SVA2RTL: Compiling SVA to Synthesizable RTL

Atsushi Kasuya

Abstract: In recent years, quick prototyping with large scale FPGA is becoming a practical method for fast verification. But monitoring and checking the correctness of the design in such environment is a very challenging task. SVA (System Verilog Assertion) can be used to provide the logical aspect of correctness with its temporal description. Our tool SVA2RTL is targeting such a prototype environment. It is designed to convert SVA description into synthesizable RTL description such that it can be instantiated in FPGA. In this paper, we show how the SVA description is converted into RTL without loosing the correctness, while maintaining the optimal gate size.

1. Introduction

SVA (System Verilog Assertion) is an integrated mechanism within System Verilog HDL to describe the expected behavior of the design using temporal description. [1][2] The description forms a monitor that keeps checking the design during the execution.

Nowadays, it is becoming popular to use large FPGA for early stage prototyping, which provides near real time execution speed. But in such an environment, the verification is always a big challenge due to the difficulty of observing the internal activities. In such situation, using SVA to construct monitors in the prototype to check the correctness of behavior may be an excellent solution, given if the SVA description can be implemented in the chip as a hardware monitor.

There are several papers presented for such purpose.

BSV translates SVA into FSM. [3] It provided a parallel evaluation mechanism on subsequence part of implication property. But actually, such parallel evaluation is required even in antecedent part or within the parallel SVA code. It may cause more resource conflict when sequence is simply converted to FSM. The authors of [4] presented the basic blocks of SVA primitives, and utilize them to construct higher level sequence and property. The authors of [5] used synthesizable assertion library to construct the SVA circuit. But neither of them provided any solution for the needs of parallel evaluation. All three paper did not consider about the local variable implementation.

Our design goal of sva2rtl compiler is to provide a practical as well as accurate solution for translating SVA property description to synthesizable RTL. Also, the compiler must support the local variable and the sequence match item mechanism, which is crucial for monitoring a modern bus protocol. In order to accomplish those goals, we derived a detailed property analysis mechanism to

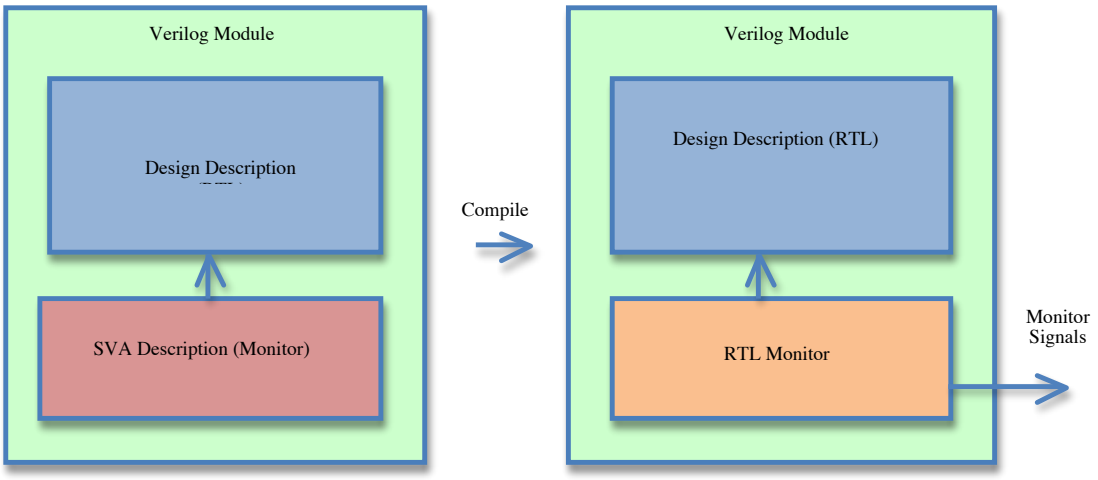
identify several types of attributes within the property, and utilize the information to generate an optimized monitor in synthesizable RTL.

In this paper, we show how the analysis is processed over SVA descriptions, and how the analysis information is used to produce the optimal RTL code.

2. The Compiler

SVA2RTL compiler takes Verilog module with SVA description, and compiles SVA part into RTL, and outputs non-SVA Verilog code as is. It adds monitor output signals to the module for observation. (The compiler only care for the concurrent assertion.)

