Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2013, Article ID 740194, 14 pages
http://dx.doi.org/10.1155/2013/740194

Research Article

Wu's Characteristic Set Method for SystemVerilog Assertions Verification

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

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

Correspondence should be addressed to Xinyan Gao; tyler.gxy@hotmail.com and Jinzhao Wu; himrwujz@yahoo.com.cn

Received 8 February 2013; Revised 9 April 2013; Accepted 9 April 2013

Academic Editor: Xiaoyu Song

We propose a verification solution based on characteristic set of Wu's method towards SystemVerilog assertion checking over digital circuit systems. We define a suitable 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 characteristic set of polynomial system. This symbolic algebraic approach is a useful supplement to the existent verification methods based on simulation.

1. Introduction

Currently, functional verification becomes an intensive challenge phase in the state-of-the-art digital systems development process. Assertion-based verification (ABV) has emerged as a promising solution to express and verify design properties.

SystemVerilog [1–3], the industry's first unified hardware description and verification language (HDVL), was developed originally by Accellera and can be viewed as an extension of the Verilog language with the added benefit of supporting object orientated constructs and assertions.

SystemVerilog was adopted as IEEE Standard 1800–2005 firstly in 2005, and the latest version is IEEE standard 1800–2009, the current version. SystemVerilog has been accepted by a wide variety of companies and has been supported by most EDA companies in their tools in practice which has totally changed the way designers specify and verify functional requirements and properties of digital systems.

Many efforts have been devoted to assertion checking solvers in recent years including model checking [4], theorem proving [5], and other simulation-based checking methods. In [6], an efficient approach to model check safety properties expressed in assertion property is studied. Proving SVA safety properties using induction-based BMC is studied in [7]. All these methods can be classified into two categories: formal and informal aspects.

On the one hand, formal method such as model checking often suffers from "state explosion" problem and cannot be applied to large scale cases. Model checking of SVAs is PSPACE complete even for a "simple subset." If more elaborated features, like intersection of regular expressions or instantiation of properties, are used, the problem becomes EXPSPACE complete.

On the other hand, informal methods such as conventional simulation for assertion checking is a well-understood and most commonly used technique, but only feasible for very small scale systems and cannot provide exhaustive checking.

A promising semiformal technique, symbolic simulation, proposed by Darringer [8] as early as 1979, can provide exhaustive checking by covering many conditions with a single simulation sequence.

In our work, to address the challenge, we propose an alternative implementation mechanism based on Wu's characteristic set by combining symbolic computation with symbolic simulation for assertions checking. This paper aims to verify whether an expected SVA property holds or not on the finite traces produced after several cycles running over a given sequential circuit.

The underlying idea is that, for any combinational circuit model, we can derive its data flow based on polynomial representation model. Meanwhile, for any sequential circuit model and a given running cycle number, we can also derive its polynomial representation by unrolling this sequential circuit for several times into a pure combinational model. In a similar way, we can get polynomial set representation model of a temporal assertion written in SVA.

By suitable restrictions of SVA and defining a constrained subset of SVA, we can get the polynomial set representations of the subset. Based on the polynomial set model, cycle-based symbolic simulation can be performed to produce symbolic traces. We then apply Wu's method symbolic algebra approach to check the zeros set relation between their polynomial representations and determine whether the expected assertion holds or not at current running cycle.

The rest of this paper is structured as follows. Preliminary knowledge with regarding to SystemVerilog used throughout this paper is introduced in Section 2. Section 3 introduces polynomial representation modeling for synchronous digital systems. Section 4 describes the cycle-based model and sequential unrolling. Section 5 will discuss sequential assertions modeling with polynomials. In Section 6, we will propose a complete verification algorithm framework based on Wu's characteristic set. Section 7 will demonstrate an example using MMP tool based on our method. In the last section, we make a short summary of our research and discuss the future work.

2. SystemVerilog Preliminary

In this section, we will give some preliminary knowledge which may be used in later sections of the paper.

SystemVerilog, as an IEEE approved hardware description language, has combined many of the best features of both VHDL and Verilog and provided superior capabilities for system architecture, design, and verification.

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.

SystemVerilog provides special language constructs and assertions [9, 10], 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.

One important capability of SystemVerilog is the ability to define assertions outside of the 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: immediate assertions and concurrent assertions.

Immediate assertions are primarily intended to be used with simulation and evaluate using simulation event-based semantics. While concurrent assertions are based on clock semantics and use sampled values of variables.

Concurrent assertions can be used in always block or initial block as a statement, a module as a concurrent block, an interface block as a concurrent block, a program block as a concurrent block.

An example of a property using sequence and formal argument is demonstrated as the following.

$$
\begin{aligned}
&\textbf{property}\ \textit{test}\ [(\textit{ready}, \textit{done}, \textit{m\_bus}, \textit{x\_bus})];\\
&\quad \textit{int}\ \textit{v\_data};\\
&\quad (\textit{ready}, \textit{v\_data} = \textit{m\_bus})\ \#\#[3:5]\\
&\quad (\textit{done}\ \&\&\ \textit{x\_bus} == \textit{v\_data});\\
&\textbf{endproperty}\ [:\textit{test}]
\end{aligned}
$$

The property *test* states that if *ready* holds then local variable *v_data* gets assigned the value of *m_bus*. Then within 3 to 5 cycles later, *done* is **TRUE** and *x_bus* should be equal to the contents of the saved local variable *v_data*. The property fails if *ready* is **FALSE** or if it follows a successful *ready*; the expression "*done* && *x_bus* == *v_data*" fails to be **TRUE** within the range from 3 to 5 cycles after *ready*.

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

The verification statement in SVA has three forms: assert, assume, and cover. In this paper, only **assume** and **assert** are involved: the statement **assert** to specify the property as a checker to ensure that the property holds for the design; the statement **assume** to specify the property as an assumption for the environment. 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.

Basically, SVA has four language layers: Boolean, sequence, property, and statement.

(1) *Boolean layer*. The boolean layer consists of boolean expressions in which each variable referenced is either a design variable or a local variable of the assertion.

(2) *Sequence layer*. The sequence layer consists of regular expressions over the boolean layer.

(3) *Property layer*. The property layer combines sequences to create temporal logic formulas.

(4) *Statement layer*. The statement layer defines whether a property is to be evaluated as an obligation, an assumption, or a coverage goal.

Moreover, SVA provides local variables as one of its distinguishing features. A local variable is used to capture the value of an expression at one point within a property and hold it for later reference, after which the local variable may be reassigned. We will provide an efficient way to handle this feature in later section of this paper.

3. Synchronous Digital System and Its Polynomial Representation

In this section, we will discuss polynomial set expressions for elementary units of combinational and sequential circuits.

We mainly focus on the register transfer level (RTL) description of the circuit systems. Previous work [11, 12] has shown that any combinational circuit can be uniquely represented by a minimum order polynomial.

Here, we give an alternative dataflow-based polynomial set representation model for our assertions checking purpose whose zero set can make such a data-flow model work well.

3.1. Combinational Logic 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**) and basic logical operations, like "**AND**", "**OR**", and "**NOT**", can be modeled by the following polynomial forms in Table 1.

Table 1: Polynomial representation for arithmetic and logical units.

