Assertion Synthesis: Enabling Assertion-Based Verification
For Simulation, Formal and Emulation Flows

Progressive, Targeted Verification through Assertions and Functional Coverage Properties

Yunshan Zhu
Co-founder and CEO
NextOp Software, Inc

Yuan Lu
Co-founder and CTO
NextOp Software, Inc

As design complexity grows, it is imperative that an understanding of the design’s structure and intent be infused into the verification process to have confidence that the RTL functional verification process is complete. Regardless of the speed of the simulator or formal engine, the verification result is only as good as the specification. Without an adequate specification, the debugging cycle will continue to increase, and design and verification teams will be unable to adequately reduce the risk of chip defects that can cause re-spin costs and schedule overruns.

Assertion-based verification helps design and verification teams using simulation, formal and emulation methodologies accelerate verification sign-off by enhancing the RTL and test specifications to include assertions and functional coverage properties, which are logic statements that define the intended behavior of signals in the design.

The emergence of assertion synthesis will allow for true proliferation of assertion-based verification by automating the painful manual process of creating adequate whitebox assertions and functional coverage properties with sufficient capacity to handle complex SoC designs (see Figure 1).

Assertion synthesis enables a progressive, targeted verification process, allowing design and verification teams to more easily uncover corner case bugs, expose functional coverage holes, and increase verification observability.

Figure 1: Assertion Synthesis as Enabling Mechanism to Assertion-based Verification
Current Functional Verification Limitations: Inadequate Specifications

Today’s verification methodologies include a combination of directed simulation, constrained random simulation, formal and semi-formal methods and emulation.

Directed simulation - utilizes blackbox checkers which test input and output behavior for each feature interaction. This approach is fundamentally not scalable due to the number of complex interactions between features.

Constrained random simulation - utilizes an external checker to define the expected behaviors of the Design under Test (DUT). During simulation, the output of the DUT is compared with the checker, and mismatches are used to identify bugs. The checker and the DUT are typically developed independently based on the architectural specification. It is difficult to write a checker to exactly match the DUT. As a result, features and interactions are often skipped by the external checkers, including performance related features, exception and interrupt conditions.

Formal verification - uses mathematical analysis to prove or disprove certain properties for all possible legal input stimuli. Complete verification using formal methods requires that users specify sufficient properties to cover all features of the design.

Emulation - uses special hardware to exercise the DUT at much higher speed than simulation. Emulation shares some of the same challenges as constrained random simulation in developing external checkers. In particular, it can be difficult to localize and debug an error when an emulation run fails.

All of these methods depend upon the quality of the specification, where we define the specification as the combination of:

- Whitebox assertions and blackbox checkers - to represent the behavior of the RTL design
- Functional coverage points - to ensure proper test stimulus.

Assertion-Based Verification: Targeted Verification via Robust Specifications

Assertion-based verification enhances directed and constrained random simulation, formal and emulation verification approaches through enhanced specifications – supplementing blackbox checkers with whitebox assertions and functional coverage properties.

- Whitebox assertions specify the behaviors of the internal logic and inject observability into the Register Transfer Level (RTL) code. In contrast, traditional blackbox checkers specify the input and output behaviors of the DUT. Features that cannot be captured using blackbox checkers can often be effectively done with whitebox assertions.
- Assertions ensure the correctness of the implementation logic; the number of assertions needed to verify the design scales linearly with the complexity of the RTL.

- Whitebox functional coverage properties expose corner case behaviors created by implementation and ensure such behaviors are targeted by simulation test vectors.

- Whitebox assertions pinpoint the problems in the RTL when they are triggered, reducing overall debugging time.

- Assertions and functional coverage properties can be shared across all verification platforms, including simulation, formal and emulation, and allow cross checking between different test environments. They also facilitate design reuse by detecting integration errors.

**Assertion-Based Verification Hindrances**

As design complexity increases, assertions and coverage properties are well-recognized as an important adjunct to the blackbox checkers used in simulation and formal verification flows. However, adoption of assertion-based verification has been slow for the following reasons:

- It is infeasible to manually generate an adequate number of assertions to the level needed for thorough identification of design problems. It is considered generally desirable to have one assertion for every 10 to 100 lines of RTL code, yet it can take hours to create, debug and maintain each assertion. Additionally, high quality assertions require different and often orthogonal perspectives, which a single designer lacks.

