Assertion Checker Synthesis for FPGA Emulation

Chengjie Zang, Qixin Wei and Shinji Kimura

Graduate School of Information, Production and Systems,
Waseda University, 2-7 Hibikino, Kitakyushu, 808-0135, Japan.
zangchengjie@asagi.waseda.jp

Abstract— In the paper, we propose a method to synthesize SystemVerilog Assertion checkers for FPGA emulation. The main idea is to synthesize assertions based on finite input memory automata (FIMA) and use embedded RAM modules to construct shift register chain for storing the history of variables. The method does not consume logic elements for storing the value and the shift register using the embedded RAM is much more efficient compared with the one uses the registers in logic elements. We also compare the proposed FIMA method with MBAC [1] method and a tool of FoCs [2].

I. INTRODUCTION

Assertion based verification is a combination of simulation based methods and formal verification methods. Assertions can help designer to detect more bugs in early stages, as well as to understand more about the design during writing assertions. SystemVerilog Assertion [3] is one of such assertion description languages combined with a widely used commercial simulation tool, which can be checked when RTL simulation.

FPGA prototyping technology is effective for accelerating simulation based functional verification. However, how to debug using FPGA is a challenge in FPGA prototype-based verification. Some works have been done to synthesize assertion checkers in order to implement them on FPGA for emulation. Formal Checkers (FoCs) [2, 4] transforms PSL/Sugar properties into assertion checkers which can be integrated into simulation environment. The state machine of the FoCs will enter an error state if the assertion fails during simulation. MBAC [1] is another tool which generates assertion checker from PSL/Sugar properties using automata based method. These checkers can help to detect bugs since they can stop the simulation and supply the information to find bugs if the assertion fails. However, many important structures are not supported by these checkers, and more works are necessary for minimizing the resource usage of assertion checkers. In this paper, we proposed a Finite Input Memory Automaton (FIMA) based synthesis method to transform SystemVerilog Assertions into hardware checkers. To save and minimize resource usage of Logic Elements (LE), we construct input memory automata by using embedded RAM modules.

II. SYSTEMVERILOG ASSERTION

SystemVerilog Assertion (SVA) is a language to describe the behavior of a system. In SVA, a sequence is used to describe a temporal behavior of the system, which is composed of Boolean expressions on signals with delay operators. A property can be constructed with multiple sequences for checking several sequential behaviors at various times. The detailed description of System Verilog Assertion is shown in [3]. In this paper, we mainly discuss the concurrent assertions. This kind of assertions is based on a clock and evaluated at every clock tick.

In SystemVerilog Assertion, sequence is used to manipulate sequential behaviors. Linear sequence includes finite Boolean expressions in a linear order of increasing time. The linear sequence is matched if all of the finite Boolean expressions are true in a finite interval of consecutive clock ticks. A single Boolean expression can be considered as a single linear sequence and which is evaluated at each clock tick. Regular expression is used to express more complex sequence, which concisely specifies finite or infinite linear sequence. In this paper, we only discuss assertions with finite linear sequence.

Sequences can be concatenated by “##”, which specifies a delay between the end of former sequence and the beginning of the next sequence. An integer number or range (named time window) denotes the length of delay. For example,

• “req ##N gnt” Specifies that if req is true on the current clock tick, then the signal of gnt shall be true on the Nth clock tick from now.

• “req ##[M:N] gnt” Specifies that if req is true on the current clock tick, then the signal of gnt shall be true within the time between the Mth and the Nth(N>M) clock tick from now.

Sequences can be connected by some operators such as “and”, “or”, “intersect”, “within” and “throughout”,
as well as some kinds repetition operators such as “[\cdot]”, “[\cdot]" and “[\cdot]", “[\cdot]”. Property defines the behavior of hardware design and there are several kinds of properties in SystemVerilog. Sequence, negation, disjunction, conjunction, implication and instantiation. Detailed description of these properties can be found in [3].

III. FINITE INPUT-MEMORY AUTOMATON BASED SYNTHESIS

In SystemVerilog assertions, “\#\#n” denotes temporal delay between sequences. The match of one sequence is determined by the present signal and a limited number of previous signals. So we use finite input-memory automaton to construct assertion checker circuit.

