Assertion Checking by Combined Word-level ATPG and Modular Arithmetic Constraint-Solving Techniques

Chung-Yang (Ric) Huang Kwang-Ting (Tim) Cheng
Department of ECE, University of California, Santa Barbara, CA 93106

Abstract

We present a new approach to checking assertion properties for RTL design verification. Our approach combines structural, word-level automatic test pattern generation (ATPG) and modular arithmetic constraint-solving techniques to solve the constraints imposed by the target assertion property. Our word-level ATPG and implication technique not only solves the constraints on the control logic, but also propagates the logic implications to the datapath. A novel arithmetic constraint solver based on modular number system is then employed to solve the remaining constraints in datapath. The advantages of the new method are threefold. First, the decision-making process of the word-level ATPG is confined to the selected control signals only. Therefore, the enumeration of enormous number of choices at the datapath signals is completely avoided. Second, our new implication translation techniques allow word-level logic implication being performed across the boundary of datapath and control logic, and therefore, efficiently cut down the ATPG search space. Third, our arithmetic constraint solver is based on modular instead of integral number system. It can thus avoid the false negative effect resulting from the bit-vector value modulation. A prototype system has been built which consists of an industrial front-end HDL parser, a property-to-constraint converter and the ATPG/arithmetic constraint-solving engine. The experimental results on some public benchmark and industrial circuits demonstrate the efficiency of our approach and its applicability to large industrial designs.

1. Introduction

Simulation is still the mainstream approach for functional verification. Various coverage metrics, for example, HDL-based code coverage, are used to assess the quality of the test-bench and determine when to stop the simulation process. However, in today's design flow, test-bench is either manually derived by designers or randomly generated from the high level description of the design and/or its environment. As a result, test-bench for some corner-case bugs cannot be easily derived and high coverage could be hard to achieve. Advanced techniques such as deterministic functional vector generation and model (property) checking intend to enhance the verification quality and reduce the developing effort for this time-consuming process.

Functional vector generation can be viewed as a constraint satisfiability problem (e.g. [1]-[5]). Although these deterministic functional vector generation techniques can enhance the coverage of hard-to-detect design errors, they still inherit the fundamental deficit of simulation — the requirement of exhaustive test patterns to prove the correctness of a property. For designs with large number of inputs or properties which require long (or infinite) sequence to prove, simulation becomes a time-consuming process and usually fails to detect some tricky corner-case bugs. An alternative is to transform the simulation property to a temporal formula and use methods like model checking to formally prove its correctness.

Model checking techniques treat the designs as finite state machines and the properties as temporal relations between states. Clarke and Emerson [6] specify the properties in computational tree logic (CTL). The correctness of the property is then verified by intersecting the backward reachable states with the initial states. In general, the set of reachable states may grow exponentially as the number of registers increases. To ease the state explosion problem, several abstraction techniques are introduced either to identify the symmetry variables (e.g. [7]), or to approximate the set of reachable states (e.g. [8]).

Symbolic model checking techniques (e.g. [9]-[11]) utilize Binary Decision Diagrams (BDDs) [12] to compactly represent the set of states and the state transition functions. In general, these approaches are capable of handling larger designs than the explicit state traversal techniques even though the BDD techniques may still suffer from the memory explosion problem. Biere et al. [13] proposes an alternative, symbolic modeling checking approach by using the boolean satisfiability (SAT) techniques. Their experimental results indicate that the temporal properties within bounded time frames can be more efficiently proved by the SAT technique and the memory usage is lower as well.

In this paper, we propose a new methodology in checking the assertion type properties for RTL design verification. Our approach combines structural, word-level, sequential automatic test pattern generation (ATPG) and modular arithmetic constraint-solving techniques. This hybrid approach utilizes the strengths of both techniques in solving different kinds of constraints in the circuit. Assertion (safety) properties, like bus contention checking, internal don't-care validation, and invariant checking, are the most commonly encountered and practical properties in RTL design verification. They can be transformed into a counter-example-generation problem which can therefore be solved by an ATPG engine.

Given an HDL design, we first synthesize/map it into a netlist of high-level primitives, called RTL netlist, including (1) Boolean gates, (2) arithmetic units, (3) comparators (data-to-control), (4) multiplexors (control-to-data), and (5) memory elements (flip-flops). The circuit can then be viewed as an interconnection of control and datapath portions with some datapath-selecting and comparison-output signals as the interface. Based on this circuit model, our constraint-solving algorithm follows a two-phase process: first we apply structural ATPG technique to solve the constraints on the control logic part and propagate the implications of the partial solutions to the datapath as much as possible. Secondly, we use analytical methods based on modular arithmetic to solve the constraints on the datapath.

