EDA Tech Forum
Technical Resource for the EE Community

EDA Tech Forum
Technical Resource for the EE Community

September 2006

Native SystemC Assertion mechanism with transaction and temporal assertion support

Native SystemC Assertion has been developed to provide better verification support in SystemC. The assertion engine supports temporal assertion primitives similar to SVA, as well as primitives to construct assertions for transaction level models at higher levels of abstraction.

SystemC [1] is rapidly becoming the language of choice for ESL-centric design methodologies. It is set to become the framework for higher-level flows above today's RTL, and has three key components: modeling, synthesis and verification. High-level modeling particularly demonstrates the language's versatility and advantages. Strong progress is also being made in higher-level synthesis. However, our view is that verification is currently the weak link.

This paper introduces temporal assertions in native SystemC that support both the cycle level (CLA) and transaction level (TLA). Based on our Native SystemC Assertion (NSCa) technology, they empower verification with transaction-level modeling (TLM) and enable a flow from TLM to RTL where TLAs can be re-used for cycle-level designs. The advantages in this approach becomes most obvious in a TLM approach that allows users to reuse testbenches originally developed for architectural modeling at the system level, in subsequent and refined design models [2] [3].

Atsushi Kasuya is CTO and chief architect of JEDA Technologies. He helped write the first hardware verification language while at Sun Microsystems and holds an MSCE from Santa Clara University and a BSEE from Seikej University.

Tesh Tesfaye is director of product development at JEDA Technologies. He has 15 years of experience in hardware design and verification at companies such as Sun Microsystems and Juniper Networks. He holds a BS ECE from the University of California, Santa Barbara.

Eugene Zhang is president and CEO of JEDA Technologies. He helped write the first hardware verification language while at Sun Microsystems and holds an MSCE from Syracuse University, New York and a MSEE and BSEE from Tsinghua University, China.

The challenge

Whether performing ESL design or developing testbenches using SystemC, identifying and fixing bugs quickly is a daunting task. Bugs missed at the ESL phase are costly to fix if they creep into silicon and ESL verification poses particular challenges.

There is no predictable way to quantify and measure how well the intent of the system designer has been implemented in RTL.

System designers usually develop ad-hoc tests to validate their designs. Unless verification engineers directly re-use these tests in test-benches challenges emerge:

1. Two sets of tests need to be developed, one for system the other for RTL validation, duplicating the effort.
2. Validation codes developed in different domains by different people are very likely to be inconsistent.
3. Managing two sets of validation codes and keeping them consistent is time consuming, particularly if a design or specification changes.

Verification engineers using SystemC face the challenge of measuring the effectiveness of their test vectors unless they resort to functional and code-coverage tools in the RTL domain using Verilog.

Notably, one method that has successfully been used in RTL is assertion-based verification (ABV). This defines checks that are continuously evaluated in the form of an expression during simulation.

The Native SystemC Assertion solution

NSCa was developed as a part of an effort to broaden and increase the verification support in SystemC [4] and to address ESL verification challenges. NSCa is a C++ assertions library that dynamically links to a SystemC simulation engine while providing the mechanism to write both temporal transaction and cycle level assertion checks directly into a SystemC/C++ code base.

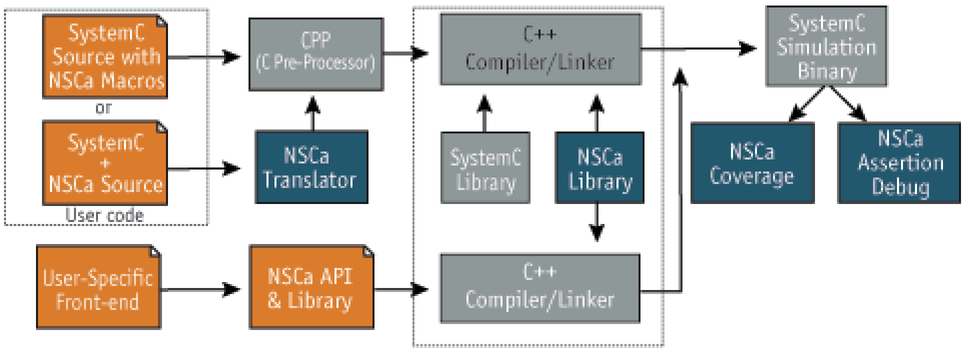


Figure 1. Assertion flow for NSCa

The natural SystemC design flow is assumed to be top down, starting with the system performance model for architectural design, then progressing through more refined levels.