Here we introduce the concept of finite sequence machine firstly. According to the definition in [5], a sequential machine M is defined by a finite input alphabet X, a finite output alphabet Z, a finite state set S, a next-state function \( \delta \), and an output function \( \lambda \). M = \( \langle S, X, Z, \delta, \lambda \rangle \), where \( s_{k+1} = \delta(s_k, x_k) \), and \( z_{k+1} = \lambda(s_k, x_k) \), for \( s_k, x_k \in X \) and \( z_k \in Z \), is said to have finite memory \( \mu \) if \( \mu \) is the least integer such that the output can be defined as follows:

\[
z_k = f(x_k, x_{k-1}, \ldots, x_{k-\mu}, z_{k-1}, \ldots, z_{k-\mu}) \text{ for } k \geq \mu.
\]

The present state \( z_k \) is determined by the inputs \( x \) of \( \mu \) time steps including current \( x_k \) and previous outputs. Finite input-memory machine is a special case of a finite sequential machine. A finite sequential machine M has finite input-memory \( \mu_i \) if \( \mu_i \) is the least integer such that

\[
z_k = f(x_k, x_{k-1}, \ldots, x_{k-\mu_i}) \text{ for } k \geq \mu_i.
\]

In finite input-memory machine, the present state \( z_k \) is determined by inputs \( x \) of \( \mu_i \) time steps including current \( x_k \).

To synthesize a finite input-memory machine, we can use register chain to store the previous values and use output decoder logic to produce the output, as shown in Fig. 1. For transforming a sequence checker, one finite input-memory machine is designed for one variable. Thus for sequence with \( N \) variables, we need \( N \) finite input-memory machines to construct the checker circuit. Output logic of these input-memory machines produce the output of sequence circuit. Next we will show the synthesis method from sequence layer and property layer respectively.

A. Synthesis Method for Sequence

In this method, we use shift registers to trace the history of variables. That means, one shift-register chain is similar to one history-list. In theory, for a history-list with depth \( N \), \( N \) registers are needed to configure the shift-register with \( N \) taps. There is one clock delay between neighbor values, so for the shift-register with depth \( N \), we should be able to simultaneously read \( N \) values.

Here is an example of linear sequence:

sequence s1:

a \#\#1 b \#\#2 c \#\#1 d;

The Boolean expression ‘a’ will be checked on each clock tick. The sequence s1 evaluates to true if ‘a’ evaluates to true in current clock, ‘b’, ‘c’ and ‘d’ evaluate to true after one, three and four clocks respectively. We can not refer the future value, so we should only refer the past signals. For this example, ‘d’ is the latest, so we consider the time based on ‘d’. To transform s1, we need to use a shift-register with depth 4 to record history of ‘a’, use a shift-register with depth 3 to record history of ‘b’, and use a shift-register with depth 1 to record history of ‘c’.

The introduction of the synthesis method starts from the linear sequence, which is the simplest sequential behavior. The following parts show how to transform a linear sequence into a hardware checker.

The inputs of a hardware checker are variables in a linear sequence and a clock signal, and the output is the evaluation result of the sequence. Firstly, we construct data structure of each input variable, Input(i) = \{P(i), Delay(i), Depth(i)\}. The next three steps present the meaning of P(i), Delay(i) and Depth(i), and show how to initialize and calculate these three parameters of each input variable. The sequence s1 is used as an example.

Step 1:

P(i) denotes the position of each variable Input(i) in the sequence.

The rightmost variable has position of 0.

For example, in sequence s1, the position of \{a, b, c, d\} is \{3, 2, 1, 0\}.

Step 2:

Delay(i) denotes the delay from its left neighbor variable in the sequence.

The leftmost variable has delay of 0;

For example, in sequence s1, the delay of \{a, b, c, d\} is \{0, 1, 2, 1\}.

Step 3:
Fig. 2 Algorithm-1: linear-sequence synthesis.

Depth(i) denotes the delay before the rightmost input variable coming.

