Recommendations for Developing an Assertion Based Protocol VIP for Formal Analysis

Amit Gurung
Cadence Design Systems, India
agurung@cadence.com

Vikas Roy
Cadence Design Systems, India
vikasr@cadence.com

Abstract

The paper discusses the issues of under constraining, over constraining and cul-de-sac which surfaces during the development of protocol based verification IP for formal analysis. The paper provides recommendations for the developer of the assertion IP, which would help in improving the overall quality.

1. Introduction

The process of developing assertions to validate a Design Under Verification (DUV) using formal analysis has been prevalent in the industry for a long time now. However the process of developing Formal Verification IP (FVIP) as a full fledged product is a relatively new development, and hence offers many unexpected challenges. These challenges, if not addressed, may adversely affect the quality and effectiveness of the FVIP.

This paper discusses a number of such challenges and offers solutions to overcome them. The proposed solutions in this paper have emerged from our experiences in developing assertion based VIP for formal analysis. Though the issues discussed in this paper are known in the model checking research world, but it is observed that engineers applying this technology are not fully cognizant of the property modeling styles which causes these issues. The paper attempts to describe these issues in a very simple manner by sighting real life examples and providing their solutions.

This paper would appeal to people interested in formal analysis in general and developers and users of assertion based components in particular. The paper would help in appreciating the complexity involved in developing FVIP and how using the proposed solutions make the FVIP robust and comprehensive.

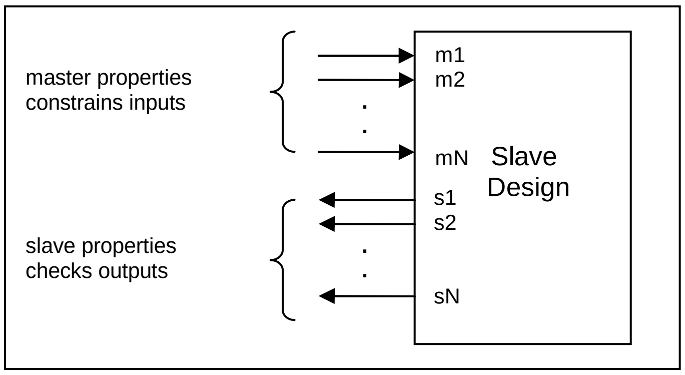
Section 2 describes FVIP development flow, Section 3 explains under constraining and causes. Section 4 discuss over constraining. Importance of Coverage is discussed in section 5.

2. FVIP Development Flow

The task of developing a FVIP begins with converting the rules described in the specification document to a set of properties which specify temporal relationship between signals on the interface. The protocol specification describes the behavior of each device connected to the interface and hence the property set can be classified according to the expectations of the device on the interface. For example, the protocol AMBA AXI has a master slave topology and hence the properties in the FVIP can be classified into master properties and slave properties. A master or slave properties describes the behavior of an ideal master or a slave device, and are prefixed with master_ or slave_ respectively.

Figure 1 describes a typical formal analysis environment for the verification of a slave design. The signals m1,m2 …mN are inputs to the slave design and signals s1,s2..sN are outputs. In verification of a slave design, the master properties in the assertion IP are used as constraints. These constraints are capable of generating all legal protocol sequences on the inputs. The slave properties in the assertion IP are used as assertions and they are proved under the assumptions of the constraints.

Fig 1


master properties constrains inputs
slave properties checks outputs
m1
m2
mN
s1
s2
sN
Slave Design

A similar environment is used for verification of a master design where the constraints and assertion reverse their roles.

3. Incomplete Specification or Under Constraining

Despite converting every rule specified in a standard specification to a property in the VIP, it is seen that the property set is not complete to describe the entire scope of the protocol. This usually occurs in protocol scenarios where the specification is silent about the design implementation.

For example in AMBA AXI protocol, the specification does not state when (or within what limits) should the slave accept a request (by asserting AxREADY) when offered by the master (by asserting AxVALID).

