Considering Static Functional Verification of Digital Systems for HDL-based Courses

Mehran Massoumi

Department of Mathematics & Computer Science
California State University, Hayward
Hayward, CA 94542
massoumi@sbcglobal.net

Abstract: Functional verification of VLSI designs is known to be a highly time-consuming phase of the design cycle. Furthermore, with increasing complexity of ASICs and programmable ICs, this problem is becoming even more challenging. An emerging approach, which helps shorten the verification cycle, is the formal approach in which mathematical techniques are used to prove properties about a design. Due to the fact that formal checks are exhaustive and no test-vectors are needed, supporting tools have gained significant momentum as an add-on solution to simulation. It is the focus of this paper to present the formal or static approach and encourage use of the available tools in design projects for Senior/Graduate-level HDL-based courses. Advantages of the static approach will be discussed by presenting property formulation for a few RTL designs. Moreover, the property language PSL (Property Specification Language), which has the promise of becoming an IEEE standard, will be used in presenting property formulations.

Introduction:

A widely practiced approach for functional verification of digital designs is the use of simulation tools and test-bench models. Although simulation is a very effective solution, designers today are continuously faced with the challenge of exhaustive and timely verification of large systems. Many designs are taped-out with corner bugs, which is the result of non-exhaustive simulation coverage. Furthermore, with the growing complexity of IC designs and programmable parts, simulation runs are becoming prohibitively long and non-convergent. One solution, which was first commercially introduced in the mid 1990s, is the use of static tools[1] in addition to simulation. The Static approach consists of formulating properties about a design (instead of using test-benches) and proving those properties or assertions using mathematical proof techniques. A proven property is exhaustive in that it holds for all possible states, and sequences of inputs of the design. In today's design community, static functional verification is gaining momentum as a solution for block-level verification of IC designs. The premise is that if individual blocks in a design are proven correct, then system-level verification will be a more manageable task. However, due to the capacity limitations of formal techniques, simulation is still needed for the system-level verification, where the size of designs can be prohibitively large for any formal tool to be useful.

Considering the significance of the verification problem and the emergence of static tools as an add-on solution to simulation, it is the focus of this paper to present the static approach and

encourage the use of available tools in the design projects for Senior/Graduate-level HDL-based courses. All commercial static tools today support Verilog[3] or VHDL[4], both IEEE standards, as the language to describe designs. Moreover, efforts are underway to formulate and standardize the property language PSL (Property Specification Language)[2], which will support both the Verilog- and VHDL-flavor. Most verification tools today support the existing definition of PSL.

Advantages of the static approach will be discussed in this paper by presenting property formulation for a few RTL designs. These advantages are key in finding costly functional problems before proceeding to the system-level verification. Typical design scenarios are presented and properties are developed in order to do an exhaustive functional verification statically. With PSL gaining momentum as a property specification language of choice, properties in this paper will be developed using PSL constructs, including temporal expressions and sequences. Furthermore, the results of verifying a set of properties and uncovering functional bugs will be presented. It will be evident from the contents of this paper that static tools can be readily integrated into HDL-based courses to complement simulation. Learning PSL and property formulation will structure the verification efforts in that it will promote a discipline for exhaustive functional coverage. This discipline is equally applicable to formulating simulation test bench models.

Property Formulation:

The most fundamental difference between the static and dynamic approaches is that the latter uses a set of stimulus input vectors to drive a design and produces a set of corresponding output values. Subsequently, these output values would have to be inspected for expected behavior. Clearly, the completeness of functional coverage is dependent upon the quality and completeness of the input stimuli. In the static approach, however, the verification is accomplished by first formulating what is referred to as properties, which capture what the design should always do and what it should never do. Subsequently, the tool will prove whether a given property will hold under all input and state conditions. If not, a trace or a set of input and state conditions will be produced illustrating why the property failed. No test bench or inspection of any simulation output is required.

A property is a logical expression similar to those in Verilog or VHDL. Additionally, a property must capture temporal relationships among expressions. A commonly used operator in property formulation is the logical implication, which can be defined to include a number of temporal characteristics. The following table illustrates the implication operators supported by PSL. In this table, both *A* and *B* represent expressions that may take several cycles before their truth-value is established.

