Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2013, Article ID 272781, 10 pages
http://dx.doi.org/10.1155/2013/272781

Research Article

Algebraic Verification Method for SEREs Properties via Groebner Bases Approaches

Ning Zhou,[1,2] Jinzhao Wu,[1,3] and Xinyan Gao[4]

[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] *Guangxi Key Laboratory of Hybrid Computation and IC Design Analysis, Guangxi University for Nationalities, Nanning 530006, China*
[4] *School of Software of Dalian University of Technology, Dalian 116620, China*

Correspondence should be addressed to Jinzhao Wu; jzwu_zh@yahoo.cn

Received 8 February 2013; Accepted 22 March 2013

Academic Editor: Xiaoyu Song

This work presents an efficient solution using computer algebra system to perform linear temporal properties verification for synchronous digital systems. The method is essentially based on both Groebner bases approaches and symbolic simulation. A mechanism for constructing canonical polynomial set based symbolic representations for both circuit descriptions and assertions is studied. We then present a complete checking algorithm framework based on these algebraic representations by using Groebner bases. The computational experience result in this work shows that the algebraic approach is a quite competitive checking method and will be a useful supplement to the existent verification methods based on simulation.

1. Introduction

With the complexity of circuits increases, it becomes an important issue to find efficient ways to express and verify design properties. Actually, verification is a very difficult and computationally intensive task. Although great advances have been made over the past decades, all these verification methods suffer from this problem in some way.

Currently, assertion based verification (ABV) has emerged as a promising solution for this problem. In particularly, an assertion specifying language named Property Specification Language (PSL) [1–3] has now become an IEEE standard and accepted by a wide variety of companies. PSL has totally changed the way how designers specify and verify functional requirements and properties of digital systems. Moreover, PSL based ABV has recently been supported by most EDA companies in their tools for both formal and runtime verification.

So far, there have been many efforts in assertion checking solvers including model checking, theorem proving (e.g., HOL [4]), and runtime verification. In [5], an efficient approach to model check safety properties expressed in PSL property has been studied. While in [6], a temporal tester was introduced as a compositional basis for the construction of automata corresponding to temporal formulas in the PSL logic for PSL assertion run-time checking.

As well known, the conventional simulation for assertion checking is a well-understood and the most commonly used technique, but only feasible for very small scale systems and cannot provide exhaustive checking, while symbolic simulation proposed by Darringer [7] as early as 1979 can provide exhaustive checking by covering many conditions with a single simulation sequence but could not handle large circuits due to exponential symbolic expressions.

In our work, to address this functional verification challenge, we propose an alternative implementation mechanism based on algebra symbolic computation combining with symbolic simulation for PSL assertion checking.

Earlier work in applications of symbolic manipulation and algebra computation has gained significant extensions and improvements. In [8], a technique framework on Groebner bases was demonstrated that computer algebra geometry method can be used to perform symbolic model checking by using an encoding of boolean sets as the common zeros of sets of polynomials. In [9], a similar technique framework based Wu's Method has been further extended to bit level symbolic model checking. In [10], an improved framework for multi-valued model checking via Groebner bases approached was proposed, which is based on a canonical polynomial representation of the multivalued logics.

All these existing articles just mainly focus on model checking via algebraic symbolic computation approaches. In our research, instead of static analysis or model checking, we extend this algebraic approach to the area of simulation-based runtime verification methods over polynomial representation models and towards PSL assertions checking.

Our aim is to verify a given temporal property holds or not on the traces produced after several cycles running over a given sequential circuit model.

The idea is that, for any pure combinational circuit model, we can derive its data-flow-based polynomial representation named PM. Meanwhile, for any sequential circuit model and a given running cycle number n, we can also derive its equivalent polynomial representation PM[n] by unrolling this sequential circuit n times and translating it into a pure combinational model. In a similar way, we can get polynomial set representation PS for any temporal assertion.

By suitable restrictions of Boolean and SERE temporal layer of PSL and redefining a hierarchy of PSL assertions, we can guarantee the availability of above polynomial set model. Based on these polynomial set models, 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 inclusion relationship between their polynomials PM[n] and PS and determine whether the temporal assertion holds or not under current running cycle n.

2. Preliminaries

In this section, we will give some preliminary knowledge throughout this paper.

2.1. Cycle-Based Symbolic Simulation. We will firstly sketch the underlying system model for simulation used in our work.

The system model we used is a cycle-based symbolic simulation model that is performed on a cycle-by-cycle basis for synchronous digital systems.

Here, the term *cycle* is defined as one iteration of the evaluation process, during which the state of the design is recomputed and may change. In other words, a cycle is the smallest granularity of time.

Intuitively, cycle-based symbolic simulation is a hybrid approach in the sense that the values that are propagated through the network can be both symbolic expressions or constant Boolean 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.

By convention, we give the model structure definition for symbolic simulation as follows.