For example, in sequence s1, the depth of \{a, b, c, d\} is \{4, 3, 1, 0\}.

Using the above three steps, we get the depth of each input variable. Based on the Depth(i) of each variable Input(i), we generate a shift-register chain with depth “Depth(i)” to trace its history. Fig. 2 shows the detailed algorithm of transforming linear sequence into a hardware checker. In this algorithm, the input variables are sorted according their positions firstly, and then the depth of each variable is calculated. The depth of each input variable denotes the number of registers in a shift-register chain.

The algorithm-1 is for simple linear sequence. More complex sequences might include regular expressions and sequence operators to connect two or more sequences. Complex sequence with regular expressions or operators can be decomposed to multiple simple linear sequences, then algorithm-1 is applied to transform these linear sequences. Next we will show how to decompose complex sequence with time window, repeaters and sequence operations. Moreover, in finite input memory automaton, since we use finite number of registers to configure shift-register chain, this kind of method can not transform operations with unbounded time window.

Consecutive repetition specifies that the operand sequence must match many successive times. Here are the example of s2 and s3.

sequence s2:
\[
a[\ast^3] \#\#1 b
\]

sequence s3:
\[
a[\ast^0:3] \#\#1 b
\]

Note that for multiple input-memory machines, we can share a shift-register chain and design output-decoder for different operations. For example, sequence s3 is decomposed into three subsequences. The variable ‘a’ of the first three subsequences can share shift registers from the fourth subsequence. Fig. 3(b) shows the register chain sharing by multiple subsequences.

2) OR Operation: OR operation is used to construct a sequence by some operand sequences, in which at least one of these operand sequences evaluates to true, the sequence evaluates to true. The end time of sequence is the same time when a match of these operand sequences. N operand sequences connected by OR operation is similar to N independent threads. Sequence might be matched at different end times. To transform this kind of sequence, N operand sequences are transformed separately.

3) First_match Operation: For sequence with time window, multiple sequences should be checked in each clock tick. First_match operator matches when the first one of these multiple sequences becomes true. Its end time is
the same as the earliest match sequence. If the earliest sequence is evaluated to true, the subsequence matches should be discarded from consideration. Here is an example of first_match operation.

sequence s4:
  first_match(a ## [2:4] b)

Each attempt of sequence s4 can result in one of the following three sequences:
  a ##2 b, a ##3 b and a ##4 b.

The sequence s4 is evaluated to true when one of the above three sequences ends firstly.

To transform the first_match operation, if it has time window, we firstly transform the time window into “OR” of N operand sequences. All of the results of earlier decided should be used for discarding the subsequent sequences. We can not synthesize the first_match operation by input-memory machine since its output-decoder includes sequential logic. Thus the operation of first_match is treated as a special case. The algorithm of transforming first_match operation is shown in Fig. 4. An example of sequence s4 is shown in Fig. 5.

4) **AND Operation**: AND operation denotes both operand sequences are expected to match. The operand sequences start at the same clock tick, but might match at different times. The end time of the composite sequence is the end time of the sequence which matches at last.

To transform AND operation, by parsing both operand sequence t1 and t2 using algorithm-1 respectively, we get the lengths |t1| = M and |t2| = N, assume N>M. The length of “t1 AND t2” is |t1 AND t2| = max{|t1|, |t2|} = N. By using input-memory based method, we get the outputs of t1 and t2 firstly, and then N-M registers are used to delay the result of short sequence. The “2-input AND” gate is used to produce the final output. For sequence with time window, firstly transform time window to “OR” operation, and then transform each “AND” operation respectively.

5) **INTERSECT Operation**: INTERSECT operation is similar to the AND operation with length restriction. In which both operand sequences must match, as well as the lengths of the two matches of sequences must be equal. Only the two operand sequences have the same length, the composite sequence has possibility to evaluate to true.

To transform INTERSECT operation, firstly, operand sequences are transformed by using algorithm-1 respectively. Then the results of two operand sequences is produced by the “2-input AND” gate directly.