- It is an unwieldy and time-consuming process to manually create sufficient functional coverage properties to identify which tests are missing. Coverage properties are a critical element to prevent defects, yet high quality coverage properties are as tedious and difficult to write as high quality assertions. Additionally, many useful functional coverage properties are non-intuitive as they address the behaviors verification engineers did not think of initially.

- The high learning curve for standard assertions languages such as SystemVerilog Assertion (SVA) and Property Specification Language (PSL) is a high barrier to adoption.

**Assertion Synthesis**

As shown in Figure 2, Assertion Synthesis is technology that automatically synthesizes:

- **High quality assertions** to capture key design constraints and specifications. The assertions offer orthogonal perspective from the RTL implementation.
- **Functional coverage properties** which expose holes in the testbench. The coverage properties are functional and are independent of the syntax of the RTL.

*Figure 2: Assertion Synthesis*

By running Assertion Synthesis in conjunction with the RTL simulator, the synthesis process can capture dynamic behaviors in the whitebox assertions and coverage properties. To be effective, assertion synthesis technology must have sufficient capacity to support full SoC designs, with run-time performance scaling linearly with respect to design complexity. The properties are automatically optimized for maximum performance, incurring minimal overhead to simulation regression. Assertions that contain temporal operators can capture sequential behaviors and to identify issues among multiple clock cycles. By automating the time-consuming process of generating assertions and coverage properties, assertion synthesis allows design and verification teams to achieve assertion-based verification in a timely and resource efficient manner.

**Assertion Synthesis as part of Assertion-based Verification Methodology**

With existing verification approaches alone, it is difficult for verification engineers to determine whether their designs have been sufficiently tested to be declared functionally correct and ready for implementation in silicon. Assertion synthesis adds a mechanism to measure the progress towards this verification confidence over sequential verification cycles. Assertion synthesis can be employed into current digital design and verification flows, according to the use model shown in Figure 3 below.

*Figure 3: Assertion Synthesis as part of Simulation, Formal and Emulation Verification Flows*
1. Engineers use their RTL design, and test information as input to the assertion synthesis tool.

2. The assertion synthesis tool automatically generates properties that are guaranteed to hold for the given stimulus set. The initial prevalence of new coverage properties provides an indicator that there are numerous holes that the verification runs have not covered. Correspondingly, the assertions provide unique ‘white-box’ observability into each block’s targeted behavior, which if triggered clearly identify design bugs.

3. Engineers classify the automatically generated properties as either assertions or coverage properties. The tool automatically outputs the classified properties in standard formats including SVA, PSL, or Synthesizable Verilog.

4. The designers bind the assertions to 3rd party simulators, emulators and formal verification tools. Throughout the iterations, the assertions are bound to the corresponding RTL, making sure any RTL modifications do not introduce new bugs. When an assertion is triggered, it pinpoints the location of a bug and therefore reduces debug turnaround time.

5. The verification engineers generate stimulus to patch coverage holes.

6. Steps 1-5 are repeated until verification sign-off. As development teams progress through their design and verification iterations, the number of coverage properties produced by assertion synthesis will decrease relative to the number of assertions, until such point that there are almost exclusively assertions with only minimal coverage properties.

The fundamental hindrance to assertion-based verification has been the time-consuming task of creating high quality assertions and functional coverage properties. Design and verification teams can now achieve verification observability with assertion synthesis, mitigating the risk of project delays and defective silicon. Furthermore, the assertions clearly describe the design intent for each block; thus the assertion continues to provide confidence that the design integrity is intact regardless of how the block is used in the design, facilitating future RTL reuse after the completion of a project.

The complete design specification becomes both the RTL and the full complement of assertions; any future incorrect usage of the RTL will be caught by the assertions.

**Case Study: Assertion Synthesis in a Random Simulation Flow**

Figure 4 illustrates an IP design that accepts linked packets, dissembles them into data cells and stores them in the memory management unit (MMU). It also reassembles the data cells into linked packets and sends them out. The IP accepts flow control signals from both
external interface and MMU. It manages the data transfer according to physical bandwidth limitation.

