2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Temporal assertions in SystemC

Mikhail Moiseev, Intel Corporation, Hillsboro, USA (mikhail.moiseev@intel.com)
Leonid Azarenkov, Intel Corporation, Hillsboro, USA (leonid.azarenkov@intel.com)
Ilya Klotchkov, Intel Corporation, Hillsboro, USA (ilya.v.klotchkov@intel.com)

***Abstract*—We propose temporal assertions in SystemC language which look similar to SystemVerilog assertions (SVA). These assertions can be declared in SystemC module scope as well as in clocked thread process function. Temporal assertion contains pre-condition expression, time parameter and post-condition, which is checked to be true if pre-condition was true at specified time in the past. Assertion expressions are checked every time the event occurs, this event is specified explicitly or taken from the current process sensitivity. The temporal assertions are automatically converted into SVA by our SystemC-to-Verilog Compiler tool during high level synthesis (HLS).**

Keywords— SystemC; SystemVerilog assertions; assertion-based verification; high level synthesis

I. INTRODUCTION

Assertion-based verification (ABV) is commonly used technique for digital design at IP level and system on chip (SoC) level. To introduce assertions in SystemC code C++ *assert* macro can be used. In addition to that, SystemC has *SC_REPORT_FATAL*, *SC_REPOR_ERROR*, *SC_REPOR_WARNING* and *sc_assert* macros. These C++ and SystemC assertions contain no timing information, so they are equivalent to SystemVerilog immediate assertions. The SystemC assertions can be used in simulation, but according to SystemC synthesizable subset standard [1] they are not taken for synthesis.

In this paper we propose temporal assertions in SystemC language. The temporal assertions intended to be used for advanced verification of design properties with specified delays. These assertions look similar to temporal SystemVerilog assertions (SVA). Each temporal assertion has pre-condition expression, time parameter, post-condition expression and event when assertion is evaluated. Every time the event occurs, the post-condition is checked to be true, if the pre-condition was true at some time in the past, as specified by the time parameter. If the pre-condition was true, but the post-condition is false, the assertion is violated, and an error is reported. Assertion event normally is a clock positive edge, negative edge or both edges, therefore we can consider a cycle when assertion is evaluated. Time parameter is represented with exact number or range of such cycles (events occurred).

There are two types of assertions proposed: SystemC module scope assertions and clocked thread function scope assertions. These assertion types are complementary and have similar features. Module scope assertions help to avoid cluttering of thread function code. Assertions in clocked thread functions allow access to function's local variables in addition to module's signals and ports. There is also a special kind of assertion which can be used inside of loops in clocked thread function. Such loop assertions intended to be applied for arrays of modules, signals or ports.

The temporal assertions are proposed to be automatically translated to SVA during high level synthesis (HLS) of the SystemC design. Module scope assertions should be translated into SVA in the corresponding SystemVerilog module. Clocked thread scope assertions should be translated into SVA in the corresponding *always_ff* block. To demonstrate translation of the assertions into SVA, we have implemented that in SystemC-to-Verilog compiler [2]. In this paper we consider SystemC synthesizable subset as specified in [1].

This paper consists of six sections. Section II discusses existing approaches for assertion-based verification in SystemC and our motivation to implement a new one. Section III gives the temporal assertions semantic and examples of the temporal assertions in SystemC code together with SVA generated. Section IV considers implementation details which are important for SystemC simulation and translation into SVA. In Section V performance evaluation of the temporal assertions is given. The last section concludes the paper.

II. BACKGROUND

Using assertion-based verification for SystemC designs are widely discussed in research papers. One of the first attempts to extend SystemC language with temporal assertions is presented in [3]. In this paper SVA are automatically translated into SystemC monitors which are integrated into design-under-test. That requires designer to mix SystemC and SystemVerilog languages and can potentially lead to naming issues. Considering the difficulties of mixing SystemC with another language, authors of [4] proposed a SystemC assertion library. The assertion library consists of number of checkers which have fixed functionality but parametrizable with checked signals, clock, reset and others.