Arithmetic or logical operation	Polynomial set representation
$y = a + b$	$\{y - a - b\}$
$y = a - b$	$\{y - a + b\}$
$y = a * b$	$\{y - a * b\}$
$y = a/b$	$\{y * b - a\}$
$y = \mathbf{NOT}\ x$	$\{1 - x - y\}$
$y = x_1\ \mathbf{AND}\ x_2$	$\{x_1 * x_2 - y\}$
$y = x_1\ \mathbf{OR}\ x_2$	$\{x_1 + x_2 - x_1 * x_2 - y\}$

We will then discuss another important control structure in digital system, multiplexer, which can be used to join many small datapaths together to make bigger datapaths.

Basically, multiplexer provides a set of condition bits, b_i $(0 \leq i \leq B)$, 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:

$$y = MUX(x_0, x_1, \ldots, x_n, s),\ i = s \Rightarrow y = x_i,\ (0 \leq i \leq n)$$

$$\Rightarrow y - \sum_{i=1}^{n-1} \left(\prod_{j\in\{0,1,\ldots,n-1\}\{i\}} ((s-j)/(i-j))\right) * x_i, \text{ with } \prod_{n-1}^{i=0} (s-i) = 0.$$

Note that for any bit-level variable x_i $(0 \leq i \leq n)$, a limitation $\{x_i * x_i - x_i\}$ should be added for each.

3.2. Sequential Unit Modeling. In sequential circuits, flip-flops (FF) and latches are fundamental building blocks which are used as data storage elements. In this paper, we will reduce all sequential units to a simple unified model described below.

The flip-flop can be equivalently modeled as a multiplexer. We have the following proposition to state this model.

Proposition 1. *For a D-typed flip-flop model (D' is the next state of D and y is the output signal of the flip-flop) with an enable signal c, then its equivalent combinational formal is $y' = MUX(D, D', s) : i = s \rightarrow y' = x_i$, $(0 \leq i < 2$, $x_0 = D$, $x_1 = D'$), whose polynomial representation form can be described as*

$$\{(y' - D) * (c - 1), (y' - D') * c, (y' - D) * (y' - D')\} \text{ or } \{y' - D * (c - 1) - D' * c\}.$$

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

This establishes the proposition. □

Proposition 2. *Let D be an FF model (D' is the next state of D and y is the output signal of the flip-flop) without enable signal; then its equivalent combinational formal polynomial algebraic model can be described as $\{y' - D\}$.*

Proof. Straightforward. □

Proposition 3. *Let D be an FF model (D' is the next state of D and y is the output signal of the flip-flop) with a reset signal; then its equivalent combinational formal polynomial algebraic model can be described as $\{y' - D * (1 - reset)\}$.*

Proof. Straightforward. □

4. Cycle-Based Simulation and Sequential Unrolling

In this section, we will sketch the underlying system model for symbolic simulation used in our work.

Basically, symbolic simulation takes in variables $(x_1, x_2, \ldots, x_m)$, as shown in Figure 1, called free variables, as input and produces output expressions in terms of the variables $(y_1, y_2, \ldots, y_n)$. The free variable can be either bit-level, integer, or constant 0 and 1. The symbolic simulator simulates simultaneously the entire set of the points that the input variables can take on.

Cycle-based symbolical simulation of a circuit for n cycles can be regarded as unrolling the circuit n times. The unrolled circuit is a pure combinational one, and the ith copy of the circuit represents the circuit at cycle i. Thus, the unrolled circuit contains all the symbolic results for n cycles.

The simulation process can be described as follows. Firstly, cycle-based symbolical 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 variable or a symbolic constant. Then, at the end of a simulation step, the expressions representing the next-state functions generally undergo a transformation-based optimization. Afterwards the newly generated functions are used as present state for the next state of simulation.

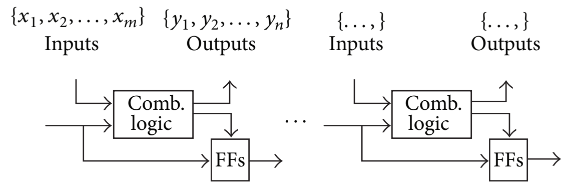


FIGURE 1: Symbolic simulation model.

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). Time-frame expansion is achieved by connecting the PPIs of the current time frame to the corresponding PPOs of the previous time frame.

5. SVA Expressions with Polynomial Set

In this section, we will discuss concurrent assertions and their temporal layer representation models.

As mentioned previously, 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.

We begin with the introduction of polynomial notions for various layers modeling.

5.1. Polynomial Notions and Algebraization. For clarity, we introduce the following symbolic notions for algebraic representation used in the rest of this paper.

(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 $[\![f]\!]_{[t]}^{[m\cdots n]}$.

Note that more detailed description of time range will be discussed in next subsection.

For a given sequential circuit, to illustrate the sequential modeling for a given cycle number clearly, we will define the following indexed polynomial representation model for the ith cycle by using the above notions.

Definition 4 (indexed polynomial representation). Let $x_{[r][l]}$ ($0 \le r \le R$, $R \in \mathbf{N}$) denote the input signals for the lth clock cycle, $m_{[s][l]}$ ($0 \le s \le S$, $S \in \mathbf{N}$) denote the intermediate signals, and $y_{[t][l]}$ ($0 \le t \le T$, $T \in \mathbf{N}$) denote the output signals. We define the time-frame expansion model $[\![M]\!]$ as **indexed polynomial representation** for the sequential circuit C as the following:

$$[\![M]\!] = \left\{\bigcup_{i=0}^{n} [\![\mathbf{C}]\!] \left(\ldots, [\![x_{[r][i]}]\!], \ldots, [\![m_{[s][i]}]\!], \ldots, [\![y_{[t][i]}]\!], \ldots\right)\right\}, \tag{1}$$

where, $[\![\mathbf{C}]\!](\ldots, [\![x_{[r][i]}]\!], \ldots, [\![m_{[s][i]}]\!], \ldots, [\![y_{[t][i]}]\!], \ldots)$ denotes the ith time-frame model of the original circuit and n denotes the number of clock cycle.

In the following, we will discuss Boolean layer modeling based on above polynomial notions.

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 SVA, the following are valid Boolean expressions:

$arrayA == arrayB$

$arrayA! = arrayB$

$arrayA[i] >= arrayB[j]$

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

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

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.

The state of a signal variable in current cycle i will be viewed as a zero of a set of polynomials. We have the zero set building rules.

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

ZS-2 : alternatively, the state of $x == 0$ (x is active low at time-frame i) can be represented by polynomial $\{x_{[i]}\}$. That is, $[\![x_{[i]}]\!] = \{x_{[i]}\}$.

ZS-3 : For any signal x and y, we have: $[\![x_{[i]} \wedge y_{[j]}]\!] = \{(x_{[i]}-1), (y_{[j]}-1)\}$ and $[\![x_{[i]} \vee y_{[j]}]\!] = \{(x_{[i]}-1)(y_{[j]}-1)\}$ hold. Where i and j are time frames.

5.2. Time Range and Its Unrolled Model. We will then 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 modeled with polynomial set.

In [7], a time range calculating algorithm is provided. Here, we will introduce some related definition and special handling for algebraization purpose. 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 compiling time, 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 $n\ge m\ge 0)$,

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

Let S := "$a\#\#[m:n]b$" ($m,n\in\mathbf{N}$ and $n\ge m\ge 0$) be a sequence; we will demonstrate how to equivalently derive its polynomial representation step by step through this example.

