Native SystemC Assertion(NSCa)

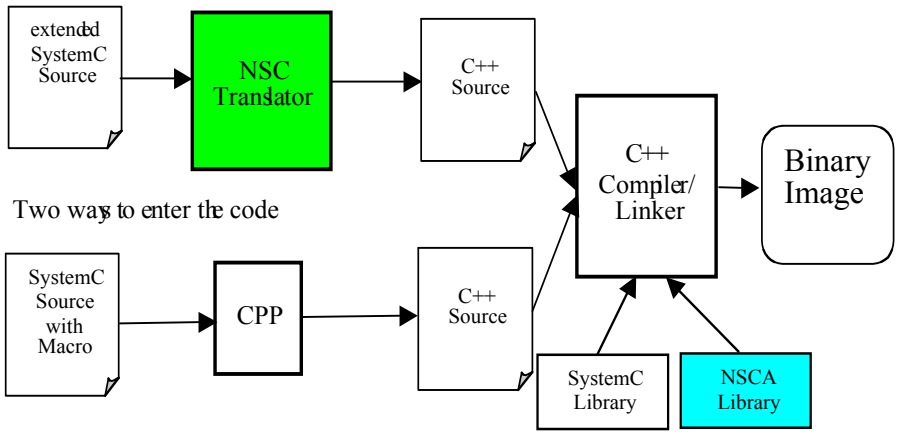
Introduction

Native SystemC Assertions(NSCa), is a scalable System to RTL verification solution that provides native assertion mechanism in SystemC. In its first release, NSCa provides assertions constructs very much similar to those in SystemVerilog Assertions(SVA).

NSCa provides the capability of writing assertions primarily in two ways. One, by providing the NSC syntax, a natural extension of the SystemC/C++ syntax to support the notations for assertion. The code written using the NSC syntax is translated to SystemC/C++ code by the NSC Translator. The second option is it uses assertion NSC macros that provide the same functionality like C macro forms. With this option, the user can stay within the standard C++ syntax and simply makes assertion macro calls.

Either using the NSC syntax or macro, the users can embed the assertion code into any SystemC program without any restriction. The code can be a property monitor that checks the design behavior throughout the entire simulation, or a single program code to check the sequence at the given time slot of the program execution. The user can use the temporal expression written directly as the condition expression of an if statement or a while statement to construct the self-checking testbench code. This flexibility extends the programmability of SystemC over the temporal event evaluation.

Figure 1 NSCa compiling/linking tool flow



NSCa functionality

The following table summarizes the assertions constructs supported by NSCa and presents the extended syntax notation or the macro definition.

Table 1: NSCa constructs summary table

Name	NSCa extended construct	NSCa macro equivalent
property declaration	nsc_property	NSC_PROPERTY(...)
assertion invocation	nsc_assert	NSC_ASSERT(...)
property anding	nsc_pand	NSC_PAND(...)
property oring	nsc_por	NSC_POR(...)
property negation with not	nsc_not	NSC_NOT(...)
implication	|->	NSC_IMPLY(...)
non-overlap implication	|=>	NSC_NOIMPLY(...)
property instance call	<property name>	NSC_CALL(...)
sequence declaration	nsc_sequence	NSC_SEQUENCE(...)
m to n cycle delay	@[m:n]	NSC_SEQ(m,n,...)
m to n consecutive repetition	[*m:n]	NSC_CREP(m,n...)
m to n goto repetition	[->m:n]	NSC_GOTO(m,n...)
m to n non-consecutive repetition	[=m:n]	NSC_NREP(m,n...)
sequence match item where <s>: sequence <m>: match item	(<s>, <m>)	NSC_MATCH(...)
sequence “and” operation	nsc_and	NSC_AND(...)
sequence “or” operation	nsc_or	NSC_OR(...)
sequence intersect operation	nsc_intersect	NSC_INTERSECT(...)
sequence-within operation	nsc_within	NSC_WITHIN(...)
sequence throughout operation	nsc_throughout	NSC_THROUGHOUT(...)
sequence-first match operation	nsc_first_match	NSC_FIRST_MATCH(...)
sequence instance call	<sequence name>	NSC_CALL(...)

