that forever holds the same Boolean value. The notion of symbolic constant was firstly introduced in STE [17] for two purposes:

(1) to encode an arbitrary Boolean constraints among a set of circuit nodes in a parametric form;

(2) to encode all possible scalar values for a set of nodes.

Assume H denotes a symbolic constant for signal logic and $\overline{H}$ denotes its negative form, if H denotes *high* then $\overline{H}$ will be *low*.

Consider (req==$\overline{H}$) and (ack==H) as an example. According to our definitions, *req* and *ack* are signals belonging to signal logic, while both (req==$\overline{H}$) and (ack==H) are of truth logic.

For example, assertion ($a[15:0] == b[15:0]$) is also a valid Boolean expression stating that the 16-bit vectors $a[15:0]$ and $b[15:0]$ are equal.

In SVA, the following are valid Boolean expressions:

(i) $arrayA == arrayB$

(ii) $arrayA! = arrayB$

(iii) $arrayA[i] >= array\ B[j]$

(iv) $arrayB[i][j+:2] == arrayA[k][m-:2]$

(v) $(arrayA[i]\&(arrayB[j])) == 0$.

Since the state of a signal variable can be viewed as a zero of a set of polynomials. We have the following.

(1) For any signal x holds at a given time step i, thus, the state of $x == 1$ (x is active-high at cycle i) can be represented by polynomial $\{x_{[i]} - 1\}$.

(2) Alternatively, the state of $x == 0$ (x is active-low at cycle i) can be represented by polynomial $\{x_{[i]}\}$.

(3) Symbolically, the state of $x == H$ (x is active-high H at the ith cycle) can be modeled as $\{x_{[i]} - H\}$.

5.3. Sequence Operator Modeling. Temporal assertions define not only the values of signals, but also the relationship between signals over time. The sequences are the building blocks of temporal assertions and can express a set of linear behavior lasting for one or more cycles. These sequences are usually used to specify and verify interface and bus protocols.

A sequence is a regular expression over the Boolean expressions that concisely specifies a set of linear sequences. The Boolean expressions must be true at those specific clock ticks for the sequence to be true over time.

SystemVerilog provides several sequence composition operators to combine individual sequences in a variety of ways that enhance code writing and readability which can construct sequence expressions from Boolean expressions.

In this paper, **throughout** operator, [1 : $] operator, and the **first match** operator are not supported by our method.

SystemVerilog defines a number of operations that can be performed on sequences. The sequence composition operators in SVA are listed as follows.

Definition 14 (sequence operator).

R ::= b//"Boolean expression" form

| $(1, v = e)$// "local variable sampling" form

| (R)// "parenthesis" form

| $(R_1\ \#\#1\ R_2)$// "concatenation" form

| $(R_1\ \#\#0\ R_2)$// "fusion" form

| $(R_1$ **or** $R_2)$// "or" form

| $(R_1$ **intersect** $R_2)$// "intersect" form

| **first_match** (R)// "first match" form

| $R[*0]$// "null repetition" form

| $R[*1:\$]$// "unbounded repetition" form

| $[*], [=], [->]$// "repetition" repeater

| **throughout**//specifying a Boolean expression must hold throughout a sequence

| **within**//specifying conditions within a sequence.

The resulted sequences constructed by operators are then used in properties for use in assertions and covers.

5.3.1. Cycle Delay Operator. In SystemVerilog, the ## construct is referred to as a cycle delay operator.

"##1" and "##0" are concatenation operators: the former is the classical regular expression concatenation; the latter is a variant with one-letter overlapping.

A *##n* followed by a number n or range specifies the n cycles delay from the current clock cycle to the beginning of the sequence that follows.

(1) Fixed-length Case

sequence *fixs;*

$a\#\#nb$

endsequence

$\Rightarrow [\![fixs]\!] = \{[\![a]\!]_t, [\![b]\!]_{t+n}\}$

(2) Time-range Case

sequence *tms*

$a\#\#[m\cdots n]b$

endsequence

$\Rightarrow [\![tms]\!] = \{[\![a]\!]_{[t]}, [\![b]\!]^m_{[t]}\} \vee \cdots \vee \{[\![a]\!]_{[t]}, [\![b]\!]^n_{[t]}\}$.

5.3.2. Intersect Operator. The two operands of intersect operator are sequences. The requirements for match of the intersect operation are as follows.

(i) Both operands must match.

(ii) The lengths of the two matches of the operand sequences must be the same.

R_1 **intersect** R_2.

R_1 starts at the same time as R_2; the intersection will match if R_1, starting at the same time as R_2, matches at the same time as R_2 matches.