There are several advantages of using structural ATPG and modular arithmetic, instead of satisfiability-based (SAT) and linear programming techniques for constraint solving: (1) We can fully utilize the high-level RTL information and perform *word-level* implication on both Boolean and arithmetic gates. We have also developed several techniques to translate the implications between Boolean and arithmetic gates such that conflicting implications can

DAC 2000, Los Angeles, California

be detected as early as possible. (2) Because the signal values in circuit netlist (RTL netlist) are finite-width bit-vectors, solving arithmetic constraints in modular instead of general integral number system will not miss the solutions that come from the modulation and therefore can avoid the *false negative effect* in generating counter examples (which is *false positive* from the viewpoint of assertion checking). (3) The *abstract state variables* in the EFSM model [5] serve as good candidates of decision points in the branch-and-bound process of ATPG. Besides, whenever the search encounters a conflict in an abstract state transition or learn that a transition can lead to a hard-to-reach state, the transition in the Extended State Transition Graph (ESTG) is recorded. This recorded information is then used in the subsequent ATPG process to speed up the search.

In addition, compared to the BDD-based symbolic model checking techniques, our method is highly memory efficient. The experimental results show that the new approach is much less sensitive to the exponential growth of the state space and thus more scalable to larger designs.

We organize the rest of the paper as follows: in Section 2, we first outline our assertion checking framework. The detailed word-level ATPG and modular arithmetic constraint-solving algorithms will be described in Sections 3 and 4, respectively. The experimental results of applying our constraint solver to some public benchmark and industrial circuits will be shown in Section 5. Section 6 concludes the paper.

2. The Assertion Checking Framework

Fig. 1 shows our assertion checking framework. The input of our framework is RTL Verilog or VHDL codes. Initialization sequence is applied to derive the set of initial states. It also requires some environmental setup which defines constraints on the circuit inputs such as clock waveform(s), one-hot constraints, etc.

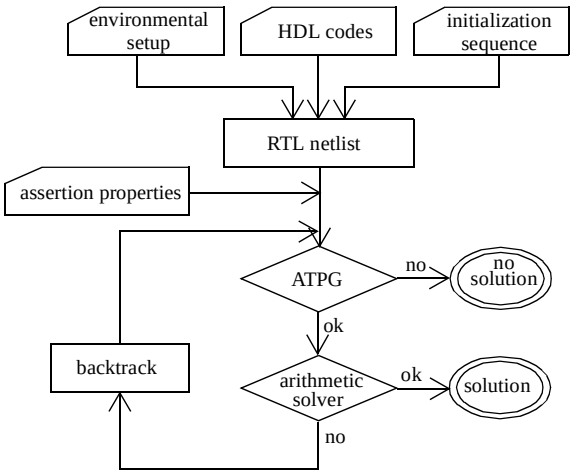

Fig. 1: The assertion checking framework.

Our framework then performs a quick synthesis to generate a flattened RTL netlist with high-level primitives. Note that in order to preserve the original design intent, we do not perform logic minimization to optimize the netlist. Instead, we record the internal don't-cares and represent them as functions of module inputs. These don't-care conditions will later be included in the justification of the ATPG constraint-solving process.

We formulate the constraints of the target assertion as a linear temporal property[14], which specifies the expected signal values and relations in an execution sequence. The assertion property is first inverted to produce a counter-example-generation problem and then translated into value requirements in different time-frames. Our constraint solver then applies word-level ATPG technique to solve the constraints in the Boolean domain and propagate the implications to the arithmetic units. If no solution is found, we can conclude that no counter example can be generated for this property and thus the assertion holds. Otherwise, we check if all the constraints in the datapath are satisfied. If yes, then a counter example is found and the assertion fails. Otherwise, we use the arithmetic solver to see if the remaining constraints are feasible. If not, we backtrack to the ATPG process and iterate for the next set of solutions. This process continues until the property is proved or the run-time exceeds the pre-set limit.

3. Word-level ATPG

The flow of our word-level ATPG algorithm is shown in Fig. 2. The assertion property is first translated into initial assignments at different time-frames and word-level logic implication of these assignments is applied to the whole circuit.

