

# Verification Methodologies in a TLM-to-RTL Design Flow

Atsushi Kasuya

JEDA Technologies Inc.  
4962 El Camino Real, Suite 105  
Los Altos, CA 94022  
1-650-964-5332

atsushi@jedatechnologies.net

Tesh Tesfaye

JEDA Technologies Inc.  
4962 El Camino Real, Suite 105  
Los Altos, CA 94022  
1-650-964-5332

tesh@jedatechnologies.net

## ABSTRACT

SoC based system developments commonly employ ESL design methodologies and utilize multiple levels of abstract models to provide feasibility study models for architects and development platforms for software engineers. Such models are evolving to finer abstract models as the development moves forward. The correctness of these models coupled with the ability of having a temporal debug environment to identify and fix model issues is critical for both hardware and software development efforts that make use of such models. This paper presents the mechanism to construct temporal assertions at models in various abstract levels and reuse the assertions on models at different abstract level.

## Categories and Subject Descriptors

J.6 [COMPUTER-AIDED ENGINEERING]: Computer-aided design (CAD)

## General Terms

Measurement, Performance, Design, Reliability, Standardization, Languages, Verification.

## Keywords

SystemC, Assertion, Verification, TLM, PV, PVT.

## 1. INTRODUCTION

Due to the complexity of SoC design, it is important for a system design team to check the functionality and performance that involves both hardware and software at the early stage of the development. In such a situation, it becomes common to employ ESL design methodologies to construct virtual platform environments. The virtual platform is evolving into finer abstract levels as the design proceed. PV model provides a functional model without timing, that can be a platform for a software design. Due to the higher abstraction level of the model, it has the advantage of simulation performance.

PVT model adds an estimated timing to provide a high level view of the system performance. This allows the system architect to study the design tradeoff to optimize the system architecture to fill full the requirements. These models can be evolved into finer

abstraction (e.g. micro architecture model), and eventually reached to RTL for hardware synthesis.

This top down design methodology is the state-of-art approach to the system level design, but requires a lot of resources, thus, to place the reusability into the model and testbench development is one of the key factors for the successful development. Using a well-defined TLM (Transaction Level Modeling) mechanism provides a way to reuse the resources (both model and testbench) at the different stages of the development.

The verification technique using assertions is commonly used in RTL design to embed the checker into the design and/or the system boundary. This assertion technique provides simple but effective checks to detect the design problems at exact timing of the false behavior. Thus, the user can debug the problem easily at the assertion failure point. The temporal assertion languages ( i.e. SVA and PSL) in RTL design is becoming a common tool to construct cycle level assertions. [1] [2]

In order to utilize such a verification methodology, we've developed 'NSCa', a native assertion mechanism in SystemC environment. [3] NSCa can construct a cycle level accuracy rule of the design as assertion expression form. At this level, the cycle level assertion property is constructed with cycle level temporal primitives that are similar to SVA and PSL.

Using the temporal logic to specify and check the software correctness has been researched in various ways. [4][5][6] Those works mainly focused on proofing the correctness statically with reach-ability analysis. This type of check requires a strict mathematical formalism and the target software must be statically analyses-able, and most of the case, requires to rewrite or transform the original code into different representation. The actual description of the specification tends to be hard to write and understand, and the analysis is not always possible for a given code.

Thus, we decided to develop a simpler description mechanism to depict a high level system behavior that can be checked dynamically. Given that basic idea, we extended the temporal primitives in NSCa to provide the mechanism to construct assertions for higher abstract models. This support is categorized to two levels. One level is for PVT model to construct timing requirement assertion based on event and a simple queue model.

Another level is for PV model to construct pure event sequence based assertion. This is achieved by a set of totally new temporal primitives that works purely on events without any clock.

This paper shows the temporal assertion mechanisms in NSCa that is designed to work for those three level models, and the examples for each level, as well as the mechanism to reuse the assertions to lower level abstraction models.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

Copyright 2007 ACM 978-1-59593-627-1/07/0006 ...\$5.00.

## 2. Cycle Accurate Temporal Assertion

The foundation of NSCa temporal assertion mechanism is based on the well-established assertion designs, SVA and PSL. NSCa allows users to use the assertions natively within the C++/SystemC code.