Therefore, we have

$$[\![R_1 \textbf{ intersect } R_2]\!] = [\![R_1]\!]_{[t]}^{dep(R_1)} \wedge [\![R_2]\!]_{[t]}^{dep(R_2)}. \quad (12)$$

The sequence length matching intersect operator constructs a sequence like the and nonlength matching operator, except that both sequences must be completed in same cycle.

*5.3.3. **and** Operator.* R_1 **and** R_2.

This operator states that R_1 starts at the same time as R_2 and the sequence expression matches with the later of R_1 and R_2 matching. This binary operator and is used when both operands are expected to match, but the end times of the operand sequences can be different.

That is, R_1 **and** R_2 denotes both R_1 and R_2 holds for the same number cycles. Then, the matches of R_1 **and** R_2 must satisfy the following:

(i) The start point of the match of R_1 must be no earlier than the start point of the match of R_2.

(ii) The end point of the match of R_1 must be no later than the end point of the match of R_2.

The sequence nonlength matching **and** operator constructs a sequence in which two sequences both hold at the current cycle regardless of whether they are completed in the same cycle or in different cycles

$$[\![R_1 \textbf{ and } R_2]\!] \Longrightarrow \bigvee_{i=l_1}^{h_1} \bigvee_{j=l_2}^{h_2} \left(\left\{ [\![R_1]\!]_{[t]}^{i} \right\} \wedge \left\{ [\![R_2]\!]_{[t]}^{j} \right\} \right). \quad (13)$$

Here, $l_i = dep(R_i) \cdot l$, $h_i = dep(R_i) \cdot h$, and $0 < i \leq 2$.

*5.3.4. **or** Operator.* R_1 **or** R_2.

The sequence **or** operator constructs a sequence in which one of two alternative sequences hold at the current cycle. Thus, the sequence (a##1b) **or** (c##1d) states that either sequence a, b or sequence c, d would satisfy the assertion

$$[\![R_1 \textbf{ or } R_2]\!] \Longrightarrow \bigvee_{i=l_1}^{h_1} \bigvee_{j=l_2}^{h_2} \left(\left\{ [\![R_1]\!]_{[t]}^{i} \right\} \vee \left\{ [\![R_2]\!]_{[t]}^{j} \right\} \right). \quad (14)$$

Here, $l_i = dep(R_i) \cdot l$, $h_i = dep(R_i) \cdot h$, and $0 < i \leq 2$.

5.3.5. Local Variables. SystemVerilog provides a feature by which variables can be used in assertions. The user can declare variables local to a property. This feature is highly useful in pipelined designs where the consequent occurrence might be many cycles later than their corresponding antecedents.

Local variables are optional and local to properties. They can be initialized, assigned (and reassigned) a value, operated on, and compared to other expressions.

The syntax of a sequence declaration with a local variable is shown below.

sequence_declaration ::=
sequence *sequence_identifier*[([*tf_port_list*])];

assertion_variable_declaration
sequence_expr;

endsequence [: *sequence_identifier*].

The property declaration syntax with a local variable can be illustrated as follows:

property_declaration ::=
property *property_identifier*[([*tf_port_list*])];

assertion,ariable_declaration
property_spec;

endproperty [: *property_identifier*].

The variable identifier declaration syntax of a local variable can be illustrated as follows:

assertion_variable_declaration ::=
var_data_type list_of_variable_identifiers.

The dynamic creation of a variable and its assignment is achieved by using the local variable declaration in a sequence or property declaration and making an assignment in the sequence.

Thus, local variables of a sequence (or property) may be set to a value, which can be computed from a parameter or other objects (e.g., arguments, constants, and objects visible by the sequence (or property)).

For example, a property of a pipeline with a fixed latency can be specified below.

property *latency*;
int x;
(*valid_in*, $x = p\_in$)|− > ##3($p\_out == (x + 1)$);
endproperty.

This property e is evaluated as follows:

When *valid_in* is **true**, x is assigned the value of *p_in*. If 3 cycles later, *p_out* is equal to $x + 1$, then property *latency* is **true**. Otherwise, the property is **false**. When *valid_in* is **false**, property *latency* is evaluated as **true**.

For the algebraization of SVA properties with local variables, in our method these local variables will be taken as common signal variables (symbolic constant) without any sequential information.

Thus, we have the polynomial set representation for property *latency*:

$[\![latency]\!]_{[t]} = \{$

$(valid\_in_{[t]} - 1)$,
$(x - p\_in_{[t]})$,
$(p\_out_{[t+3]} - (x + 1))\}$.

5.3.6. Repetition Operators. SystemVerilog allows the user to specify repetitions when defining sequences of Boolean expressions. The repetition counts can be specified as either a range of constants or a single constant expression.

Nonconsecutive repetition specifies finitely many iterative matches of the operand Boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at or after the last iterative match of the operand, but before any later match of the operand.