The compiler accepts two types of SVA directives, 'assert' and 'cover', and RTL generates the 'error' signals and 'cover' signals corresponding to the SVA directives. When more than one assertion are declares in the module, the vectored output signal will be generated.



The following table shows supported commands in SVA2RTL compiler.

Category	Supported Command	Note
Assertion Type	Concurrent Assertion	
Assertion Directive	assert cover	'assume' is not supported
sub-assertion construction	property sequence	
property operator	and or not if |->\ |=>\	

sequence operator	##N ##[N:M] [*N] [*N:M] [->N:M] and or intersect throughout first_match()	zero cycle delay is not supported. Repetition only supports non-branching sequence. Infinite goto-repetition is not supported. 'within' is not supported.
Assertion System Function	$rose $fell $stable $sampled $onehot $onehot0	$isunkown is ignored, and always return False. $countones, $isunbounded functions are not supported. .ended, .triggerd methods are not supported.
Local Variable	local variables for sequence match items are supported.	

3. Assertion Analysis and Optimization

3.1 Error Check Attribute

A typical SVA assertion takes the following form.

```
label:  assert property (  <actual property> ) ;
```

In the <actual property> parts, typically it takes a implication with 'I->', 'I=>' operators. (A direct sequence as a property may be used for a relatively simple signal observation such as 'Reset'.)
Let's consider a sequence as the following.

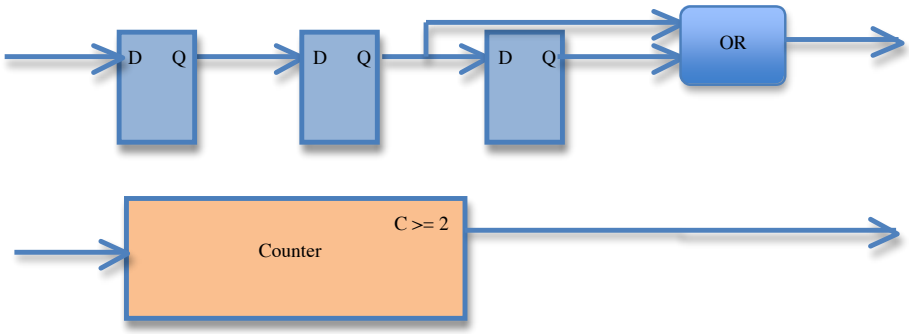
```
A ##2  B  |->  C ##2  D
```

Here, A, B, C, D are Boolean expressions. (SVA2RTL assumes that any Boolean expressions are synthesizable RTL, and uses it as is.)
The left hand side sequence (antecedent) is considered 'Error Free', means it is not a monitoring level error, when a sequence does not hold.
The right hand sequence (subsequence) must holds once it is triggered. We call it 'Error Check' (EC for short) attribute.

(Note that in the case of NOT property, even the antecedent part is required to detect error.)

3.2 Simple Pipe Sequence

In the property context, both ‘A ##2 B’ and ‘C ##2 D’ are considered to be ‘simple-pipe’, which contains no branching element, and the delays are short enough to be implemented with FF pipeline.
Implementing the delay in FF pipeline has several advantages over the counter implementation.
For example, a branching delay ‘##[2:3]’ can be implemented with FF pipeline, or a counter as the following figure.



If this delay element is used in the ‘Error Free’ segment, the FF pipeline implementation is non-blocking and no additional parallel logic is required, even it has a branching attribute. This is because each FF stage represents the timeline of the input signal. The multiple triggers are kept in the FF stages without interfering each other. Multiple outputs may be merged, but it does not change the logical aspect of the sequence evaluation toward the next stage.
On the other hand, the counter implementation can only take a single trigger input while it’s counting. So, a following new trigger must be handled in a different evaluation unit properly.
The difference between those implementations is expressed as non-blocking/blocking attribute. If an element is non-blocking, parallel evaluation is not required.
As of this nature, SVA2RTL uses the pipeline delay implementation by default for low number of delay length. The threshold to switch to counter implementation is set to 10, and it can be modified by a compiler option.

3.3 Parallel Evaluation Attributes

In order to handle multiple triggers simultaneously, we have to generate parallel evaluation mechanism that consists of multiple evaluation elements and a dispatcher to select one that is not busy. The decision to generate such parallel evaluation units must be carefully done to avoid the explosion of the size of the

circuitry. SVA2RTL uses two types of attributes associated to a SVA node, to optimize the RTL generation.