There are two ways to write assertions with NSCa. We provide the NSC syntax, which naturally extends the SystemC/C++ syntax to support assertion notations. The user code written in this extended syntax is translated into SystemC/C++ code by the *NSC Translator*. The second option is to use assertion macros that provide the same functionalities in C macro forms. With this option, the users can stay within the standard C++ formal syntax. The table 1 shows the temporal primitives for cycle accurate level assertion. They are derived from SVA, and by its nature, are also similar to PSL.

Table 1. Cycle Accurate Temporal Primitives

NSC Notation	Macro Name	Function
nsc_property	NSC_PROPERTY	property declaration
nsc_always	NSC_ALWAYS	always property declaration
nsc_assert	NSC_ASSERT	property assertion invocation (spawn a thread)
nsc_pand	NSC_PAND	property-and operation
nsc_por	NSC_POR	property-or operation
nsc_not	NSC_NOT	property-not operation
->	NSC_IMPLY	implication
=>	NSC_NOIMPLY	non-overlap implication
<property name>	NSC_CALL	property instance call
nsc_sequence	NSC_SEQUENCE	sequence declaration
@ [ m : n ]	NSC_SEQ(m,n, ...)	m to n cycle delay
( <s> , <m> )	NSC_MATCH	sequence match item where <s> : sequence, <m> : match item
[ * m : n ]	NSC_CREP(m,n, ...)	m to n consecutive repetition
[ -> m : n ]	NSC_GOTO(m,n, ...)	m to n goto repetition
[ = m : n ]	NSC_NREP(m,n, ...)	m to n non-consecutive repetition
nsc_and	NSC_AND	sequence-and operation
nsc_or	NSC_OR	sequence-or operation
nsc_intersect	NSC_INTERSECT	sequence-intersect operation

nsc_within	NSC_WITHIN	sequence-within operation
nsc_throughout	NSC_THROUGHOUT	sequence-throughout operation
nsc_first_match	NSC_FIRST_MATCH	sequence-first match operation
<sequence name>	NSC_CALL	sequence instance call

The assertion expression can be used as a native C++ expression within the SystemC thread context. For example, the following code uses a temporal expression as a condition within an if statement.

```

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

```

By using such a powerful temporal expression into the SystemC code, the users can write a compact, self-checking testbench code effectively.

## 3. OCP Compliance Checker

By utilizing the cycle level temporal assertion mechanism, we developed a complete set of OCP 2.0 Protocol Compliance Checker. [7] The Figure 1 shows the structure of the NSCa OCP checker.

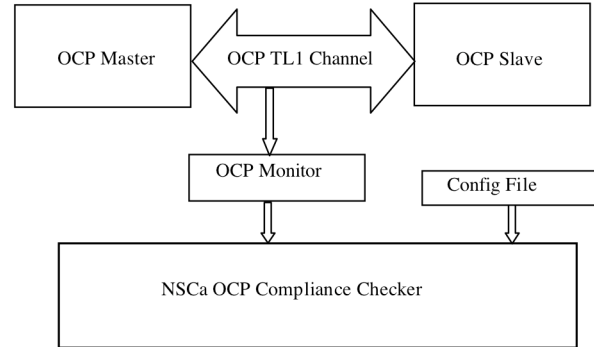


Figure 1 NSCa OCP Compliance Checker

The checker provides assertions that covers the check list recommended by the OCP-IP verification working group. NSCa monitors the assertion activities, and provides the path coverage information over the protocol rules to assess the completeness of the test cases.

## 4. Assertions on PVT Model

In addition to the temporal logic primitives presented above, NSCa contains Transaction Level Assertion (TLA) primitives that provide support to construct high level assertions for PVT model. TLA primitives are categorized into three types: the asynchronous evaluation primitives, the mutex primitives, and the high level

assertion primitive classes. The first two types are provided as NSCa assertion engine primitives, while the high level assertion primitive classes are provided as a C++ class library.

### 4.1 Asynchronous Evaluation Primitives

The asynchronous primitives are used to synchronize an assertion with alternative clock, or event. The primary design goal of this type of primitives is to capture an event from a transactor and construct a high level assertion sequence from it. This primitive can also be used for specifying multi-clock temporal assertion as well. There are two types of asynchronous evaluation primitives, as shown in Table 2.

