First Look

OneSpin Advances Formal Assertion/RTL Debug Automation

By Bill Murray

07/15/09

OneSpin has launched RootCauseAnalyzer, a new enhancement of its OneSpin 360MV formal assertion-based verification (ABV) solution, which automates the analysis and debug of formal SystemVerilog assertions (SVA) and RTL code. The company states that it eliminates much of the manual effort required to find the root cause of failing SVA. Moreover, it enables the efficient debug of the complex, hierarchically-coded, specification-level assertions that are critical to boosting formal ABV quality and productivity.

Michael Siegel, OneSpin's product marketing director, said "The tool's automation eliminates a barrier to the adoption of specification-level assertions, which leverage advanced SVA constructs such as nested function calls, and predefined sequences and properties. The effort-intensive manual debug of such assertions hinders many users from exploiting them to their maximum potential. Users often resort to a larger collection of simpler assertions. This not only fails to use the full power of assertions and assertion reuse, it also complicates review of the assertions against the specification. Moreover, it imposes a significantly higher assertion maintenance effort if the RTL changes."

How big a productivity boost can the tool deliver? Siegel said "It can reduce debug time by 2X to 10X, depending upon assertion and design complexity – the higher the complexity, the greater the gain. Assertion and design debug accounts for about 30 to 40 percent of the total formal verification time and effort, so our debug productivity improvements can significantly reduce verification schedules and design cost."

How long does it take users to get up and running? Siegel said "The components of the RootCauseAnaylzer are easy to learn. It takes less than an hour to become productive."

Debugging Assertions

What do you do when an assertion fails? How do you determine the cause of the failure? Is the assertion code incorrect? Is there an illegal input scenario that the constraints failed to exclude? Or is there a bug in the RTL code? Or is it a combination thereof? Siegel said "Without highly automated debug support, it requires significant manual effort to answer these questions. This manual effort can put a serious dent in the team's productivity, even in the case of relatively simple assertions that verify local RTL behavior."

According to Siegel, the debug productivity hit can be considerably greater when the team uses specification-level assertions, which verify RTL behavior at the level of, for example, complete design transactions. Systematic verification of design transactions increases verification productivity, so they are a powerful enhancement to ABV flows. But the resulting assertions can consist of hundreds of lines of hierarchically-structured code spanning hundreds or thousands of clock cycles, posing a significant debug problem when they fail. So, how do you debug them efficiently?

Siegel said "We have solved this problem by automating the most effort-intensive steps in SVA analysis and RTL debug."

According to Siegel, the company's new RootCauseAnalyzer comprises four individual components that constitute a four-step debug flow:

- The WaveformAnalyzer (WA) identifies the clock cycle where the assertion is first violated, and generates diagnostic information to speed understanding of counterexamples.
- The StructuralAssertionAnalyzer (SAA) – an SVA code debugger – automatically identifies the failing parts of the assertion and shows the user where to start assertion debug.
- The TemporalFaninAnalyzer (TFA) automatically traces the signals involved in the assertion failure to related RTL design signals, thus enabling efficient exploration of signal relationships across all relevant clock cycles and module boundaries.
- The ActiveCodeAnalyzer (ACA) automatically identifies those regions in the RTL source code that are involved in the assertion failure.

Siegel said "The RootCauseAnalyzer's four step process eliminates a great deal of tedious, manual search and analysis. The StructuralAssertionAnalyzer is particularly key to reducing debug effort and speeding the whole debug process. In hierarchically described assertions, it analyzes the definitions of all referenced functions, sequences and properties, and pinpoints the failing clause across the entire hierarchical definition of the assertion – a daunting task for manual debug."

Does the tool interoperate with third-party debug automation tools? Siegel said "It feeds downstream RTL debug tools such as SpringSoft's Debussy and Verdi. However, it does not need these tools to do its job."

Debug Example

In our interview, Siegel illustrated the flow using an example of an AHB-to-Wishbone bridge – an OpenCores design with 9,000 lines of code and 6,500 flip-flops (see figure 1). The bridge allows an AHB master to perform single or burst read/write accesses to a Wishbone slave.