Assume the starting time frame is t; then: the sequence S will start $(n-m+1)$ sequences of evaluation which are

$$\Longrightarrow\begin{cases} a\#\#mb \\ a\#\#(m+1)b \\ \vdots \\ a\#\#(n-m+1)b, \end{cases} \quad \text{respectively.} \tag{2}$$

Correspondingly, we can derive their algebraic forms as follows:

$$\Longrightarrow\begin{cases} [\![a\#\#mb]\!]_{[t]} = [\![a_t\wedge b_{t+1}]\!] \\ [\![a\#\#(m+1)b]\!]_{[t]} = [\![a_t\wedge b_{t+2}]\!] \\ \vdots \\ [\![a\#\#(n-m+1)b]\!]_{[t]} \\ \quad = [\![a_t\wedge b_{t+n-m+1}]\!], \end{cases} \quad \text{respectively.} \tag{3}$$

Then we have the equivalent form of above representation set as

$$\Longrightarrow [\![S]\!]_{[t]} = [\![(a_t\wedge b_{t+m})\vee\cdots\vee(a_t\wedge b_{t+n})]\!]. \tag{4}$$

Finally, we can have their polynomial set forms by following the zero set building rules discussed previously: **ZS-1**, **ZS-2**, and **ZS-3**.

Similarly, we can also derive the equivalent algebraic representation of $[\![S_1\#\#[m:n]S_2]\!]_{[t]}$ by following the same steps.

5.3. 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:

$$T = [l\cdots h] = \{xl\le x\le h\} \quad (\text{here, } x,l,h\in\mathbf{N}). \tag{5}$$

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 range, of T_1 and T_2 is defined as $T_1+T_2 = [(l_1+l_2)\cdots(h_1+h_2)]$.

Definition 5 (maximum sequential depth). The maximum sequential depth of a SVA expression F or a sequence, written $dep(F)$, is defined recursively:

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

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

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

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

(5) $dep(F_1$ **and** $F_2) = max(\text{dep}(F_1), dep(F_2))$, if F_1, F_2 are sequences of SVA;

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

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

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

Note that if $dep(F) = [m\cdots n]$, we have $dep(F)\cdot l = m$ and $dep(F)\cdot h = n$.

5.4. 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 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.

In the following, we will discuss the algebraic representation of the supported sequence operators.

(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. The cycle delay in SVA has two basic forms.

(a) A "##n" followed by a number n specifies the n cycles delay from the current clock cycle to the beginning of the sequence that follows.

(b) A "##$[m\cdots n]$" followed by a rang $[m\cdots n]$ specifies the $[m\cdots n]$ cycles delay from the current clock cycle to the beginning of the sequence that follows.

It is easy to derive the algebraic representation forms for this operator by following the method discussed in Section 5.2.

(2) **intersect** operator. The two operands of intersect operator are sequences. The requirements for match of the intersect operation are as follows:

(a) both operands must match,

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

"R_1 **intersect** R_2" denotes that 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.

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

(3) **and** operator. This operation "R_1 **and** R_2" 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 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 that both R_1 and R_2 hold for the same number cycles. Then, the matches of "R_1 **and** R_2" must satisfy the following:

(a) the start point of the match of R_1 must not be earlier than the start point of the match of R_2;

(b) the end point of the match of R_1 must not be 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 complete in the same cycle or in different cycles.

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

(5) **repetition** operator. 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 expressions.

The syntax of repetition operator can be illustrated as the following:

non_consecutive_rep ::= [= *const_or_range_expr*].

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

From above discussion, we have the following rules for polynomial set-based representation.

Rule 1. Assume that R, R_1, and R_2 are valid SVA sequences; the rules in Table 2 can be used to construct the corresponding polynomial set representations.

5.5. Local Variables Representation. An important feature in SVA is that variables can be used in assertions which is highly useful in pipelined designs. These local variables are optional and local to properties. They can be initialized, assigned or reassigned a value, operated on, and compared to other expressions.

Table 2: Polynomial representation for sequence operators.

NO.	Sequence	Polynomial set representation
1	$a\#\#n\ b$ $a\#\#[m\cdots n]\ b$	$\left(\left\{\llbracket a\rrbracket_{[t]}, \llbracket b\rrbracket_{[t+n]}\right\}\right)$ $\vee_{i=m}^{n}\left(\left\{\llbracket a\rrbracket_{[t]}, \llbracket b\rrbracket_{[t+i]}\right\}\right)$
2	R_1 **intersect** R_2	$\left\{\llbracket R_1\rrbracket_{[t]}^{dep(R_1)}\right\} \wedge \left\{\llbracket R_2\rrbracket_{[t]}^{dep(R_1)}\right\}$
3	R_1 **and** R_2	$\vee_{i=l_1}^{h_1}\vee_{j=l_2}^{h_2}\left(\left\{\llbracket R_1\rrbracket_{[t]}^{i}\right\} \wedge \left\{\llbracket R_2\rrbracket_{[t]}^{j}\right\}\right)$ Here, $l_i = dep(R_i)\cdot l; h_i = dep(R_i)\cdot h$ $(1 \leq i \leq 2)$
4	R_1 **or** R_2	$\vee_{i=l_1}^{h_1}\vee_{j=l_2}^{h_2}\left(\left\{\llbracket R_1\rrbracket_{[t]}^{i}\right\} \vee \left\{\llbracket R_2\rrbracket_{[t]}^{j}\right\}\right)$ Here, $l_i = dep(R_i)\cdot l; h_i = dep(R_i)\cdot h$ $(1 \leq i \leq 2)$
5	$R\ [= n]$	$\vee_{i=0}^{n}\left(\left\{\llbracket R_1\rrbracket_{[t]}^{i*dep(R)}\right\}\right)$

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

A simple example of property with a local variable is specified below:

```
property dataout;
  logic[31 : 0] x;
  @(posedge clk)($rose(load), x = data) |=> ##[0 :
  5]
(ready && dout && data == x) |=> ##[0 :
3]done && q == x;
endproperty
```

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, or objects visible by the sequence (or property)).

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, the polynomial set representation for property *dataout* will be translated in the following form:

$$\llbracket dataout\rrbracket_{[t]} = \left\{\cdots\left(\llbracket data\rrbracket_{[t]} - x\right), \ldots, \left(\llbracket q\rrbracket_{[t]} - x\right)\right\}. \tag{6}$$

5.6. Property Operator Modeling. In SVA, a property that is a sequence evaluates to true if, and only if, there is a nonempty match of the sequence. Property expressions are built using sequences, other sublevel properties, and simple Boolean expressions via property connectives.

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

These individual elements are combined using property operators which are listed as follows:

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 | −> P) ("implication" form);
| (R | => P) ("implication" form);
| **disable iff**(b) P ("reset" form)

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

(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 implication operator takes a sequence as its antecedent and a property as its consequent. For each successful match of the antecedent sequence, the consequence sequence 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.

(2) **NOT** operator. "**NOT** P" states that the evaluation of the property returns the opposite of the evaluation of the underlying P.

(3) **AND** operator. "P_1 **AND** P_2" states that the property evaluates to **true** if, and only if, both P_1 and P_2 evaluate to **true**.

(4) **OR** operator. "P_1 **OR** P_2" states that the property evaluates to **true** if, and only if, at least one of P_1 and P_2 evaluates to **true**.

(5) **IF-ELSE** operator. This operator has two valid forms which are listed as follows.

(a) **IF** ($dist$) P_1. A property of this form evaluates to **true** if, and only if, either $dist$ evaluates to **false** or P_1 evaluates to **true**.