Table 2. Asynchronous Evaluation Primitives

NSC Notation	Macro Name	Function
@@ nsc_event	NSC_AWAIT ( nsc_event )	Wait for an nsc_event notification.
( <sequence> ) @ nsc_event	NSC_ASYNC ( nsc_event, <sequence> )	<sequence> is evaluated with nsc_event as the default clock.

The first asynchronous primitive is to simply capture the notification on an event. For example,

```
nsc_always ( @@ addr_phase_done ) |-> check_data_phase();
```

Note that ‘nsc\_always’ loop is evaluated at the first element of the property, so the always loop is executed on the notification of addr\_phase\_done event.

This primitive can be used with consecutive repetition, as:

```
nsc_sequence ack_five () {
    ( @@ ack_done ) [*5]
}
```

The evaluation of this sequence checks the ack\_done event for 5 times.

The second asynchronous primitive is used to overwrite a given clock with an alternative one. The sequence encapsulated in this primitive is evaluated with the given clocking event. For example:

```
nsc_sequence alt_clk_seq() {
    ( @[5] ( y == 0 ) [*1,5] @[0,5] ( x == 0 ) ) @ alt_clk
}
```

With this primitive, the assertion can be written over multiple clock cycles. However, the real intention for this primitive is to capture the event notification from a transactor. The following example shows the usage of this primitive in such a context.

```
nsc_property check_rd_cmd() {
    always ( ( cmd.read() == rd_cmd ) @ addr_phase_done )
    |-> check_rd_sequence();
}
nsc_property check_wr_cmd() {
    always ( ( cmd.read() == wr_cmd ) @ addr_phase_done )
    |-> check_wr_sequence();
}
```

Unlike running on a multiple clock, the property above can run purely on the notification of the events. Thus, this assertion is well suited for transaction level modeling tasks.

### 4.2 Mutex Primitives

Mutex primitive provides the mutual exclusion mechanism within an assertion. It works just like SystemC’s sc\_mutex construct, providing a mutual exclusion locking capability that only allows a single assertion flow to enter in the critical region at the same time while blocking all other assertion accesses. The mutex primitive is identified with a name string. The mutex call with the same name is exclusively controlled, and only one active assertion evaluation can enter the critical region. The mutex works as FIFO queue, and the schedule of the evaluation threads is managed by the arrival order.

Table 3 shows the mutex primitives. There are two types of mutex control mechanisms. In the table, the first one is to specify a sequence to be mutually exclusive. The second and third are pared to specify the entry point and exit point of the mutex individually.

Table 3. Mutex Primitives

NSC Notation	Macro Name	Function
nsc_mutex ( name_str, <sequence> )	NSC_MUTEX(..)	Acquire the mutex, evaluate <sequence> in mutually exclusive, then release the mutex

With the first mechanism, the sequence evaluation is blocked when other evaluation is in the critical region. Once the mutex is obtained, then the mutex is blocked while the inside sequence(<sequence>) is being evaluated. The mutex is released at the completion of the <sequence>.

Here’s an example of the mutex usage:

```
nsc_always
( @@ addr_phase ) |->
nsc_mutex ( “data_mutex”, @@ data_phase );
```

In this example, the right hand evaluation starts every time the addr\_phase is notified. If the bus structure allows the use of pipelined operation, it is possible that multiple evaluation of the right hand sequence can be invoked by multiple addr\_phase

notifications prior to the assertion of `data_phase`. In such case, the mutex “`data_mutex`” guarantees that only one evaluation of the right hand side catches one `data_phase` notification. This mechanism is similar to using `sc_mutex` within a transactor to prevent from multiple threads accessing the bus resource at the same time.

When the mutex is released after receiving the notification on `data_phase`, the next evaluation thread obtains the mutex and waits for the next notification on `data_phase`. Note that the notification is stateless, thus this new evaluation thread does not capture the previous notification before the mutex release.

### 4.3 High Level Assertion Primitive Classes

The predominant objective of any high level assertion for PVT model is to enumerate and capture the performance and/or functional requirements of a system during high level modeling in the form of assertions. Unlike temporal assertions that are used for cycle-based checking protocol between signals, high level assertions require more analytical and statistical checking. Coming up with a set of predefined primitives forms tailored for high level assertions is difficult because such analytical and statistical checks are application oriented. For example the checking needs for networking systems is different from that of imaging systems.