In such a case, an additional property (which is not specified in the standard protocol specification document) should be included, which mandates that AxREADY should eventually be asserted when AxVALID is asserted.

In the absence of the above mentioned property, formal analysis of a master DUT could result in many failures in transaction level checks due to AxREADY never being asserted, as the formal analysis tool was free to exercise any value of AxREADY in order to fail an assertion.

The above mentioned scenario is a case of incomplete specification or under constraining. Under constraining results in generation of illegal sequences by the formal tool.

Recommendation 1:

All protocol scenarios (especially relating to latency) where the specification is silent about design implementation, should be judiciously included in the property set.

4. Over Constraining

During an analysis of a design, an assertion on the output is proved for a given set of constraints written on the inputs. Constraints describe the legal set of sequences which can be applied on the inputs. Over constraining refers to reducing the state space which the formal tool explores, to a reduced subset of the legal permissible set.

Every protocol specification constrains the state space of formal analysis to an optimal set. Any further additional constraining leads to an over constrained set. In such cases, though the constraints generate legal sequences of inputs, they are not capable of generating all legal sequences as described in the specification.

Since the formal analysis tool has no knowledge of the entire legal state space specified by the specification, it is not possible for the tool to detect an over constrained environment. Hence it is imperative that adequate steps are taken to detect an over constrained environment.

4.1 Indirect Constraining

It is common to make use of auxiliary HDL code to simplify a property implementation. In some cases the combination of HDL logic and constraints leads to constraining of primary inputs, even if no direct constraints are applied on them. This type of constraining might be unintended and may lead to over constraining of the primary inputs.

The following example of a combinational circuit, is used to demonstrates this effect. We want to check that the output port out2 does not take a value 3'd4 as specified by the assertion A1. Since we want to check this for all values of inputs, we have not written any direct constraints on them. Now assume that there is a wrongly placed constraint as specified by C1.

```
module (
input [1:0] in1,
input [1:0] in2,
.
.
input [1:0] inN
output [2:0] out1,
output [2:0] out2
.
.
output [2:0] outN
);

assign out1 = in1 + in2;
assign out2 = ~out1;

// assignment of other outputs with different
// implementation logic
..
endmodule

// psl A1 : assert never(out2 == 3'd4);
// psl C1 : assume never(out1 == 3'd3);
```

If all values of inputs were exercised on the design, the assertion A1 should fail, but with the inclusion of a wrong constraint C1 in the formal environment, the assertion results in a pass. Hence the constraint C1 over constraints the formal environment such that those values of in1 and in2 are not exercised which results in out1 attaining a values of 3. Moreover all the other assertions written on other output ports are also proved in this reduced state space.

Hence as demonstrated by the above example, the results of a formal run are as good as its constraints. Over constraining may lead to erroneous results being reported. Though the above example is very simple to understand, complex circuits may have very complex relationship between constraints and often it is extremely difficult to detect over constraining.

Recommendation 2: Be cautious when selecting constraints for analysis. Avoid writing constraints on the output of a design.

In general, it is a good practice to write constraints on the inputs of a circuit only. It can be observed that the above constraint which causes indirect constraining of the primary inputs was written on the output of the circuit.

4.2 Cul-de-sac

Assertions can be of the form such that it has an enabling condition and a fulfilling condition. During formal analysis if such an assertion passes with its enabling condition also getting covered, one normally assumes that the fulfilling condition is bound to happen when the enabling condition occurs. However, there are cases when this assumption does not hold true. There can be finite paths in the design which prevents the tool from exploring new states while proving a property. This would result in the fulfilling condition not getting covered during the analysis. This condition is referred to as cul-de-sac or dead end.