Using high-level synthesis tools, such a model can produce a functionally equivalent cycle-accurate RTL model that represents the final hardware implementation. In such a flow, decisions made at the architectural design phase should be carried down to the final system model to confirm that the implementation has held to its goals. A well thought out TLM scheme allows the reuse of testbenches as well as functional/performance assertions from the architecture model down to the final implementation level. To provide such a mechanism, we need to construct such high-level assertions that represent the design goals with the TLM.

Existing standards for assertion, Property Specification Language (PSL) [5] and SystemVerilog Assertions (SVA) [6], mainly target the temporal transition at the cycle-accurate modeling level. The primitives in both languages are designed for checking the cycle-by-cycle behavior of signals. Our findings suggest that this is not well suited to a TLM design flow. As a TLM flow employs a top- down methodology, the specification of high-level assertions needs to be done in a similar manner. If high-level assertions can be constructed over the TLM testbench, TLA can be adopted in the lower abstraction model without modification. Thus, the advantage of the TLM approach for testbench construction can be extended further.

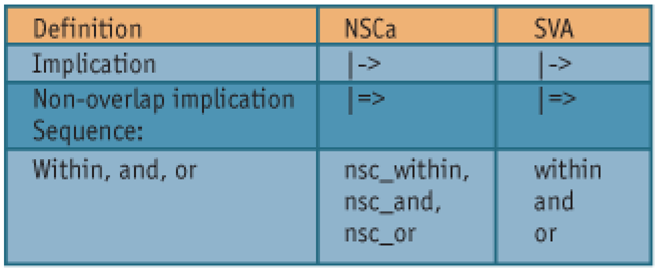
NSCa flow

Figure 1 illustrates the assertion flow for NSCa. It implements a scalable high-performance assertion engine that can be used in a number of ways. From the NSCa assertion flow diagram, one can gather the following: User's can express temporal assertions in one of two forms:

1. System architects and verification engineers who prefer the pure C++ syntax can optionally use NSCa macros to describe assertion checks.
2. Users can write checks using an extended C++ syntax that resembles SVA or PSL syntax.

NSCa includes an assertion coverage and debug facility that can be used by engineers to further debug assertion checks.

Users writing their own C++ models that do not use the SystemC class library can further use the NSCa engine in a standalone mode along with their own custom front-end to reap all the advantages of NSCa.



Definition	NSCa	SVA
Implication	|->	|->
Non-overlap implication Sequence:	|=>	|=>
Within, and, or	nsc_within, nsc_and, nsc_or	within and or

Figure 2. Similarities between NSCa and SVA

Transaction-level assertions using NSCa

Unlike temporal assertions that perform evaluations on every clock, TLAs use SystemC events for synchronization or process/threads. Performing TLA natively in SystemC offers the architect a number of advantages.

High-level performance analysis checks can easily be written using NSCa's TLA primitives. These perform the needed checks during simulation in real-time as opposed to requiring post-processing.

This in turn means that performance anomalies are detected right at the failure point, identifying the root cause quickly.

Meanwhile on the fly checks do not require the storage of large amounts of data for later analysis, eliminating the need for large disks for simulation runs.

TLAs also have the path for further refinement and or reuse when RTL assertions are developed.

```
// req , then gnt within 5 cycle,
//    then req = 0
if (
   ! nsc_sequence(
      req.read() == 1 @[1,5] gnt.read() == 1
         @1 req.read() == 0
      )
   )
{
  cout << "Error: request/grant sequence broken!"
       << endl ;
}
```

Figure 3. NSCa temporal assertion checks using extended C++ syntax

Cycle-level assertions(CLA) using NSCa

The temporal assertion primitives in NSCa are derived from SVA. At this level, assertion evaluations take place on every clock. Using NSCa, one can then build simple-to-complex property/sequence checks to implement interface and/or protocol-level checks in fewer lines than would be needed if using plain SystemC. Figure 2 illustrates the similarities between NSCa and SVA.

As stated earlier, there are two ways of expressing NSCa temporal assertion checks, using the extended C++ syntax (Figure 3) or in the NSCa MACRO (Figure 4).

So how effective are my assertion checks?