Given this difficulty, NSCa provides a small number of relatively simple primitive classes as a C++ class library. By extending these classes, users can further construct primitive classes that fit their application’s specific needs.

Currently, three types of primitive classes are provided: `nsc_simple_queue`, `nsc_id_queue`, and `nsc_coverage_bin`. The first two classes are to monitor various queue type activities. Those two queue classes have common member functions, `arrive()` and `depart()`. The function `alive()` should be called when an entry is arrived to the queue, and the function `departure()` should be called when the entry is leaving from the queue.

The `nsc_coverage_bin` class is a generic coverage bin class essential to construct a user specific functional coverage.

## 5. OCP TL2 Example

The OCP interface standard is an established interface protocol for SoC designs [8]. The communication protocol is implemented in various layers of SystemC Transaction Level Modeling (TLM) to provide proper abstract levels for various phases of the system development.

The OCP TL2 transactor is designed for architectural evaluation and high level modeling. It is a timed protocol model but not cycle-accurate. The transactors are executed in the event-driven scheme. In such an abstraction level, we cannot use the ordinal temporal assertion functionality as there is no notion of clock. But the TLA mechanism can implement reasonable assertions for such a model.

Figure 2 shows the structure of the assertion. The events to notify the request end and response start are extracted from the TL2 transactors, and connected to NSCa assertion.

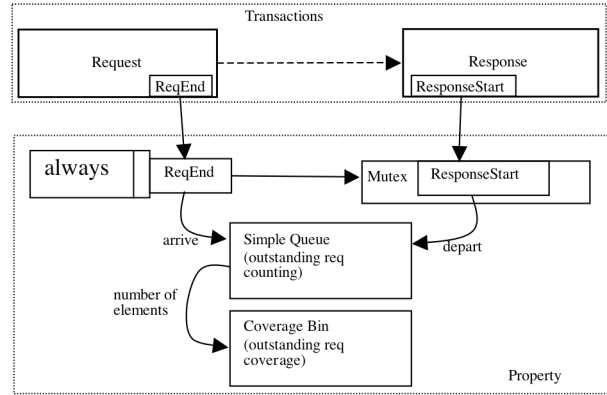


Figure 2. Outstanding Request Monitor

The actual code that implements the assertion is the following:

```
class nsc_a_ocp_tl2_assertion {
..
public:
    nsc_simple_queue req_que ;
    nsc_coverage_bin req_cov ;
    // constructor
    nsc_a_ocp_tl2_assertion(nsc_a_ocp_tl2_events* evnts ) {
..
        // necessary initialization
        req_que.set_limit( NUM_OUTSTANDING_REQ ) ;
        req_cov.set_max_bin_mum(NUM_OUTSTANDING_REQ+1) ;
    }
    // The assertion property declaration.
    property ocp_outstanding_request() {
        int t ;
        always (
            @@ReqEnd, t = req_que.arrive(), req_cov.covered(t)
        ) |->
            nsc_mutex( “ocp_res_end_mutex”
                ( @@ResponseStart, req_que.depart() )
            );
    }
}
```

The timing of a request accepted by the target, to the timing of a response sent by the target is captured into a simple queue structure to measure the outstanding request. The number of outstanding requests is collected in the coverage bin. The response event is protected with the mutex to maintain the pipeline behavior. At the acceptance of the event `ResponseStart`, `req_que.depart()` is called to notify it to the queue.

## 6. Assertion on PV Model

For PV model (or other pure software activities) where a sequence cannot be detected as the simulation time transition, using events for driving assertion may not work, as a functional model is computed without any preemption. Thus, a different mechanism to create a trigger connection between user models and assertions is necessary. In order to capture such a transition within a PV model, the callback mechanism is needed to report an event occurrence at the point of transaction processing, such that an event within the system is reported as a function call (callback). We have defined this mechanism of this on the TLM 2.0 example by extending Analysis Port interface. [9] [10]

In order to construct a meaningful assertion over PV model, a new set of TLA primitives is defined. [11] These primitives are designed to create a proper sequence without depending on the clock. The table 4 shows the PV TLA primitives.