Operation	Function
A -> B	If A is true then B must hold. Both A and B are evaluated concurrently
A <-> B	If A is true then B must hold and if B is true then A must hold. Both A and B are evaluated concurrently
A |-> B	If A is true then B must hold. B is evaluated in the last cycle of A
A |=> B	If A is true then B must hold. B is evaluated after the last cycle of A

Consider a Moore state machine, which detects "110" on input *din* and asserts its output *match* if a match is found. Verilog statements for this state machine are listed below.

```
always @(posedge clk)
        shift_reg <= {shift_reg[1:0], din};

assign match = shift_reg[2] && shift_reg[1] && !shift_reg[0];
```

In general, the best approach is to start from the specification and formulate PSL assertions that capture the expected functionality. Moreover, in large designs, knowledge of the design architecture is helpful in achieving full coverage efficiently. The following property reflects the expected behavior.

```
assert always (din && (next din) && (next[2] !din)) -> (next[3] match);
```

Furthermore, it should be proved that there exists no other condition under which a match is detected. This is accomplished by reversing the original property, as shown below.

```
assert always (next[3] match) -> (din && (next din) && (next[2] !din));
```

Uses of the PSL "next" operator, in the above assertions, capture the temporal relationships on both sides of the implication operator. The entire property spans four clock cycles, which includes the present cycle and three cycles into the future. The truth-value of the left-hand-side in the first property is established in three cycles and that of the right-hand side in four cycles. The following two assertions make use of the other PSL implication operators but capture the same identical behavior as the above assertion.

```
assert always {din; din; !din} |-> {true; match};
assert always {din; din; !din} |=> {match};
```

Both sides of the implication consist of a sequence, enclosed in {…}. A sequence specifies any number of events, which are to take place in multiple clock cycles. As shown, a semi colon separates each event. For example, the sequence "{din; din; !din}" specifies the expected input pattern and in case of "|=>", the right-hand-side expression is evaluated after the input pattern is detected. However, in case of "|->", the right hand side is evaluated concurrent with the cycle in which "!din" is detected, hence the use of the keyword "true". All of the above properties pass verification.

In addition to considering design specification when formulating properties, one can prove useful design characteristics that may result in updating the specification and/or finding a functional error. A useful assertion for this design is to prove that once a match is found, there shall be no more match for at least the pursuing two clock periods.

```
assert always match -> (next_a[1:2] !match);
```

However, an assertion, which checks for at least three cycles between consecutive matches, fails as expected and a signal trace showing wire and state values for four clock cycles is generated.

assert always match -> (next_a[1:3] !match);

	t0	t1	t2	t3
din	1	1	0	0
match	1	0	0	1
shift_reg[2:0]	110	101	011	110

The shift register points out that a pattern of "110" has already been received and as a result *match* is asserted at time t0. Moreover, starting at time t0 another pattern of "110" is initiated resulting in a match at time t3, thus violating the property.

Formal Verification vs. Simulation:

Formal or static tools provide certain functional coverage, which is not readily achievable by simulation, if at all possible. This is illustrated in the verification of the following Verilog description, which is a 4-bit BCD counter with *load* and *enable* inputs.

```
module cnt(clk, rst, load, enable, cout, data);
        input clk, rst, load, enable;
        input [3:0] data;
        output reg [3:0] cout;

        always @(posedge clk or posedge rst)
                if (rst) cout <= 0;
                else if (load) cout <= (data > 9) ? 0 : data;
                else if (enable) begin
                        cout <= (cout >= 9) ?  0 : cout + 1;
                end
endmodule
```

Considering the design specification, one may contrive a set of properties as shown below.

```
assume always !rst;
assert always (load && (data <= 9)) -> ((next cout) == data);
assert always (load && (data > 9)) -> ((next cout) == 0);
assert always (enable && !load && (cout < 9)) -> ((next cout) == cout + 1);

assert always (enable && !load && (cout >= 9)) -> ((next cout) == 0);
assert always (!enable && !load) -> ((next cout) == cout);
```

The above properties can be readily verified using a simulator as well, although such verification is not likely to be exhaustive. Essentially, in simulation environments, assertions must always hold as test vectors are applied to the design. The exhaustiveness of the coverage per assertion is

determined by how well the test bench is generated. Many simulators support PSL assertions today.