The syntax of repetition operator can be illustrated as follows:

$$non\_consecutive\_rep ::= [= const\_or\_range\_expr]. \quad (15)$$

The number of iterations of a repetition can be specified by exact count.

For example, a sequence with repetition can be defined as

$$a \mid=> b\,[= 3]\,\#\#1\,c. \quad (16)$$

The sequence expects that 2 clock cycles after the valid start, signal "b" will be repeated three times.

We have $[\![R[= n]]\!] = \bigvee_{i=0}^{n}(\{[\![R]\!]_{[t]}^{i*dep(R)}\})$.

5.4. Property Operator Modeling. In general, property expressions in SVA are built using sequences, other sublevel properties, and simple Boolean expressions via property operators.

In SVA, a property that is a sequence is evaluated as true if, and only if, there is a nonempty match of the sequence. A sequence that admits an empty match is not allowed as a property.

The success of assertion-based verification methodology relies heavily on the quality of properties describing the intended behavior of the design. Since it is a fairly new methodology, verification and design engineers are often faced with a question of "how to identify good properties" for a design. It may be tempting to write properties that closely resemble the implementation.

Properties we discussed in this paper can be classified as.

(1) Design Centric. Design centric properties represent assertions added by the RTL designers to characterize white box design attributes. These properties are typically towards FSMs, local memories, clock synchronization logic, and other hardware-related designs. The design should conform to the expectations of these properties.

(2) Assumption Centric. Assumption centric properties represent assumptions about the design environment. They are used in informal verification to specify assumptions about the inputs. The assume directive can inform a verification tool to assume such a condition during the analysis.

(3) Requirement/Verification Centric. Many requirement properties are best described using a "cause-to-effect" style that specifies what should happen under a given condition. This type of properties can be specified by implication operator. For example, a handshake can be described as "if request is asserted, then acknowledge should be asserted within 5 cycles".

In general, property expressions are built using sequences, other sublevel properties, and simple Boolean expressions.

These individual elements are combined using property operators: **implication**, **NOT**, **AND**, **OR**, and so forth.

The property composition operators are listed as follows.

Definition 15 (property operator).

P ::= R/* "sequence" form */

|(P)/* "parenthesis" form */

|**not** P/* "negation" form */

|$(P_1$ **or** $P_2)$/* "or" form */

|$(P_1$ **and** $P_2)$/* "and" form */

|$(R\mid- > P)\,(R\mid => P)$/* "implication" form */

|**disable iff**(b) P/* "reset" form */.

Note that **disable if and only if** will not be supported in this paper. Property operators construct properties out of sequence expressions.

5.4.1. Implication

Operators. The SystemVerilog implication operator supports sequence implication and provides two forms of implication: overlapped using operator |− >, and nonoverlapped using operator | =>, respectively.

The syntax of implication is described as follows:

$$\begin{aligned} &sequence\_expr \mid - > property\_expr \\ &sequence\_expr \mid => property\_expr. \end{aligned} \quad (17)$$

The implication operator takes a sequence as its antecedent and a property as its consequent. Every time the sequence matches the property must hold. Note that both the running of the sequence and the property may span multiple clock cycles and that the sequence may match multiple times.

For each successful match of the antecedent sequence, the consequence sequence (right-hand operand) is separately evaluated, beginning at the end point of the matched antecedent sequence. All matches of antecedent sequence require a match of the consequence sequence.

Moreover, if the antecedent sequence (left hand operand) does not succeed, implication succeeds vacuously by returning **true**. For many protocol related assertions, it is important to specify the sequence of events (the antecedent or cause) that must occur before checking for another sequence (the consequent or effect). This is because if the antecedent does not occur, then there is no need to perform any further verification. In hardware, this occurs because the antecedent reflects an expression that, when active, triggers something to happen.

A property with a consequent sequence behaves as if the consequent has an implication first-match applied to it.

5.4.2. *NOT*

Operator. The operator **NOT** *property_expr* states that the evaluation of the property returns the opposite of the evaluation of the underlying *property_expr*.

5.4.3. *AND*

Operator. The formula "P_1 **AND** P_2" states that the property is evaluated as **true** if, and only if, both P_1 and P_2 are evaluatd as **true**.

5.4.4. *OR*

Operator. The operator "P_1 **OR** P_2" states that the property is evaluated as **true** if and only if, at least one of P_1 and P_2 is evaluated as **true**.

5.4.5. *IF-ELSE*

Operator. This operator has two valid forms which are listed as follows.

(1) **IF** (*dist*) P_1.

A property of this form is evaluated as **true** if, and only if, either *dist* is evaluated as **false** or P_1 is evaluated as **true**.

(2) **IF** (*dist*) P_1 **ELSE** P_2.

A property of this form is evaluated as **true** if, and only if, either *dist* is evaluated as **true** and P_1 is evaluated as **true** or *dist* is evaluated as **false** and P_2 is evaluated as **true**.