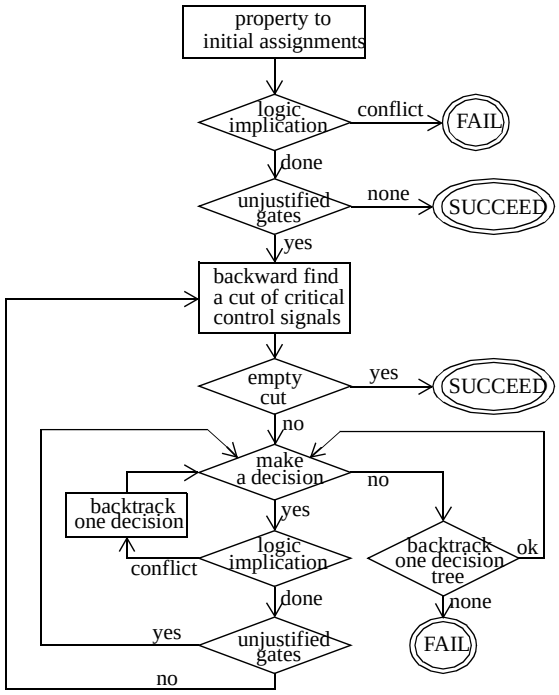

Fig. 2: The word-level ATPG algorithm.

3.1 Word-level Logic Implication

We use different kinds of data structures to represent the legal values for different kinds of gates. This enables forward and backward word-level logic implications not only on Boolean gates but also on arithmetic elements. It also allows translation of the implications between Boolean gates and arithmetic elements.

Boolean gates. We utilize 3-valued logic encoding (i.e. 0, 1, x, where x means unknown) to perform parallel implication for bit-wise logic gates. For example, suppose a 4-bit AND gate has input values **a** = 4'b10xx, **b** = 4'bxxxx, and output **y** = 4'bx00x. If the input **b** receives the new implication value 4'b1x1x, it will forward imply a new value 4'b100x at output **y**, which in turns backward implies a new value 4'b100x at input **a**.

Arithmetic units. For arithmetic units like adders, we perform 3-valued forward and backward simulation in order to propagate as much known-value information as possible. For example, if a 4-bit adder has output value 4'b0111, and one of its input has value 4'b1x1x (Fig. 3), then by subtracting 4'b1x1x from 4'b0111, we can learn that the other input must at least have the value 4'b1x0x, and the adder must have carry-out value equal to 1.

Besides, as will be shown in the next section, the solutions to a linear arithmetic network can be represented as a closed matrix

form: $\mathbf{x} = \mathbf{x_0} + \mathbf{N} * \mathbf{f}$, where $\mathbf{x}$ is the vector of the input variables, $\mathbf{x_0}$ is a particular solution, $\mathbf{N}$ is a coefficient matrix and $\mathbf{f}$ is a set of free variables.

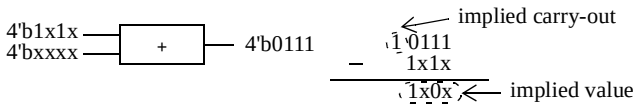

Fig. 3: Word-level implication for an adder circuit.

Comparators. We use a pair of bit-vectors to record the maximum and minimum values for each of the comparator inputs. For example, suppose a 4-bit "greater" (>) gate has output value 1 (TRUE) and input values "in_a = 4'bx01x" and "in_b = 4'b1x0x" (Fig. 4). By setting all the x's to 0's and then to 1's, we learn that "in_a" has the minimum and maximum values [min_a, max_a] equal to [2, 11], and "in_b" has [min_b, max_b] = [8, 13]. However, for the "greater" gate to be evaluated "TRUE", it implies "min_a" must be greater than "min_b", and "max_b" must be smaller than "max_a". Adjusting the values of "min_a" and "max_b", we have [min_a, max_a] = [9, 11] and [min_b, max_b] = [8, 10]. To map the new ranges back to 3-valued logic, we use the following rules:

(Rule 1) Only bits with value "x" can have new boolean implications.

(Rule 2) More significant bits must have implication prior to less significant ones.

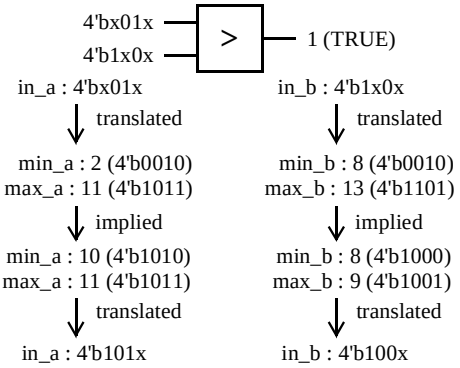

Fig. 4: Implication for comparators

While Rule 1 is trivial in logic implication, Rule 2 is based on the fact that only the most significant "x" bit can divide the original range into two disjoint sub-ranges. For this example, implication on the second highest bit of input **b** (with original value 4'b1x0x) can split the original range [8, 13] into two distinct sub-ranges [8, 9] (implied "0") and [12, 13] (implied "1"). On the other hand, implication on the least significant bit will produce two overlapped ranges [8, 12] and [9, 13]. Therefore, to have the new implied range [8, 10], it is mandatory that the second highest bit be implied "0" because this new range has null intersection with the other implied range [12, 13]. Implication on the least significant bit cannot draw any conclusion on this.