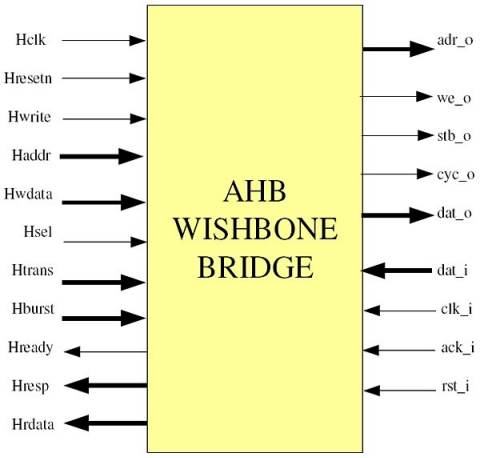

Figure 1: The AHB-2-Wishbone bridge – an OpenCores design

The following hierarchical assertion uses named properties, sequences and functions to capture a single-write transaction of the bridge (see figure 2(a)).

Figure 2: A hierarchical assertion for a single-write bridge transaction (Source: OneSpin)

The hierarchical assertion refers to the property single_write_wait (see figure 2(b)), which is itself hierarchically described using predefined objects. Coding, understanding and reuse are structured and simplified by the use of the predefined sequences SingleWriteStart and keep_stable (see figure 2 (c)), and the nested functions wb_write_ctrl and wb_write_addr. For example, the parameterized sequence keep_stable states that a set of control signals is kept stable until an expected signal arrives within a number of clock cycles defined by a parameter MAX_WAIT.

The assertion fails on the AHB-2-Wishbone design and the formal verification tool generates a counterexample showing a design behavior that contradicts the assertion.

Debug Step 1 – Waveform Analysis

The WaveformAnalyzer displays diagnostic information to speed analysis of counterexamples that show the assertion failure (see figure 3).

According to Siegel, the WaveformAnalyzer indentifies the clock cycle where the assertion is first violated, and colors signals to speed understanding of the counterexamples. It shows in red which signals are involved in the assertion failure, and shows by means of the vertical yellow line that the assertion fails in the third clock cycle and that the signals adr_o, haddr and ack_i are somehow involved in the failure.

Siegel said "When working with a simple assertion, such as one that states that two signals are expected to have the same values, the WaveformAnalyzer often gives sufficient information about the signals involved in the assertion failure to understand which part of the assertion needs further inspection. However, in more complex assertions, it is much harder to locate which part of the assertion actually causes the failure. That's where the StructuralAssertionAnalyzer comes in."

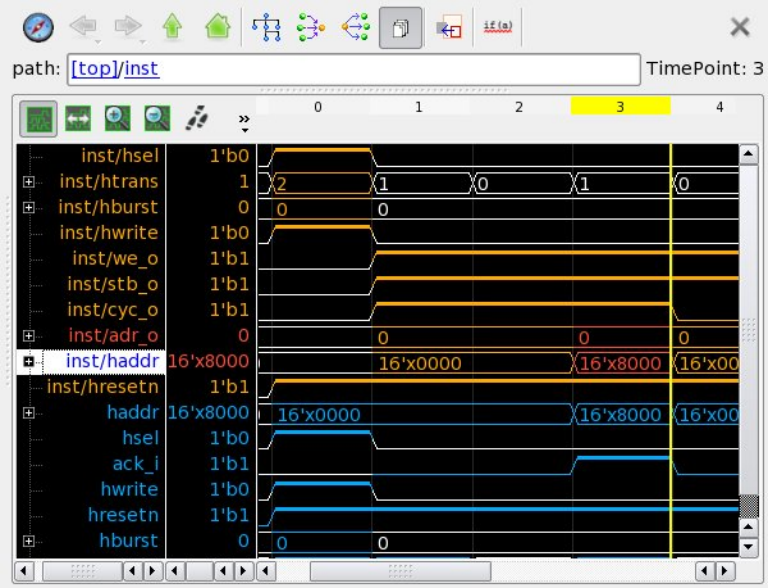

Figure 3: WaveformAnalyzer displays diagnostic information (Source: OneSpin)

Debug Step 2 – Assertion Failure Identification

The StructuralAssertionAnalyzer is a SVA code debugger that automatically identifies the failing parts of the assertion, and marks them red, showing the user where to start assertion debug. As stated earlier, it analyzes all referenced functions, sequences and properties, and pinpoints the failing clause across the entire hierarchical definition of the assertion (see figure 4).

Here, it automatically identifies and highlights the expression "adr_o=haddr" in function wb_write_add_fn as the reason why the assertions fails. All objects not violated by the counterexample are left unmarked.