From previous discussion, we have the following proposition for property reasoning.

Proposition 16. *Assume that P, P_1, and P_2 are valid properties in SVA, the following rules are used to construct the corresponding verification process. Here, AssChk*$(Model, Property)$: {**ture**, **false**} *is a self-defined checking function that can determine whether a given "Property" holds or not with respect to a circuit model "Model".*

(1) P_1 **OR** $P_2 \Rightarrow AssChk(M, [\![P_1]\!]) \vee AssChk(M, [\![P_2]\!])$

(2) P_1 **AND** $P_2 \Rightarrow AssChk(M, [\![P_1]\!]) \wedge AssChk(M, [\![P_2]\!])$

(3) **NOT** $P \Rightarrow \neg AssChk(M, [\![P]\!])$

(4) **IF** (*dist*) $P_1 \Rightarrow \neg AssChk(M, dist) \vee AssChk(M, [\![P_1]\!])$

(5) **IF** (*dist*) P_1 **ELSE** $P_2 \Rightarrow (AssChk(M, dist) \wedge AssChk(M, [\![P_1]\!])) \vee (\neg AssChk(M, dist) \wedge AssChk(M, [\![P_2]\!]))$

(6) $S \mid -> P \Rightarrow$ *if* $(\neg AssChk(M, [\![S]\!]))$*return* **true** *else* $(AssChk(M, [\![S]\!]_{[t]}) \wedge AssChk(M, [\![P]\!]_{[t+dep(S)-1]}^{dep(P)}))$

(7) $S \mid=> P \Rightarrow$ *if* $(\neg AssChk(M, [\![S]\!]))$*return* **true** *else* $(AssChk(M, [\![S]\!]_{[t]}) \wedge AssChk(M, [\![P]\!]_{[t+dep(S)]}^{dep(P)}))$.

6. Verification Algorithm

In this section, we will describe how an assertion is checked using Groebner bases approach. Firstly, we will discuss a practical algorithm using Groebner bases for assertion checking.

6.1. Basic Principle. As just mentioned, our checking method is based on algebraic geometry which is the study of the geometric objects arising as the common zeros of collections of polynomials. Our aim is to find polynomials whose zeros correspond to pairs of states in which the appropriate assignments are made.

We can regard any set of points in k^n as the variety of some ideal. We can then use the ideal or any basis for the ideal as a way of encoding the set of points.

From Groebner Bases theory [10, 12] every nonzero ideal $I \subset k[x_1, \ldots, x_n]$ has a Groebner basis and the following proposition evidently holds.

Proposition 17. *Let C and S be polynomial sets of $k[x_1, \ldots, x_n]$, and $\langle GS \rangle$ a Groebner basis for $\langle S \rangle$, and then we have that*

$$\langle C \rangle \subseteq \langle S \rangle \Leftrightarrow \forall c \in C : remd(c, GS) == 0 \text{ holds.}$$

Theorem 18. *Suppose that $A([\![A]\!] = \{a_1, a_2, \ldots, a_r\})$ is a property to be verified. T is the initial preconditions of C, and M is a system model to be checked. Let $[\![T]\!]$ and $[\![M]\!]$ be the polynomial set representations for T and M, respectively, constructed by previous mentioned rules. Let $H = [\![T \cup M]\!] = \{h_1, h_2, \ldots, h_s\} \subseteq k[x_1, \ldots, x_n]$, $I = \langle H \rangle$ (where, $\langle H \rangle$ denotes the ideal generated by H) and $GB_H = gbasis(H, \prec)$, then we have*

$$\begin{aligned} &\left((1 \notin GB_H) \textbf{ and } remd(C, GB_H) == 0 \right) \\ &\Longleftrightarrow \left((1 \notin GB_H) \textbf{ and } \bigwedge_{i=0}^{r} (remd(g_i, GB_H) == 0) \right) \\ &\Longleftrightarrow (M \mid= A) \text{ holds.} \end{aligned} \tag{18}$$

Proof. (1) The polynomial set $[\![M]\!]$ of system model M describes the data-flow relationship that inputs, outputs, and functional transformation should meet. The initial condition $[\![M]\!]$ can be seen as extra constraint applied by users. There should not exist contradiction between them; that is, the polynomial equations of $[\![T \cup M]\!]$ must have a common solution.

By Hilbert's Nullstellensatz theory, if we have polynomials $[\![T \cup M]\!] = \{f_1, \ldots, f_s\} \in k[x_1, \ldots, x_n]$, we compute a reduced Groebner basis of the ideal they generate with respect to any ordering. If this basis is {1}, the polynomials have no common zero in k^n; if the basis is not {1}, they must have a common zero.