The attribute ‘Branching’ tells if a node can generate multiple triggers on its output. Considering an implication property ‘X |-> Y’, where X and Y are sequences, the start point at the beginning of X is considered to be ‘Branching’. This ‘Branching’ attribute will be propagated to the next element in general.

Another attribute to consider is ‘Blocking’ attribute as shown in previous counter delay implementation. This ‘Blocking’ attribute tells that this element can not take more than one trigger while it’s busy, so parallel evaluation unit must be generated if the previous stage is ‘Branching’. When SVA2RTL encounters such a condition, it generates multiple evaluation units, with a dispatcher. After the dispatcher, it is guaranteed that there’s only one trigger within one of the parallel units. Thus, the ‘Branching’ attribute is cleared. But if there is another branching element (e.g. ##[3:5]) exists in the following sequence, the Branching condition must be set again at the output of the branching element. This means that nested parallel evaluation units are required within the parallel unit.

Now, let’s consider a sequence with such branching delay:

```
A ##[3:5]  B  ##100 C
```

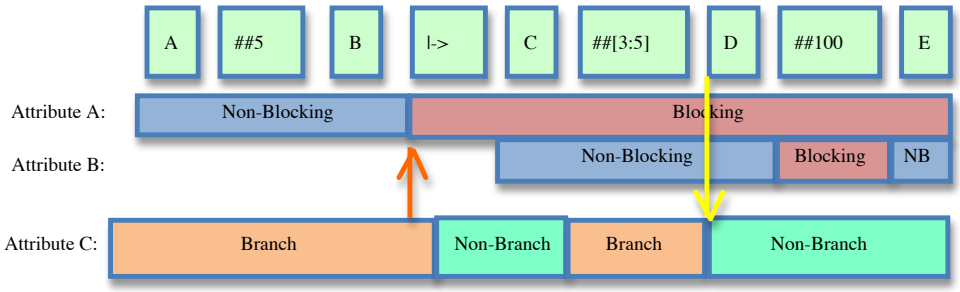
Here, we assume this sequence is within one of the parallel evaluation such that input to A is not branching. Still the first branching delay (##[3:5]) will generates the ‘Branching’ attribute, so that the following element with blocking attribute must be parallel. But most of the case, such branching delay is intended to capture the condition B happens once within several cycle, not expected that condition B triggers the following sequence multiple times. Considering such design intension, SVA2RTL puts ‘Filter’ attributes to the Boolean expression following the branching delay. The filter attribute terminates the Branch attribute propagation to avoid unnecessary parallel units. This filter effect can be disabled with a compiler option.

For implication (e.g. X |-> Y), the subsequence (Y part) has the implied first_match function. Thus, even if the sequence Y is non-blocking, it can take only one active evaluation at a given time. So the subsequence part always has ‘Blocking’ attribute.

Here, we can check how the following property is analyzed and converted into RTL as an example.

```
A ##5 B |-> C ##[3:5] D ##100 E
```

The two attributes, blocking and branching are attached to the property as the following Figure.

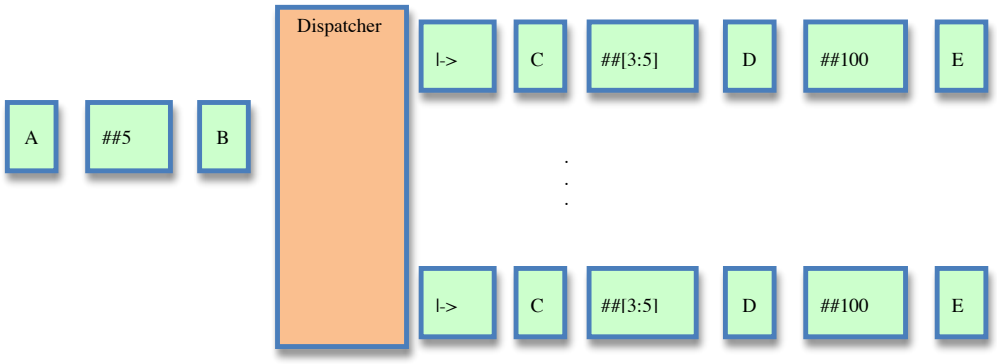


In the figure, Attribute A shows that the antecedent part of the implication is non blocking, as ##5 is short enough to be implemented as a pipeline. And the entire consequence part of the implication is blocking, due to implied first_match function.

