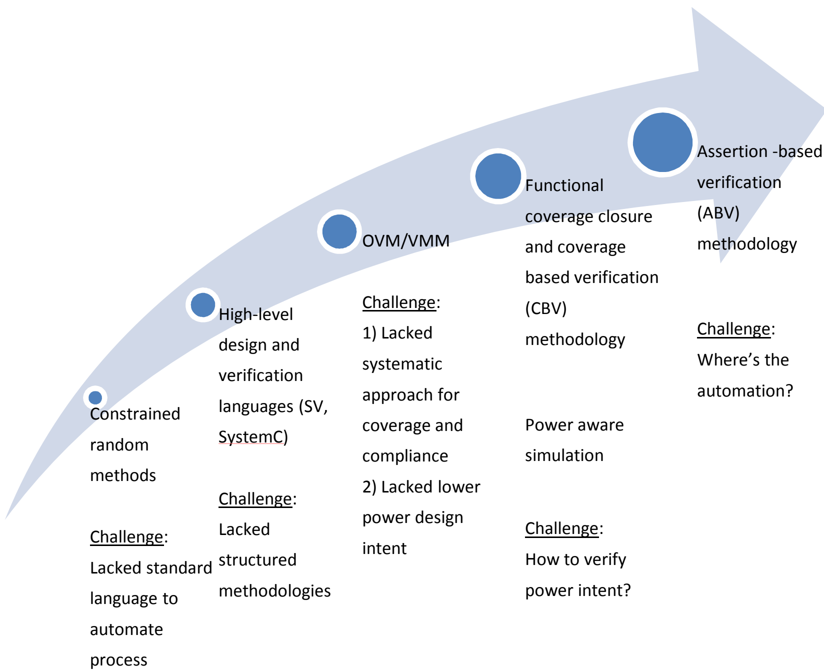
Microarchitecture Property Synthesis Automates Assertion and Coverage Based Verification

By Chris Browy

The evolution of functional verification has been exceptional over the last 10 years including the introduction of SystemVerilog, reusable testbench methodologies such as VMM and OVM, and raw simulator and formal tool capacity, performance, and debug capabilities.



However the challenges we now face involve the automation and methodology of using assertions and functional coverage which are not mainstream today. For example if you ask engineers what they think functional coverage means you will get different answers.

There is no one doubt that assertions and functional coverage can be extremely useful. Assertions check the design conforms to a set of rules of operation. Functional coverage measures the range of operation of a design for a given input domain. Together their use can result in attaining much higher quality of verification and isolate problems much faster. Adding assertions and coverage to a verification plan creates a clear set of objective requirements with measurable goals for achieving verification closure. Assertions and functional coverage are both based on properties which describe either the erroneous or intended behavior, respectively. In fact, SystemVerilog properties can be used in 3 different ways by simulators and formal tools:

- assume property describes the rules regarding the input domain
- assert property describes the rules of design operation

- cover property describes the range of design operation

However what it takes to achieve these goals is non-trivial. The big problem is in knowing what the right set of assertions and coverage to develop that addresses the verification plan fully.

The experts will tell you that there should be between 1 to 4 assertions for every 10 lines of RTL code. They refer to this as the assertion density. The introduction of OVL was an attempt to provide a structured library of assertions that could be instantiated in the RTL, however this turns out to be as cumbersome as writing them from scratch and is not generally considered to be a successful industry initiative. There is obviously a big gap between what the mainstream does today and what the experts advocate.