Consider the same example with slight modification. The circuit is sequential in nature with clock and reset.

```
module (
input clk;
input resetn;
input [1:0] in1,
input [1:0] in2,
output [2:0] out1,
);

always @(posedge clk or negedge resetn)
   if (~resetn)
      out1 <= 3’d0;
   else
     out1 <= in1 + in2;

..
endmodule
// psl default clock = posedge clk;
// psl C2 : assume never(out1 == 3’d3);
// psl A2 : assert always {in1==1 & in2==2}
// |=> {out1 == 3’d6};
// psl A3 : assert always {in1==0; in2==1;
// in1==0 & in2==3 } |=> {1’b0};
```

In the above example C2 is a wrongly placed constraint. The constraint states that out1 can never take a value 3. Since there are no constraints on inputs, a state is possible in which in1 and in2 has values such that their present sum is 3. The enabling condition (also called trigger check) gets covered in a formal run. However the design logic is such that out1 would have a value 3 in the next clock. Hence we see that the design implementation conflicts with a constraint in the future. This introduces a finite path in the design which truncates with in1 and in2 taking values such that their sum is 3.

If an assertion is written similar to A2 and A3, such that the last state of the enabling condition is the last state on a finite path then the analysis would always report the assertion as a pass, no matter what is written as the fulfilling conditions. In the above example assertions A2 and A3 results in a pass. Results of analysis of a formal environment having finite paths are normally not intuitive and mostly misleading.

Recommendation 3: Always write a corresponding cover check for all assertions

We propose a solution here to detect erroneous passes which were a result of cul-de-sac issues.

For a property of the form

```
// psl P : assert always {SERE1} |=> {SERE2};
```

write a cover check of the form

```
// psl P_cover : cover {SERE1;SERE2};
```

To ensure a valid pass of an assertion (i.e. when an enabling condition occurs, only the fulfilling condition can and does occurs), the assertion should pass and the cover should also have a witness. Hence it is recommended that every assertion should be checked for validity by writing a corresponding cover check.

The above recommendation applies to properties which are selected as to work as constraints as well. Writing a corresponding cover check for constraint ensures that the constraint is not redundant in the formal setup. If the cover check of a constraint fails, it indicates an over constrained environment. This usually helps in debugging the cause of over constraining.

4.3 Context of a property

Though the earlier section lays great emphasis on refraining from writing constraints on outputs of a design, there are cases such that in order to model complex part of the protocol specification constraints involve both the inputs and outputs. In a bus based design a lot of transaction information has be latched when a transfer occurs. A transfer normally occurs when a master device places some request on the bus and the slave accepts them. Hence it is seen that constraints involving information latched from a transfer would inadvertently involve both the master and slave signals.

Recommendation 4: For properties involving signals from both the inputs and outputs of a design, always get the context of the property right. Context of a property implies knowing whether a property is a slave property or a master property.

The following case describes an incorrect property modeling style which causes ambiguity in the context of property.

In AMBA AXI protocol, a write request transfer occurs when both AWVALID (a master output signal) and AWREADY (a slave output signal) are asserted. Similarly a write response transfer occurs when both BVALID (a slave output signal) and BREADY (a master signal) are asserted. The protocol mandates that every write request should be returned with a response. Now consider a coarse implementation of the same

```
// This wire implies a write transfer
wire write_request_transfer= AWVALID & AWREADY;

// This wire implies a write response transfer
wire write_response_transfer = BVALID & BREADY;

// psl resp_for_every_req : assert always (
// write_request_transfer ->
// eventually! (write_response_transfer) );
```

The context of the property “resp_for_every_req” is not correct as it does not clearly describe whether the fulfilling condition is the responsibility of a master device or a slave device. It is not clear that for verification of a slave design, whether this property is to be picked as constraints or as an assertion and the same logic applies to verification of a master device too. Having an incorrect context of a property may lead to over constraining or cul-de-sac issues discussed earlier.