(b) **IF** ($dist$) P_1 **ELSE** P_2. A property of this form evaluates to **true** if, and only if, either $dist$ evaluates to **true** and P_1 evaluates to **true** or $dist$ evaluates to **false** and P_2 evaluates to **true**.

From the previous discussion, we have the checking rules for property reasoning.

Rule 2. Assume that P, P_1, and P_2 are valid SVA properties, S denotes a SVA sequence, and M denotes the system model to be checked; the following transformation rules in Table 3 are used to construct the corresponding verification process, where $AssChk(Model, Property)$: {**ture**, **false**} is a self-defined justification function which can take either polynomial sets or original models as input arguments and is used to determine wether a given $Property$ holds or not with respect to a circuit model $Model$. Additionally, ¬, ∨, and ∧ are standard logical connectives.

Table 3: Polynomial representation for property operators.

Property operation	Polynomial set representation
P_1 **OR** P_2	$AssChk([\![M]\!],[\![P_1]\!]) \vee AssChk([\![M]\!],[\![P_2]\!])$
P_1 **AND** P_2	$AssChk([\![M]\!],[\![P_1]\!]) \wedge AssChk([\![M]\!],[\![P_2]\!])$
NOT P	$\neg AssChk([\![M]\!],[\![P]\!])$
IF ($dist$) P_1	$\neg AssChk([\![M]\!], dist) \vee AssChk([\![M]\!],[\![P_1]\!])$
IF ($dist$) P_1 **ELSE** P_2	$(AssChk([\![M]\!], dist) \wedge AssChk([\![M]\!],[\![P_1]\!])) \vee (\neg AssChk([\![M]\!], dist) \wedge AssChk([\![M]\!],[\![P_2]\!]))$
$S\vert -> P$	$if\ (\neg AssChk([\![M]\!],[\![S]\!]))\ return$ **true** $else\ \left(AssChk\left([\![M]\!],[\![S]\!]_{[t]}\right) \wedge AssChk\left([\![M]\!],[\![P]\!]^{dep(P)}_{[t+dep(S)-1]}\right)\right)$
$S\vert => P$	$if\ (\neg AssChk([\![M]\!],[\![S]\!]))\ return$ **true** $else\ \left(AssChk\left([\![M]\!],[\![S]\!]_{[t]}\right) \wedge AssChk\left([\![M]\!],[\![P]\!]^{dep(P)}_{[t+dep(S)]}\right)\right)$

5.7. Constrained Subset of SVAs. SystemVerilog assertions formally define the syntax and grammar of an assertion language that can define design properties and constraints, including temporal (i.e., spread over several cycles) characteristics. The atoms of the SVA concurrent assertions are so called sequences. A sequence is a regular expression that describes the behavior of signals over time. A property of SVA is composed of sequences by property operators.

As described in [13], the total set of SVA can be 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, only a subset of these groups can be supported.

Formally, we will give the following definition for this subset.

Definition 6 (constrained subset of SVA). The constrained subset of SVA supported by algebraic symbolic polynomial representation is defined recursively as the following:

(1) $R ::= b|(v = e)|(R)|(R\#\#m)|(R\#\#[m\cdots n]R)|(R$ **or** $R)|$ $(R_1$ **intersect** $R_2)|R[= n]|(R_1$ **and** $R_2)$,

(2) P ::= $R|(P)|$ **NOT** P $|(P_1$ **AND** $P_2)$ | $(P_1$ **OR** $P_2|(R$ | $-> P)|$ $(R$ |=> $P))$,

where b denotes a Boolean expression and all Boolean operators are supported, R (R_1, R_2) denotes a sequence, P (P_1, P_2) denotes a property, v denotes a local variable name, and e denotes an expression.

For sequence operators, unspecified upper bound time range and **first-match** operator are excluded from the constrained subset. Additionally, property operator **disable iff** will not be supported also.

In this paper, only a constrained subset of SVA language defined above can be supported by our method.

Firstly, we translate the properties described by the constrained subset of SVA 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 finite polynomial set.

Then, the verification problem is reduced to proving zero set inclusion relationship which can be resolved by characteristic set-based algebraic approaches.

6. Characteristic Set Based Verification Principle and Algorithm

In this section, we will discuss the checking algorithm based on Wu's characteristic set. We first recall some fundamental knowledge in Wu's method.

6.1. Wu's Method Preliminary. Wu's method [14, 15] is an efficient algorithm for solving multivariate polynomial equations introduced by Wen-Tsun Wu. It also has succeeded in mechanical geometry theorem proving. For further details and elementary introduction, we refer the reader to [16].

Let $P_i = \mathbf{k}[x_1, \ldots, x_i]$ denote the polynomial ring in variables $x_1, x_2, \ldots, x_i$ with coefficient $\mathbf{k}$, for every i in the range $\{1, \ldots, n\}$, and $\mathbf{k}$ is an algebraically closed field.

In what follows, without loss of generality, we assume that the given variables ordering is fixed as

$$x_1 \prec x_2 \prec \cdots \prec x_n. \tag{7}$$

Assume $p \in \mathbf{k}[x_1, \ldots, x_n]$ and let $deg(p, x_i)$ be the maximum degree of p with respect to x_i.

Definition 7. Let $p \in P_i$; the greatest variable $v \in \{x_1, \ldots, x_i\}$ is called the *main variable* of p such that $deg(p, v) \neq 0$, denoted by $mvar(p)$. The biggest index i is called the class, denoted by $class(p)$. If we assume p as a univariate polynomial with $mvar(p) = x_i$, we can write $p = cx_i^d + r$, where $d = deg(p, x_i)$. In this case, c and d are, respectively, the *initial* and the *main degree* and are denoted by $init(p)$ and $mdeg(p)$.

Definition 8 (pseudo division). For any two polynomials $f, g \in P_i$, with $class(f) = j$. There exist a nonnegative number α and polynomials q and r, such that $init(f)^{\alpha} g = qf + r$. Where r is denoted by $prem(g, f, x_j)$. We denote the remainder r on pseudo division (pseudo remainder) of f by g with respect to the variable y by $prem(f, g, y)$.

Definition 9 (Wu's characteristic set). Let PS and CS be two polynomial system, and CS is a triangular set; a triangular set is called a characteristic set of PS if

$$\forall p \in PS, \quad prem(p, CS) = 0. \tag{8}$$

Theorem 10 (zero decomposition theorem). *Let $Zero(PS)$ denote the zero set of PS and CS be a characteristic set of P; then*

$$Zero(PS) = Zero\left(\frac{CS}{init(CS)}\right) \cup Zero(PS_i), \tag{9}$$

where $PS_i = PS \cup init(CS)_i$.

Now we briefly outline the essential steps about Wu's method on geometry theorem proving [17].

(1) Algebraize a geometric theorem by converting it into a system of algebraic equations. We will translate the theorem's hypotheses into a set of hypotheses equations $\{f_1, \ldots, f_r\}$,

$$\begin{aligned} f_1 &= f_1(x_1, \ldots, x_n), \\ f_2 &= f_2(x_1, \ldots, x_n), \\ &\vdots \\ f_r &= f_r(x_1, \ldots, x_n), \end{aligned} \tag{10}$$

and the theorem's conclusion into a polynomial $g(x_1, \ldots, x_n)$.