Attribute B shows the internal blocking attribute within the consequence. The delay '##100' is blocking, as it is too long to be implemented with a pipeline.

Attribute C shows the transition of Branch/Non-Branch attribute. The precedence part has 'Branch' attribute propagated from its beginning. At the red arrow, the branch attribute hits the blocking attribute, so the parallel evaluation must be performed. Once the property is in the consequence part, it becomes non-branch, as the dispatcher guarantes that there is only one active evaluation at given time. But within the consequence, ##[3:5] has 'Branch' attribute, as it will generate multiple triggers. Then the attribute is filtered by Boolean evaluation 'D', and back to 'non-branch'. If this filter effect is deactivated (it can be done with a compile option), the branch attribute is propagated, and hits the blocking attribute on '##100', so this element must also be parallelized.

As the result of this analysis, SVA2RTL generates the parallel evaluation on consequence part of the implication as shown in the following figure.



The number of parallel evaluation units can be set with a compile option. The user can select the optimal number for the system requirement. When more triggers than the available evaluation units are entered into dispatcher, the dispatcher generates

Overwrap signal. By monitoring this signal over the prototype execution, the user can detect that the assertion result may not be accurate due to the overwrap, and tune the monitor with proper number of evaluation units.

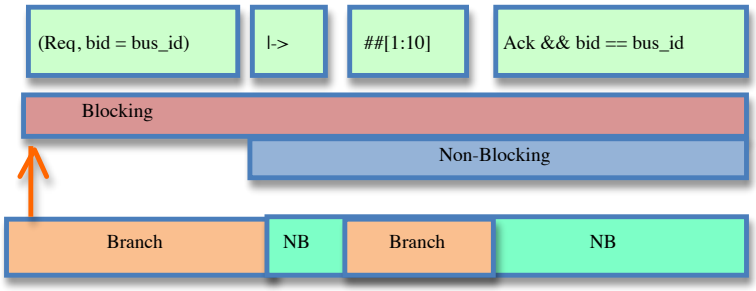
3.4 Local Variable and Blocking Attribute

Many modern bus and interconnect used in high performance systems utilize split transaction access, and multiple outstanding request can be issued at given time. The order of replies does not follow the order of requests. In such a protocol, a reply corresponding to a request must be identified by ID field in the transaction packets. In order to construct a monitor for such bus/interconnect protocol, local variable and the sequence match item to sample a data is crucial mechanism.

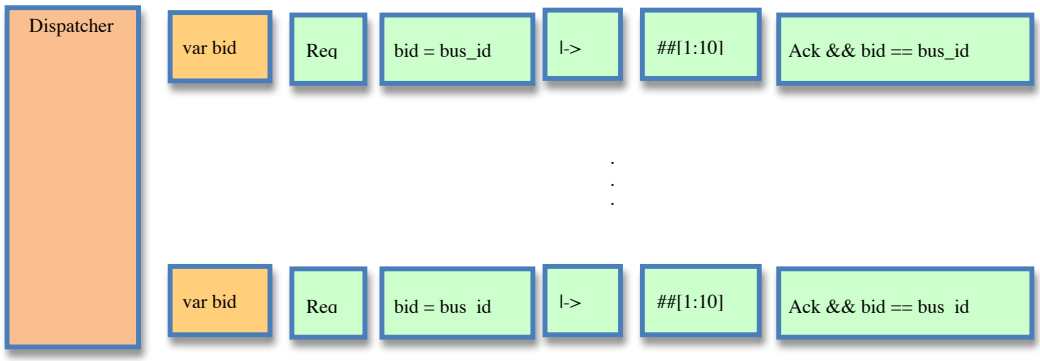
Considering the following implication property:

```
( Req ,  bid = bus_id ) |->  ##[1:10] ( Ack && bid == bus_id )
```

The similar attribute evaluation shown before is done over the property, but this case, the block attribute is extended to the beginning of the property due to the local variable usage.

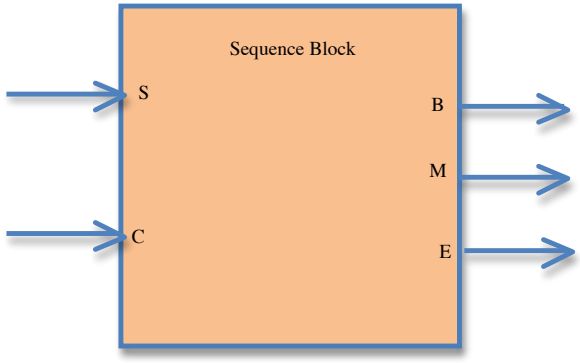


As the result, the parallel evaluation units are generated at the beginning of the property, with the local variable instantiation on every evaluation units.



