

A Simulation Expert's Guide to Formally Verifying Software Status and Interrupts

Neil Johnson
Ciena Corporation
Ottawa, ON
njohnson@ciena.com

Abstract—Verification is more than just simulation. This paper is a case study meant to encourage simulation experts to consider formal property checking for features that are hard to reach in simulation. Specifically, it documents a strategy for proving software status and interrupts, both of which are habitually difficult to verify as part of a UVM testbench. Included are motivations for formal property checking, aspects of planning and documentation, generalized implementation guidelines and lessons learned.

I. INTRODUCTION

Simulation is the workhorse verification technology in semiconductor development as measured by tool revenue [1] and rates of technology adoption [2].

In the period between 2012 and 2020, data from the Wilson Research Group Functional Verification Study shows adoption rates for simulation related techniques for IC/ASIC teams as being approximately 2x greater on average than formal property checking. For the same period, adoption rates for simulation related techniques are approximately 3x higher on average than automatic formal techniques.

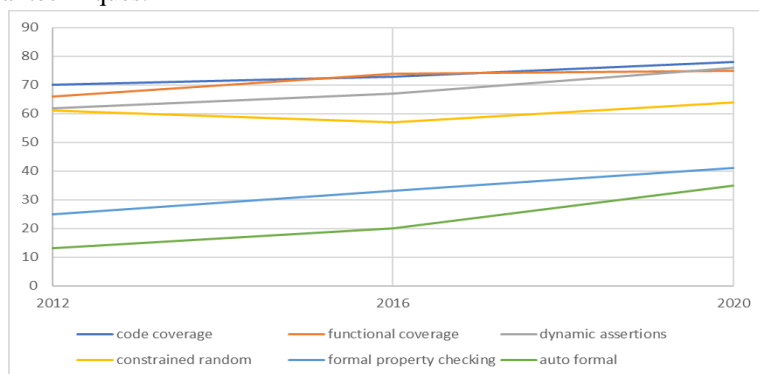


Figure 1 - Simulation and Formal Technology Adoption Trends

There are many logical reasons why simulation dominates verification, for example relative tool cost and abundance of industry expertise. But this paper contends that many of the decisions verification teams make are habitual, including the decision to use simulation over alternative technologies such as formal. Through a preference for the status quo, verification teams traditionally relying on simulation translates to simulation being the most likely technology for future projects [3].

This paper highlights the potential of formal property checking for simulation experts. As such, the intent is to innovate by bridging the gap between formal and simulation technologies—as opposed to innovating within formal verification proper—to encourage complementary tool usage and cross-technology problem solving within verification teams.

A. Verification Thought Leadership

For years, verification thought leadership has clearly focused on simulation with an emphasis on UVM-based simulation strategies relative to formal. This focus is particularly evident when analyzing keyword search data from DVCon proceedings between 2010 and 2022 [4].