(2) Triangulize the system of hypotheses equations using pseudo-division. Then we will obtain the hypothesis equations denoted as

$$\begin{aligned} f_1 &= f_1(u_1, \ldots, u_d, x_1), \\ f_2 &= f_2(u_1, \ldots, u_d, x_1, x_2), \\ &\vdots \\ f_r &= f_r(u_1, \ldots, u_d, x_1, \ldots, x_r). \end{aligned} \tag{11}$$

(3) Perform successive pseudo division on the transformed hypotheses in triangular form $\{f_1, f_2, \ldots, f_r\}$ by the conclusion equation $g(x_1, \ldots, x_n)$, yielding a final remainder r_0:

$$r_0 = prem\left(\cdots prem\left(g, f_r, x_{p_r}\right), \ldots, f_1, x_{p_1}\right). \tag{12}$$

If this final remainder $r_0 \equiv 0$, we will say that the conclusion g follows from the hypotheses $\{f_1, \ldots, f_r\}$.

As mentioned above, the characteristic set of a polynomial system is a special triangular form. We can use the characteristic set of hypotheses equations as the triangular form, denoted by $charset(f_1, \ldots, f_r)$. Roughly speaking, the soul of Wu's method on theorem proving is to check whether the zero set of the expected consequent equations includes the zero set of hypotheses equations or not.

Our checking method is based on algebraic geometry that studies of geometric objects arising as the common zeros of collections of polynomials. The aim is to find polynomials whose zeros correspond to pairs of states in which the appropriate assignments are made.

Let $zero(PC)$ and $zero(PS)$ denote the zero set for circuit polynomial set PC and assertions polynomial set PS respectively. Let $C \vDash s$ denote that assertion s holds under model C.

We aim to attempt to determine whether assertion s holds or not. Therefore, we have

$$(s \vDash C) \Longleftrightarrow Zero(PC) \subseteq Zero(PS). \tag{13}$$

Theorem 11. *Suppose that $G = [\mathscr{A} \Rightarrow \mathscr{C}]$ is a property, $\mathscr{A}$ is the precondition of $\mathscr{C}$, and M is a system model. Let PA and PM be the polynomial set representations for $\mathscr{A}$ and M, respectively, constructed by previous mentioned rules. PC is the polynomial set representation for $\mathscr{C}$.*

Let $H = A \cup M$, $PH = PA \cup PM = \{h_1, h_2, \ldots, h_s\} \subseteq k[x_1, \ldots, x_n]$, $CS = charset(PH)$; $charset(PH)$ denotes the characteristic set of PH, and $prem(PC, CS)$ is the function which computs the remainder by successive pseudo division; then we have

$$prem(PC, CS) == 0 \Longleftrightarrow (M \models G). \quad (14)$$

Proof. Firstly, let $zero(PA)$ and $zero(PM)$ denote the zero sets for circuit polynomial set $\mathscr{A}$ and assertions polynomial set M, respectively. Since $PH = PA \cup PM = \{h_1, h_2, \ldots, h_s\} \subseteq k[x_1, \ldots, x_n]$, let $zero(PH)$ denote the zero sets for $zero(PA \cup PM = \{h_1, h_2, \ldots, h_s\})$. Let $H \vDash C$ denote that the assertion C holds under model H.

Our aim is to attempt to determine whether assertion C holds or not. Thus, we have

$$(M \vDash G) \Longleftrightarrow (H \vDash C) \Longleftrightarrow zero(PH) \subseteq zero(PC). \quad (15)$$

As mentioned previously, Wu's method can be used to determine the inclusion relationship of zero sets. Thus we have

$$\begin{gathered} prem(PC, charset(PH)) == 0 \\ \Longleftrightarrow zero(PH) \subseteq zero(PC). \end{gathered} \quad (16)$$

Let $CS = charset(PH)$, which denotes the characteristic set of PH; according to (15) and (16), we have

$$\left(prem(PC, CS) == 0\right) \Longleftrightarrow (M \models G). \quad (17)$$

This establishes the theorem. □

6.2. Checking Algorithm. From above discussion, we have the following core decision algorithm.

Algorithm 12 (Assertion Checking: $AssChk(\mathbf{M}, G)$).

```
Input: Circuit model M, an assertion $G = [\mathscr{A} \Rightarrow \mathscr{C}]$;
Output: Boolean: true or false;
BEGIN
  /* Step 0: initialize input signals via testbench */
00  InitSignals();
01  $\mathbf{M} = \emptyset$; $PS_A = \emptyset$; $H = \emptyset$; $PS_C = \emptyset$;
  /* Step 1: build polynomial set for antecedent $\mathscr{A}$ */
02  $PS_{\mathscr{A}} = BuildPS(\mathscr{A})$;
  /* Step 2: build polynomial set for consequent $\mathscr{C}$ */
03  $PS_{\mathscr{C}} = BuildPS(\mathscr{C})$;
  /*Step 3: calculate the $PS_{\mathscr{A}} \cup \mathbf{M}$*/
04  $H = PS_{\mathscr{A}} \cup \mathbf{M}$;
  /*Step 4: calculate the characteristic set of $H$*/
05  $CS := Charset(H)$;
  /*Step 5: check the final remainder by successive
pseudo-division */
06  if($prem(PS_{\mathscr{C}}, CS) \neq 0$){
07    return false;
    }
08  return true; /* Assertion does hold */
END
```

7. Case Study and Experiment

In this section, we will firstly study a simple circuit to show how the SVA properties are verified by polynomial representations and characteristic set computation-based approaches. We then show a scalable experiment with the classic synchronous arbiter circuit to evaluate the performance of our approach.

7.1. Circuit and Assertion Modeling. As an example, consider the synchronous circuit in Figure 2, whose polynomial set can be constructed as follows:

$$\begin{aligned} Set_{adder} = \{ & \\ & f1 = \left\{m3' - (1 - m1)\right\}, \\ & f2 = \left\{m2' - m3\right\}, \\ & f3 = \left\{m1' - m2\right\}, \\ & f4 = \{m4 - (1 - m1) * m3\}, \\ & f5 = \{m5 - (1 - m3) * (1 - m2)\}, \\ & f6 = \{m6 - m1 * m2\}\}. \end{aligned} \quad (18)$$

Throughout this paper, an internal variable name $m_{[i]}$ refers to the corresponding signal m in the original circuit, and the subscript denotes the time-frame i (or the ith clock cycle). For convenient, variable m' (i.e., $m_{[i+1]}$) denotes the next state of m ($m_{[i]}$) by omitting time-frame i.

To illustrate the problem clearly, we define polynomial set representation $PM_{[i]}$ for the ith time frame as follows:

$$[\![PM_{[i]}]\!] = \{f1_{[i]}, f2_{[i]}, f3_{[i]}, f4_{[i]}, f5_{[i]}, f6_{[i]}\}. \quad (19)$$

Assume 8 cycles are needed to check; therefore, we then have the model representation: $[\![PM]\!] = \bigcup_{i=0}^{8} [\![PM_{[i]}]\!]$.

For any boolean variable, we must add an extra constraint polynomial: for all $a, \mathscr{V}(a) \in \{0, 1\} : \{a * a - a\}$.

The circuit produces 3-phase nonoverlapping outputs, which can decode the overlapping outputs v_1, v_2, and v_3 to nonoverlapping outputs A, B, and C as shown in Figure 3.