The property can be implemented with the correct context when responsibilities are distributed for individual devices. For example the same protocol scenarios when expressed as follows set the expectation of the devices correctly.

```
// mandate the slave to provide a response by
// asserting BVALID when a write request
// transfer occurs
// psl slave_resp_for_every_req : assert always
// ( write_request_transfer ->
// eventually! (BVALID) );

// mandate the master to eventually accept all
// write responses
// psl master_eventally_accept_write_response :
// assert always (BVALID ->
// eventually! (BREADY) );
```

The above mentioned master property also falls under the ambit of recommendation1.

4.4 Over constraining due to design implementation

Assertion IP developers devote huge amount of time and effort in ensuring that the bundled property set is the most optimal set to describe the full scope of the protocol specification. Comprehensive testing strategies are used to prove that is no over constraining introduced by the assertion themselves. But it is not true that the same set of properties would not be over constrained when applied onto different designs.

Consider the following example. The inputs in1 and in2 are not directly constrained as we want to check all assertions for all possible combinations of the inputs. Design1 and Design2 have different implementation of the output port out1. Constraint C1 states that in a cycle where in1 and out1 are 1, in2 should be 1.

```
module master (
    input                 in1;
    input                 in2;
    output                out1;

assign out1 = in1 & in2;   //Design1
assign out1 = in1 | in2;   //Design2

endmodule

// psl C1 : assume always ((in1 & out1) ->
in2);
```

Now let us analyze the combination of values possible on the ports in1, in2 and out1 for both the designs

For design1

```
in1  in2  out1
--- ---- ------
0    0    0
0    1    0
1    0    0
1    1    1
```

For design2

```
in1  in2  out1
--- ---- ------
0    0    0
0    1    1
1    0    1  ← conflicts with C1
1    1    1
```

A formal analysis tool always abides to the constraint of the environment while proving an assertion. In Design1, the tool exercises all combination of in1 and in2 for the analysis. In Design2, since the design implementation is such that if the tool were to apply values in1 == 1 and in2 == 0, the output would be 1 which would conflict with the constraint C1. Hence in order to obey the constraint, the tool never exercises these values while proving all the assertions in the design. This results in over constraining of the primary inputs even though no direct constraints were applied on them.

Recommendation 5: Be aware that certain design implementations may reduce the input state space, hence always rely on coverage before accepting the results of an analysis.

5. Coverage

Coverage analysis of a formal environment is an important part of a formal analysis. As described in earlier sections of this paper, there are myriad reasons for over constraining or under constraining in a formal environment. Coverage metric can provide meaningful and sometimes startling information about the quality of the analysis. It is imperative that coverage information be reviewed before accepting the results of formal analysis. Coverage statistics on the output of a design provides information about the subset of the protocol the design has implemented. Coverage statistics of the inputs provides information of the quality of the test sequences exercised on the design.

A formal VIP should provide the following type of coverage.

Sanity covers: These checks are very basic in nature but are extremely effectively in detecting over constrained environment. They check whether a port can ever have a value 0 or 1 and whether the value on the port can ever rise or ever fall.

Value covers: These covers checks whether a multibit port can take all possible values

Positive covers: These covers checks whether specific legal protocol sequences occur on the interface. Failure of these covers on the inputs of a design indicates an over constrained environment.

Negative covers: These cover checks whether specific illegal protocol sequences do not occur on the interface. A witness of these covers on the inputs of a design indicates an under constrained environment.

Cross covers: These covers checks whether specific combinations of values can occur on the interface.

6. Conclusions

The paper explains the causes of under-constraining, over-constraining and cul-de-sac during the development of Formal VIP. The recommendations for eradicating these problems have also been discussed. The paper also touches upon the importance of coverage in formal analysis.

7. Acknowledgments

The authors wish to acknowledge the assistance provided by the members of formal analysis tool development team in explaining the tool behavior in these complex cases.

8. References

1. AMBA AXI 1.0 Specification
2. FAQs : Incisive Formal Verifier