Definition 1 (simulation model). The symbolic *Simulation Model* for synchronous digital system is a tuple $\Sigma = (X_0, X, Y, M, S, F, n)$, where

(i) X_0 is a finite set of input assignment including numeric value and symbolic value, Boolean or integer;

(ii) X is a finite set of primary input variables;

(iii) $Y = \{y_i \mid 1 \leq i \leq N_Y\}$ is a finite set of primary output variables;

(iv) $M = \{m_i \mid 1 \leq i \leq N_M\}$ is a finite set of intermediate variables;

(v) n is the sequential depth of the network or running cycles;

(vi) $F = \{y_1, y_2, \ldots, y_m\}$ is a finite output function regarding input or intermediate variables, and note that each $y_i = f_i(x_1, x_2, \ldots, x_n, @)$ $(1 \leq i \leq n)$ is defined on $X \bigcup M$.

Given sequential depth n of the network, a synchronous sequential logic network can be transformed into a pure combinational function of delayed input variables with delay less than or equal to n, that is,

$$Y = F(X, X@1, \ldots, X@n, M, M@1, \ldots, M@n). \quad (1)$$

The behavior of a circuit is defined by its excitation function Y that serves a role similar to the transition relation or next-state functions of temporal logic model checkers.

The 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 (X_0). Each of the primary input signals will be assigned a distinct symbolic variable or a symbolic constant. Then, at the end of a simulation step, the expressions representing the next-state functions generally undergo a parametric transformation based optimization. After parameterization, the newly generated functions are used as present state for the next state of simulation.

In this paper, simulation based verification is to check whether the given assertion is satisfied or not after running a few cycles.

2.2. PSL Preliminary. PSL is a hierarchical language and its syntax is very declarative and structural. Generally, PSL contains four layers: Boolean, temporal, verification, and modeling layers.

(i) Modeling Layer. Modeling layer is needed to define the verification environment specially for formal verification tools. This layer is used to model behavior of design inputs and to model auxiliary parts of the design that are needed for verification.

(ii) Verification Layer. Verification layer is more related to the description of verification tools where notions like assume and guarantee are present. This layer is used to tell the verification tool what to do with the properties described by the temporal layer.

(iii) Temporal Layer. Temporal layer is the essence of PSL where complex temporal relations between signals can be expressed. This layer can describe properties that involve complex temporal relations which are evaluated over a series of evaluation cycles.

(iv) Boolean Layer. Boolean layer is used to build expressions for the other layers, specifically the temporal layer. Boolean expressions are evaluated in a single evaluation cycle.

PSL allows the engineer to define assertions describing the system's behavior once and reuse them between different forms of formal, semiformal, or functional verification. With PSL, it is possible to perform assertion based runtime verifications of the design while simulation properties are checked.

According to PSL specification [1, 3, 11], every assertion written in PSL can be broken down into parts that can be attributed to one of those four layers.

The Boolean layer comprises all Boolean expressions including signal names as well as HDL expressions and PSL expressions (especially all built-in function calls like, e.g., *prev*(*b*) and *rose*(*b*) and the logical implication and other operators).

The Boolean layer forms an underlying basis for the whole assertion architecture. In this paper, we will limit our discussion only to a special subset of the Boolean layer for our purpose. We then further build a restricted simple subset of SERE layer for temporal property specification and verification over this constrained Boolean layer.

3. System Polynomial Representation Model

In this section, we will discuss polynomial modeling for combinational and sequential circuits. Previous work [12] has shown that any combinational circuit can be uniquely represented by a minimum order polynomial. Here, we give an alternative data-flow based polynomial set representation model for our assertions checking purpose whose zero set can make such a data-flow model work well.

3.1. 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 following polynomials:

(1) $y = a + b \Rightarrow (y - a - b)$,

(2) $y = a - b \Rightarrow (y - a + b)$,

(3) $y = a * b \Rightarrow (y - a * b)$,

(4) $y = a/b \Rightarrow (y * b - a)$.

The basic logic operations [13] like "**AND**," "**OR**," and "**NOT**" can be modeled by the following forms:

$$\begin{gathered} y = NOT \quad x \Longrightarrow (1 - x - y), \\ y = x_1 \quad AND \quad x_2 \Longrightarrow (x_1 * x_2 - y), \\ y = x_1 \quad OR \quad x_2 \Longrightarrow (x_1 + x_2 - x_1 * x_2 - y). \end{gathered} \tag{2}$$

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

$y = x_1 \oplus x_2$ (or $y = x_1$ **XOR** x_2) $\Rightarrow (y - (x_1 + x_2 - x_1 * x_2) * (1 - x_1 * x_2))$.

For all bit level variable x_i $(0 \le i \le n)$, a limitation $x_i * x_i - x_i$ should be added.

3.2. Branch and Sequential 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, $(0, \dots, T - 1)$, 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 $x * x - x$,

$$\begin{gathered} y = \text{MUX}(x_0, x_1, \dots, x_n, s), \quad i = s \Longrightarrow y = x_i, \\ (0 \le i \le n) \Longrightarrow y - \sum_{i=1}^{n-1} \left(\prod_{j \in \{0,1,\dots,n-1\}\{i\}} \left(\frac{(s-j)}{(i-j)} \right) \right) \\ * x_i, \quad \text{with} \prod_{i=0}^{n-1} (s - i) = 0. \end{gathered} \tag{3}$$

Each flip-flop (FF) in the circuit can be modeled as a multiplexer, as illustrated in Figure 1. We have the following proposition to state this model.