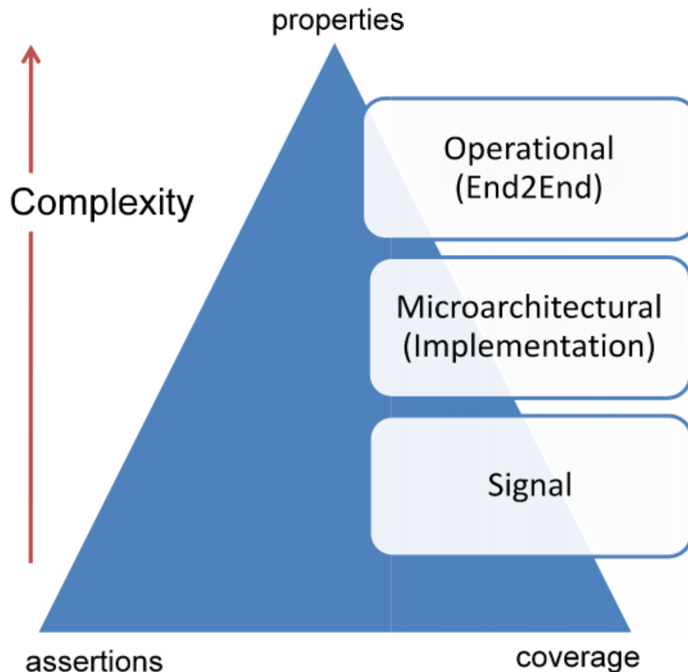
Assertions can take many forms.

- Operational
- Microarchitectural
- Signal-level

Operational assertions support modeling input-output properties of the end to end behavior. They are often protocol specific and can involve 10s to 100s of clock cycles. Key parameters going into an assertion often are derived from testbench monitors which extract key state information about the protocol stages.

For example the ARM AMBA AXI protocol checker distributed by ARM has roughly 125 properties and over 1500 lines of behavior code to support the property declarations.

These complexities yield the claims of why assertions are not more widely used, specifically that manually writing SystemVerilg assertions with SVA is daunting especially for designers.



Microarchitectural assertions and coverage address the design implementation which is comprised of a set of hardware functions. Microarchitectural assertions and coverage address the interactions between the functions to isolate problems. These properties may be temporal and involves up to 10s of clock cycles.

For example, a design may have 2 FSMs that interact via handshaking signals. When this condition is found, handshaking assertions and coverage can be added to ensure the handshake completes in the required time and the FSMs don't deadlock or livelock.

Another example may involve a FIFO and data processing block. Here it is obvious to include assertions to detect FIFO underflow or overflow. In addition however the FIFO and data processing block interactions should also be checked to ensure there is no queue leak condition where the FIFO pops an element yet the data processing function fails to properly sink this element due to a load enable problem.

Signal-level assertions are the easiest to write and typically involve timing invariants and are combinatorial.

Assertion synthesis delivers the advancements in automation and methodology for using assertions and coverage effectively. Specifically Insight deals with the synthesis of microarchitectural assertions and coverage and takes on some of the basic problems discussed earlier. Designers typically don't want to manually write assertions for their RTL code. Verification engineers tend to focus only on black box verification and operational assertions and coverage because they may not fully understand the microarchitectural implementation. This leaves a gap in the assertions and coverage plan and makes for uneven and ad hoc use of assertions and coverage.

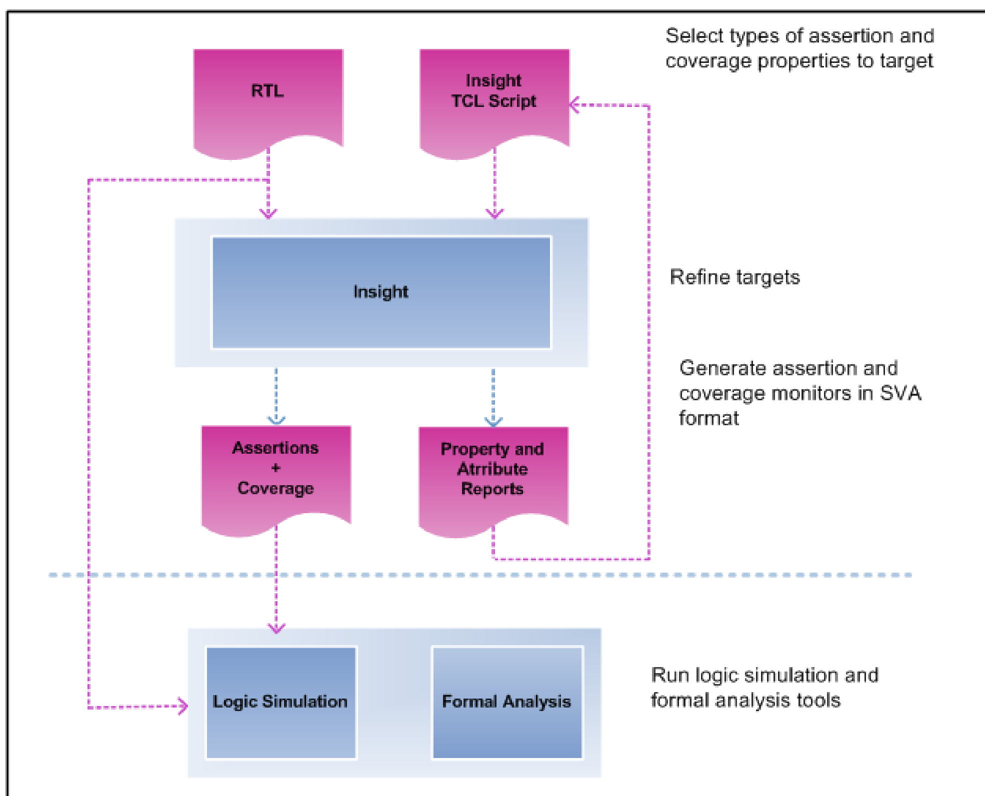
Assertion and coverage synthesis at this level involves automatically understanding the microarchitecture from the RTL and generating a comprehensive set of assertions and coverage.