Table 4. PV Temporal Primitives

NSC Notation	Macro Name	Function
tla_always	TLA_ALWAYS	always sequence declaration
tla_alwaysif tla_then	TLA_ALWAYSIF	always if sequence declaration
tla_repeat	TLA_REPEAT	repeat sequence
tla_within	TLA_WITHIN	inclusion on a sequence in antother sequence
tla_without	TLA_WITHOUT	exclusion on a sequence in antother sequence
tla_followedby	TLA_FOLLOWEDBY	a sequence followed by another sequence
tla_coswitch tla_case	TLA_COSWITCH	concurrent switch (branch)
tla_until	TLA_UNTIL	repeated evaluation of a sequence until another sequence is detected

A brief explanation of each primitives is the following. Detailed explanation can be found in the reference. [11]

1) tla\_always (<tra\_sequence> )

This primitive defines a property that is always evaluated. Note that there will be only one active evaluation of the <tla\_sequence> under this property. The next evaluation will only start when currently active evaluation of the sequence is completed.

2) tla\_alwaysif (<tla\_sequence> ) tla\_then (<tla\_sequence> )

This primitive evaluates the second tla sequence whenever the first sequence is observed. Thus, multiple thread of second tla sequence may be evaluated concurrently.

3) tla\_repeat ( N, <tla\_sequence> )

The tla\_repeat primitive defines the repeated occurrence of the tla\_sequence. The first integer parameter N defines the number of repetition.

4) (<tla\_sequence> ) tla\_within (<tla\_sequence> )

The first and second tla\_sequence are evaluated concurrently. The first tla\_sequence must be detected before the second tla\_sequence is detected.

5) (<tla\_sequence> ) tla\_without (<tla\_sequence> )

The first and second tla\_sequence are evaluated concurrently. The first tla\_sequence must not be detected before the second tla\_sequence is detected. At the detection of the second sequence, the evaluation of the first sequence will be terminated as well.

6) ( M, <tra\_sequence> ) tla\_followedby ( N, <tla\_sequence> )

The first integer parameter M defines the number of occurrence of the first sequence. Assigning negative value to the first integer means this must be endlessly checked.

The second integer parameter N defines the degree of freedom. If N is defined to be 0 (zero), then the second sequence must strictly follow the first one. A non zero positive value of N defines how many outstanding first sequence can happen before the second sequence is detected.

4) (<tla\_sequence> ) tla\_until (<tla\_sequence> )

This primitive evaluates first sequence until the second sequence is observed. When a match of the first sequence is detected, it will be evaluated again from the top, until the second sequence is detected.

## 7. A Simple PV TLA Example

Using the callback filter layer and the TLA primitives, we can define various properties that the system must always follow.

One very simple application is to define the restriction over the register access sequence in an IO device. Assume that there's two registers reg\_a and reg\_b within an IO device that must be accessed atomically. Thus, accessing reg\_a must followed by accessing reg\_b. This is a typical situation in a device with address register and target register, that the address to the actual internal register must be set to the address register, then the target register access must be followed.

Here, we assume that two callback entries REG\_A\_ACCESS and REG\_B\_ACCESS are declared with proper filtering such that it provides a callback event on the write access to those registers.

The following property can be used to monitor the violation.

```
tla_always (
    (-1, REG_A_ACCESS() )
    tla_followedby ( 0, REG_B_ACCESS() )
)
```

Here, the first number -1 means the infinite check of this sequence, the second number 0 means there's no freedom, so that the reg\_a access must strictly followed by reg\_b access.

This property can be also written as

```
tla_alwaysif( REG_A_ACCESS() )
tla_then(
    ( REG_B_ACCESS() ) tla_without ( REG_A_ACCESS() )
)
```

At PV model level, application dependent access rules can be written as the assertion properties. For example, the function layer of the USB device can be modeled without the USB device layer,

and the host application can directly talk to the device via a simplified channel. In such a case, the device class level access rules can be placed as a TLA assertion. Such assertion set is reusable when other models in different abstract levels are developed.

## 8. Assertion Reuse

Reusing the verification code to various level of abstraction is an important factor for successful verification. [12] A good set of assertion in various abstraction level can be considered as executable specification. Assertions developed at higher abstract models can be reused in the lower abstract models. This is useful to confirm the requirements and constraints for the system defined at the higher abstraction model are carried over and maintained at the lower level, down to the final implementation. In order to reuse such high level assertions, the transactors at the different abstraction level should carry the common logical trigger point. JEDA Technologies has suggested the extension to Analysis Port in TLM 2.0. [13] It is important to have such a standard carries the guide line for developing transactors in various level such that the verification task can easily be connected to the existing testbench code. When proper guideline is given, it is relatively easy to find those useful connection points and put the required trigger points to the transactors.