Proposition 2. *For a D FF model (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*

$$\begin{gathered} (y' - D) * (c - 1), \qquad (y' - D') * c, \\ (y' - D) * (y' - D'), \\ or \\ y' - D * (c - 1) - D' * c. \end{gathered} \tag{4}$$

Proof. Let D be the current state and let y' denote the next state of the flip-flop. When the clock value is 0, y' has the same value as D so that the FF maintains its present state; when the clock 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 3. *Let D be a FF model (D' is the next state), without enable signal; then its equivalent combinational formal polynomial algebraic model can be described as* $(y' - D)$.

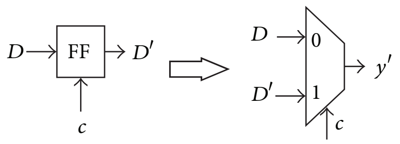

Figure 1: Flip-flop model.

3.3. Sequential Unrolling. Generally, for a sequential circuit, 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.

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

For example, PM[i] is defined as follows. $\mathrm{PM}[i] = \{(x1_{[i]} - m2_{[i]} - y3_{[i]}), \ldots\}$, where $x1$ denotes signal variable name while $x1_{[i]}$ denotes variable state in ith simulation cycle. If the given running cycle is n, then we have the system representation: $\mathrm{PM} = \{\bigcup_{i=0}^{n} \mathrm{PM}[i]\}$.

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

$$\mathrm{FM} = \left\{ \bigcup_{i=0}^{n} \mathrm{FM}[i] \right\}, \tag{5}$$

where $\mathrm{FM}[i] = \mathbf{C}(x1_{[i]}, \ldots, m1_{[i]}, \ldots, m1_{[i]}, \ldots, x1_{[i+1]}, \ldots, m1_{[i+1]}, \ldots, y1_{[i+1]}, \ldots)$ denotes 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.

3.4. Sequence Operator Modeling. In this paper, only a so-called simple subset of PSL will be considered, which subsumes the properties in which time advances monotonically, from left to right through the property: if an entity (a Boolean Expression or a SERE) needs to be evaluated at a given time, all other entities right of it do so far not need to be known. Many properties not in the simple subset can be rewritten by the simple subset. The most properties to be verified can be expressed within the bounds of the simple subset.

For SEREs, only the following features are supported by our modeling method:

(1) standard Boolean expressions,

(2) fixed length Kleene closure,

(3) SERE concatenation,

(4) SERE fusion,

(5) SERE disjunction,

(6) length-matching SERE conjunction.

By the constrained simple subset of PSL, the user can specify a safety property using only nonnegated weak operators. Intuitively, a safety property is used to ensure that "something bad does not happen" which is important in formal verification. Because safety properties are easier to verify, this approach is only able to deal with safety properties.

(1) **Next** Operator

It indicates that the property will hold if its operand holds at the next cycle. For example,

$$\textbf{assert}\,(req- > next\ ack) \tag{6}$$

states that if signal *req* is asserted then *ack* will be asserted at next cycle:

$\Rightarrow \mathbf{N}^{i}(req)$ **and** $\mathbf{N}^{i+1}(ack)$.

(2) **Semicolons** Operator

Semicolons operator, a semicolon(;), is used to join two SEREs (or two AL expressions, or a AL expression and a SERE) in such a way that the right-hand SERE starts the cycle after the left-hand SERE ends.

For example, $G = \{\textbf{assert}\,(req; ack)\}$ states that when signal *req* is asserted then *ack* will be asserted at next cycle:

$\Rightarrow \mathbf{N}^{i}(req)$ and $\mathbf{N}^{i+1}(ack)$

$\Leftrightarrow \mathbf{N}^{i}(req\ \textbf{is}\ H)$ and $\mathbf{N}^{i+1}(ack\ \textbf{is}\ \overline{H})$,

where $(0 \leq i \leq dep(G))$.

(3) **Fusion** Operator

The fusion operator, a colon (:), is used to join two SEREs (or two AL expressions, or a AL expression and a SERE) in such a way that there is a single cycle of overlap between them: the right-hand SERE just starts the same cycle that the left-hand SERE ends.

For example, $G = \{\textbf{assert always}\,(req : ack; gnt)\}$ states that when signal *req* is asserted then *ack* and *gnt* will be asserted at next cycle:

$\Rightarrow \mathbf{N}^{i}(req\ \textbf{is}\ H)$ and $\mathbf{N}^{i}(ack\ \textbf{is}\ H)$ and $\mathbf{N}^{i+1}(gnt\ \textbf{is}\ H)$,

where $(0 \leq i \leq dep(G))$.

(4) **Repeat** Operator

Repeat operators allow the user to build more sophisticated SEREs, using variations on the SERE repetition operators $[*n]$, $[= n]$, and so forth. Consecutive repetition operators provide a shortcut to typing the same sub-SERE a number of times.

In this paper, we only consider fixed times repeat operator $[= n]$.