B. **Synthesis Method for Property**

A property defines the behavior of hardware design and there are seven kinds of properties in SystemVerilog Assertions: Sequence, negation, disjunction, conjunction, if...else, implication and instantiation. To transform property, we firstly decompose it into some independent properties. Each independent property is constructed separately and then the synthesis result is generated according property operators. Operators of negation, disjunction and conjunction are similar to operators of NOT, OR and AND respectively in sequence layer. Disjunction operator is used for a disjunction property, in which at least one of the property evaluates to true, the disjunction property evaluates to true. Conjunction operator is used for a conjunction property, in which only if all of the properties evaluate to true, the conjunction property evaluates to true. Next we discuss implication operator in detail.

SVA supplies the implication operator, which is similar to if-else structure. The left hand side (LHS) of the implication is called “antecedent” and the right hand side (RHS) is named “consequent”. In each clock, if the antecedent succeeds, then the consequent is evaluated. Otherwise the property is assumed to be “vacuous success”. There are two kinds of implications, overlapped implication and non-overlapped implication.

- 1) **Overlapped implication**
  property P_overlapped;
  @(posedge clk) S1 |− > S2;
  endproperty

If sequence S1 evaluates to true on a given positive clock edge, then the sequence S2 should match on the same positive clock edge.
• 2) Non-overlapped implication

property P_non-overlapped;
    @(posedge clk) S1 => S2;
endproperty

If sequence S1 evaluates to true on a given positive clock edge, then the sequence S2 should match on the next positive clock edge.

To transform implication operation, before using algorithm-1, we firstly transform implication operation to linear sequences. For overlapped implication, the last Boolean expression of LHS and the first Boolean expression of RHS should both be true at the same clock cycle. For example, an overlapped property: a ##1 b ##1 c => d ##1 e, can be expressed as a ##1 b ##1 c ##0 d ##1 e. For non-overlapped implication, it can be transformed to a sequence LHS ##1 RHS. Assuming the lengths of sequences in LHS and RHS are |S_{LHS}| = M and |S_{RHS}| = N, the length of a sequence for overlapped is |S_{LHS}| > |S_{RHS}| = M + N and the length of non-overlapped implication is |S_{LHS}| = |S_{RHS}| = M + N + 1. We use the algorithm-1 to transform LHS and RHS sequences. The output of implication operation is produced by combinational logic, as block 3 in Fig. 6.

Here we use an non-overlapped implication example to show how to synthesize the property to an input memory automaton based circuit.

property implication_p1;
    @(posedge clk) a ##1 b ##1 c => d ##1 e;
endproperty

In this property, on each positive clock edge, LHS sequence will be checked. If the sequence of LHS is true, then at next positive clock edge, the sequence of RHS will be evaluated. Otherwise the simulator will give a result of vacuous success. If the sequence of RHS is also true, the simulator will give a result of real success, otherwise it will return fail.

The implication assertion can be transformed to three parts in circuit, LHS, RHS and implication operator. Sequences of LHS and RHS are transformed by the input-memory automaton based method. These three parts are corresponding to Block 1, Block 2 and Block 3 in Fig 6.

For the sequence with time window, the time window can be explained as logic “OR” of operand sequence threads in parallel, as shown in Fig. 7, which is an implication example with time window.

IV. Flip-flop Usage Reduction

FIMA based method takes registers to record the history of Boolean expressions in sequences. For mapping a hardware design on FPGA, assertion checkers take resources. Complex hardware design takes large amount of resources in FPGA. When the usage of FPGA is about 70~80 percent, the wire routing resource becomes sensitive to design. Assertion for verification is expected not to influence the implementation of the design, but checkers take wire routing resource when mapping. To release the burden on FPGA resource, we construct input memory automata using embedded RAM modules. Fig. 8 shows how to construct shift register by embedded RAMs. We use the RAM module with two read/write ports and each RAM stores two 1-bit data. For one RAM module, by setting the “wren” as ‘1’ and connecting output-1 to input-2, we get a 1-bit shift-register with depth of 2. There are 2 taps in each RAM and we can construct a shift-register with more taps by connecting multiple RAM modules. Fig. 8 shows a shift-register with 4 taps which is constructed by two RAM modules and Fig. 9 shows the timing report of RAM based shift-register with 4 taps.

