Tales from the Tech Trenches

SVA Properties for pipelined protocols

Doulos - March, 2009

All engineers who attend our courses are invited to send their questions to us in the weeks and months that follow the course, as they learn to apply the knowledge they gained during the few days of the course. The problem we're looking at here came from an email from an antendee, and he described his requirement to verify an interface that is simple to describe but harder to check.

SystemVerilog Assertions (SVA) have helped in verifying many designs and for some groups they have driven the adoption of SystemVerilog. They work particularly well on complex control logic with many states; the sort of code that is hard to exercise completely in simulation. They can also be useful for providing robust specifications of interface behaviour, a place where differences in interpretation can lead to hard-to-find bugs. However, the effective use of SVA is distinctly non-trivial. The grammar and syntax of SVA can be hard to penetrate, and even when you know the grammar it can be hard to create a property that you are sure describes your desired behaviour.

We assume readers are familiar with at least the basics of SVA.

The Problem

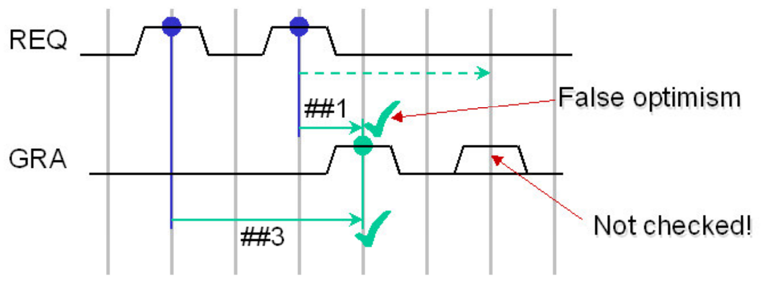
Two blocks communicate using a pair of wires, consisting of a request (REQ) and a grant (GRA). Requester asserts REQ to issue a request, which may happen in consecutive clock cycles. When Granter has completed the work associated with a request, it asserts GRA. Again, this may be in consecutive cycles. Granter can maintain four outstanding requests, and there must be two or more cycles between a request and its corresponding grant.

Issues to solve

It's the second part of the specification that presents the problem; how do we allow for overlapping activity in our assertion? Leaving aside the spacing of REQ and GRA, a simple property such as this

```
assert property (REQ |-> ##[1:4] GRA);
```

is not going to handle the situation, as we can see here:



The first of the two expected GRA pulses will cause the completion of two evaluations of the property, each started on the REQ pulse.

To deal with this, we need to solve two problems:

1. Often in an overlapping protocol there is some field in the data to enable matching of request to grant, but in our example this is not the case.
2. Each evaluation of the assertion must know to which request it relates, so that it can match itself to the correct grant.

SystemVerilog to the rescue!

The first problem has to be solved by adding some external code to count requests and grants. This is very simple:

```
bit [1:0] req_num, gra_num;
  always @(posedge clock) begin
    if (reset) begin
      req_num = 0;
      gra_num = 0;
    end else begin
      if (req) req_num++;
      if (gra) gra_num++;
    end
  end
```

As the maximum number of outstanding requests is four, it's convenient to use a two-bit counter.

The second problem means that we have to be able to identify each evaluation of the property; to put it another way, each assertion thread must be unique somehow. To do this, we make use of SVA's facility for local variables in properties. A variable declared in a property is created anew each time an assertion evaluation starts, and thus each evaluation can be distinct. For example:

```
property data_pipe;
  logic [31:0] v;
  ( $rose(load), v = data_in ) |=>
   ##[1:10] (done && (data_out == v));
endproperty
```

Notice the comma-separated lists of actions at each stage in the property; when the first item is found to be true, the list of actions that follow is performed. This mechanism can be used to record the values that are used in the evaluation of the property, and these values can then be passed out for use in functional coverage.