Therefore, $(1 \notin GB_H)$ should hold.

(2) Evidently, by previous proposition, it is easy to have that $remd(C, GB_H) == 0$ should hold.

Thus, we have that this theorem holds. □

Input:
(1) circuit model M;
(2) initial condition T;
(3) an assertion A;
Output: Boolean: *true* or *false*;
BEGIN
/* *Step 1*: initialize input signals via testbench */
(00) $InitSignals(\vec{T})$;
(01) $\mathscr{M} = \emptyset; \mathrm{PS}_A = \emptyset; H = \emptyset; \mathrm{PS}_T = \emptyset$;
/* *Step 2*: build polynomial model */
(02) $\mathscr{M} = [\![M]\!] = Build\mathrm{PS}(M)$;
/* *Step 3*: build polynomial set for initial condition T */
(03) $PS_T = [\![T]\!] = Build\mathrm{PS}(T)$;
/* *Step 4*: build polynomial set for consequent $\mathscr{A}$ */
(04) $PS_A = [\![A]\!] = Build\mathrm{PS}(A)$;
/* *Step 5*: calculate the $\mathrm{PS}_T \cup \mathscr{M}$ */
(05) $H = \mathrm{PS}_{\mathscr{T}} \cup \mathscr{M}$;
/* *Step 6*: calculate the Groebner base of $\langle H \rangle$ */
(06) $\mathrm{GB}_H := g\mathrm{basis}(H, \prec)$;
/* *Step 7*: determine the basis is {1} or not */
(07) if($1 \in \mathrm{GB}_H$) {
(08) return *false*; }
/* *Step 8*: check every polynomial */
/* $A = \{a_1, a_2, \ldots, a_{|A|}\}$ */
(09) $i = 1$;
(10) *while*($i \leq |A|$){
(11) if($remd([\![a_i]\!], \mathrm{GB}_H) \neq 0$) {
(12) return *false*; }
(13) i++;
(14) } /* endwhile */
(15) return *true*; /* Assertion does hold */
END;

Algorithm 1: "*AssChkPolyBuild*(M, T, A)".

6.2. Checking Algorithm. For simplicity, in this section, we only provide the key decision algorithm.

Firstly, the original system model is transformed into a normal polynomial representation and the assertion as well. Then, calculate the hypothesis set and its Groebner basis using the Buchberger algorithm [18] and their elimination ideals. Finally, examine the inclusion relationship between elimination ideals to determine whether the assertion to be checked holds or not.

From above discussion, we have the following process steps and detailed algorithm description. Algorithm 1.

For convenience, we can derive another version checking procedure named "*AssChk*($[\![M]\!], [\![A]\!]$)" which can accept polynomial representations as inputs without explicit polynomial construction process.

Further, by applying Theorem 18 and the checking algorithm "*AssChk*($[\![M]\!], [\![A]\!]$)", we can easily verify all supported properties written in SVA.

7. An Example

In this section, we will study a classical circuit to show how SVA properties are verified by polynomial representation and algebra computation method.

Consider the Johnson counter circuit in Figure 3. Johnson counters can provide individual digit outputs rather than a binary or BCD output, as shown in Table 3. Notice that each legal count may be defined by the location of the last flip-flop to change states and which way it changed state.

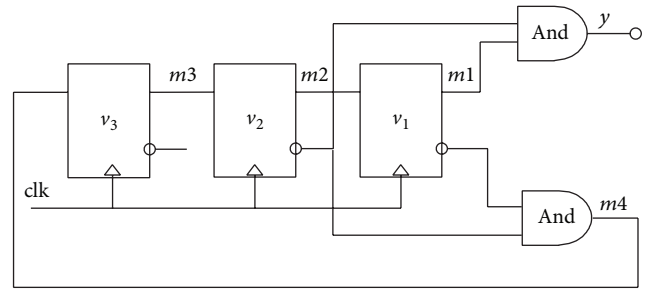

Figure 3: Johnson counter.

Table 3: Table of Johnson counter output.

Counter (clock)	$v3(m3)$	$v2(m2)$	$v1(m1)$
0	0	0	0
1	1	0	0
2	1	1	0
3	0	1	1
4	0	0	1
5	0	0	0

7.1. Algebraization of Circuit and Assertion. Johnson counter provides individual digit outputs, as shown in Table 3.

The polynomial set for Johnson counter circuit in Figure 3 can be constructed as follows:

$$\begin{aligned} Set_{\text{adder}} = \{ & f1 = \{m3' - m4\}, f2 = \{m2' - m3\}, \\ & f3 = \{m1' - m2\}, \\ & f4 = \{m4 - (1 - m1) * (1 - m2)\}, \\ & f5 = \{y - m1 * (1 - m2)\}\}, \end{aligned} \tag{19}$$

where $m1'$ denotes the next state of $m1$. For the ith cycle, we use $x1_{[i]}$ to denote variable name in current cycle.