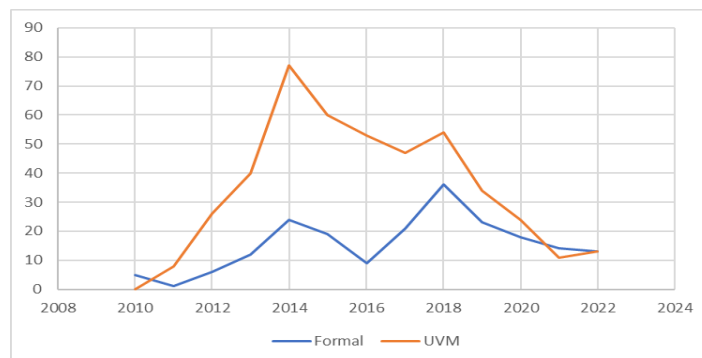


Figure 2 - DVCon Keyword Search Data 2010-2022: Number of Keyword Hits

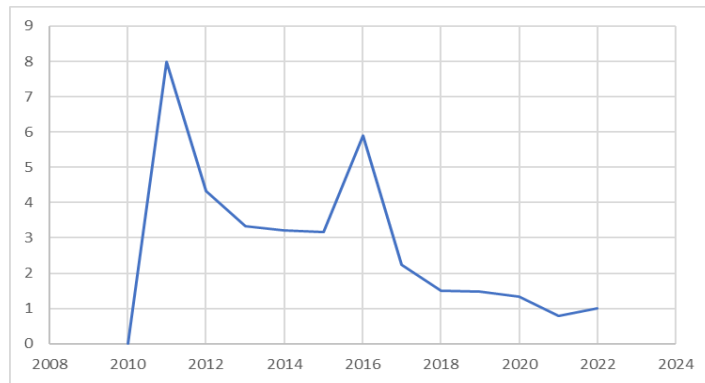


Figure 3 - DVCon Keyword Search Data 2010-2022: Ratio of 'UVM'/'Formal' Keyword Hits

Over the entire 12 year period, 'UVM' is more than twice as common as 'formal'. For the period of 2011-2016—the five years following release of UVM version 1.0—'UVM' was roughly four times more common than 'formal'.

B. Limitations of Simulation Architecture

The primary considerations guiding testbench architecture typically revolve around core functionality. For example, architectural decisions for: telecom applications focus on packet traffic; DSP applications focus on processing algorithms; and SoC level integration focus on bus interconnect and sub-system compatibility.

Rightly, verification planning and architecture are tailored to core functionality because it is fundamental to the application. Secondary features may be part of the initial planning but rarely with the priority comparable to that of core functionality.

Likewise, testbench implementation and sanity testing tend to begin with core functionality. Aside from common register accessibility tests¹, implementation of software status checking is often delayed until core functionality testing is well underway. Beyond simple prioritization of core over secondary features, an additional factor leading to delay in software status checking is that software status generally depends on core functionality. Without sane core functionality, it can be difficult or impractical to simulate software status.

Finally, implementation of status checking tends to be overlaid on core functionality testbench infrastructure. As such, it is common to see core architectural decisions unintentionally made to the detriment of secondary features. Detriment in this case means the architecture is sub-optimal for secondary features because it imposes: productivity restrictions that force sub-optimal approaches or bloated infrastructure; and/or technical restrictions that limit necessary visibility or control. Both types of restrictions can be attributed in part to the practice of TLM abstraction.

C. Downsides to TLM Abstraction

Whereas telecom, DSP and integration applications are well suited to transaction-level modeling found in UVM testbenches, status and interrupt checking is not. First, software state is typically a broad set of pin-level observations that are difficult to abstract. As such, packaging device state as a TLM transaction is a practice that provides limited architectural advantage. Further, because device state is often tightly coupled to RTL implementation, a low-level checking strategy is preferable. Raising abstraction through TLM can damage the necessary coupling and complicate the checking strategy.

II. SOFTWARE STATUS CHECKING WITH FORMAL PROPERTY CHECKING

This case study documents an example of separating verification of core and secondary functionality into independent efforts. Secondary functionality in this case study is software status and interrupts which are verified using formal property checking. Core functionality of the DUT—verified with simulation—is beyond the scope of this paper.

Verification through two independent efforts delivers two advantages. First, it enables effective architectural decision making and implementation options. Simulation architecture for core functionality is built as a UVM-based testbench while the software status infrastructure is built as pin-level Systemverilog properties. Second it allows for earlier verification of software status than happens otherwise because no dependencies exist between the two efforts.

A. Sub-system Under Test: Timing Block

The *timing block* sub-system under test (UUT) is part of the Ciena Wavelogic series of products. It is a highly configurable sub-system with the primary function of synchronizing optical data streams through the generation of variable data rates using PLLs.

¹ Common register accessibility tests would include those defined as sequences in the `uvm_reg` package such as `uvm_reg_bit_bash_seq` or `uvm_reg_access_seq`. These tests are used to verify the connectivity and accessibility of the registers proper, not the downstream logic the register is connected to.

The IO is a combination of software control register input, software state output, input/output data rates and other general IO. A critical feature of the UUT is that the processor interface and corresponding software registers are external to the sub-system, as shown in Figure 4. As such, all interactions with the UUT happen directly through dedicated control and status struct connections. A significant benefit of this partitioning is that it avoids interactions through a true processor interface; properties that model read/write protocols were not necessary².