Likewise, we can learn that the most significant bit of "in_a" must have implied value "1". Therefore, we will have the new ranges for "in_a" and "in_b" equal to [10, 11] and [8, 9], respectively. Mapping these new ranges back to 3-valued logic, we will have implications "in_a = 4'b101x" and "in_b = 4'b100x".

Multiplexors. Because we represent the value of a multiple-bit bus as a 3-valued bit-vector (a cube), we can use cube union of the input values to derive the implication on the output of a multiplexor. On the other hand, if one of its inputs has null cube intersection with the output, then it implies that the control signal cannot have the value that selects this input.

Registers/Flip-flops. Similarly, we can derive implication for the asynchronous "set" and "reset" signals of a data register by examining its data input and output values. For example, if the data output bits have all been assigned to zero and at least one of its input bits has been assigned to one, we can learn that its "reset" signal must be asserted.

Based on the rules above, we perform logic implication whenever a decision on a Boolean gate assignment is made. If any conflict occurs during the implication process, the process backtracks. However, unlike the bit-level logic implication where the single-bit signal can be implied only once, i.e. from "x" to "0" or "1", a word-level signal can be implied multiple times. Therefore, when an implication process returns a conflict and the process backtracks to its previous condition, we cannot just reset the signals to "x" but need to recover them to their previously partially-implied values.

After implication, we check if there is any unjustified logic gate, that is, its 3-valued simulation value is different from its output implied value, or the values of the control flip-flops do not cover the initial states. If yes, the Boolean constraints are not satisfied and the justification procedure is called.

3.2 Justification Process

After implication, a branch-and-bound algorithm is employed to justify the value requirements on the control logic. In this phase we only make decisions on control signals and leave the requirements in the datapath portion unjustified. This greatly reduces the effort of ATPG process as we avoid the enumeration of the possibly enormous number of datapath decision points. In addition, whenever there is a new decision of assigning a logic value to a single-bit control signal, word-level logic implication is followed immediately to reduce the search space and to detect early contradiction in value assignments.

The justification process begins by backward, breadth-first traversing the circuit from the unjustified gates and stopping at a cut of candidate decision points including control PIs, flip-flops, comparator outputs, and multiple-fanout internal logic gates. Note that if the number of decision candidates is too large, using all of them as the decision points may make the decision-making process less efficient. Therefore, if the number of decision candidates exceeds a limit, based on the number of fanouts of each candidate, a subset of them is selected as the decision nodes.

The list of candidate decision-making gates is then sorted based on their bias of being assigned "1" or "0" to meet the requirements of the unjustified gates. Note that observability in general is not a problem in this application as we can add *watch points* (in RTL simulation) wherever is necessary. Therefore, in calculating the order of the decision points, the controllability/observability measures used for traditional stuck-at-fault testing [15][16] is not appropriate. In our method, we backward compute the *legal-1/ legal-0 probability* for the signals between the unjustified and decision-making gates.

Definition 1 (legal-1/legal-0 probability). The *legal-1(legal-0) probability* of a signal is the probability of its being assigned to "1"("0") to satisfy its output logic value.

For example, if a 2-input AND gate has output value "0", and both inputs with value "x", then there are 3 different legal assignments which can satisfy this unjustified value: {(0, 0), (0, 1), (1, 0)}. Therefore, the legal-1 probability for the input signal is 1/3 because only one out of three legal assignments is value "1" for each of the inputs. On the other hand, if the 2-input AND gate has output value "1", its input legal-1 probability is 1.0 since this is the only legal assignment. Note that the summation of legal-1 and legal-0 probabilities is equal to 1.0.

We can generalize the rules above and have the backward legal-1 probability calculation as:

(Rule 3) For signals with Boolean value "1", the legal-1 probability = 1.0. On the contrary, for signals with Boolean value "0", the legal-1 probability = 0.0.

(Rule 4) Suppose the legal-1 and legal-0 probability for a gate out-

put are p_1, and p_0, and it has unjustified output value and n unknown inputs (with value "x"). The input legal-1 probability q_1 (for different gate types) is:

INVERTER: $q_1 = p_0$.

AND: $q_1 = p_1 + \frac{2^n - 2^{n-1} - 1}{2^n - 1} \times p_0$

OR: $q_1 = \frac{2^{n-1}}{2^n - 1} \times p_1$

(Rule 5) The legal-1 probability of a fanout stem is set to the average of the legal-1 probabilities of its fanout branches.