Assertion-based verification can be applied at SystemC Transaction-level Modeling (TLM) which provides higher level of abstraction and commonly used for early prototyping and resource estimation. In [5] an implementation of a TLM assertion framework is discussed. To introduce assertions a custom specification language is proposed. The assertions are translated into SystemC and joined into design-under-test. As soon as some of existing EDA simulation tools support mixed-language simulation, it is possible to combine SVA assertions in SystemVerilog modules with SystemC design modules. That idea is used in [6] for dynamic verification of SystemC/TLM designs. In the paper problem of adaptation of clock based SVA concurrent assertions for clockless designs is solved. Our temporal assertions are intended for cycle accurate SystemC designs, extension for TLM is not considered up to now.

Besides dynamic assertion-based verification there are formal and semi-formal methods to check assertions. An approach to formal verification of SystemC/TLM designs based on partial order reduction and symbolic simulation is presented in [7]. In work [8] aspect-based technique is used to describe temporal properties for SystemC/TLM designs. Assertions specified in Property Specification Language (PSL) are used in [9] to generate monitoring logic. The formal verification of the assertions is done with bounded model checking. Applying semi-formal approach based on static analysis for SystemC cycle accurate design is presented in [10]. In this paper special language for functional assertions is proposed. The temporal assertions proposed in this paper are planned to be used for dynamic ABV, but there is no restriction to used them with formal methods.

Our motivation to develop the temporal assertions is to extend SystemC language with powerful and easy-to-use features for IP level and SoC level verification. Unlike of some approaches discussed above, we do not invent a new assertion language, but try to implement SystemC assertions similar to SVA as much as possible in modern C++. That simplifies usage of the assertions by design and verification engineers who have experience with SVA.

SVA have several forms of notation and include a lot of advanced features like combining assertion sequences, declaring assertion dedicated variables, providing assertion system functions and others. The proposed assertions cover only main functionality of SVA, which has been chosen based on our experience in digital design verification. So, this implementation can be considered as a start point, which can be extended with other desirable SVA features in the future.

III. TEMPORAL ASSERTIONS IN SYSTEMC CODE

A. Assertion macros

The proposed temporal assertions are represented with macros in SystemC code. The assertions can be added into SystemC code in module scope and in thread process function scope. These two ways are complementary and can be mixed. Assertions in module scope can access module fields. Such assertions require an assertion event which normally should be clock positive, negative or both edges. Using assertions in module level allows to separate them from design logic to avoid cluttering of thread function code. Assertions in thread function scope can additionally access local variables of the function that allows to reuse some pre-evaluated expressions from design logic. There is special kind of assertions which can be used inside of loops in clocked thread function. Such loop assertions intended to be applied for arrays of modules, signals, ports or others.

There are four assertion macros, two for module scope and two for thread function scope:

- *SCT_ASSERT (RHS, EVENT)* – module scope assertion,
- *SCT_ASSERT (LHS, TIME, RHS, EVENT)* – module scope assertion,
- *SCT_ASSERT (LHS, TIME, RHS)* – function scope assertion,
- *SCT_ASSERT_LOOP (LHS, TIME, RHS, ITER)* – loop scope in function assertion.

Temporal assertions parameters:

- *LHS* – antecedent assertion expression (pre-condition),
- *TIME* – temporal condition is specific number of cycles or cycle interval,
  - *SCT_TIME(N)* – time delay, *N* is number of cycles,
  - *SCT_TIME(N, M)* – time interval, *N* and *M* are number of cycles.
- *RHS* – consequent assertion expression, checked to be true if pre-condition was true (post-condition),
- *EVENT* – cycle event,
- *ITER* – loop iteration counter variable(s), comma separated in arbitrary order.

Assertion expressions *RHS* and *LHS* are arithmetical or logical expressions, which can be evaluated into *true* or *false*. To reduce assertion length, *SCT_TIME* can be omitted, so instead of *SCT_TIME(1)* short form *(1)* can be used. The same works for time interval, instead of *SCT_TIME(3,2)* short form *(3,2)* works.

B. Assertions in module scope

Temporal assertions in module scope have the following semantic:

- *SCT_ASSERT (RHS, EVENT)* – assertion with expression *RHS*, checked when *EVENT* occurs, equivalent to *SCT_ASSERT (true, SCT_TIME(0), RHS, EVENT)*;
- *SCT_ASSERT (LHS, TIME, RHS, EVENT)* – assertion with pre-condition *LHS*, post-condition *RHS*, time parameter *TIME*, evaluated and checked when *EVENT* occurs.

Assertion expression in module scope can operate with signals, ports, template parameters, constants and literals. Member data variables (not signals/ports) access in assertion leads to data race and therefore prohibited.