![Diagram of linked packets](image)

**Figure 4: IP to process linked packets**

The following code sample defines some of the internal registers used for flow controls.

```verilog
always @(posedge clk or negedge rst)  
  if(!rst)  
    rel_cnt <= 0;  
  else  
    rel_cnt <= wr_rel_cnt ? rel_cnt + wdata :  
      inc ? rel_cnt - 1 : rel_cnt;

...  
always @(posedge clk or negedge rst)  
  if(!rst)  
    buff_cnt <= 0;  
  else if(wr_buff_cnt)  
    buff_cnt <= load_data;  
  else  
    buff_cnt <= dec ? buff_cnt - dec_cnt :  
      buff_cnt;
```

Assertion Synthesis is run using the nightly simulation regression tests of the IP design. Some of the properties generated are listed in the 1st Column in Table 1.

<table>
<thead>
<tr>
<th>Properties Generated by Assertion Synthesis</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>wr_rel_cnt l-&gt; rel_cnt == 0</td>
<td>Update only happens when rel_cnt is 0; coverage</td>
</tr>
<tr>
<td>dec l-&gt; buf_cnt &gt;= @buf_cnt</td>
<td>buf_cnt cannot underflow; assertion</td>
</tr>
<tr>
<td>pkt1_valid l-&gt; pkt0_nxt_ptr != pkt1_nxt_ptr</td>
<td>Packet link list cannot have a self loop; assertion</td>
</tr>
</tbody>
</table>

**Table 1: Properties Generated by Assertion Synthesis**

The first property reveals an important functional coverage hole. The RTL allows new values for rel_cnt to be written at any time. However, assertion synthesis reports that new values are written only when rel_cnt is 0 in all of the regression tests. Note this coverage hole cannot be found using traditional code coverage methods.

The second property implies that buf_cnt cannot underflow (@ is the next time operator where @buf_cnt denotes buf_cnt’s value at next cycle).

There is an important programming requirement that the next pointer of a linked packet cannot point to itself. As showed in the 3rd property, assertion synthesis concisely captures the programming requirement using internal pipeline signals. In contrast, blackbox checkers using only interface signals will be much more complex involving deeply nested temporal operators to parse and compare two sequential packet pointers.
Based on the analysis above, the designer classifies the properties into assertions and coverage, at which point the assertion synthesis tool can then output them in proper System-Verilog Assertion (SVA) or Property Specification Language (PSL) format. An example is showed as follows.

```verilog
class pc_2nf6td;
@ (posedge clk) disable iff (rst !== 1'b0)
   dec I> #1 ($past(buf_cnt) >= buf_cnt);
endproperty : pc_2nf6td
assert_pc_2nf6td : assert property(pc_2nf6td)
else begin
   $display("TIME[%0d] NEXTOP ERROR: assertion pc_2nf6td in <tb.dut> failed",$time);
   $assertoff(0, assert_pc_2nf6td);
end
```

Engineers can now use the SVA/PSL in their existing verification flow by binding the RTL of the IP, and they develop additional tests to patch the coverage hole on rel_cnt. When the tests are added, the RTL is modified, or the IP is integrated in a system, the assertions will continue to ensure the integrity of the design.

About NextOp

NextOp Software, Inc. is focused on delivering assertion-based verification solutions that allow design and verification teams to uncover bugs, expose functional coverage holes, and increase verification observability. NextOp’s BugScope Assertion Synthesis is the first product to automatically generate whitebox assertions and functional coverage properties in SVA, PSL and Verilog formats. Assertion Synthesis’s properties are used to drive progressive, targeted verification via robust, executable design specifications for existing simulation, formal and emulation flows.


Biographies

**Dr. Yunshan Zhu**  
President and CEO, NextOp Software.  
Prior to NextOp, Dr. Zhu was a member of the Advanced Technology Group at Synopsys. Dr. Zhu also worked as a visiting scientist and a post-doc at Carnegie Mellon University where he co-invented the bounded model checking algorithm. Dr. Zhu did his undergraduate study at University of Science and Technology of China and received his Ph.D. in Computer Science from University of North Carolina at Chapel Hill.

**Dr. Yuan Lu**  
Chief Technology Officer, NextOp Software.  
Prior to NextOp, Dr. Lu was a Principle Scientist in Enterprise Switching Group at Broadcom. Dr. Lu received his BS from Shanghai Jiao Tong University and his Ph.D. in Electrical and Computer Engineering from Carnegie Mellon University.