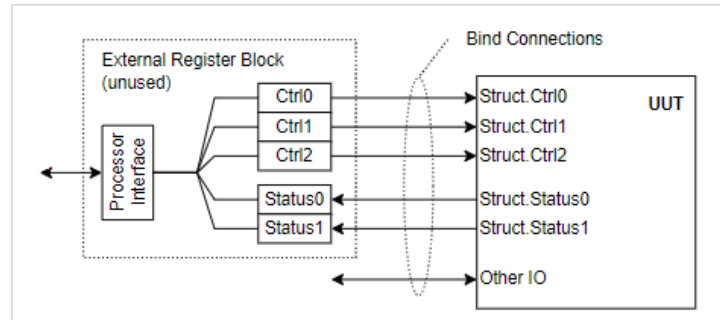


Figure 4 - Software State Connectivity and Unused External Registers

B. Checker Strategy

The preferred approach in this case study is to interact with properties via top-level IO while avoiding use of internal nodes. This approach is chosen because (a) it forces properties to be created with system-level perspective; and (b) allows for indirect checking of tangential logic (i.e. logic between primary inputs and the RTL being targeted by a property).

Using the system-level perspective is intentional in that the engineer creating properties interacts with the DUT similar to how an embedded software developer does, through the setting of control inputs and sampling of status outputs. As such, it offers an opportunity to validate the sanity of interactions from a user's perspective in a way that localized testing using internal nodes cannot.

C. Method

1) Capturing Dependencies

The first step in planning involves capturing the relationship between software state and relevant IO. This is done through a visual mapping. Dependencies are identified using design documentation and the RTL implementation. As an example, Figure 5 highlights dependencies for Status0.StateA that include Ctr0.FieldA and IO.PinA.

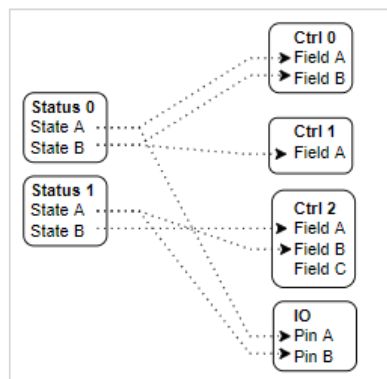


Figure 5 - Software Status Mapping

2) Estimating Feasibility

Being part of a formal technology adoption exercise, a critical aspect is estimating feasibility of formal property checking relative to simulation. As such, status outputs are analyzed independently for feasibility. Initial assessments hinge on expected difficulty of modeling state expectations relative to control offered by control register and related inputs.

- A status output is assumed feasible if direct, easy to model relationships exist.
- Further exploration is required for status outputs with indirect, complex IO relationships.

3) Checker Strategy

The checker strategy is a more detailed analysis of each status bit that describes the modeling relationship. The intention and description of the required modeling is captured for each status output along with expected limitations. Where necessary,

² It is assumed that modeling a processor interface protocol would have caused a significant, possibly prohibitive, increase in formal proof times

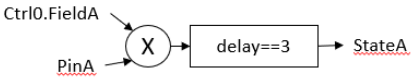
the strategy includes block diagrams or schematics for relationship details. TABLE I. is a generalized version of the checker strategy.

TABLE I. CHECKER STRATEGY TEMPLATE

Strategy	Status Outputs	Limitations	Status	Other
Brief description of the associated checker with modeling considerations, interactions, behaviours, etc relevant to the SW status output.	Pin/port name	Potential limitations or risks that may restrict the strategy or intended formal proof outcome	Not started/ In progress/ Done	Relevant schematic, if necessary.

An example strategy for Status0.StateA that includes the relationship with its dependencies is shown in TABLE II.

TABLE II. CHECKER STRATEGY TEMPLATE

Strategy	Status Output	Limitations	Status	Other
The ctrl input is a scaling factor and the pinA is a primary input. Vary pinA and the scaling factor to prove the status output is asserted for scaled values gt 100	Status0.StateA	Max scaling factor is 73	In progress	

4) Implementation

Checkers are implemented as formal properties using SVA. They are declared in a dedicated checker module then connected to the UUT using a *bind* construct. Binding is done at the sub-system top-level. Internal UUT signals are not used.