Due to the space limitation, we omit the detailed derivation of the rules here. After the legal-1 probabilities of the decision gates are computed, we calculate their *legal assignment bias* as:

Definition 2 (legal assignment bias). Let the legal-1 probability of a gate be p_1, its *legal assignment bias* is:

$\frac{p_1}{1-p_1}$ if p_1 >= 0.5; (bias value = "1")

$\frac{1-p_1}{p_1}$ if p_1 < 0.5; (bias value = "0")

Note that the legal assignment bias is always greater or equal to 1. Having the legal assignment bias for each decision point, we make the decision at the gate with the highest bias first. If we are proving an assertion property, that is, it is likely that the counter example does not exist or, if exists, is hard to find, we first assign the complement of the bias value so that the conflicting condition occurs early and thus help trim down the decision space. On the other hand, if our objective is to generate a witness sequence which is likely to exist, we assign the bias value (instead of its complement) first.

Once the gates between the unjustified gates and the decision points are all justified, we further traverse backward to see if there is any unjustified boolean gate. If yes, we update the list of decision points and repeat the justification process again for the new set of unjustified gates. Otherwise, we should have reached the primary inputs and the datapath-control boundary and all the constraints on the control logic should have been satisfied.

4. Arithmetic Constraint Solver

After the constraints on the control logic are satisfied, we further check whether the requirements on the datapath are also satisfied. If not, we apply modular arithmetic techniques to solve the constraints on the datapath portion. Note that although the datapath constraints may across many timeframes, we can create a combinational model by treating the state elements (D flip-flops) as buffers and adding necessary new variables for the inputs of each timeframe. As a result, the datapath constraints can be represented by a set of arithmetic equations.

The arithmetic constraint solver is based on the modular number system, which is required as the values of the hardware signals are represented as fixed-width bit-vectors. Besides, using the modular instead of the general integral arithmetic is critically important to prevent the *false negative* effect. For example, suppose we have a multiplier with two 3-bit inputs a and b, and a 4-bit output c. The following nonlinear arithmetic constraints describe the multiplier:

$$\begin{cases} a \times b = c \\ 0 \le a, b < 8 \\ 0 \le c < 16 \end{cases}$$

Consider an initial assignment of c = 12 and a = 4. The solution b = 3 can be easily derived by direct implication. However, b = 7 is also a solution because (4 * 7) modulo 16 = 12 as c is a 4-bit signal. Therefore, if the local solution (a, b) = (4, 3) does not satisfy other imposed constraints while (a, b) = (4, 7) does, using a solver/algebra not based on the modular number system will result in a false negative conclusion.

Generally speaking, constraints on datapath can be divided into two types: linear and nonlinear. Nonlinear arithmetic constraints are those derived from multipliers and shifters. Since completely solving them could be very difficult, if not impossible, we apply some analytical approaches like prime number factoring to heuristically enumerate the possible solutions and substitute them into the arithmetic equations so that the constraints become linear and can be solved by our linear constraint solver.

4.1 Linear Constraint Solver

Linear constraints, on the other hand, arise from adders, subtractors, and multipliers with one constant input. They make up most of the arithmetic units for industrial circuits in various applications. Given a linear subcircuit with m outputs and n inputs, we can transform it to a problem of m linear equations with n variables and further formulate it into the matrix form as $\mathbf{A}*\mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a $m*n$ matrix representing the coefficients in the m equations, $\mathbf{x}$ is a $n*1$ column matrix containing the n variables, and $\mathbf{b}$ is a $m*1$ column matrix for the output constraints. Solving the input vectors that can satisfy the output constraints is equal to finding the solution to the matrix equation[1].

For example, consider a 2-input, 2-output linear circuit with all signals of 3-bit wide. Suppose the output constraint is (5, 4) and the circuit under this constraint can be expressed in the following matrix format:

$$\begin{bmatrix} 1 & 1 \\ 2 & 7 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \end{bmatrix}$$, where x and y are the input signals.

Solving it in the integral domain, we first multiply the first row by 2 and subtract it from the second row. We have:

$$\begin{bmatrix} 1 & 1 \\ 0 & 5 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ -6 \end{bmatrix}$$

It is clear that there is only one non-integral solution (x, y) = (31/5, -6/5). However, under modulo-2^3 number system we can find a solution (x, y) = (3, 2). We will demonstrate later that this solution can be derived by calculating the *multiplicative inverse* of bit-vectors.

In the rest of section, we provide some theoretic details of our linear constraint solver which is capable of finding all solutions to a given set of linear constraints under modular number system and expressing them in a closed form.