There is an example of temporal assertions in SystemC module:

```
static const unsigned T = 3;
static const unsigned N = 4;
sc_clk_in clk{"clk"};
sc_in<bool> req{ "req"};
sc_out<bool> resp{"resp"};
sc_signal<sc_uint<8>> val{"val"};
sc_vector<sc_signal<bool>> enbl{"enbl", N};
...
SCT_ASSERT(req || !resp, clk.pos());
SCT_ASSERT(req, SCT_TIME(1), resp, clk.pos());
SCT_ASSERT(req, (2), val.read() == N, clk.neg());
SCT_ASSERT(val.read() == 0, SCT_TIME(3,1), val.read() == 1, clk);
SCT_ASSERT(enbl[0], (3,1), enbl[1], clk);
SCT_ASSERT(!resp, SCT_TIME(T+1,T), resp, clk);
```

Listing 1. Assertions in module scope

```
`ifndef SVA_OFF
sctAssertLine48 : assert property (@(posedge clk) 1 |-> req || !resp );
sctAssertLine49 : assert property (@(posedge clk) req |=> resp );
sctAssertLine50 : assert property (@(negedge clk) req |-> ##2 val == 4 );
sctAssertLine51 : assert property (@(clk) val == 0 |-> ##[1:3] val == 1 );
sctAssertLine52 : assert property (@(clk) enbl[0] |-> ##[1:3] enbl[1] );
sctAssertLine53 : assert property (@(clk) !resp |-> ##[3:4] resp );
`endif // SVA_OFF
```

Listing 2. Generated SVA in module scope

C. Assertions in clocked thread function

Temporal assertions in clocked thread can be placed in reset section or right after reset section before main infinite loop. Temporal assertions in function scope semantic:

- *SCT_ASSERT (LHS, TIME, RHS)* – assertion with pre-condition *LHS*, post-condition *RHS*, time parameter *TIME*, evaluated and checked every time when thread process activated;
- *SCT_ASSERT_LOOP (LHS, TIME, RHS, ITER)* – loop assertion in function scope, effectively run for each loop iteration, has additional loop counter variable(s) *ITER* parameter.

These assertions can operate with member signals, ports and others like module scope assertions. They also can operate with all local variables and module member data variables (not signals/ports) which are modified in this process. Accessing member data variables modified in another process leads to data races and therefore prohibited.

There is an example of temporal assertions in SystemC clocked thread function:

```
void thread_proc() {
   // Reset section
   SCT_ASSERT(req, SCT_TIME(1), ready);     // Assertions in reset section
   wait();
   SCT_ASSERT(req, SCT_TIME(2,3), resp);    // Assertions after reset section

   // Main loop
   while (true) {
      ...                                   // No assertion in main loop
      wait();
}}
```

Listing 3. Assertions in clocked thread

Assertion in reset section is generated in the end of *always_ff* block, that makes it active under reset. Assertion after reset section is generated in else branch of the reset if, that makes it inactive under reset.

```
always_ff @(posedge clk or negedge nrst) begin
   if (~nrst) begin
      ...
   end else
   begin
      ...
   `ifndef SVA_OFF
      assert property (req |-> ##[2:3] resp);  // Assertions after reset section
   `endif // SVA_OFF
   end
`ifndef SVA_OFF
   assert property (req |=> ready);            // Assertions from reset section
`endif // SVA_OFF
end
```

Listing 4. Generated SVA in module scope

Loop assertions can be placed in any loop with statically determined number of iterations and one counter variable. Such loop cannot have *break*, *continue*, and *wait()* calls. *goto* statement is not considered, as it is not supported for synthesis [1]. Loop assertions can be placed in nested loops. There is an example of loop assertions:

```
static const unsigned N = 4;
static const unsigned M = 3;
sc_vector<sc_signal<bool>> enbl {"enbl", N};
sc_vector<sc_vector<sc_signal<bool>>> actv{"actv", N};  // Initialized as N x M
...
void thread_proc() {
   // Reset section
   for (int i = 0; i < N; ++i) {
      SCT_ASSERT_LOOP(enbl[i], SCT_TIME(1), !enbl[i], i);
      for (int j = 0; j < M; ++j) {
         SCT_ASSERT_LOOP(actv[i][j], SCT_TIME(2), actv[i][M-j-1], i, j);
   }}
   wait();
   while (true) {
      ...
      wait();
}}
```