For combinational logic parts in hardware checkers, we also use RAM modules to implement them by storing the truth table in the memory. By initializing the value of a RAM module, we can implement a gate logic. For example, to implement the Block 3 of Fig. 6, we use one port
TABLE II
Comparisons of different methods.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FF</td>
<td>LUT</td>
<td>MHz</td>
</tr>
<tr>
<td>a1</td>
<td>4</td>
<td>4</td>
<td>422.12</td>
</tr>
<tr>
<td>a2</td>
<td>6</td>
<td>7</td>
<td>422.12</td>
</tr>
<tr>
<td>a3</td>
<td>12</td>
<td>14</td>
<td>422.12</td>
</tr>
<tr>
<td>a4</td>
<td>4</td>
<td>4</td>
<td>422.12</td>
</tr>
<tr>
<td>a5</td>
<td>6</td>
<td>3</td>
<td>422.12</td>
</tr>
<tr>
<td>a6</td>
<td>48</td>
<td>41</td>
<td>422.12</td>
</tr>
<tr>
<td>AS7</td>
<td>6</td>
<td>16</td>
<td>422.12</td>
</tr>
</tbody>
</table>

TABLE I
Benchmark assertions.

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>@(posedge clk) a ##1 b##1 c</td>
</tr>
<tr>
<td>a2</td>
<td>@(posedge clk) a</td>
</tr>
<tr>
<td>a3</td>
<td>@(posedge clk) a ##1 b</td>
</tr>
<tr>
<td>a4</td>
<td>@(posedge clk) a</td>
</tr>
<tr>
<td>a5</td>
<td>@(posedge clk) first_match(a ##[2:4] b)</td>
</tr>
<tr>
<td>a6</td>
<td>@(posedge clk) a</td>
</tr>
<tr>
<td>AS7</td>
<td>Arbiter, Chart 2 [6] (4 assertions)</td>
</tr>
</tbody>
</table>

V. Experimental Results

To evaluate FIMA based method, we compare the synthesis results generated by three methods, FIMA based method, MBAC [1] method and a tool of FoCs [2]. The assertions used in FoCs are written in the PSL/Sugar specification language, while our tool is based on SystemVerilog. To do the comparison, we describe the same assertion by PSL language and SystemVerilog Assertion language respectively and then generate hardware checkers by these three methods. The benchmark assertions are shown in Table I, a1~a6 are written by ourself and AS7 (including 4 assertions) is selected from [6]. For hardware emulation, we use Altera EP1S80F1508C6 FPGA to map the assertion checkers. The VerilogHDL assertion modules are synthesized by Quartus II 7.2.

Table II shows the mapping result by using different transformation methods. Some properties are not supported by FoCs tool, such as “first_match”. “N.S.” is used to denote the properties which are not supported by tool. We check the proposed method by two ways, LE based shift-register and RAM based shift-regtser. From table II, MBAC method takes smaller number of FFs and larger number of LUTs than the proposed FIMA(LE) method. Compared with MBAC, in FIMA(LE) method, we can reduce the number of LUTs by increasing the number of FFs. FoCs tool seems to generate much larger circuit compared with method of FIMA(LE) and MBAC.

In FIMA(RAM), embedded RAM modules are used to construct shift-registers, as well as to implement combinational logic. From Table II, constructing shift register chain by RAM modules does not use FF, and assertion checkers work at 150~200 MHz.

VI. Summary and Conclusions

In this paper, we propose a finite input-memory automaton based assertion checker synthesis method to transform SystemVerilog Assertions into hardware checkers for FPGA prototyping. To save logic elements, the delay chain of finite input memory is constructed by embedded RAM modules. We compare the proposed FIMA method with MBAC method and a tool of FoCs. The results show that proposed FIMA(LE) use more FF and less LUTs compared with MBAC. By using embedded RAM module, the proposed FIMA(RAM) method transforms assertions without using logic elements.

Acknowledgements

This research was supported in part by Ambient SoC Global COE Program of Waseda University of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References