For example, $G = \{\textbf{assert}\,(req[n]; ack)\}$ states that when signal *req* is asserted n times then *ack* will be asserted at next cycle. We then have

$\Rightarrow \mathbf{N}^{1}(req\ \textbf{is}\ H)$ and $\cdots$ and $\mathbf{N}^{n}(req\ \textbf{is}\ H)$

and $\mathbf{N}^{n+1}(ack\ \textbf{is}\ H)$.

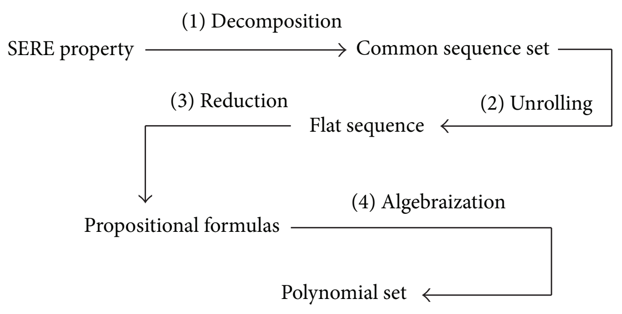

Figure 2: Algebraization steps of SEREs.

4. Translation of SERE

In this section, we will mainly discuss the hierarchical modeling method of SERE. The temporal layer contains "Sequential Extended Regular Expressions" (SEREs) which allow describing the relation between Boolean layer expressions over time.

Firstly, we discuss the general algebraization process of SERE from a symbolic computation point of view.

4.1. Algebraization Process. The algebraization process of SERE properties can be demonstrated in Figure 2. The properties written in SERE will be unrolled and checked against the design for bounded time steps in our method. Note that only a constrained subset of SERE 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 SERE into flat sequences according to the semantics of each supported operator.

Secondly, the unrolled flat sequences will be added 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.

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

4.2. Boolean Layer Modeling. The PSL Boolean layer forms an underlying basis for the whole assertion architecture. In this paper, we limit our discussion only to the Boolean layer and a special constrained subset of it.

While the Boolean layer consists of Boolean expressions that hold or do not hold at a given cycle, the temporal layer provides a way to describe relationships between Boolean expressions over time.

In this paper, we distinguish between signal logic and Boolean proposition logic.

Therefore, we have the following two definitions.

Definition 4 (signal logic). In digital circuit systems, signal logic (**SL**, for short) is defined as follows:

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

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

(iii) if a signal s is assigned a symbolic value, then its signal value is defined as U.

Definition 5 (symbolic trajectory logic). The definition of trajectory evaluation logic (TEL) is extended as the following grammar:

$$f ::= n \textbf{ is } 0 \mid n \textbf{ is } 1 \mid w \textbf{ is } \mathbf{N} \mid f_1 \textbf{ and } f_2 \mid P \longrightarrow f \mid \mathbf{N}(f), \tag{7}$$

where "**is**" is used to state the value of a Boolean or word-level node in the circuit. Defined recursively over V, where p is a Boolean expression over V; n is a node or variable name; f, f_1, f_2 are TEL formulas; **N** is the next-time operator.

For example, a symbolic trajectory assertion, assume $V = \{a\}$, then $[(\text{in } \textbf{is } a) \wedge \mathbf{N}(\texttt{true}) \Longrightarrow \mathbf{N}(\text{out } \textbf{is } \overline{a})]$.

Let numeric subscript denote time frame number for each variable, then we have $\text{in}_{[0]}$ **is** a, $\text{out}_{[1]}$ **is** $\overline{a}$.

In this paper, all temporal operators in PSL SEREs specification will be modeled by next operator **N**.

We will introduce a notion of symbolic constant to PSL inspired from GSTE [14].

Definition 6 (symbolic constant). A *symbolic constant* [14] is a rigid Boolean or integer variable that forever holds the same boolean value. The notion of symbolic constant is introduced in an assertion 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.

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

Here, we provide a formal syntax definition for assertion proposition logic, namely, *Assertion Boolean Logic*.

If x_1, x_2, m, and n are of **SL**, then we have $x_1 = x_2$, $m = x_1 \& x_2$, $m = x_1 \| x_2$, $m = !x_2$, and $(m = x_1) \wedge (n = x_2)$ are all of valid **AL** and can also be verified by using polynomial model.

Definition 7 (assertion Boolean logic layer syntax). If $a \in \mathbf{SL}$ and $H \in \text{BC}$, then $a = H$ is an atom Boolean formula; [*Atom Boolean Formula*].

Built-in functions: *stable*(), *rose*(), *fell*(), *isunknown*(), *onehot*(), and *onehot*0() are of atom Boolean formulas.

If a is an integer signal logic variable (denoted by $a \in \mathbf{ISL}$) and symbolic constant $I \in IC$, then $a = I$ is also an atom Boolean formula; [*Atom Boolean Formula*].

If a_1 and a_2 are atom Boolean formulas, then

(1) a_1 & & a_2 [*Standard Logic* "*AND*"],

(2) $a_1 \| a_2$ [*Standard Logic* "*OR*"],

(3) $!a_1$ [*Standard Logic* "*NOT*"],

(4) $a_1- > a_2$ [*Standard Logic* "*Implication*"] are Boolean formulas.