Listing 5. Assertions in loop

```
always_ff @(posedge clk or negedge nrst) begin
   ...
`ifndef SVA_OFF
    for (integer i = 0; i < 4; ++i) begin
        sctAssertLine70: assert property (enbl[i] |=> !enbl[i]);
    end
    for (integer i = 0; i < 4; ++i) begin
        for (integer j = 0; j < 3; ++j) begin
            sctAssertLine72: assert property (actv[i][j] |-> ##2 actv[i][3-j-1]);
        end
    end
`endif // SVA_OFF
end
```

Listing 6. Generated SVA in loop

IV. Temporal Assertions Implementation

A. *Temporal assertions in SystemC simulation*

The temporal assertions are implemented with *SCT_ASSERT* and *SCT_ASSERT _LOOP* macros. In SystemC simulation these macros replaced with:

1. Dynamic allocation of a *sct_property_expr* class which captures *LHS* and *RHS* as lambdas, and gets time parameters. This class *operator()* evaluates lambdas and stores pre- and post-condition traces in specified time interval, checks post-condition if pre-condition was true and reports error in case of violation.
2. Registration of the *sct_property_expr* instance in a static map with hash which is calculated for assertion string, process name and loop iteration(s). That ensures one *sct_property_expr* instance for an assertion in each module instance and loop iteration.
3. Create spawned method process sensitive to *EVENT* or current thread process event, which runs *sct_property_expr()*.

For assertions in process function it needs to get current process sensitivity events. For clocked thread process *sc_process_b* class, there is *m_static_events* field contains required sensitivity. To get access to *m_static_events* getter function *get_static_events()* has been added into *sc_process_b*. That is one SystemC patch which is required. We suppose, this can be fixed in future Accellera SystemC versions.

Lambda functions for assertion expressions capture all parameters by reference. That allows to have updated values of module members and function local variables. For *SCT_ASSERT _LOOP* each loop iteration determines a new assertion instance, so iteration variables should be captured by value. That is the reason why loop counter variables are specified in *ITER* parameter.

We have implemented the temporal assertions in C++11 as it is enough to express desired properties of assertion expressions and time condition.

B. Temporal assertion translation into SVA

Temporal assertions are planned to be translated into SVA during HLS of the SystemC design. Existing HLS tools are typically based on static analysis of the design code. Some HLS tools combines static analysis with dynamic elaboration, but process functions and other evaluation stage logic should be statically analyzed. To simplify code analysis and translation of temporal assertions in function scope the *SCT_ASSERT* macro is replaced with *sct_assert_in_proc_func()* function call. This function has the same parameters as *SCT_ASSERT* macro. That works if *__HLS__* is defined. For *SCT_ASSERT* in module scope there is variable of `sct_property_mod` type declared. The variable name constructed with line number where *SCT_ASSERT* placed. The `sct_property_mod` gets the same parameters as the *SCT_ASSERT* macro. Having specified function/constructor call with assertion parameters should be enough to extract all required information from Abstract syntax tree (AST), Control flow graph (CFG) or any other code representation.

```
// Assertion functions for SCT_ASSERT in function scope
template<class T1, class T2>
void sct_assert_in_proc_func(bool lhs, bool rhs, const char* name, T1 lo, T2 hi)
{}
template<class T1>
void sct_assert_in_proc_func(bool lhs, bool rhs, const char* name, T1 time)
{}

// Assertion class for SCT_ASSERT in module scope
struct sct_property_mod {
    explicit sct_property_mod() {}
    template<class T1, class T2>
    explicit sct_property_mod(bool lhs, bool rhs, sc_event_finder& event,
                              const char* name, T1 lo, T2 hi) {}
    template<class T1, class T2>
    explicit sct_property_mod(bool lhs, bool rhs, sc_port_base& event,
                              const char* name, T1 lo, T2 hi) {}
    ...
}
```

Listing 7. Assertion function for code analysis

V. EVALUATION AND ANALYSIS

The temporal assertions introduce additional computation in dedicated processes, therefore it is important to estimate their performance and SystemC simulation slow down. Table I shows SystemC simulation time increase for two artificial designs with various sets of assertions and without them. SystemC simulation done by Accellera SystemC. Assertions in module scope and in process functions create the same spawned process, therefore have the same performance, so we evaluated only assertions in module scope. Both artificial designs have simple testbench with clocked thread process, which works as stimulus and result checker.