4. RTL generation

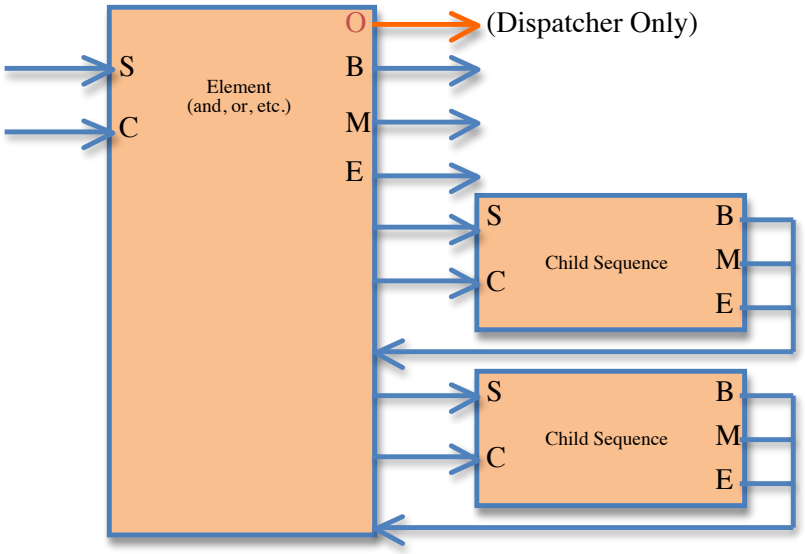
A basic block of compiled sequence has logically two input signals and 3 output signals, beside the system signals such as clock and reset.



S : Start signal to start the evaluation of the sequence block
C: Clear signal to clear all pending evaluation in the block
B: Busy signal to indicate that the block is busy evaluating the previous S input
M: Match signal to indicate that the sequence matches
E: Error signal to indicate that the sequence failed.

All the sequence element of SVA can be expressed with this block, and its combination sequence can be constructed by connecting this basic block.
As this block represents a sequence, first_match() function can be implemented by simply connecting the Match signal to its Clean signal.

The operator elements such as AND, OR, Dispatcher, etc., has multiple interfaces to its children sequences, and represent itself with the 5-signal interface to its sibling elements.
For the dispatcher, the additional signal ‘O’ (Overwrap) will be generated to detect the overwrap condition due to shortage of parallel evaluation elements.



With this basic structure, SVA2RTL compile generates RTL according to the sequence and the attributes, and constructs it by connecting child elements into larger block with the same interface.
Based on the error free attribute, some optimization is performed to omit the error signal generation when unnecessary.

5. Conclusion

With the detailed analysis of the property, SVA2RTL compiler can generate optimal RTL code from a SVA description. The compiler supports majority of SVA constructs, the concurrent evaluation mechanism, local variable and sequence match item mechanism which are crucial features for constructing effective monitor for modern high performance interconnect protocols.

The complier evaluates the attributes of each element in a SVA sequence and generates synthesizable RTL code, while maintaining the accuracy of the original description. The complier utilizes the pipeline implementation of delay element as much as possible and takes the advantage of its non-blocking nature for RTL generation.

The mechanism of using the two attributes, branching and blocking, shows a simple but effective method to generate necessary parallel evaluation units for general SVA descriptions.

6. References

[1] SystemVerilog 3.1a Language Reference Manual, Accellera, 2004

[2] Faisal I. Haque, Jonathan Michelson, Khizar A. Khan: The Art of Verification with SystemVerilog Assertions, Verification Central, 2006

[3] Micheal Pellauer, Mieszko Lis, Donald Baltus, Rishiyur Nikhil: Synthesis of Synchronous Assertions with Guarded Atomic Actions, 2005

[4] Sayantan Das, Rizi Mohanty, Pallab Dasgupta, P.P. Chakrabarti: Synthesis of System Verilog Assertions, 2006

[5] Ivan Kastelan, Zoran Krajacevic: Synthesizable SystemVerilog Assertions as a Methodology for SoC Verification, 2009