Assertion proposition logic (**AL**) for PSL is defined as standard Boolean logic. A Boolean expression of **AL** is an expression that is evaluated in a single cycle and has the value *true* or *false*. Boolean connectives for **AL** are interpreted in the standard.

For example, assertion ($a[15:0] == b[15:0]$), given in the Verilog flavor of PSL, is a valid Boolean expression which means $a[15:0]$ and $b[15:0]$ are equal.

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. Algorithm Framework

In this section, we will describe how an assertion is checked using Groebner basis approach.

As we all know, in traditional numeric simulation [15], PSL assertion checking process can be described as follows. Firstly, the design file with PSL codes is compiled into local executable binary code via simulation tools (such as, QuestaSim or ModelSim). The designer then provides a testbench file to set input values, running cycles, and other parameters. Finally, the designer performs simulation by starting "run" command to produce traces for assertion checking.

Firstly, we will sketch some of the key notions of Groebner bases theory [16, 17] and symbolic computation.

5.1. Groebner Bases Preliminary. 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.

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

Theorem 8 (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 $I_1 \times I_2 = \langle f_i \times g_j \mid 1 \leq i \leq r, 1 \leq j \leq s \rangle$.

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.

Definition 9 (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 work [18], 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 $G : \forall g \in G$, 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.

Theorem 10 (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 lex order where $x_1 \succ x_2 \succ \cdots \succ x_n$. Then, for every $0 \leq l \leq n$, the set*

$$G_l = G \cap k\left[x_{l+1}, \ldots, x_n\right] \tag{8}$$

is a Groebner basis of the lth elimination ideal I_l.

Theorem 11. *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 11 can also be taken as the definition of a Groebner basis. Then we will get an efficient algorithm for solving the ideal membership problem. Assumed that we know a Groebner basis G for the ideal in question, we only need to compute a remainder with respect to G to determine whether $f \in I$.

5.2. Verification Principle Based Theorem Proving. As just mentioned in previous section, our checking method is based on algebraic geometry theory. Algebraic geometry 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 system states in which the appropriate assignments are made.

In our method, we regard any set of points in k^n as the variety of some ideal. We can use the ideal or any basis for the ideal as a way of encoding the set of states. The verification problem is then transformed into ideal membership problem that can be solved by computation algorithms.

From Groebner Bases theory [16, 18] every nonzero ideal $I \subset k[x_1, \ldots, x_n]$ has a Groebner basis and the following proposition evidently holds.

Proposition 12. *Let C and S be polynomial sets of $k[x_1, \ldots, x_n]$, and $\langle GS \rangle$ is a Groebner basis for $\langle S \rangle$, then one has $\langle C \rangle \subseteq \langle S \rangle \Leftrightarrow \forall c \in C : remd(c, GS) = 0$.*

All supported SEREs properties can be classified into two categories.

(1) *Implication-typed*: Properties of this type have an explicit antecedent that can be taken as an initial precondition. If the precondition is conflict with the system model, this property will be viewed as *false*. Otherwise, further checking process will be performed.

(2) *Sequence-typed*: Properties of this type have no explicit antecedent, and therefore an initial condition should be provided by the testbench. If the precondition is in conflict with the system model, this sequence property will also be viewed as *false*. Otherwise, further checking process will be performed.

Theorem 13. *Suppose that G (If $G = [A \Rightarrow C]$ is an implication-typed property, then A denotes the antecedent; otherwise, G is a sequence-typed property, then A is the precondition) and M is a system model. Let PA and PM be the polynomial set representations for A and M, respectively, constructed by previous mentioned rules. Let $H = PA \cup PM = \{h_1, h_2, \ldots, h_s\} \subseteq k[x_1, \ldots, x_n]$, $I = \langle H \rangle$ (where $\langle H \rangle$ denotes the ideal generated by H), $\{c_1, c_2, \ldots, c_r\}$ denotes the polynomial set representation for C, $GB_H = gbasis(H, \prec)$, then one has*

$$((1 \notin GB_H) \textbf{ and } remd\,(C, GB_H) == 0)$$
$$\Leftrightarrow ((1 \notin GB_H) \textbf{ and } \textstyle\bigwedge_{i=0}^{r} (remd\,(c_i, GB_H) == 0))$$
$$\Leftrightarrow (M \models G).$$

Proof. By Hilbert's Nullstellensatz theory and previously mentioned notions, it is easy to have the conclusion. □

5.3. Checking Algorithm. For a practical assertion checking process, it needs to build complicated syntax analysis tree for a given assertion and call the basic checking functions to perform checking. For simplicity, we only provide the core decision algorithms and the basic process flow.

Firstly, the original circuit is sliced with respect to the given assertion G. Polynomial representation for sliced circuit model, antecedent, and consequent will then be built, respectively. Finally, we calculate the hypothesis set and its Groebner bases to determine whether the assertion holds or not.