Assume the initial values of the registers A, B, and C are all 0, they will be $\{1, 0, 0\}$, respectively, after 1 cycle running and $\{1, 1, 0\}$ after 2 cycles. This property can be specified by the following SVA assertions:

```
property PA;
  (m3 = 0 && m2 = 0 && m1 = 0)
  |=> ##1(m3 == 1 && m2 == 0 && m1 == 0)
  |=> ##1(m3 == 1 && m2 == 1 && m1 == 0)
  |=> ##1(m3 == 1 && m2 == 1 && m1 == 1);
endproperty
```

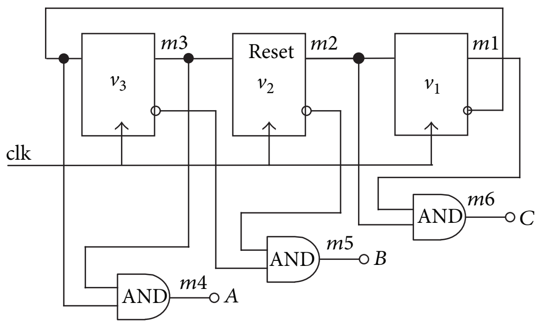


Figure 2: A 3-stage johnson counter.

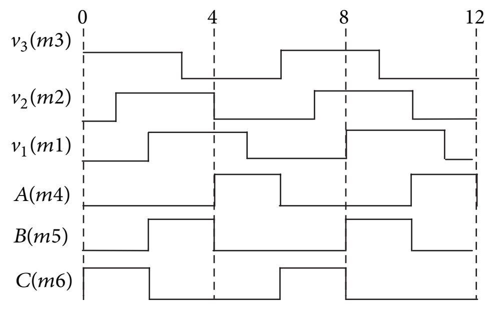


Figure 3: A 3-stage johnson counter waveform.

property *PB*;

$m6| => \#\#2\ m5\ |=> \#\#2m4$;

endproperty

property *PC*;

$m5\#\#!m5\ | => \#\#[1 : 3]\ !m5$;

endproperty

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

7.2. Assertion Checking Using MMP. We ran this example by using MMP [18] (Mathematics Mechanization Platform). MMP is a group of symbolic computation and automated theorem proving softwares based on Wu's Method. Before running, we manually translated all models into polynomials for this example.

In this paper, all experiments are conducted on a PC with an intel 2.40 GHz CPU (intel i5 M450) and 1024 MB of memory.

As shown in MMP outputs, the given circuit has been modeled as polynomial set CM (its characteristic set is denoted by cs), and the assertion results as $(m3_{[2]} - 1)$. The checking process is listed below:

[> $CM := [\cdots]$;

$/*\{f1_{[i]}, f2_{[i]}, f3_{[i]}, f4_{[i]}, f5_{[i]}, f6_{[i]}\}$ Circuit Model $*/$

[> $vars := (m1_{[0]}, m2_{[0]}, m3_{[0]}, m4_{[0]}, m5_{[0]}, m6_{[0]},$
$m1_{[1]}, m2_{[1]}, m3_{[1]}, m4_{[1]}, m5_{[1]}, m6_{[1]},$
$m1_{[2]}, m2_{[2]}, m3_{[2]}, m4_{[2]}, m5_{[2]}, m6_{[2]}, \ldots$

[> $cs := charset(CM, vars)$;

[> $c1 := (m3_{[2]} - 1)$;

[> $ret := premas(c1, cs, vars) = 0$

From the running result, the return value of ret is 0 which means cs can be pseudo divided with no remainder by $(m3_{[2]}-1)$. Similarly, the expected polynomials $(m2_{[2]}-1)$ and $(m1_{[2]})$ both return 0. Thus, from the previously mentioned verification principles, it is easy to conclude that the assertion holds under this circuit model after 2 cycles.

The example took about 0.07 seconds and 810 KB of memory when applying Wu's method in MMP environment. This case can be a fairly complete illustration of how the checking algorithm works.

7.3. Experiment with Classic Arbiter Circuit. The synchronous arbiter circuit [19], depicted in Figure 4, is one of the most popular test cases of model checkers. We also test our approach with this circuit currently to illustrate the performance of our approach. The convert algorithm which can automatically translate the circuit netlist into polynomial set has been implemented in C++.

This arbiter circuit is used to grant access on each clock cycle to a single client among a number of clients contending for the use of common resources.

Here, we briefly explain how the arbiter works. For more detailed information regards to the arbiter, please refer to [19]. The basic part of the arbiter is the colored cell which is repeated k times to form a round robin chain. Each cell has a request input req_i ($1 \le i \le k$, here, k denotes the number of client) and an acknowledge output ack_i. The grant output of cell i is passed to cell i and indicates that no clients of index less than or equal to i are requesting. Hence a cell i may assert its acknowledge output ack_i if its grant input gi_i is asserted. Each cell has a register T which stores one when the token is present. The T registers form a circular shift register which shifts up one place each clock cycle.

Each cell also has a register W for waiting which is set to one when the request input is asserted and the token is present.

The register remains set while the request persists until the token returns. At this time the cell's override oo_i and acknowledge outputs ack_i are asserted.

For clarity, we use the first subscript of a variable name which denotes the cell number j and the second subscript which denotes the current time-frame i in this example.

The corresponding polynomial set for each cell j ($0 \le j \le k-1$) and time-frame i ($1 \le i \le n$) can then be constructed as follows:

$$f_1 : \left\{w_{[j][i]} + t_{[j][i]} - w_{[j][i]} * t_{[j][i]} - m1_{[j][i]}\right\}$$

$$f_2 : \left\{m2_{[j][i]} * req_{[j][i]} - m2_{[j][i]}\right\}$$

$$f_3 : \left\{w'_{[j][i]} - m2_{[j][i]}\right\}$$

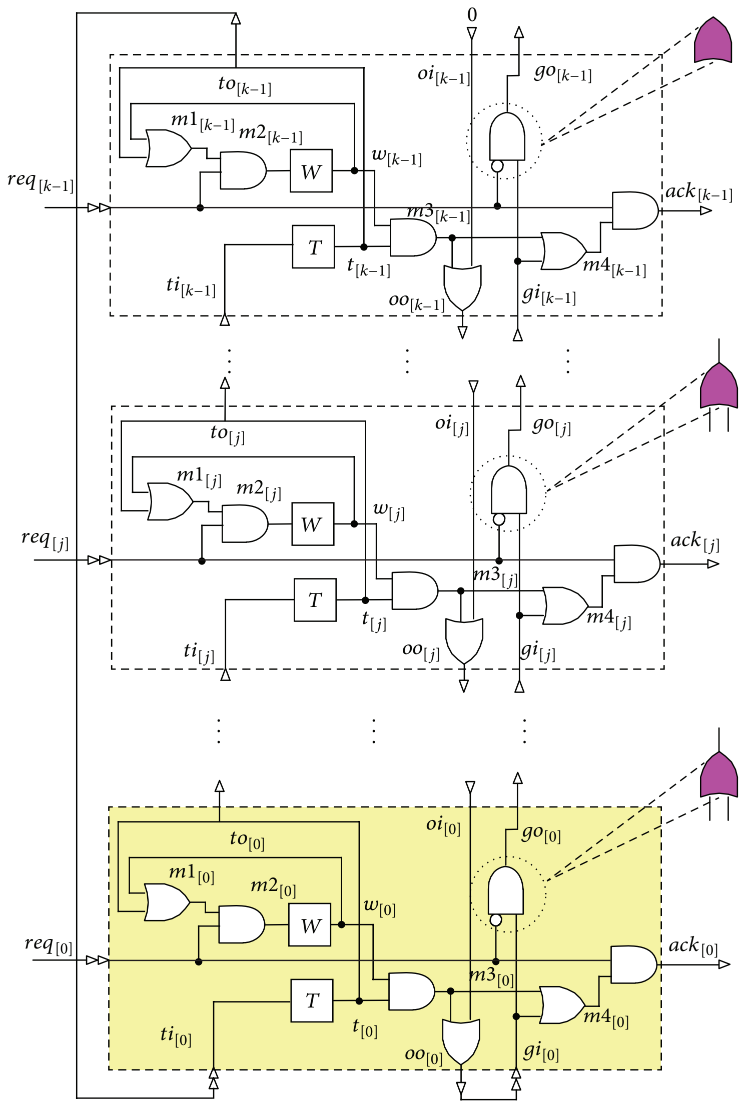


FIGURE 4: Chain of synchronous arbiter circuit.