Similarly, another Johnson counter circuit with an error is shown in Figure 4, the corresponding polynomial set can be described by

$$\begin{aligned} Set_{\text{adder}} = \{ & f1, f2, f3, \\ & f4' = \{m4 + m1 + m2 \\ & \quad - (1 - m1) * (1 - m2)\}, f5\}. \end{aligned} \tag{20}$$

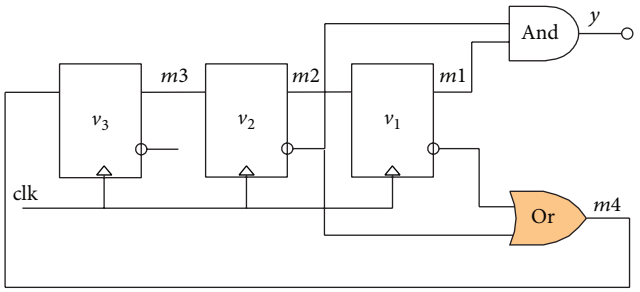

Figure 4: Johnson counter with error.

To illustrate the problem clearly, we define polynomial set representation $PM[i]$ for ith cycle as follows:

$$\begin{aligned} PM\,[i] = \{&m3_{[i+1]} - m4_{[i]}, m2_{[i+1]} - m3_{[i]}, \\ &m1_{[i+1]} - m2_{[i]}, m4_{[i]} \\ &- \left(1 - m2_{[i]}\right) * \left(1 - m1_{[i]}\right), \\ &y - m1_{[i]} * \left(1 - m2_{[i]}\right)\}. \end{aligned} \tag{21}$$

Support that the circuit will run 6 cycles for verification, therefore, we have $PM = \{\bigcup_{i=0}^{5} PM[i]\}$.

For any Boolean variable a, we will add an extra constraint: $a*a-a$. Thus, we define the corresponding constraints set as follows: $CNS[i] = \{c_{[i]} * c_{[i]} - c_{[i]}, c \in \{m1, m2, m3, m4, y\}\}$.

In the same manner, we have $CNS = \{\bigcup_{i=0}^{5} CNS[i]\}$.

The sequential property for this counter circuit can be specified by the following SystemVerilog assertions.

property *JCPA*;

($m3 = 0$ && $m2 = 0$ && $m1 = 0$)
| => ##1($m3 == 1$ && $m2 == 0$ && $m1 == 0$)
| => ##2($m3 == 1$ && $m2 == 1$ && $m1 == 0$)
| => ##3($m3 == 0$ && $m2 == 1$ && $m1 == 1$);

endproperty

property *JCP*1;

@(clk)($m1 == 0$)| => ##2 ($m1 == 0$);

endproperty

property *JCP*2;

@(clk)($m2 == 0$)| => ##2 ($m2 == 1$);

endproperty

property *JCP*3;

@(clk)($m3 == 0$)| => ##2 ($m3 == 1$);

endproperty.

Table 4: Polynomial forms.

Name	Precondition	Expected consequent
JCPA	$\{m1_{[0]}, m2_{[0]}, m3_{[0]}\}$	
	cycle 1	$\{m1_{[1]} - 1, m2_{[1]}, m3_{[1]}\}$
	cycle 2	$\{m1_{[2]} - 1, m2_{[2]} - 1, m3_{[2]}\}$
	cycle 3	$\{m1_{[3]}, m2_{[3]} - 1, m3_{[3]} - 1\}$
JCP1	$\{m1_{[0]}\}$	$m1_{[2]}$
JCP2	$\{m2_{[0]}\}$	$m2_{[2]} - 1$
JCP3	$\{m3_{[0]}\}$	$m3_{[2]} - 1$

We will afterwards demonstrate the verification process step by step.

The circuit model to be verified is as shown below:

$$SM = PM \cup CNS. \tag{22}$$

The property of this counter can be specified as the following SVA assertion:

assert property (*JCP*1)

assert property (*JCP*2).

The Algebraization form of the above properties can be modeled as in Table 4.

7.2. Experiment Using Maple. We run this example by using Maple 13. Before running, we manually translated all models into polynomials. The experiment is performed on a PC with a 2.40 GHz CPU (intel i5 M450) and 1024 MB of memory. It took about 0.04 seconds and 0.81 MB of memory when applying Groebner method.