Such assertion code can be used within a SystemC thread. The evaluation of such a temporal expression is executed with the default sensitivity of the thread (e.g. a clock assigned to the thread). Each property or sequence evaluation is a blocking operation, but the evaluation of multiple nodes within a property or a sequence is done concurrently. In other words, the evaluation is done in the non-deterministic finite automaton. For example, a sequence with range of delay, “@[1,5] <sequence>” will produce five active evaluation nodes from cycle one to cycle five. It may be possible that the following <sequence>

produces another level of multiple nodes per those five active nodes. The NSCa engine produces the multiple evaluation nodes at runtime and evaluates the required sequence accordingly.

With our NSC front end, the user can use the extended NSC syntax to represent the temporal expression in SystemC code. The following example shows a sequence is used within an if statement.

Listing 1.1 NSCa code snippet to check req-gnt relationship

```
// 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: requuest/grant sequence broken!\n";
}
```

The same expression can be written with assertion macros, as shown below.

Listing 1.2 NSC macro example to check req-gnt relationship

```
// 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: requuest/grant sequence broken!\n";
}
```

The following code example shows the manual implementation of the sequence example. In order to implement the temporal evaluation, it can not be written as a single line expression in the if condition, and it must use various state flags to represent the sequence.

Even though the example does not have a non-deterministic situation, the manual construction of such a sequence is very hard to understand at a glance. When it requires the non-deterministic evaluation (multiple active nodes within the state machine), it would be very hard to check if the code itself is correctly representing the expected behavior.

Listing 1.3 SystemC equivalent to check req-gnt checking

```
// req, then gnt within 5 cycle, then req = 0
#define EXIT_STATE 1000
int flag =0;
int state = 0;
int count = 0;
while( state < EXIT_STATE) {
   switch(state) {
      case 0:
        if ( req.read() == 0 ) state = 1;
        else state = EXIT_STATE;
        break;
      case 1:
        if ( gnt.read() == 1 ) state = 2;
        else if ( ++count > 4) state = EXIT_STATE;
        break;
      case 2:
        state = EXIT_STATE;
        if ( req.read() == 1 ) flag = 1;
        break;
    }
    if ( state < EXIT_STATE ) wait();
}

if ( !flag ) {
   count << "Error: request/grant sequence broken! \n";
}
```

NSCa example

An assertion monitor for DUT can be added as an individual thread in a SystemC module. The following example code shows a simple monitor is attached as a thread in a module.

Listing 1.4 NSCa assertions inside a SystemC code

```
SC_MODULE(dff) {
  sc_in <bool> R;
  sc_in <bool> D;
  sc_out <bool> Q;
  sc_in_clk CLK;

  SC_CTOR(dff) {
   SC_METHOD(toggle);
   sensitive_pos << CLK;
   sensitive << R;
   SC_THREAD(monitor); // monitor is added as a thread
   sensitive_pos << CLK;
  }
  void toggle() {
     if ( R.read() ) Q.write(0) else Q.write(D.read());
  };
  void monitor() {
     wait(); wait();
     nsc_property(
        nsc_always(
           (R.read() == 1) |=> (Q.read() == 0);
        )
     );
  };
};
```

The monitor code use ‘always’ property to evaluate the property for every cycle. The implication condition will detect the proper timing to check the following sequence to be matched.

Within a SystemC thread, an assertion can be invoked as the assert statement (starting with nsc_assert keyword). When the assert statement is used, it spawns a dynamic thread from the location (using sc_spawn() function), and the assertion evaluation is executed concurrently without blocking the caller thread execution. The following code shows the example of invoking a monitor as an assert statement.