Definition 3 (multiplicative inverse of bit-vector). The *multiplicative inverse x* of an n-bit bit-vector *a* is defined as: { x | (a * x) *modulo* 2^n = 1 } [17]. We denote it as *multiplicative_inverse*(*a*).

Note that while multiplicative inverse exists for every non-zero real number, in integral number domain only integers 1 and -1 have multiplicative inverse. In modulo-2^n number system, only odd numbers have one and only one multiplicative inverse [18]. For example, for 3-bit-wide bit-vectors, 3 is 3's multiplicative inverse because 3 * 3 = 9 and 9 modulo 2^3 = 1. On the other hand, 2 does not have any multiplicative inverse.

Although there exists no multiplicative inverse for even bit-vectors, we can extend the concept of multiplicative inverse to the *multiplicative inverse with product k*:

Definition 4 (multiplicative inverse of bit-vector with product k). The *multiplicative inverse x* of an n-bit bit-vector *a* with multiplication product *k* is defined as { x | (a * x) *modulo* 2^n =

1. Note that we use the term "column/row matrix" instead of "column/row vector" in order not to get confused with the term "bit-vector".

k }. We denote it as $multiplicative\_inverse_k(a)$.

For example, for 3-bit-wide bit-vectors, 3 is 6's multiplicative inverse with product 2 because 6 * 3 = 18 and 18 modulo $2^3 = 2$. Especially, 0 does not have any multiplicative inverse with non-zero product, but every bit-vector is the multiplicative inverse of 0 with product 0.

As for the linear constraint example above, we can modulate the equation with 2^3 and have:

$$\begin{bmatrix}1 & 1\\ 0 & 5\end{bmatrix} \cdot \begin{bmatrix}x\\ y\end{bmatrix} = \begin{bmatrix}5\\ 2\end{bmatrix}$$

Solving the second row by $multiplicative\_inverse_2(5)$, we can have y = 2. Substitute it back to the first row, we can derive x = 3.

A bit-vector may have none or several multiplicative inverses. The following theorem gives the number of multiplicative inverses for a given bit-vector.

Theorem 1. Given a non-zero n-bit-wide bit-vector a with greatest odd factor a', it can be expressed as $a = a' * 2^m$, where m is an integer.

(T1.1) a has *exactly one* multiplicative inverse with product k if and only if a is an odd number, that is, $m = 0$. Moreover, $multiplicative\_inverse_k(a) = multiplicative\_inverse(a) * k$.

(T1.2) a has *no multiplicative inverse* with product k if and only if a is an even number and k is not a multiple of 2^m.

(T1.3) a has *exactly 2^m multiplicative inverse* with product k if and only if a is an even number and k is a multiple of 2^m.

For example, for 3-bit-wide bit-vectors, 6 (= $3 * 2^1$) has no multiplicative inverse with product 3 because 3 is not a multiple of 2^1, but has exactly 2 multiplicative inverse of product 4 as {2, 6}.

Furthermore, we can represent all the 2^m multiplicative inverse bit-vectors in (T1.3) in a closed form as shown in the following theorem:

Theorem 2. Given a non-zero, even, n-bit-wide bit-vector a with the greatest odd factor a' expressed as: $a = a' * 2^m$. If k is a multiple of 2^m as $k = k' * 2^m$, and b is the only multiplicative inverse of a' with product k', that is, ($a' * b$) modulo $2^n = k'$, then the multiplicative inverse of a with product k can be represented in the following closed form:

$multiplicative\_inverse_k(a) = (b + 2^{(n-m)} * t)$ modulo 2^n, where t is a *free integer* between 0 and $2^m - 1$.

For example, for 4-bit-wide bit-vectors, let $a = 6 = 3 * 2^1$, and k = $10 = 5 * 2^1$ which is a multiple of 2. By (T1.3), we know that a = 6 has exactly 2 multiplicative inverse with product k = 10. Because the multiplicative inverse of 3 with product 5 is 7 (3 * 7 = 21 modulo $2^4 = 5$), we can represent all the multiplicative inverse of 6 with product 10 as: "$7 + 2^3 * t$", for t = 0 or 1.

We extend the concept of multiplecative inverse and apply it to solve the linear bit-vector matrix constraints using the *Gauss-Jordan Elimination method*. The solutions can be represented in a closed form as:

$$\mathbf{x} = \mathbf{N} * \mathbf{f} + \mathbf{x_0}$$

where **N** is called the *null matrix* (because multiplying it with the constraint matrix **A** will result in a zero matrix), **f** is a column matrix containing some free variables, and $\mathbf{x_0}$ is a solution and can be derived from **A**, **N** and **b** in linear time. Applying different values of the free variables in **f**, we can obtain different values of **x** and each of which is a solution of $\mathbf{A} * \mathbf{x} = \mathbf{b}$.