When the design process moved forward, it is not always true that the testbench carries the transactors to communicate with the design model. When multiple SoC modules are instantiated and they are communicating each other, there's no transactor to extract the assertion event on that path. In such case, an adaptor to extract the proper trigger point from the signal level activity may be needed. In such a case, it is a possible solution to use the cycle level assertion mechanisms explained in previous chapter to construct the trigger point extractor. [13] Instead of writing a code directly, using CLA primitives allows the users to write compact, maintainable event extractor.

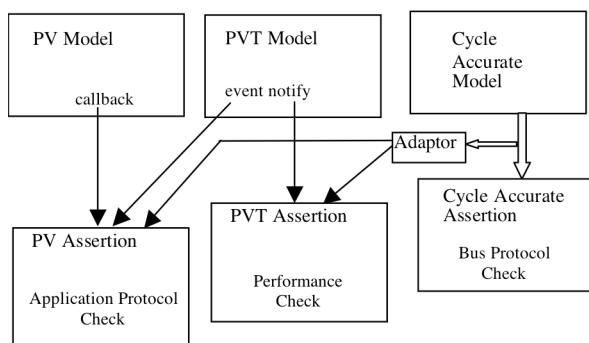


Figure 3 Assertion Reuse

## 9. Conclusion and Future Work

In this paper, we've explained our native SystemC assertion mechanism and demonstrated some ideas for constructing temporal assertions for higher level models, including PVT and PV model. A set of new assertion primitives to handle the temporal logic beyond the cycle accurate level is defined. The

ability to express the PV model level temporal assertion allows users to define requirements for the embedded software in the assertion form.

We've suggested an extension to analysis port in OSCI TLM 2.0 draft to be able to utilize the port mechanism for various verification tasks, includes callback event for assertion. It is important to have the guideline and examples within such standard to integrate the verification activities easily. In order to reuse the assertion properties at different abstraction level, the transactors must be carefully defined and developed throughout the design and verification process. We are working on TLM WG and Verification WG to provide the requirement and examples.

Since TLA primitives shown here is our new development with a limited usage in the actual system, we are working with customers to develop assertions for various applications with this mechanism. We will refine those primitives to make the mechanism more effective as our experiences accumulated.

The NSCa primitives shown in this paper are submitted to OSCI Verification WG as a candidate of the verification standard. We'll keep working on the standardization effort at the WG.

## 10. REFERENCES

- [1] [1800-2005 IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language](#)
- [2] [Accellera PSL v1.1 LRM](#), Accellera
- [3] [NSCa Native SystemC Assertions](#), JEDA Technologies, Inc., Product Information
- [4] [Specifying Systems](#), Leslie Lamport, Addison-Wesley ISBN0-321-14306-X
- [5] [The BLAST Query Language for Software Verification](#), Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar, Proceedings of the 11th International Static Analysis Symposium
- [6] [Using SPIN Model Checking for Flight Software Verification](#), Peter R. Gluck, Dr. Gerard J. Holzmann, 2002 IEEE Aerospace Conference Proceedings
- [7] [NSCa OCP checker](#), JEDA Technologies, Inc., Product Information
- [8] [OCP Specification 2.2 Release](#), ocpip.org
- [9] [Utilizing Analysis Port for Verification Callback](#), JEDA Technologies, Inc. Submitted document to OSCI Verification WG
- [10] [Temporal Assertion over Analysis Port Interface](#), JEDA Technologies, Inc. Submitted document to OSCI Verification WG
- [11] [NSCa PV primitive document](#), JEDA Technologies, Inc. Submitted document to OSCI Verification WG
- [12] [Maintaining Consistency Between SystemC and RTL System Designs](#), Bruce, A. Nightingale, A. Romdhane, N. Hashmi, M.M.K. Beavis, S. Lennard, C, 2006 DAC
- [13] [TLA Reuse method.](#), JEDA Technologies, Inc. White Paper