$$f_4 : \left\{t'_{[j][i]} - ti_{[j][i]}\right\}$$
$$f_5 : \left\{m3_{[j][i]} - t_{[j][i]} * w_{[j][i]}\right\}$$
$$f_6 : \left\{m3_{[j][i]} + oi_{[j][i]} - m3_{[j][i]} * oi_{[j][i]} - oo_{[j][i]}\right\}$$
$$f_7 : \left\{go_{[j][i]} - \left(1 - req_{[j][i]}\right) * oo_{[j][i]}\right\}$$
$$f_8 : \left\{gi_{[j][i]} + m3_{[j][i]} - gi_{[j][i]} * m3_{[j][i]} - m4_{[j][i]}\right\}$$
$$f_9 : \left\{ack_{[j][i]} - m4_{[j][i]} * req_{[j][i]}\right\}. \tag{20}$$

Additionally, the chain relation between cells can be modeled as below:

$$l_1 : \left\{go_{[j][i]} - gi_{[j+1]\ [i]}\right\}$$
$$l_2 : \left\{oi_{[j][i]} - oo_{[j+1]\ [i]}\right\}$$
$$l_3 : \left\{to_{[j][i]} - ti_{[j+1]\ [i]}\right\}$$
$$l_4 : \left\{to_{[k-1]\ [i]} - ti_{[0]\ [i]}\right\}$$
$$l_5 : \left\{oo_{[0][i]} - gi_{[o][i]}\right\}. \tag{21}$$

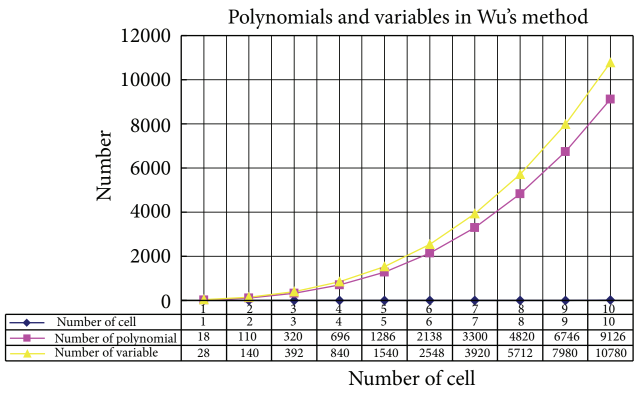
The desired properties of the arbiter circuits can be described as follows:

(1) no two acknowledge outputs are asserted simultaneously,

(2) every persistent request is eventually acknowledged,

(3) acknowledgment is not asserted without request.

In the following description, if clear from the context, the time-frame index will be omitted for convenience. Note that s, t $(0 \le s, t \le k-1)$ and $ack_{[s]}, ack_{[t]} \in \{0, 1\}$. Intuitively, properties expressed in CTL can be listed as follows:



	1	2	3	4	5	6	7	8	9	10
Number of cell	1	2	3	4	5	6	7	8	9	10
Number of polynomial	18	110	320	696	1286	2138	3300	4820	6746	9126
Number of variable	28	140	392	840	1540	2548	3920	5712	7980	10780

FIGURE 5: Polynomials and variables for safety property checking.

(1) $\bigwedge_{s\neq t} AG\neg(ack_{[s]} \wedge ack_{[t]})$,

(2) $\bigwedge_{s} AGAF(req_{[s]} \Rightarrow ack_{[s]})$,

(3) $\bigwedge_{s} AG(ack_{[s]} \wedge req_{[s]})$.

Note that the arbiter can handle the access of n clients to a common resource. If a client asserts a request, the client will wait at most $2n$ clock cycles before it is served. Therefore, in our test, it is sufficient to verify the properties that the circuit is unrolled $2n$ time frames.

Evidently, these properties can be equivalently expressed in SVA codes. In [7], the authors have provided a detailed assertion description for the case that client number is 3.

In this example, we will follow their idea and explain how to translate them into algebraic polynomials as below:

$$\begin{aligned}
&\bigwedge_{s\neq t} AG\neg\left(ack_{[s]} \wedge ack_{[t]}\right) \\
&\Longleftrightarrow \\
&ack_{[0]} + \cdots + ack_{[s]} + \cdots + ack_{[k-1]} = 1 \text{ or } 0 \\
&\Longleftrightarrow \\
&\left\{ack_{[0]} + \cdots + ack_{[s]} + \cdots + ack_{[k-1]} - 1\right\} \\
&\text{or } \left\{ack_{[0]} + \cdots + ack_{[s]} + \cdots + ack_{[k-1]}\right\}.
\end{aligned} \tag{22}$$

Let $\mathscr{C}_1 = \{ack_{[0]} + \cdots + ack_{[s]} + \ldots + ack_{[k-1]} - 1\}$ and $\mathscr{C}_2 = \{ack_{[0]} + \cdots + ack_{[s]} + \ldots + ack_{[k-1]}\}$. The polynomial representation of consequent can then be formulated as $\mathscr{C} = \mathscr{C}_1 \vee \mathscr{C}_2$.

Assume $G = [\mathscr{A} \Rightarrow \mathscr{C}]$ and let $G_1 = [\mathscr{A} \Rightarrow \mathscr{C}_1]$ and $G_2 = [\mathscr{A} \Rightarrow \mathscr{C}_2]$. If $G_1 \vee G_2$ holds, then G holds.

Similarly, the precondition of this property is that only one single token is present in this circuit chain. We have the polynomial representation of antecedent: $\{t_{[0]}+\cdots+t_{[s]}+\cdots+t_{[k-1]} - 1\}$ $(t_{[i]} \in \{0,1\})$.

More specifically, if the subscript for time frame is considered, we can define $\mathscr{C}_{[k][n]} := \bigwedge_{j=0}^{k-1}\{(\sum_{i=0}^{n-1} ack_{[j][i]} - 1) \vee (\sum_{i=0}^{n-1} ack_{[j][i]})\}$ and $\mathscr{A}_{[k][n]} := \bigwedge_{j=0}^{k-1}\{(\sum_{i=0}^{n-1} t_{[j][i]} - 1)\}$; here, n denotes the number of time frames and k denotes the number of cells.

In this experiment, we also tested the circuit with an artificial error that an "*AND*" gate is replaced by an "OR" gate by mistake, as shown in Figure 4. We mainly demonstrate whether or not our method is able to find bugs. The corresponding polynomial set model can be updated by replacing f_7 with $\{go_{[j][i]} + req_{[j][i]} * oo_{[j][i]} - req_{[j][i]} - oo_{[j][i]}\}$.

Consequently, in our test with 3 cells, it took only 9.37 MB of memory and 10.5 seconds to find the result $ret \neq 0$ which means the circuit did not meet the required property. Then we concluded that there must be some design errors in the circuit.