As shown in Figure 6, property structure is based on a template for the four phases of automated software unit tests: setup, exercise, verify and teardown [5]. Properties explicitly contain a setup, exercise and verify phase. Given no shared state exists between property execution threads, the teardown state is irrelevant and therefore unused. Two forms of this template are used, both use the implication operator.

The first form, shown in Figure 6, follows the order of evaluation used in software unit testing. The *setup* and *exercise* are located in the antecedent while the *verify* is located in the consequent. *Setup* is generally used to specify static control register and/or general IO settings that do not change during evaluation. *Execute* describes some observed dynamic event that satisfies the vacuity requirement of the property to enable the *verify* phase. *Verify* confirms the expected response of the DUT given the *setup* and *exercise* conditions.

```

property form1;
  @(posedge clk)
  /* setup */ and
  /* execute */ |->
  /* verify */
endproperty

```

Figure 6 - Phased SVA Template

A second form of the template shown in Figure 7 reverses the order of the execute and verify phases. In this form, the expected outcome in the *verify* phase is detected first as part of the antecedent. Once detected, the *execute* is evaluated in the consequent to confirm the expected outcome is valid. To account for the phase re-ordering, the *execute* phase uses the *\$past* operator to confirm the triggering event was responsible for the expected outcome.

```

property form2;
  @(posedge clk)
  /* setup */ and
  /* verify */ |->
  $past(/* execute */)
endproperty

```

Figure 7 - Phased SVA Template with Verify and Execute Phases Reversed

Examples for Status0.StateA properties using both form1 and form2 styles are show in Figure 8 and Figure 9.

```

property stateA form1;
  @(posedge clk)
  (ctrl10.Field0 < 73) and
  (1, exp_stateA = (ctrl10.Field0 * IO.pinA) > 100) |->
  1 ##3 status0.stateA == exp_stateA
endproperty

```

Figure 8 – Example Phased SVA Template

```

property stateA_form2;
@(posedge clk)
(ctrl0.Field0 < 73) and
status0.stateA |->
$past(status0.stateA,3) * $past(IO.pinA,3) > 100
endproperty

```

Figure 9 – Example Phased SVA Template with Verify and Execute Phases Reversed

Both templates are recommended though advantages for each are difficult to generalize. One factor that guides application is the complexity of the *execute* phase. If the *execute* phase contained minimal variability (i.e. a single sequence/expression or multiple sequences/expressions with sequential ordering) and a simple relationship with the expected result, the first form is used. Conversely, if the *execute* phase contains multiple sequences/expressions with parallel execution, the second form is used. It is common to start implementing a property using the first form then change to the second as unforeseen complexity presents itself. In some situations, both forms are used to prove different variations of behavior on a single status output (i.e. variations to confirm \$rose, \$unchanged and \$fell conditions separately).

To maintain reasonable proof run-times³, static variables are used whenever possible. Static variables are constrained using assume properties.

5) Documentation and Review

All checkers are documented and reviewed with relevant members of the development team. Documentation is again captured in a tabular format for ease-of-review. Plain English is used to describe checker details. The intended audience for documentation is the design engineer who is responsible for the RTL implementation.

TABLE III. CHECKER DOCUMENTATION TEMPLATE

Category	Property	Parameterization	Configuration	When Condition...	Check Condition...
Property Grouping	Property Name	Relevant checker module parameters (if applicable)	Setup sequence description	Antecedent sequence description	Consequent sequence description

D. Results and Observations

1) Bugs Found

During the formal property checking effort described in this case study, a total of six bugs were found in the process of proving 15 different status outputs.

One RTL bug found during this case study can be attributed to using a system-level perspective. It involved a configuration that was judged to be non-viable and therefore the corresponding RTL was removed.

Two bugs were found as a result of logic between the primary IO and the logic being targeted by formal properties. It is assumed that without interacting via primary IO these bugs would have gone undetected. The two bugs related to a continuous assignment that used an incorrect shift operator and a logic pipeline that was improperly synchronized.

The remaining three bugs were found in logic specifically targeted by the formal property.

2) Infrastructure and Coding Style

Compared to UVM-based testbenches built for simulation, the infrastructure required for this formal property effort was relatively low. Relevant code for property definition, instantiation and binding to the DUT tended to be localized within a single file.