Developing the Property

In our example, each time a REQ occurs, we save the value of the satellite REQ counter in a local variable and use that to search for a grant that matches:

```
property req_gra_check;
     bit [1:0] num;
     (req, num = req_num)
       |->
       ##[2:$] gra && (num == gra_num)
        );
endproperty
```

Thus when a REQ is seen, the number is recorded. This then implies that two or more cycles later GRA must be asserted and the satellite grant counter have reached the correct value.

This is not adequate, however. Our specification says that the correctly-numbered GRA must occur two cycles after the REQ to which it refers. Our property does not currently include this behaviour, as there is no specification of what can or should occur during the delay of two or more cycles. To this end, I must add to the property the requirement that during the delay, if GRA is asserted, the grant counter must not have the number corresponding to the request we are following in this thread. Got that? Luckily, the code is easier than the text (a major plus for SVA there):

```
property req_gra_check;
     bit [1:0] num;
     (req, num = req_num)
       |->
          !(gra && (num == gra_num))[*2]
            ##[1:$]
          gra && (num == gra_num)
        );
endproperty
```

This now says that on a request with count N, grant N must be inactive for two cycles, before going active later. The repetition number $ means "end of simulation", and you must think very carefully if this is appropriate. My property will never fail if grant N never occurs.

This property is OK, but it does not deal with the other part of the requirement, that there be no more than four outstanding requests. How can we deal with that? Remember that our satellite counter loops through the values one to four. We can simply write that if request N occurs, grant N should occur before another request N:

```
property req_gnt_check;
     bit [1:0] num;
     (req, num = req_num)
       |->
        ( ##1 !(req && (req_num==num))[*1:$] )
          intersect
        (
          !(gra && (num == gra_num))[*2]
          ##[1:$]
          gra && (num == gra_num)
        );
  endproperty
```

Notice the use of the `intersect` operator. This length-matching sequence "and" allows us to specify that the two operands must start and finish in the same cycle Thus, however long it takes for the grant N to occur, there must not be another request N during the whole of that time. If one occurs, the property fails to hold.

Concluding Thoughts

We have shown how some useful features of SVA can help create a property that dsecribes our stated behaviour. We have also illustrated that SystemVerilog properties require a great deal of thought and attention in their development. They make it possible to define unambiguous specifications of bahaviour, but they also make it possible to write a property that has little to do with the behaviour you thought it had.

This is a serious issue: how can we make sure that an assertion of even this moderate complexity is correct and complete? It is a question that is asked in almost every SystemVerilgo course we run. There are no really good answers, alas. Following much discussion with course attendees, we have come to some conclusions about assertions get used in a real project:

- You often get assertions wrong. Usually no problem; they fail, you debug, you fix it. Just like anything else. At least you have cross-checking of design vs. assertion, and since the mental processes you go through to build those two things are so very different, the risk of common-mode error is fairly small.
- My assertions are probably incomplete, much in the same way that the specification is probably incomplete too. That doesn't detract from the fact that some high-quality checking, even if incomplete, is much better than none. Assertions are an easy way to add more checking. And it's easy to add more of them later, when you spot the fact that you missed something.
- Assertion coverage keeps us honest, providing we're assiduous about following up anything suspicious-looking in the coverage report. As a trivial example, an assertion that fires more times than the clock.

- SVA (with its local variables, actions on proposition match, and assertion action blocks) gives us plenty of opportunity to instrument my assertions so we can check they're firing when and how we think they should. Not the whole story, but one more useful tool.

A new generation of tools is beginning to appear that use formal methods and various forms of fault modelling to identify possible brokenness in your design that would not be caught by any of your assertions. Of course, they can't say whether your design is actually broken because they don't know the specification you're working to, but they will help in identifying areas of your design and testbench where checking is weak.

Even though SystemVerilog Assertions can be hard to write and harder to debug, we consider that assertion-based verification is A Good Thing. It presents another weapon in the armoury of the team attempting to produce, correctly, ever more complex devices to ever tighter timescales.