These experiments show assertions with different complexity of left and right expressions have similar performance. That means simulation resources mostly depends on number of assertions. Assertions with single time and time interval have almost the same performance, if time interval is small (less than 10). Increasing assertion time interval, requires more simulation time. In our practice, most of temporal assertions added into industrial designs have single time or time interval with high time less than 5.

Table I. Performance estimation for artificial examples

Design	Process number	Assertions	SystemC simulation time increase
Summator	2	One assertion w/o pre-condition, single time	10%
		One assertion with pre-condition, single time	12%
		One assertion with pre-condition, time interval (1,3)	13%
		One assertion with pre-condition, time interval (10,30)	15%
FIFO	5	One assertion checks FIFO is not empty after push	4%
		Four assertions check main FIFO properties	17%

We have evaluated the temporal assertions in several industrial designs. The assertions have been used for checking design top level interface properties and module internal properties. Temporal assertions helped us to prevent a few real bugs.

Table II presents simulation time increase for SystemC and Verilog simulation for industrial designs A, B, C, D, and E. For each of the SystemC designs with and without temporal assertions we generated Verilog code with SystemC-to-Verilog compiler [2]. The temporal assertions were translated into equivalent SVA by the tool. SystemC simulation done by Accellera SystemC, Verilog simulation done by one of commercial tools. Both SystemC and Verilog simulations used the same SystemC testbench. SystemC simulation time with assertions increase is 4-16%, that is comparable with Verilog simulation time increase 5-15%. So, we can conclude the temporal assertion performance is similar to SVA performance.

Table II. Performance estimation for industrial designs

Design	Process number	Assertion number	SystemC simulation time increase	Verilog simulation time increase
A	71	19	11%	15%
B	68	21	16%	15%
C	101	22	5%	5%
D	109	41	14%	11%
E	212	38	4%	6%

VI. CONCLUSION

SystemC language becomes more and more popular for digital design and verification, that needs new features to meet different design flow requirements. The proposed temporal assertion can be used by many engineers to improve coverage of the design properties to be checked during simulation. Support of the assertion translation into SVA allows to preserve checking IP properties after integration into SoC.

This paper demonstrates possibility to implement a subset of SVA in C++11 and can be considered as a proposal to SystemC language workgroup.

REFERENCES

[1] "SystemC Synthesizable Subset Version 1.4.7". http://accellera.org/downloads/standards/systemc.

[2] M. Moiseev, R. Popov, I. Klotchkov, "SystemC-to-Verilog Compiler: a productivity-focused tool for hardware design in cycle-accurate SystemC," in proceedings of DvCon`19.

[3] A. Habibi and S. Tahar, "On the extension of SystemC by SystemVerilog assertions," Canadian Conference on Electrical and Computer Engineering 2004, Niagara Falls, Ontario, Canada, 2004, pp. 1869-1872 Vol.4.

[4] W Ecker, V Esen, J Smit, T Steininger, M Velten "Implementation of a systemc assertion library", in proceedings of IP Based SoC Design (IP/SOC), 2005.

[5] W. Ecker, V. Esen, T. Steininger, M. Velten and M. Hull, "Implementation of a Transaction Level Assertion Framework in SystemC," 2007 Design, Automation & Test in Europe Conference & Exhibition, Nice, 2007, pp. 1-6.

[6] H. Sohofi and Z. Navabi, "Assertion-based verification for system-level designs," Fifteenth International Symposium on Quality Electronic Design, Santa Clara, CA, 2014, pp. 582-588.

[7] H. M. Le, D. Große, V. Herdt and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, 2013, pp. 1-6.

[8] M. Kallel, Y. Lahbib, R. Tourki and A. Baganne, "Aspect-based ABV for SystemC transaction level models," 2009 International Conference on Microelectronics - ICM, Marrakech, 2009, pp. 304-307.

[9] D. Grosse, H. M. Le and R. Drechsler, "Induction-Based Formal Verification of SystemC TLM Designs," 2009 10th International Workshop on Microprocessor Test and Verification, Austin, TX, 2009, pp. 101-106.

[10] M. Glukhikh, M. Moiseev, H. Richter, "A static analysis approach for formal verification of SystemC designs", in processding of Radio electronic and computer systems, 2013, № 5. pp. 227–232.