Properties required minimal abstraction and were generally described in terms of interactions on pin-level IO ports. Functions and sequences were used to generalize common interactions though the purpose of these generalizations was more to avoid duplicate code than raising abstraction.

One goal of the effort was to minimize use of RTL modeling (aka: helper logic) outside of properties. In some cases, however, helper logic was used to capture behaviour difficult to describe within a property. These situations usually involved event detection in complex temporal logic or timers.

In situations where logic timing was deterministic, cycle accurate properties were defined. In cases where timing was not deterministic or where timing was not critical to the feature being targeted, properties were relaxed to apply over an acceptable timeframe.

3) Technology Ramp-up

Most of the ramp-up time associated with this effort is attributed to inexperience with Systemverilog assertions.

³ Reasonable proof times were 10 minutes or less to enable short debug cycles. Up to an hour was acceptable for complex proofs. Anything over an hour generally triggered a refactoring cycle (i.e. trial-n-error experimentation with coding styles to find faster proofs).

E. Tooling

The method described in II is deployed using Confluence for planning and documentation with Siemens PropCheck used for formal proofs.

III. LESSONS LEARNED

A. Use of RTL Models

RTL modeling was critical for improving run-time performance of formal proofs. In this case study, two specific modules were identified as being suitable for RTL modeling. Common characteristics between the two involved (a) deep logic pipelines of roughly 10-15 clock cycles; and (b) arithmetic calculations with operands of greater than 32 bits.

Eventually, it was found that an RTL model that limited the logic pipeline to a single clock cycle was sufficient for significant run-time improvement. In this case study, proofs with RTL models typically converged in less than 60 minutes whereas original RTL implementations either (a) took up to eight hours to converge and were deemed impractical; or (b) failed to converge within eight hours.

Prior to models being used, the original RTL implementations were verified in isolation, also done with formal property checking.

B. Verilog Configurations vs. Black-boxing/Binding

For connecting RTL models, two alternatives were used: black-box directives with a bind statement and Verilog configurations.

The first attempt at connecting RTL models was using a combination of black-box directives and a Verilog bind statement. As illustrated in Figure 10, the black-box directive was used to remove the original RTL implementation; the bind statement was used to connect the corresponding RTL model. This approach was adequate, however, it required a tcl directive to specify the black-box and introduced a potential maintenance issue where the bind statement required knowledge of connections in the parent module⁴; if connections in the parent module changed, the bind would also need to change.

```
tcl.do:
  formal compile -d top
  netlist blackbox instance top.parent.child_0
  formal verify

parent.sv:
module parent();
  RTL_original child_0(.clk(p_clk),
                      .rst_n(p_rst_n),
                      .a(p_a));
endmodule

bind.sv:
// bind the model to the parent signals
bind top.parent RTL_model model_0(.clk(p_clk),
                                   .rst_n(p_rst_n),
                                   .a(p_a))
```

Figure 10 - Connecting an RTL Model Using a Black-box and Bind

A second, preferable method involved the use of Verilog configurations. As seen in Figure 11, the Verilog configuration lists each instance of the RTL model to be used with the path to the original implementation. This approach was preferable because it could be handled within the language—though the addition of a configuration file—without requiring other vendor specific tcl directives. More importantly it avoided any knowledge of connectivity within the parent module and potential connection maintenance problems. It did require scripting updates to specify an alternate top-level design module, but this was easily absorbed as a new option to Ciena proprietary wrapper scripts.

⁴ Initially it was assumed that binding an RTL model to some instance in a design, then black-boxing that instance would provide the required functionality. However, this had the unintended effect of black-boxing both the original RTL and the RTL model.

```

tcl.do:
// change the top-level module to Verilog config
formal compile -d model_top
formal verify

config.sv:
config model_top;
design work_top;
default liblist work;
instance top.parent.child_0 use RTL_model;
endconfig

```

Figure 11 - Connecting an RTL Model Using a Verilog Configuration

C. Assuming Checker Execution State

A significant difference between simulated tests and formal property checking—one that is especially pertinent for engineers entering formal property checking with a simulation background—is the concept of checker execution state (i.e. the state in which a particular check is executed).