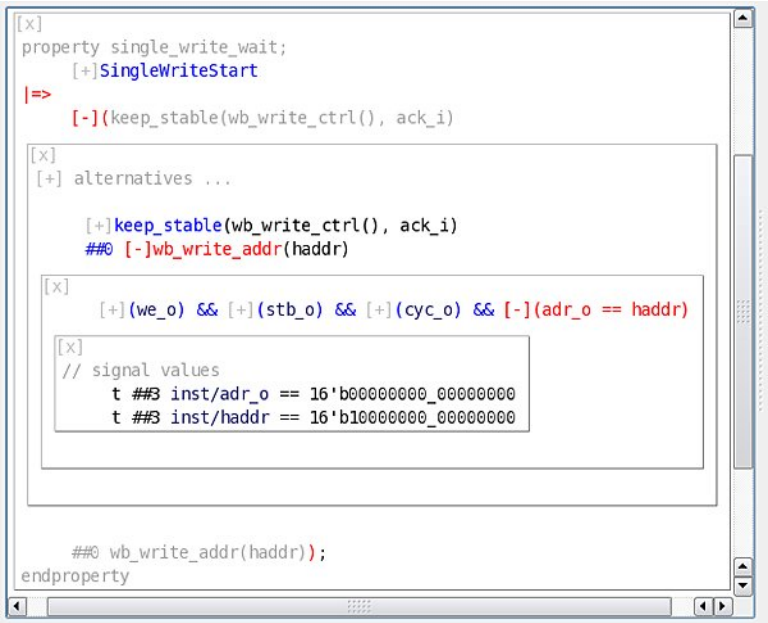

Figure 4: The StructuralAssertionAnalyzer automatically identifies failing parts of the assertion (Source: OneSpin)

Where assertions use repetition operators, such as "[0:MAX_WAIT]" in the sequence keep_stable, the question arises: which parts of the assertions match until which clock cycles in the counterexample?

The StructuralAssertionAnalyzer performs an automatic analysis which instantiates ranges in SVA repetition operators with a concrete value that is best suited for debugging. In this example, it has instantiated the wait period with the value 2, showing that in the consequent of the assertion the control signals remained stable for two cycles and then the assertion violation occurred in the third cycle in the clause marked red. Analysis across the assertion hierarchy and the instantiation of repetition operators enables automatic annotation of all objects referenced in the assertion with values – including all objects in referenced functions, sequences and properties – for all clock cycles relevant to the failure, based on the counterexample. Value annotations for 'adr_o' and haddr' are shown in the screenshot for the third clock cycle.

Siegel said "Without the StructuralAssertionAnalyzer's automation, every one of these analysis steps would have to be done manually. In particular, the use of functions, sequences and properties, repetition operators and local variables add significantly to debug complexity and effort."

Debug Step 3 – Signal Trace Analysis

The TemporalFaninAnalyzer automatically traces the signals involved in the assertion failure to related design signals, thus enabling efficient exploration of signal relationships across all relevant clock cycles and – more importantly – across module boundaries. It automates otherwise tedious manual analysis of signal dependencies across clock cycles and the search for design signals involved in the failure across the modules of the design.

As can be seen in figure 5, the signal adr_o at clock cycle 3 depends on a register addr_temp at clock cycle 3, which itself depends on the input signal haddr at clock cycle 2. The last column 'Time' indicates the signal dependencies across cycles, the column 'Module' shows in which module the respective signals are defined.