Example: Assume all the signal buses in the linear circuit of Fig. 5 are 4-bit wide, and an initial assignment for output x = 2 and y = 10 are given. The linear constraints can be expressed as an integer matrix equation: $\begin{bmatrix}3 & -1 & 0 & -2\\ 1 & 2 & -2 & 0\end{bmatrix} \cdot \begin{bmatrix}a\\ b\\ c\\ d\end{bmatrix} = \begin{bmatrix}2\\ 10\end{bmatrix}$

Modulating the coefficients by 16 (2^4), we have:

$$\begin{bmatrix}3 & 15 & 0 & 14\\ 1 & 2 & 14 & 0\end{bmatrix} \cdot \begin{bmatrix}a\\ b\\ c\\ d\end{bmatrix} = \begin{bmatrix}2\\ 10\end{bmatrix}$$

We then apply the *Gauss-Jordan Elimination method* (based on the extended concept of multiplecative inverse) to solve the equation and obtain the solution **x** in a close form:

$$\begin{bmatrix}a\\ b\\ c\\ d\end{bmatrix} = \begin{bmatrix}14 & 6\\ 10 & 0\\ 1 & 3\\ 0 & 1\end{bmatrix} \cdot \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}10\\ 0\\ 0\\ 6\end{bmatrix}$$, where i and j are free variables between 0 and 15.

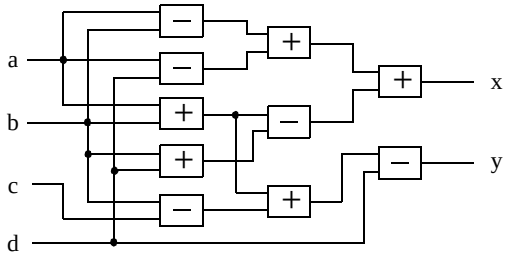

Fig. 5: A linear circuit example.

The computational complexity of solving the linear constraints (finding all solutions) under the modular number system is $O(n^3)$, where n is the number of input variables. This can be proved by construction. Due to the space limit, the proof is omitted here.

5. Experimental Results

We have implemented the combined word-level ATPG and arithmetic constraint-solving techniques and integrated them with a commercial HDL parser/logic synthesizer. The framework is applied to several public benchmark and industrial circuits. Table 1 shows the statistics of these designs.

Table 1: Circuit statistics

ckt name	#lines	#gates	#FFs	#ins	#outs
addr_decoder	52	307	86	7	64
token_ring	157	4902	536	518	132
arbiter	303	2443	24	69	25
alarm_clock	719	1277	33	7	40
industy_01	11280	380k	9922	293	733
industy_02	5726	25520	96	60	25
industy_03	694	2623	0	70	64
industy_04	599	924	0	79	32
industy_05	47	210	7	13	1

* ckt name: circuit name
* #lines: lines of Verilog codes
* #gates: number of gates
* #FFs: number of flip-flops
* #ins: number of inputs
* #outs: number of outputs

We conduct the experiments by applying several different asser-

tion properties for each circuit. The properties are denoted as "p1", "p2",...etc.

For *addr_decoder*, we first check that for any randomly-selected memory cell we can write the data to it successfully (p1). We also prove that it is impossible to have two address lines selected at the same time (p2). The properties for *token_ring* and *arbiter* are the same. We assert that the bus-selecting signals for the clients are one-hot, (p3 and p5) and each client can access the bus after waiting certain periods (p4 and p6). There are three properties associated with the design *alarm_clock*: first we show that the clock will be reset to time "12:00" after it passes "11:59" (p7). Second, we generate a witness sequence that brings its "hour" display to "2" after the alarm clock is powered on (p8), and third, we assert that it is impossible for the "hour" output to display "13" (p9). For the industrial designs *industry_01* and *industry_05*, we prove that their internal don't-cares are also external to the circuits (p10 and p14). That is, it is impossible to reach these internal don't-care states and therefore we can use them to optimize the circuits. Designs *industry_02*, *industry_03*, and *industry_04* contain 152, 128, and 32-bit bus signals, respectively. We assert that there is no bus contention for these designs, that is, either the tri-state enabling signals are one-hot, or when more than two of the enabling signals are on, their tri-state input data values must be consensus (p11, p12, and p13).

The experimental results are shown in Table 2. We conducted the experiments on a Sun UltraSparc 5 workstation with 512MB memory. It shows that our algorithm is very efficient especially on the memory usage. This is mainly because the memory consumption for the ATPG algorithm is linear with respect to the circuit size times the number of timeframes. Since we always try to make the generated sequence as short as possible and use the extended state transition graph to record the illegal states, the number of timeframes will not become arbitrarily large. Besides, because we utilize word-level primitives instead of Boolean gates to represent the netlist, the number of gates can be reduced significantly. Therefore, our approach is promising for large industrial designs.