Consider a FIFO overflow check. FIFO overflow is executed only in states where the FIFO is full. Using simulation, that state is usually reached procedurally; a test will first fill the FIFO, then push an additional element to verify assertion of the overflow signal.

```

task proc_test();
repeat (FIFO_FULL_LEVEL) begin
@ (posedge clk);
fifo.write = 1;
end

@ (posedge clk);
fifo.write = 1;

@ (posedge clk);
assert (fifo_overflow);
endtask

```

Figure 12 – A Procedural Test For FIFO Overflow

Using formal property checking, a procedural approach to reaching the checker execution state is also possible. For example, the property declared in

```

property proc_overflow;
@ (posedge clk)
fifo.write [*FIFO_FULL_LEVEL] ##1 fifo.write |->
1 ##1 fifo_overflow;
endproperty

```

Figure 13 emulates the procedure from Figure 12.

```

property proc_overflow;
@ (posedge clk)
fifo.write [*FIFO_FULL_LEVEL] ##1 fifo.write |->
1 ##1 fifo_overflow;
endproperty

```

Figure 13 – A Procedural Property For FIFO Overflow

A second approach to writing the overflow property is a declarative one that ignores the procedure required to reach the execution state and instead simply describe that state as a prerequisite to the check.

```

property decl_overflow;
@ (posedge clk)
fifo.fill_level == FIFO_FULL_LEVEL && fifo.write |->
1 ##1 fifo_overflow;
endproperty

```

Figure 14 – A Declarative Property to Assume Checker Execution State

The advantage of the declarative approach is 2-fold. First, it removes the burden of describing *how* the checker execution state is reached. Second, it gives the formal tool the flexibility to explore all possible paths to the checker execution state as it searches for counter-examples. These advantages combine to increase both productivity during development and robustness of formal proofs. Notably, this declarative technique cannot be used in simulation unless additional modeling interventions are employed.

D. Property Debug

Formal tooling consistently found counter-examples that could be attributed to incomplete or incorrect property definitions. This was especially relevant during initial ramp-up but persists as the engineering team gains experience. For

engineers new to Systemverilog assertions and formal property checking, it is recommended to expect property debug to form a sizable portion of any ramp-up.

IV. FURTHER CONSIDERATIONS

A. Cut Points

While the strategy of strict interaction with the DUT via primary IO proved useful in finding bugs in the software interface and tangential logic, it also resulted in situations where formal property checking became infeasible. For example, one such situation required the tooling to explore state of a filter output deep inside the design. While the proof that relied on the filter output was relatively simple, the complex logic between the primary inputs and filter inputs combined with the complexity of the filter itself made a converging proof impractical. Bypassing complex logic in situations such as these would allow for proving a larger set of software status outputs.

An option for bypassing complex logic involves the addition of cut points. A cut point is a virtual DUT input that the formal tool may use as a control point just as any other input may be used as a control point [6]. Any logic driving a net that has been specified as a cut point is effectively ignored.

While not ideal, it is recommended cut points be considered in cases where interaction through primary IO becomes infeasible and corresponding proofs deliver measurable value.

V. CONCLUSIONS

The intention of this paper is to highlight the potential of formal property checking for simulation experts. While a large body of knowledge exists to support the use of simulation, formal property checking historically has received less support from functional verification thought leaders while complementary approaches that use both simulation and formal are relatively unexplored.

One such area well suited to formal property checking within a broader simulation-based approach is software status and interrupts. It is recommended that simulation experts explore software status and interrupts as a possible first step toward complementary tool usage within functional verification.

VI. REFERENCES

1. Electronic System Design Alliance, "Electronic Design Market Data Report". The ESD Alliance. First quarter. 2022.
2. H. Foster, "The 2020 Wilson Research Group Verification Study". *Verification Horizons*.
<https://blogs.sw.siemens.com/verificationhorizons/2020/10/27/prologue-the-2020-wilson-research-group-functional-verification-study/>.
3. W. Samuelson and R. Zeckhauser, "Status Quo Bias in Decision Making," *Journal of Risk and Uncertainty*, Vol 1, No. 1, pp. 7-59, March 1988.
4. <https://dvcon-proceedings.org/>
5. G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional, 2007.
6. *Questa PropCheck User Guide*. Siemens EDA. 2022.