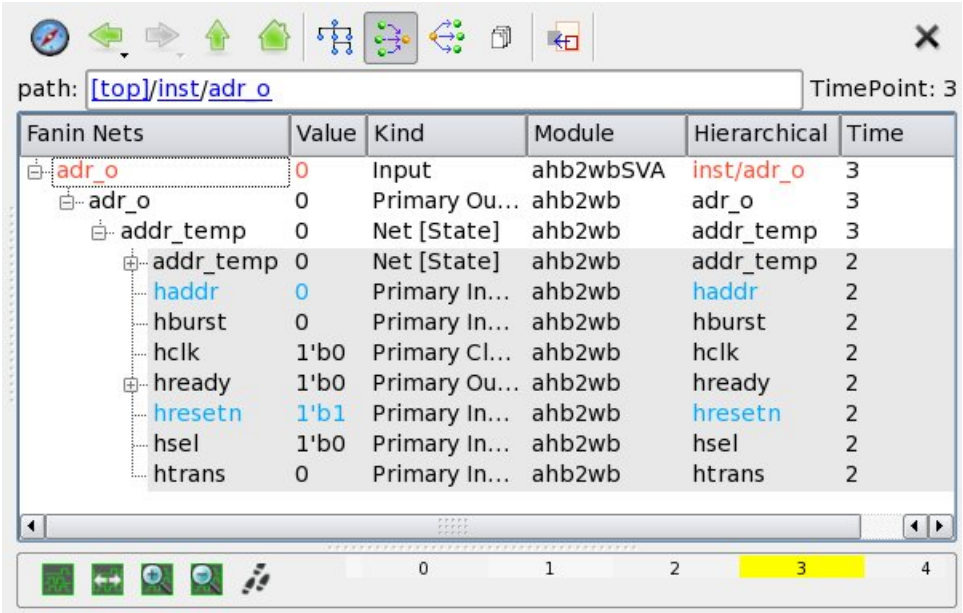

Figure 5: The TemporalFaninAnalyzer automatically traces signals involved in the assertion failure (Source: OneSpin)

Debug Step 4 – RTL Failure Identification

Finally, the ActiveCodeAnalyzer automatically identifies those regions in the RTL source code that are involved in the assertion failure and marks them red (see figure 6). Conditional code that does not contribute to the failure remains unmarked.

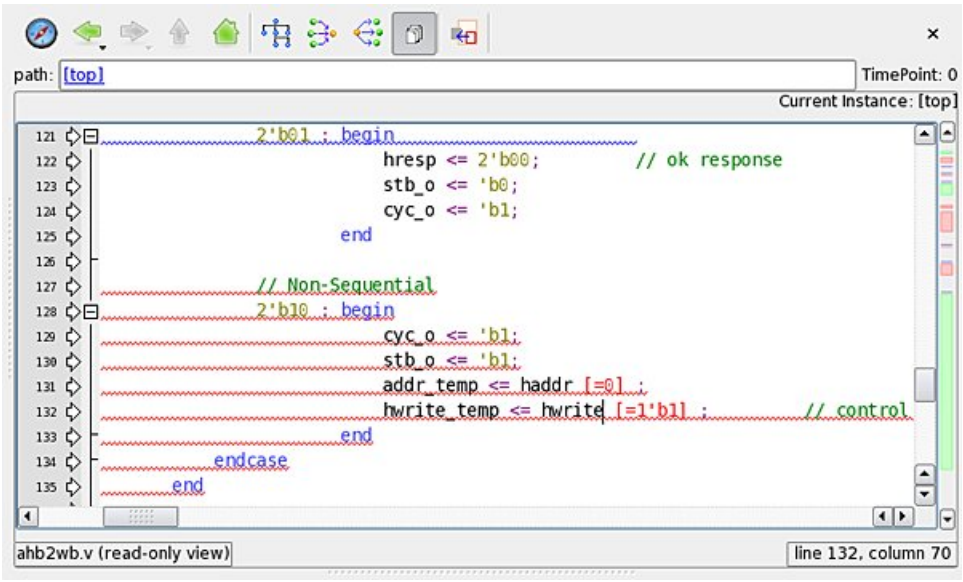

Figure 6: The ActiveCodeAnalyzer automatically identifies relevant RTL regions (Source: OneSpin)

The analysis and display of active code regions is generated for all relevant clock cycles of the counterexample, thus automating RTL code exploration across clock cycles, focusing and speeding RTL source code debug. In this case, further analysis shows that the root cause of the assertion failure is that the input address from the AHB master is output by the bridge one cycle later to the Wishbone master side than stated in the assertion. A cross-check with the specification shows that the design behavior is correct; thus the assertion must be corrected.

Availability

The RootCauseAnalyzer is available now in OneSpin's 360 MV family of formal verification solutions. OneSpin will demonstrate the new tool at the Design Automation Conference 2009 in San Francisco, California. Booth 3465 in the North Hall.

Further Reading

- OneSpin Offers Step-by-Step Formal Verification Suite. SCDsource.
- Formal – Rocket Science or Mainstream Technology? A Deeper Look. SCDsource.
- Mixing Formal and Dynamic Verification, Parts 1 and 2. User survey and interviews. SCDsource.