Listing 1.5 NSCa properties written as a monitor

```
SC_MODULE(foo) {
   sc_in <bool>
   ...

   SC_CTOR(foo) {
      SC_THREAD(foo_thread);
      sensitive_pos << CLK;
      ...
   }

   nsc_property sequence_check() {
      int id;
      nsc_sequence(
         @[1,10] (ack.read() ==0) nsc_match(id=req_id.read()
         @[1,100] (reply.read() ==1 && rep_id.read() == id )
      )l
    };

   void foo_thread() {
     ..
     // involking a property to check the ack -> reply sequence with
     // a proper id match
     if ( strobe.read() == 1 )
        nsc_assert @(nsc_posedge CLK) sequence_check() ;
     ..
    };
};
```

Assertions reuse, NSCa to SVA and vice-versa

Despite working in either RTL, ESL or both levels at any given time, the need for assertions solution that works for both environments from the same code-base to reduce the design and debug cycle is very important. Given that SystemVerilog and SystemC will be the language of choice designing systems, going back and forth between the two languages in both design and verification environments will become the norm. In the case of assertions, the need for having a single code base that can work in both SystemVerilog and SystemC is non-existent today.

At Jeda Technologies we realize the importance of code reuse, primarily assertions reuse in the ESL and RTL design cycles. The way NSCa was defined was deliberate to make it functionally equivalent and very similar like SystemVerilog Assertion(SVA) for this very same reason, to maximize reuse. The following table best illustrates the similarities of NSCa and SVA:

Table 2: tabular representation of the NSCa vs. SVA assertion constructs

Name	NSCa construct	SVA construct
property declaration	nsc_property	property
assertion invocation	nsc_assert	assert
property anding	nsc_pand	and
property oring	nsc_por	or
property negation with not	nsc_not	not
implication	|->	->
non-overlap implication	|=>	|=>
property instance call	<property name>	<property name>
sequence declaration	nsc_sequence	sequence
m to n cycle delay	@[m:n]	##[m:n]
m to n consecutive repetition	[*m:n]	[*m:n]
m to n goto repetition	[->m:n]	[->m:n]
m to n non-consecutive repetition	[=m:n]	[=m:n]
sequence match item where <s>: sequence <m>: match item	(<s>, <m>)	(<s>, <m>)
sequence “and” operation	nsc_and	and
sequence “or” operation	nsc_or	or
sequence intersect operation	nsc_intersect	intersect
sequence-within operation	nsc_within	within
sequence throughout operation	nsc_throughout	throughout
sequence-first match operation	nsc_first_match	first_match
sequence instance call	<sequence name>	<sequence name>

The close similarity of NSCa with SVA eases the translation of one construct into the other and vise-versa an easier task than being otherwise. Nevertheless, this fact alone will not result in the realization of completely reusable assertions code for SystemC and SVA levels but also how the assertions are written, i.e. inline vs. as separate monitors even has greater impact.

Based on our experience as well as interactions with customers, we present the following guideline on how to write reusable assertions for both the ESL and RTL levels of your design cycle.

1. Inline vs. monitor assertions:

   Assertions will have to be written as monitors, as opposed to being embedded very

close to the design. This means, the DUT will have to be treated as a black box and the assertion checks will only have to be performed at the interfaces of the design.

2. Checks in separate files:

   The advantage of having the assertion checks in a separate file is that it makes the checks all self contained and eases the work of translators from NSCa to SVA and vice-versa. Some may opt of writing the assertions in separately, perhaps at the bottom end of a design which will also work. The only thing it has to conform to is to what is discussed in the next section, separate property and sequence checks for each assertion check.

3. Separate properties and sequence checks:

   Ideally you should have sequence and property checks that are small, performing one to few checks. Again, this also lends itself to easing the work of the translators where translating a single property/sequence is much easier from doing the same for inter-dependent large and complex assertion checks. If you happen to have very complex checks, it would be ideal to break them down into smaller common pieces as well.