Insight requires only very low effort from the designer or verification engineer to deploy. It can be run on a block or a full chip and should be run on each configuration supported in logic simulation or formal analysis.

The user can control Insight processing for:

- what microarchitectural features to recognize
- hierarchy scope of design instances to include or exclude
- whether to output assertions or coverage only, or both

Insight supports Tcl commands to navigate the sets of microarchitecture objects to apply assertion and coverage synthesis. Integration with the simulation or formal analysis environment is done through binding the SystemVerilog assertions and coverage to module instances or by generating separate module that can be included as another top-level module



Insight recognizes a range of microarchitectural features that involve the control and interface logic behavior of the design:

- FSM
- Queue (FIFO, Stack)

- Memory
- Arbiter
- Counter
- Key control functions
- Bus
- Clock domain crossing
- Power controller (when UPF is provided)

FSM assertions detect deadlock and livelock when involving multiple interacting FSMs. FSM coverage tracks state and state transition coverage. Optionally Insight can limit its state transition coverage properties to only the most interesting, multi-way FSM state transition conditions. During the analysis Insight will also provide a warning if any next state transitions in the RTL are infeasible since this may suggest a bug. A high-level query function will show the FSM attributes such as the FSM states and input variables involved in each of the state transition conditions.

The following example shows a FSM that has 5 states. The state transition variables for each state transition are listed. A long form of this report will show the full, multi-level equations for each variable which is generated from the symbolic analysis. The query command used to generate the short form report is:

```
insight_query_object cpu8080.state -info 2
```

```
============== FSM Analysis Report ==============
Variable: cpu8080.state
Type: State register (FSM)
   S1, Reset, Signal wait, -> S1 (reset)
      -> S2 (reset)
   S2-> S1 (reset)
      -> S3 (reset)
   S3, Signal wait, -> S1 (reset)
      -> S4 (reset, waitr)
      -> S3 (reset, waitr)
   S4, Signal wait, -> S4 (reset, opcode)
      -> S1 (reset, opcode, sign, parity,
           carry, zero)
      -> S5 (reset, opcode)
   S5, Signal wait, -> S1 (reset, intr, ei)
      -> S5 (reset, intr, ei)
=================================================
```