Formal tools, on the other hand, guarantee exhaustiveness per assertion. Moreover, certain properties such as proving existence of only a finite set of conditions under which an operation takes place is not easily possible using test vectors. Consider the following property, which verifies exhaustively that output *cout* may undergo one and only one of the following four operations. In other words, *cout* may be (1) incremented, (2) reset to zero, (3) set to input data, or (4) not changed.

```
assert always ((next cout) == cout + 1) || ((next cout) == 0) ||
                ((next cout) == data) || ((next cout) == cout);
```

Such properties explore the design space and uncover corner problems, which may result in changing the design or the specifications. Another case is to verify that the only condition that will cause the counter to be incremented is when *enable* is active, *load* is inactive, and *cout* is less than 9.

```
assert always ((next cout) == cout + 1) -> (enable && !load && (cout < 9));
```

This property fails verification and the following trace table is generated.

	t0	t1
cout[3:0]	0000	0001
data[3:0]	0001	0001
enable	0	0
load	1	0
rst	0	0

This trace illustrates that the value of *cout* is incremented from 0000 to 0001 at time t1 but the conditions are not as expected. *load* is active and the value of *data*, which happens to be the increment of *cout* is clocked into the counter. This serves as a useful knowledge about the design and may cause functional errors in interfacing modules. Once the property is reformulated to include this new condition, it will pass verification, which proves existence of only two conditions under which *cout* is incremented.

```
assert always ((next cout) == cout + 1) -> ((enable && !load && (cout <= 9)) ||
                (load && (data == (cout + 1))));
```

Lastly, the following assertion proves that the output *cout* will always be a valid BCD.

```
assert always (cout <= 9) && (cout >= 0);
```

In other words, there exists no conditions, post initial reset, under which the value of *cout* will be greater than 9.

Why Static Tools in HDL-courses?

As noted in the above discussion, formal tools yield a higher degree of confidence that a design meets its specification and that no corner issues exist. Additionally, such tools offer a unique way of exploring the design space without any test vectors. Integrating assertion-based verification as well as static tools into HDL-based courses are favorable for a number of reasons listed below.

- Many companies in the design community are adopting assertion-based verification in their tool flow. Today, PSL assertions are being used in both static and dynamic environments. Therefore, learning PSL prepares students for emerging methodologies in simulation. Unlike static tools, the assertion-based verification within simulation environments is test-vector driven and not exhaustive. Hence, an assertion may only fail if a particular test-vector causes a violation.
- Many IC design companies are adopting static tools for block-level verification. The exhaustive functional coverage that such tools provide is very beneficial in achieving complete functional coverage and avoiding recalls of parts with corner bugs.
- PSL is emerging as a standard assertion language. Such development attests to the fact that assertion-based verification is of significant interest among designers. PSL is a natural extension of Verilog or VHDL and can be readily learned.
- Static tools provide a unique way of exploring the design space and therefore help students observe all corner behaviors of any given design. Such knowledge and experience fosters the discipline for writing accurate specifications as well as for attention to details during the design.
- Static tools do not require any changes to existing verification flows. Property verification can be integrated seamlessly into any simulation environment.

Conclusion:

Property formulation using PSL was presented in this paper. Such properties or assertions can be verified in both static and dynamic verification environments. However, the static approach yields an exhaustive functional check per property for all possible input sequences. Moreover, in contrast to simulation, static checks provide a way of exploring the design space for corner bugs. Considering the importance of PSL in both static and dynamic verification and considering the advantages of using static verification, integrating static PSL verification in HDL-based courses will no doubt better prepare students for today's design challenges.

Bibliography:

[1] Averant Inc., www.averant.com, Solidify User's Manual, Version 3.0.19, December 2004.

[2] Accellera Property Specification Language, Reference Manual Version 1.1, June 2004.

[3] IEEE Standard Verilog Language Reference Manual, IEEE Standard Number 1364-2001, 2001.

[4] IEEE Standard VHDL Language Reference Manual, IEEE Standard Number 1076-2002, 2002.

Biography:

MEHRAN MASSOUMI received his Ph.D. in Electrical and Computer Engineering from the University of Arizona in 1994. He has worked with a number of Design Automation companies, where he was responsible for HDL synthesis products. Currently, he is with Averant Inc, a verification company he co-founded in 1998, and has also been lecturing at the CSU, Hayward.