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

An important advantage of our algorithm is that it only requires a comparatively small amount of state variables to verify a given assertion due to slicing reduction.

```
Input: Circuit model C, an assertion $G = [\mathscr{A} \Rightarrow \mathscr{C}]$;
Output: Boolean: true or false;
BEGIN
     /* Step 0: initialize input signals via testbench */
(0)     $InitSignals(\overrightarrow{X_0})$;
(1)     $\mathbf{S} = \emptyset; \mathbf{M} = \emptyset; PS_A = \emptyset; H = \emptyset; PS_C = \emptyset$;
     /* Step 1: build polynomial model */
(2)     $\mathbf{M} = BuildPS(\mathbf{S})$;
     /* Step 3: build polynomial set for antecedent $\mathscr{A}$ */
(3)     $PS_{\mathscr{A}} = BuildPS(\mathscr{A})$;
     /* Step 3: build polynomial set for consequent $\mathscr{C}$ */
(4)     $PS_{\mathscr{C}} = BuildPS(\mathscr{C})$;
     /* Step 4: calculate the $PS_{\mathscr{A}} \cup \mathbf{M}$ */
(5)     $H = PS_{\mathscr{A}} \cup \mathbf{M}$;
     /* Step 5: calculate the Groebner base of $\langle H \rangle$ */
(6)     $GB_H := gbasis(H, \prec)$;
     /* Step 6: calculate the Groebner base of $\langle H \rangle$ */
(7)   if($1 \in GB_H$){
(8)         return false; }
(9) if($remd(PS_{\mathscr{C}}, GB_H) \neq 0$){
(10)        return false; }
(11)  return true; /* Assertion does hold */
END ;
```

Algorithm 1: Assertion checking: *AssChk* $(\mathbf{C}, G)$.

From the above discussion, we have the process steps and detailed algorithm description in Algorithm 2.

Firstly, the original circuit is transformed into a normal polynomial representation and the assertion as well. Then, calculate Groebner bases using the Buchberger algorithm [19] and their elimination ideals. Finally, examine the relation between elimination ideals and determine whether the assertion holds or not.

6. A Case Study

In this section, we will study a case to show how PSL SERE properties are verified by polynomial representation and algebra computation.

6.1. Circuit and PSL Modeling. As an example, consider the 3-bit synchronous counter circuit **C** in Figure 3, whose polynomial set can be constructed as follows. In this circuit, there exists a design bug that "**AND**" gate is replaced by "**OR**" gate incorrectly. Now, let us show how to check this error using our symbolic algebraic method:

$$\begin{aligned} &PSet_{\text{counter}} \\ &\quad = \{ (y1 - (m1 + m4 - m1 * m4) * (1 - m1 * m4)), \\ &\qquad (y2 - (m2 + m3 - m2 * m3) * (1 - m2 * m3)), \\ &\qquad (1 - m3 - y3), (1 - m4 - m3 * m2), \\ &\qquad (m1' - y1)(m2' - y2)(m3' - y3) \}, \end{aligned} \tag{9}$$

```
*Input:* Circuit model C, a temporal assertion s, running cycles cycles;
*Output:* Boolean: true or false;
BEGIN
(1)  $i = 0$;
(2)  switch(operator(s)){
(3)      case always :{
(4)          while($i < cycles$){
(5)              if(!AssChk(C, s, i)){;
(6)                  return false;}
(7)              $i+ = dep(s)$}
(8)          } /∗ end while ∗/
(9)      case eventually:{
(10)         while($i < cycles$){
(11)             if(AssChk(C, s, i)){;
(12)                 return true;}
(13)             $i+ = dep(s)$}
(14)         } /∗ end while ∗/
(15)     }/∗ end case ∗/
(16)     case never :{
(17)         while($i < cycles$){
(18)             if(AssChk(C, s, i)){;
(19)                 return false;}
(20)             $i+ = dep(s)$}
(21)         } /∗ end while ∗/
(22)     }/∗ end case ∗/
(23)         deafult :{
(24)         return AssChk(C, s, i);
(25)         }/∗ end switch ∗/
(26)     }
END ;
```

Algorithm 2: Assertion checking: *TemporalAssChk* (**C**, s, *cycles*).

where $x1'$ denotes the next state of $x1$. For the ith cycle, we use $x1_{[i]}$ to denote variable name in current cycle.

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

$$\begin{aligned} \mathrm{PM}[i] &= \{(y1_{[i]}-(m1_{[i]}+m4_{[i]}-m1_{[i]}*m)*(1-m1_{[i]}*m4_{[i]})), \\ &\quad (y2_{[i]}-(m2_{[i]}+m3_{[i]}-m2_{[i]}*m3)*(1-m2_{[i]}*m3_{[i]})), \\ &\quad (1-m3_{[i]}-y3_{[i]}), (1-m4_{[i]}-m3_{[i]}*m2_{[i]}), \\ &\quad (m1_{[i+1]}-y1_{[i]}), (m2_{[i+1]}-y2_{[i]}), (m3_{[i+1]}-y3_{[i]})\}. \end{aligned} \tag{10}$$

Therefore, we have $\mathrm{PM} = \{\bigcup_{i=0}^{7} \mathrm{PM}[i]\}$.

For any boolean variable a, we will impose an extra constraint: $a*a-a$. Thus, we should define the corresponding constraints set as follows: $\mathrm{CNS}[i] = \{a_{[i]}*a_{[i]}-a_{[i]}\}$ for all bit-level variables in the ith cycle.

In the same manner, we have $\mathrm{CNS} = \{\bigcup_{i=0}^{7} \mathrm{CNS}[i]\}$.