More detailed experimental results concerning polynomials and variables for safety property verification of the circuit are listed in Figure 5.

7.4. Discussion. So far, several methodologies are developed to address the verification problem. In this section, we will discuss these methods by the arbiter example shown in Figure 4.

(1) In [19], a BDD-based model checking is proposed. The performance of this method is fairly promising. The size of the transition relation increases linearly as the number of cells. But the OBDD representing the set of states grows exponentially in the number of cells. The performance of BDD-based symbolic model checking procedure is plotted in Figure 6. For more detailed information about this test case, please refer to [19].

(2) Due to advances in SAT-solving techniques [20], SAT-based bounded model checking (for short, BMC) can often handle much larger designs than BDDs. In [7], the authors presented a SAT-based BMC to verify the arbiter circuit. They found that the size of the instance for the safety property is proportional to the number of arbiter clients and the liveness property has quadratic size.

(3) In our method, the worst case complexity of our symbolic algebraic method checking is very high due to intermediate expression swell. As shown in Figure 5, the number of polynomials to be generated for liveness property checking has quadratic size of the number of cells k as well as the number of variables. Currently, from the runtime consumed, the SAT solver still shows better performance than our method. Fortunately, a great deal of optimization techniques can be applied to improve the performance.

As an example, we will explain how to apply variable projection technique to optimize the calculating process. Assume the given circuit has been modeled as polynomial set *CM* and variable order list is *tord* (its projection for the zero set on $pord \subseteq tord$ is denoted by *PC*). We will firstly calculate the projection to eliminate the unnecessary elements:

$$PC = proj\left(CircuitModel, [\,], ord, pord, \text{"}alg\text{"}\right) \tag{23}$$

and then the zero set: $wsolve(PC, ord)$. Correspondingly, the performance will be improved greatly.

8. Conclusion

In this paper, we presented Wu's method-based verification approach for SVA properties checking.

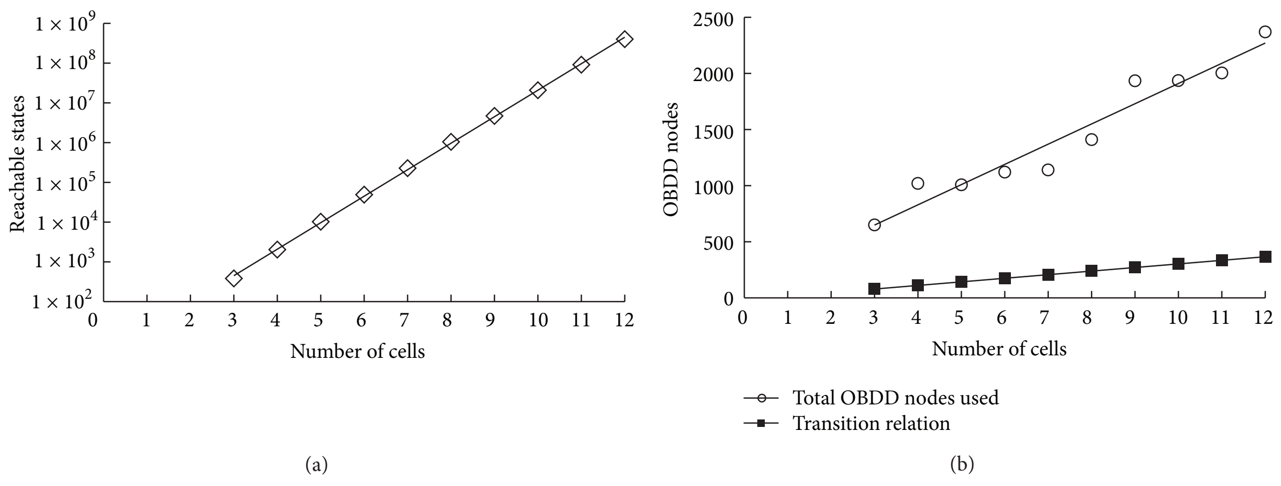


Figure 6: Performance-synchronous arbiter example.

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 MMP environment. For synchronous sequential circuits, we adapted a parameterized polynomial set modeling method based on time-frame expansion. We also defined a constrained subset of SVAs which is powerful enough for practical purpose and proposed a practical algebraization method for each sequence and property operator of this subset. This method allows users to deal with more than one state and many input combinations every cycle due to symbolic simulation. The advantage comes directly from the fact that many vectors are simulated at once using symbolic value.

Basically, our approach may provide a useful supplement to existing methods based on OBDD or SAT and may also provide important theoretical insights by allowing the application of important results in symbolic computation to the assertion checking problems. We plan to extend the work presented in several directions.

(1) We plan to optimize the solver based on Wu's method. As we know, Wu's method-based mathematics mechanization platform is a powerful computation environment for general purpose. On the one hand, we will apply variables projection and other mathematics approaches to improve the effectiveness of this algorithm. On the other hand, we also plan to implement our core steps by using the programming language in MMP.

(2) In this paper, we only test some classic cases mentioned in previous papers to illustrated the feasibility of our method currently. In the future, we plan to test the benchmarks from ISCAS'89 and ISCAS'85 and other industrial examples to provide a comprehensive evaluation of our approach and compare to the state-of-the-art where circuits with several thousands of gates and dozens of input signals.

(3) In this paper, we only concentrated our attention on the basic assertion checking algorithm, but the other important features such as counterexample generation were not discussed. We will also work on those issues in the future.

Furthermore, in the field of symbolic computation, there is a rich collection of solutions to solving polynomials, methods for inclusion of zero sets, and so forth. We are interested in applying some of the useful results and methods to verification area.

Acknowledgments

The project is supported by the Fundamental Research Funds for the Central Universities (1600-852014) and partly 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 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," in *Proceedings of the IEEE P1800/D3*, pp. 1–1304, November2011.

[3] "System Verilog," http://www.systemverilog.org/.

[4] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 2000.

[5] 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.

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

[7] 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.

[8] J. A. Darringer, "Application of program verification techniques to hardware verification," pp. 375–381, 1979.

[9] C. Spears, *System Verilog for Verification*, Springer.

[10] J. Bergeron, E. Cerny, A. Nightingale, and A. Hunter, *Verification Methodology Manual for System Verilog*, Springer.

[11] J. Smith and G. De Micheli, "Polynomial circuit models for component matching in high-level synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, pp. 783–800, 2001.

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

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

[14] W. T. Wu, "On the decision problem and the mechanization of theorem-proving in elementary geometry," *Scientia Sinica*, vol. 21, no. 2, pp. 159–172, 1978.

[15] W. T. Wu, "Basic principles of mechanical theorem proving in geometries," *Journal of Automated Reasoning*, vol. 2, no. 4, pp. 221–252, 1986.

[16] D. Wang, *Elimination Methods.*, Springer, Wien, NY, USA, 2001.

[17] J. Elias, "Automated geometric theorem proving," *The Montana Mathematics Enthusiast*, vol. 3, no. 1, pp. 3–50, 2006.

[18] X. S. Gao and Q. Lin, "MMP/geomete—a software package for automated geometric reasoning," in *Proceedings of the 4th International Workshop of the Automated Deduction in Geometry (ADG '02)*, vol. 2930 of *Lecture Notes in Computer Science*, pp. 44–66, September 2002.

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

[20] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proceedings of the 38th Design Automation Conference*, pp. 530–535, June 2001.