[>*with*(*Groebner*)

[> $CM := \cdots$ /*Circuit Model*/

[> $TDEG := tdeg($

$m1_{[0]}, m2_{[0]}, m3_{[0]}, m4_{[0]},$
$m1_{[1]}, m2_{[1]}, m3_{[1]}, m4_{[1]},$
$m1_{[2]}, m2_{[2]}, m3_{[2]}, m4_{[2]},$
$m1_{[3]}, m2_{[3]}, m3_{[3]}, m4_{[3]},$
$m1_{[4]}, m2_{[4]}, m3_{[4]}, m4_{[4]},$
$y_{[0]}, y_{[1]}, y_{[2]}, y_{[3]})$

[> $CGB := Basis(CM, TDEG)$.

Thus, we can have the Groebner basis as follows:

[> $[y1_{[3]}, y1_{[2]}, y1_{[1]}, y1_{[1]}, m3_{[4]}, m2_{[4]},$

$m1_{[4]} - 1, m4_{[3]}, m3_{[3]}, m2_{[3]} - 1, m1_{[3]} - 1,$
$m4_{[2]}, m3_{[2]} - 1, m2_{[2]} - 1, m1_{[2]}, m4_{[1]} - 1,$
$m3_{[1]} - 1,$
$m2_{[1]}, m1_{[1]}, m4_{[0]} - 1, m3_{[0]}, m2_{[0]}, m1_{[0]}]$

[> $ret := NormalForm(m3_{[1]} - 1, CGB, TDEG)$

[> $ret = 0$.

Table 5: Computation result table.

Number	Polynomial	GBase	Mem	Time	Return	Result
0	$\{m1_{[1]}\}$	$1 \notin CGB$	0.80 M	0.02 S	0	Yes
	$\{m2_{[1]}\}$	$1 \notin CGB$			0	
	$\{m3_{[1]} - 1\}$	$1 \notin CGB$			0	
1	$\{m1_{[2]}\}$	$1 \notin CGB$	0.81 M	0.03 S	0	Yes
	$\{m2_{[2]} - 1\}$	$1 \notin CGB$			0	
	$\{m3_{[2]} - 1\}$	$1 \notin CGB$			0	
2	$\{m1_{[3]} - 1\}$	$1 \notin CGB$	0.81 M	0.04 S	0	Yes
	$\{m2_{[3]} - 1\}$	$1 \notin CGB$			0	
	$\{m3_{[3]}\}$	$1 \notin CGB$			0	
3	$\{m1_{[2]}\}$	$1 \notin CGB$	0.81 M	0.04 S	0	Yes
4	$\{m2_{[2]}\}$	$1 \notin CGB$	0.81 M	0.04 S	0	Yes
5	$\{m3_{[2]}\}$	$1 \notin CGB$	0.81 M	0.04 S	0	Yes

Table 6: Computation result table.

Number	Polynomial	GBase	Mem	Time	Return	Result
0	$\{m1_{[1]}\}$	$1 \notin CGB$	0.80 M	0.02 S	0	Yes
	$\{m2_{[1]}\}$	$1 \notin CGB$			0	
	$\{m3_{[1]} - 1\}$	$1 \notin CGB$			0	
1	$\{m1_{[2]}\}$	$1 \notin CGB$	0.81 M	0.03 S	0	Yes
	$\{m2_{[2]} - 1\}$	$1 \notin CGB$			0	
	$\{m3_{[2]} - 1\}$	$1 \notin CGB$			0	
2	$\{m1_{[3]} - 1\}$	$1 \notin CGB$	0.81 M	0.14 S	0	NO
	$\{m2_{[3]} - 1\}$	$1 \notin CGB$			0	
	$\{m3_{[3]}\}$	$1 \notin CGB$			−1	

As shown in maple outputs, the given circuit has been modeled as polynomial set CM (its Groebner bases is denoted by CGB) and assertion expected result as $\{m3_{[1]} - 1\}$. From the running result, we have $1 \notin CGB$ and return value of $NormalForm$ is 0 which means CGB is divided with no remainder by $\{m3_{[1]} - 1\}$.

Thus, from the previously mentioned verification principles, it is easy to conclude that the assertion $JCP1$ holds under this circuit model after 3 cycles.

More detailed experiment results for verification of circuit shown in Figure 3 are listed in Table 5.

Conversely, we check these assertions against the circuit shown in Figure 4 in a similar way. More detailed experiment results are demonstrated in Table 6.

From above table, when checking $JCPA$ assertion, the result $ret := NormalForm(m3_{[3]}, CGB, TDEG) \neq 0$ so that we can conclude the assertion does not hold. Therefore, there must exist unexpected errors in the original circuit.

This case is a fairly complete illustration of how the checking algorithm works.

8. Conclusion

In this paper, we presented a new method for SVA properties checking by using Groebner bases based symbolic algebraic approaches. To guarantee the feasibility we defined a constrained subset of SVAs, which is powerful enough for practical purposes.

We first introduce a notion of symbolic constant without any sequential information for handling local variables in SVAs inspired from STE. We then proposed a practical algebraization method for each sequence operator. For sequential circuits verification, we introduce a parameterized polynomial set modeling method based on time frame expansion.