The sequential properties of this counter circuit can be specified by the following assertions:

$G_1 = \{$**assert always** $(m1 = \overline{H}\,\&\,m2 = \overline{H}\,\&\,m3 = \overline{H}) \mid\Rightarrow (m1 = \overline{H}\,\&\,m2 = \overline{H}\,\&\,m3 = H)\}$,

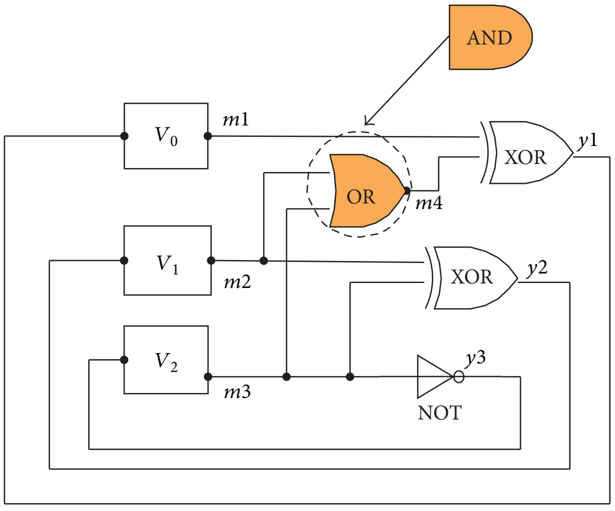

Figure 3: Synchronous counter.

$G_2 = \{$**assert always** $(m1 = \overline{H}\,\&\,m2 = \overline{H}\,\&\,m3 = H) \mid\Rightarrow (m1 = \overline{H}\,\&\,m2 = H\,\&\,m3 = \overline{H})\}$,

$G_3 = \{$**assert always** $(m1 = \overline{H}\,\&\,m2 = H\,\&\,m3 = \overline{H}) \mid\Rightarrow (m1 = \overline{H}\,\&\,m2 = H\,\&\,m3 = H)\}$, and the rest may be deduced by analogy.

Table 1: Polynomial representations for properties to be verified.

No.	Precondition	Expected consequent
0	$m1_{[0]}, m2_{[0]}, m3_{[0]}$	N/A
Cycle1	N/A	$(m1_{[1]}, m2_{[1]}, m3_{[1]} - 1)$
Cycle2	N/A	$(m1_{[2]}, m2_{[2]} - 1, m3_{[2]})$
Cycle3	N/A	$(m1_{[3]}, m2_{[3]} - 1, m3_{[3]} - 1)$
Cycle4	N/A	$(m1_{[4]} - 1, m2_{[4]}, m3_{[4]})$
Cycle5	N/A	$(m1_{[5]} - 1, m2_{[5]}, m3_{[5]} - 1)$
Cycle6	N/A	$(m1_{[6]} - 1, m2_{[6]} - 1, m3_{[6]})$
Cycle7	N/A	$(m1_{[7]} - 1, m2_{[7]} - 1, m3_{[7]} - 1)$

Afterward, we will demonstrate the verification process step by step.

Firstly, we calculate the sequential depth and have

$$dep(G_1) = 2, dep(G_2) = 2, \text{and } dep(G_2) = 2.$$

Secondly, to verify a given property hold or not, we have to build a system model with 8 cycles at most and check $dep(G_1) = 2$ steps.

The circuit model to be verified is below:

$$\mathrm{SM} = \mathrm{PM} \bigcup \mathrm{CNS}. \tag{11}$$

The properties of this counter can be specified as the following PSL assertions listed in Table 1.

6.2. Assertion Checking Using Maple. We run this example by using Maple 13 software. Before running, we manually translated all models into polynomials. The experiment is performed on a Computer with a 2.40 GHz CPU (Intel i5 M450) and 512 MB of memory. It took about 0.04 seconds and 0.81 MB of memory to find this error when applying Groebner method:

$[>with(Groebner)$

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

$[>\mathrm{TDEG} := tdeg($

$m1_{[0]}, m2_{[0]}, m3_{[0]}, m4_{[0]}, m1_{[1]}, m2_{[1]},$
$m3_{[1]}, m4_{[1]}, m1_{[2]}, m2_{[2]}, m3_{[2]}, m4_{[2]},$
$y1_{[0]}, y1_{[1]}, y1_{[2]}, y2_{[0]}, y2_{[1]}, y2_{[2]},$
$y3_{[0]}, y3_{[1]}, y3_{[2]})$

$[>\mathrm{CGB} := Basis(\mathrm{G}, \mathrm{TDEG})$

$[>ret := Normal\,Form(m3_{[0]} - 1, \mathrm{CGB}, \mathrm{TDEG})$

$[>ret = 0.$

As shown in maple outputs, the given circuit has been modeled as polynomial set CM (its Groebner basis is denoted by CGB) and assertion representation as $(m3_{[0]} - 1)$. From the running result, we have return value of *Normal Form* is 0 which means CGB be divided with no remainder by $(m3_{[0]} - 1)$. Thus, from the previously mentioned verification principles, it is easy to conclude that the SERE assertion G_1 holds under this circuit model after 1 cycle. Other results are shown in Table 2.