Further analysis capabilities involve analyzing the cross FSM signal dependencies and cyclical sequential loops which can contribute to livelock and deadlock conditions. For example here FSM, fsm1, is found to be dependent on fsm2 outputs, which is in turn dependent on fsm3 outputs. Insight also finds that fsm1 inputs logic cone can be traced all the way up to top-level module inputs, fsms_testbench.a and

fsms_testbench.b, which are driven from the testbench. FSM dependency analysis can analyze dependencies spanning the design hierarchy so that all FSM interactions can be easily understood. A general FSM dependency analysis is performed for all FSMs in the design using the following command.

```
insight_fsm_dependency_analysis
    [insight_get_objects -type fsm]
```

```
======== FSM Dependency Analysis Report ========

Variable fsms_testbench.dut.fsm1 is affected by
the following variables at time 675, cycle 6:
Module inputs:
  fsms_testbench.a (seq_depth= 1)
  fsms_testbench.b (seq_depth= 1)
Other registers:
  fsms_testbench.dut.fsm2 (seq_depth= 1)
  fsms_testbench.dut.fsm3 (seq_depth= 2)
  fsms_testbench.dut.temp (seq_depth= 3)
====== End of FSM Dependency Analysis Report ====
```

Queue assertions detect overflow and underflow. A queue leak assertion is also supported and involves the interface between of the queue and the load enable of the sink logic. Here is the Tcl script for queue leak. First Insight builds a list of all FIFOs in the design and then the register leak check is applied to all FIFO outputs.

```
set fifos [insight_get_objects -type fifo]
set fifo_list [split $fifos " "]
foreach fifo $fifo_list {
   insight_asyn_property_gen $fifo -reg_leak }
```

Queue coverage properties support monitoring the range of queue operation such as # of time the queue was empty/full, and the # of passthroughs.

Arbiter assertions check

- request/grant timeout
- fairness
- exclusive grant access

Insight can recognize a number of arbiter forms however the user may also provide a Tcl command to identify an arbiter or its arbitration algorithm. Arbiter coverage request/grant latencies.

Power control assertions and coverage are generated from the RTL and corresponding UPF files. Insight reads the UPF file and generates a number of assertions.

- No X on isolation registers after its reset has been sequenced
- Exclusion of separate retention save / restore signals
- Proper sequence order of power signals
- Power state coverage combinations from power state table

Other microarchitecture features covered by Insight include:

- Memory and register checks detect X conditions on read/write and address range checks.
- Bus assertions detect multiple drive conditions of the same value or X.
- Counter coverage tracks counter wrap-around situations.
- Key RTL conditions are analyzed and assertions and coverage generate warning on the operation of priority and full case statements, X conditions on conditional expressions, and coverage monitors are generated for high fanin/fanout control signals
- Cross clock domain crossing assertions detect data stability problems of the source data registers

Case Study

The following examples summarize Insight results on several real examples. Design A is a TLP ingress port accumulator block of a PCI Express switch design. Design B is a level 2 cache controller.

	Design A	**Design B**
FSM	48 recognized 104 state cover properties 364 state transition cover properties 103 deadlock assert properties	112 recognized 188 state cover properties 4953 state transition cover properties 187 deadlock assert properties
Counter	85 recognized 79 wrap cover properties	179 recognized 133 wrap cover properties
Memory	62 memories 310 assert properties	137 memories 573 assert properties
FIFO	21 recognized 105 assert properties 147 cover properties	11 recognized 55 assert properties 77 cover properties
Arbiter	0 recognized	2 recognized 6 assert properties 32 cover properties

	Design A	Design B
Lines of RTL	39,000	38,000
# of flops	17,899	50,000
Size (gates)	750 K	3.0 M
Runtime	1 min	22 min

The coverage results obtained by running a teststuite regression with the coverage properties included indicated that many design features were not being fully exercised. This feedback helped the verification teams go back and enhance the testsuite for higher coverage.

Conclusion

Assertion and coverage synthesis provides the critical advances in methodology and automation of effective assertion- and coverage-based verification. Microarchitecture-level assertion and coverage synthesis augments any input-output operational assertions that designers and verification engineers may develop manually on their own.

Microarchitectural assertions and coverage monitor key hardware features as well as their interactions to detect violations and limited range of operation covered by a testsuite or formal tools. Isolating problems in this manner pinpoints design issues closest to their source.

Insight takes very little effort level to deploy since it works directly on the block or chip RTL without requiring a testbench. Its microarchitecture-level assertions and coverage synthesis enhance the coverage baseline of a design and provide critical objective measures needed to achieve verification closure and increase the overall quality of functional verification.