Our approach is based on polynomial models construction for both circuit models and SVA assertions. This method is to eventually translate a simulation based verification problem into a pure algebraic zero set determination problem by a series of proposed steps, which can be performed on any general symbolic algebraic tool.

This method allows users to deal with more than one state and many input combinations every cycle. This advantage comes directly from the fact that many vectors are simulated at once using symbolic value.

In summary, in this research, by suitable restrictions of SVA assertions, we can guarantee the availability of polynomial set representation. Based on this polynomial set model, symbolic simulation can be performed to produce symbolic traces and temporal relationship constraints of signal variables as well. We then apply symbolic algebra approach to check the zeros set relation between their polynomial sets and determine whether the temporal assertion holds or not under current running cycle.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The project is supported by the National Natural Science Foundation of China under Grant no. 11371003, the Natural Science Foundation of Guangxi under Grants nos. 2011GXNSFA018154, 2012GXNSFGA060003, and 2013GXNSFAA019342, the Science and Technology Foundation of Guangxi under Grant no. 10169-1, the Scientific Research Project no. 201012MS274 from Guangxi Education Department, the Fundamental Research Funds for the Central Universities under Grants nos. DUT14QY05 and HCIC201204 of Guangxi Key Laboratory of Hybrid Computation and IC Design Analysis Open Fund, and the Baguio scholarship Project of Guangxi. The authors would like to thank their colleagues for participating in the research. They also appreciate the anonymous reviewers for their helpful comments.

References

[1] IEEE System Verilog Working Group, IEEE Standard for SystemVerilog C Unified Hardware Design, Specification, and Verification (IEEE Std 1800-2005). IEEE, 2005.

[2] "IEEE draft standard for system verilog—unified hardware design, specification, and verification language," IEEE P1800/D3, 2011.

[3] R. Wille, G. Fey, M. Messing, G. Angst, L. Linhard, and R. Drechsler, "Identifying a subset of SystemVerilog assertions for efficient bounded model checking," in *Proceedings of the 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD '08)*, pp. 542–549, September 2008.

[4] L. Darringer, "Application of program verification techniques to hardware verification," in *Proceedings of the IEEE-ACM Design Automation Conference*, pp. 375–381, 1979.

[5] G. S. Avrunin, "Symbolic model checking using algebraic geometry," in *Proceedings of the 8th International Conference on Computer Aided Verification*, vol. CAV96, pp. 26–37, Springer, London, UK, 1996.

[6] W. B. Mao and J. Z. Wu, "Application of Wus method to symbolic model checking," in *Proceedings of the 2005 international Symposium on Symbolic and Algebraic Computation (ISSAC '05)*, pp. 237–244, ACM Press, Beijing, China, 2005.

[7] J. Wu and L. Zhao, "Multi-valued model checking via groebner basis approach," in *Proceedings of the 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE '07)*, pp. 35–44, IEEE Computer Society Press, June 2007.

[8] N. Zhou, J. Wu, and X. Gao, "Algebraic verification method for SEREs properties via Groebner bases approaches," *Journal of Applied Mathematics*, vol. 2013, Article ID 272781, 10 pages, 2013.

[9] X. Gao, N. Zhou, J. Wu, and D. Li, "Wu's characteristic set method for SystemVerilog assertions verification," *Journal of Applied Mathematics*, vol. 2013, Article ID 740194, 14 pages, 2013.

[10] J. Little, D. Cox, and D. O'Shea, *Ideals, Varieties, and Algorithms*, Springer, New York, NY, USA, 1992.

[11] T. Becker and V. Weispfenning, *Gröbner Bases: A Computational Approach to Commutative Algebra*, Springer, New York, NY, USA, 1993.

[12] B. Buchberger, "Groebner bases: an algorithmic method in polynomial ideal theory," in *Multidimensional Systems Theory*, pp. 184–232, 1985.

[13] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers C*, vol. 27, no. 6, pp. 509–516, 1978.

[14] S. Hoereth and R. Drechsler, "Formal verification of word-level specifications," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 52–58, Munich, Germany, 1999.

[15] Y. M. Ryabukhin, "Boolean ring," in *Encyclopaedia of Mathematics*, Springer, Michiel, Germany, 2001.

[16] S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti, "Synthesis of system verilog assertions," in *Proceedings of the Conference on Design, Automation and Test in Europe: Designers' Forum (DATE '06)*, pp. 70–75, European Design and Automation Association, Leuven, Belgium, March 2006.

[17] C.-J. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–190, 1995.

[18] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms An Introduction to Computational Algebraic Geometry and Commutative Algebra*, Undergraduate Texts in Mathematics, Springer, New York, 3rd edition, 2007.