Table 2: Result table.

Cycle no.	Polynomial	Result
Cycle1	$m1_{[0]}, m2_{[0]}, m3_{[0]} - 1$	$ret = 0$
Cycle2	$m1_{[1]}, m2_{[1]} - 1, m3_{[1]}$	$ret \neq 0, m1_{[1]}$ fails
Cycle3	$m1_{[2]}, m2_{[2]} - 1, m3_{[2]} - 1$	Stop

From Table 2, when checking G_2 assertion, the result $ret := Normal\,Form\,(m1_{[1]}, \mathrm{CGB}, \mathrm{TDEG}) = 1 \neq 0$ so that we can conclude the assertion does not hold and there must exist some error in the original circuit. This case is a fairly complete illustration of how our checking algorithm works.

7. Conclusion

In this paper, we presented a new method for constrained SERE temporal assertions checking by combining symbolic simulation with symbolic algebraic approaches. We modified the original PSL specification to adapt our verification requirements and rebuilt a new constrained class of boolean and temporal layer.

We first introduce a notion of symbolic constant for data path verification, which can gain great state coverage for simulation based verification. This method allows users to deal with more than one state and many input combinations at a time. This advantage comes directly from the fact that many vectors are simulated at once using symbolic value.

We then defined a constrained simple subset of SERE and proposed an practical algebraization method for each temporal 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 circuits and assertions. In other words, symbolic simulation is performed on data-flow model and its unrolled form in polynomial representation. Our method is to eventually translate a simulation based verification problem into a pure algebraic zero set determination problem by previously mentioned steps, which can be performed on any general symbolic algebraic tool. An experimental evaluation using maple has shown that the method is extremely efficient and useful.

Furthermore, we can summarize the advantages of our checking method as follows:

(1) from the real case, we see that SERE properties verification can be achieved easier using symbolic algebraic than traditional method. Complex test bench or test vector is not essential for this approach;

(2) this advantage comes directly from the fact that many vectors are simulated at once using symbolic value;

(3) for assertion property verification, an efficient slicing reduction technique can be applied to gain performance improvement.

Basically, our method can be taken as a useful theoretical insight for verification methodology.

Finally, we plan to explore further tradeoffs and combine numeric computation with symbolic simulation for boosting performance, in particular, and to apply this method to more industrial case studies.

Acknowledgments

The project is supported by the National Natural Science Foundation of China under Grant no. 60973147, the Natural Science Foundation of Guangxi under Grant no. 2011GXNSFA018154, the Science and Technology Foundation of Guangxi under Grant no. 10169-1, Guangxi Scientific Research Project no. 201012MS274, and Grants (HCIC201102) of Guangxi Key Laboratory of Hybrid Computation and IC Design Analysis Open Fund. 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 standard for property specification language (psl)," IEEE Std 1850-2005, 2005.

[2] C. Eisner and D. Fisman, *A Practical Introduction to PSL*, Integrated Circuits and Systems, Springer, New York, NY, USA, 2006.

[3] "IEEE standard for property specification language (psl)," IEEE Std 1850-2010, 2010, Revision of IEEE Std 1850-2005.

[4] T. Tuerk, K. Schneider, and M. Gordon., "Model checking PSL using HOL and SMV," in *Proceedings of the 2nd International Haifa Verification Conference on Hardware and Software, Verification and Testing (HVC '06)*, E. Bin, A. Ziv, and S. Ur, Eds., pp. 1–15, Springer, Berlin, Germany, 2006.

[5] T. Launiainen, K. Heljanko, and T. Junttila, "Efficient model checking of PSL safety properties," *IET Computers & Digital Techniques*, vol. 5, no. 6, pp. 479–492, 2011.

[6] A. Pnueli and A. Zaks, "PSL model checking and run-time verification via testers," in *Proceedings of the 14th international conference on Formal Methods (FM '06)*, pp. 573–586, 2006.

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

[8] G. S. Avrunin, "Symbolic model checking using algebraic geometry," in *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)*, pp. 26–37, 1996.

[9] W. Mao and J. Wu, "Application of Wu's method to symbolic model checking," in *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC '05)*, pp. 237–244, July 2005.

[10] 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, June 2007.

[11] C. Eisner and D. Fisman, *A Practical Introduction to PSL*, Integrated Circuits and Systems, Springer, New York, NY, USA, 2006.

[12] J. Smith and G. De Micheli, "Polynomial methods for component matching and verification," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '98)*, pp. 678–685, November 1998.

[13] Y. M. Ryabukhin, "Boolean ring," in *Encyclopaedia of Mathematics*, M. Hazewinkel, Ed., Springer, 2001.

[14] C. J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, 1995.

[15] K. H. Chang, W. T. Tu, Y. J. Yeh, and S. Y. Kuo, "A simulation-based temporal assertion checker for psl," in *Proceedings of IEEE International Symposium on Micro-NanoMechatronics and Human Science*, pp. 1528–1531, 2003.

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

[17] T. Becker and V. Weispfenning, *Groebner Bases: A Computational Approach to Commutative Algebra*, vol. 141, Springer, New York, NY, USA, 1993.

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

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