The NSCa package includes a comprehensive coverage analysis capability so a user can measure the effectiveness of property/sequence checks. NSCa provides two types of assertion coverage mechanism and these are:

Assertion path coverage

```
// req , then gnt within 5 cycle,
//    then req = 0
if (
   ! NSC_SEQUENCE(
        NSC_BOOL( req.read() == 1 ) &&
        NSC_SEQ( 1, 5, NSC_BOOL(gnt.read() == 1) ) &&
        NSC_SEQ( 1, 1, NSC_BOOL(req.read() == 0) )
      )
   )
{
  cout << "Error: request/grant sequence broken!"  << endl ;
}
```

Figure 4. NSCa temporal assertion checks using NSCa MACRO

Typical NSCa expressions entail the evaluation of multiple sub-expressions. Suppose you have run extensive simulation runs and would like to find out which precise conditions you have not covered. You would use assertion path coverage (APC) to obtain that data.

```
a nsc_and b nsc_or (@[1:2] c @[3:4] d)
                  | --- a nsc_and b
                  | --- @[1:2] c @[3:4] d
                  | --- @1 c @3 d
                                  | --- @1 c @4 d
                                  | --- @2 c @3 d
                                  | --- @2 c @4 d
```

Figure 5. Assertion path coverage

The NSCa expression in Figure 5 shows that for expressions "a,b,c,d" there are multiple paths that will possibly yield true. Using a high-performance and scalable database, APC will keep track of detailed information on the evaluation of every expression path. This will help you efficiently identify which conditions have been missed.

Assertion activation coverage

Whereas APC keeps detailed statistics on sub-expression evaluation granularity, assertion activation coverage (AAC) works at a property level. This facility in NSCa collects different levels of statistics that can be used by users to further understand failures of assertions. The information provided by AAC shows:

- How many times a property has been activated;
- How many times the property check passed or failed;
- When in the simulation the property check failed

NSCa assertion failure debugging

NSCa includes an assertion debug facility that one can use to quickly debug failed assertion evaluations. This GUI-based environment has the ability to zoom in on the most basic logical computation node of an expression and clearly highlight the cause of the failure. A sample window from a debug session is shown in Figure 6.

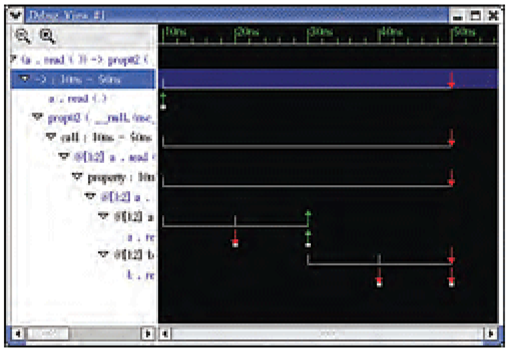


Figure 6. Sample window from debug session

Conclusion and future work

We have discussed how Native SystemC Assertions provides both temporal primitives and transaction-level assertion capabilities. Several values can be derived from using NSCa.

- It ensures that that the architect's system-level design specifications are maintained through the implementation phase of a design.
- It allows protocol/interface checks to be easily written within the framework of SystemC/C++.
- It aids the rapid identification of system design problems before implementation.

JEDA has published a proposal for NSCa standards to the SystemC Verification Working Group and is actively working with other members of the group to realize this goal.

The company's founders have a deep experience in the field, having invented other hardware verification languages, Vera and Jeda-X, in the past. The team is now actively working to bring other SystemC verification automation solutions to market.

References

1. SystemC homepage, www.systemc.org, 2006
2. Thorsten Grotker, Stan Liao, Grant Martin, and Stuart Swan. System Design with SystemC, Kluwer Academic Publishers, 2002
3. Anssi Haverinen, Maxime Leclercq, Norman Weyrich, and Drew Wingard. SystemC based SoC Communication Modeling for the OCP Protocol, www.ocpip.org/socket/systemc, OCP-IP, 2003
4. NSCa product data, www.jedatechnologies.net, 2006
5. Property Specification Language Reference Manual, Version 1.1, Accellera, 2004
6. System Verilog 3.1a Language Reference Manual, Accellera

JEDA Technologies
4962 El Camino Real Suite 105
Los Altos, CA
94022
USA

T: +1 650 964-5332
F: +1 650 964-5334
www.jedatechnologies.net