Table 2: Experimental results

ckt_name	prop.	cpu time	memory
addr_decoder	p1	0.08	0.01
	p2	0.09	0.01
token_ring	p3	1.88	1.57
	p4	1.45	1.53
arbiter	p5	0.14	0.12
	p6	0.59	0.20
alarm	p7	0.36	0.88
	p8	1.31	2.74
	p9	137.05	9.76
industry_01	p10	14.79	54.66
industry_02	p11	20.37	17.89
industry_03	p12	1.25	2.85
industry_04	p13	0.40	1.59
industry_05	p14	0.03	0.02

* cpu time is in seconds and memory usage is in Mega bytes.
They include flattening the circuit and solving the constraints.

6. Discussion and Conclusion

We propose a combined word-level ATPG and modular arithmetic constraint-solving approach for the RTL assertion property checking. The experimental results show that our methods are very memory efficient and thus suitable for large industrial designs. However, we found that there is still some high-level information in the RTL design that we can utilize to speed up our algorithms. For example, there are usually many local finite state machines in the design and the transition relationship for each individual machine is usually very easy to extract in the parsing and synthesis steps. Storing the local state transition graph and using them to guide the ATPG justification process can avoid entering illegal states and therefore can generate the test vectors more efficiently. In addition, recognition of other high-level modules like counters, and shift-registers,...etc, can also help improve the efficiency.

In addition to applying our techniques to assertion properties, we also plan to use them for model checking more general properties in the future. We will study algorithms on efficient state hashing of the extended state transition graph and detecting loops of execution sequence.

Reference

[1] R. Vemuri and R. Kalyanaraman, "Generation of Design Verification Tests from Behavioral VHDL Programs Using Path Enumeration and Constraint Programming", IEEE Trans. on VLSI System, Val. 3, No. 2, June 1995.

[2] C.R. Ho, "Validation Tools for Complex Digital Designs", Ph.D. Thesis, Stanford University, Dec. 1996.

[3] F. Fallah, S. Devadas, and K. Keutzer, "Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability", Proc. DAC, June 1998, pp. 528-533.

[4] K.T. Cheng and A.S. Krishnakumar, "Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model", ACM Trans. on Design Automation of Electronic System, Vol. 1, No. 1, Jan. 1996, pp. 57-79.

[5] R. C-Y Huang, and K-T. Cheng, "A New Extended Finite State Machine (EFSM) Model for RTL Design Verification", Proc. IEEE International High Level Design Validation and Test Workshop, Nov. 1998, pp. 47-53.

[6] E. M. Clarke and E. A. Emerson, "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic", Lecture Notes in Comp. Science, Vol. 131, May, 1981, pp. 244-263.

[7] "Murphi description language and verifier", http://verify.stanford.edu/cgi-bin/wrap/dill/Murphi.

[8] I-H. Moon, J-Y. Jang, G.D. Hachtel, F. Somenzi, J. Yuan, and C. Pixley, "Approximate Reachability Don't Cares for CTL model checking", Proc. International Conference on Computer-Aided Design, Nov. 1998, pp. 351-358.

[9] K.L. McMillan, "Symbolic Model Checking: An Approach to the State Explosion Problem", KAP, 1993.

[10] "The SMV System", http://www.cs.cmu.edu/~modelcheck/smv.html.

[11] The VIS Group, "VIS: A system for Verification and Synthesis", Proc. of the 8th International Conference on Computer Aided Verification, July 1996, Springer Lecture Notes in Computer Science, p428-432.

[12] "R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Transactions on Computers, Vol. C-35, No. 8, August, 1986, pp. 677-691.

[13] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking Using SAT Procedures Instead of BDDs", Proc. DAC, June 1999, pp. 317-320.

[14] A. Pnueli, "A Temporal Logic of Concurrent Programs", Theoretical Computer Science, 13:45-60, 1981.

[15] L.M. Goldstein and E.L. Thigen, "SCOAP: Sandia COntrollability/Observability Analysis Problem", Proc. Design Automation Conference, June 1980, pp. 190-196.

[16] F. Brglez, "On Testability Analysis of Combinational Networks", Proc. ISCAS, May 1984, pp. 221-225.

[17] Allenby, "Rings, Fields, and Groups", Edward Arnold Ltd., 1983.

[18] B.W. Jones, "Modular Arithmetic", Blaisdell Publishing Company, 1964.