

A C++ Class Structure for System-Level Assertion-Based Verification

Abstract

Assertion-based verification (ABV) is becoming a popular method of capturing requirements for a design and checking the behavior of the design in simulation against its expected behavior. A number of assertion languages are already widely used in industry to perform ABV on register-transfer level (RTL) designs. Meanwhile, new system-level design languages are helping designers manage the growing complexity of simulation models. However, there is little support for ABV in either system-level languages or the development environments currently available for them. This paper presents a scalable C++ class structure designed to provide basic ABV capabilities for simulation of system-level models. We also describe how we tested the class structure on two design examples.

1. Introduction

An assertion is a statement regarding the expected behavior of some part of a system, individually or in relation to other parts. A designer or verification engineer can capture the specifications of a design by writing a set of assertions that defines how the behaviors of different parts of the system should be related. Then, when the system is tested in simulation, the actual behaviors that occur can be compared with the expected behaviors which the assertions specify. When a behavior violates an assertion, the violation indicates a bug in the part of the system that is responsible for the behavior. Assertions can therefore automatically detect any failure of the system to meet the specifications. System verification becomes much more thorough when assertions are used, because they can monitor behaviors within the system that may not always affect the functional output. Assertions also enhance verification productivity by pinpointing the source of each failure.

We can define a behavior as the occurrence of a specific condition in a system. An assertion then becomes a rule stating whether or not the condition should occur and what relationship (if any) its occurrence should have to the occurrence of other conditions. Thus,

the simplest type of assertion is one which says that a particular condition should always (or never) occur in a simulation. For example, we might specify that the read and write signals to a memory should never both be high at the same time. A more complex assertion could specify a temporal relationship between the occurrences of two or more conditions. For example, we could say that if a device requests access to a shared resource, it should be granted access to the resource within a fixed number of clock cycles. Naturally, there are many possible ways to relate occurrences of multiple conditions over time. The true power of assertions lies in their ability to capture such relationships in an unambiguous fashion.

All assertions, then, involve a condition which may or may not have a causal relationship to other conditions in the system. We refer to the first or only condition in an assertion as the *triggering condition*. By “first”, we mean that among the conditions in an assertion, the triggering condition should chronologically occur first in simulations. If an occurrence of this condition is related to occurrences of other conditions, then we refer to the latter conditions as *implied conditions*. We define any collection of one or more implied conditions in an assertion as the *subcondition* of that assertion. With this terminology in mind, we created the assertion types listed in Table 1. These types are based on assertion types found in Sugar [1], an industry-standard assertion language for RTL design verification.

Generally speaking, for each assertion type, condition A is the triggering condition, and conditions B and C (where present) are implied conditions. We can think of $\text{never}(A)$ as a special case where “not A” is the triggering condition. A cycle can be represented by either a single period of a system clock or a single iteration of a control loop in an untimed simulation. Note that when implied conditions are present, they are assumed to occur on later cycles than the triggering condition. This is because the system under test would require some number of clock periods or iterations to respond to the triggering condition. We constructed a C++ class to represent each of our assertion types. Below we describe the

Table 1. Assertion types.

Assertion type	Description
always(A)	Condition A must occur on every cycle of a simulation.
never(A)	Condition A must never occur on any cycle.
implies(A, B)	For every occurrence of A, there must be exactly one occurrence of B on a later cycle.
within(A, B, n)	For every occurrence of A, there must be exactly one occurrence of B no more than n cycles later.
before(A, B, C)	For every occurrence of A, there must be exactly one occurrence of B before the next occurrence of C.
before_(A, B, C)	For every occurrence of A, there must be exactly one occurrence of B either before or on the same cycle as the next occurrence of C.
until(A, B, C)	For every occurrence of A, B must occur on every subsequent cycle until the next occurrence of C.
match(A, B)	For a given sequence of values "A", there should be a matching sequence of values "B" occurring elsewhere in the system at some later time.

computational models which we used to construct the classes.

2. Computational models

Each assertion is implemented in the form of finite state machines: an assertion state machine to monitor the triggering condition, and a subcondition state machine to monitor the subcondition. The assertion state machine has four possible states, as described in Table 2. When we say that an assertion is in a given state, we are referring to the state of the assertion state machine.

The assertion state machine begins in the `not_trig` state, where it waits for the triggering condition to occur. The assertion can only enter its `pass` state when both the triggering condition and the subcondition (if there is one) have occurred in a manner that satisfies the requirements

Table 2. Assertion states.

Assertion state	Description
<code>not_trig</code>	The initial state, indicating that the triggering condition has not occurred
<code>pass</code>	Indicates a complete and valid occurrence of both the triggering condition and the subcondition (if there is one)
<code>fail</code>	Either the triggering condition or some part of the subcondition has violated the requirements of the assertion
<code>waiting</code>	All or part of the subcondition is expected and has yet to occur

of the particular assertion type. If at any time a monitored condition violates those requirements, the assertion enters its fail state. The waiting state is only valid for an assertion that includes a subcondition, and it signifies that the assertion state machine is waiting for some part of the subcondition to occur.

It should be noted that we do not consider the fail state to be a trap state in any of our assertions. That is, an assertion only remains in the fail state for those cycles where violations of the assertion's requirements actually occur. This scheme allows the assertions to report multiple failure points in the course of a simulation. Partly due to the lack of trap states, the final state of an assertion is not necessarily the state it is in for the last cycle of the simulation; it also depends on the number of cycles the assertion spends in each state. For more details on determining the final state, see section 3.5, "Assertion Checker".

For any assertion that monitors implied conditions, we define a subcondition state machine to watch for all implied conditions whenever there is an occurrence of the triggering condition A. The state value of the subcondition state machine is factored into the state table for the assertion state machine. This enables us to use the same assertion state machine with any subcondition to implement many complex assertion types. The same four state names are used in subcondition state machines as in the assertion state machine. However, in the context of the subcondition, the states are defined as shown in Table 3.

3. Class types

Table 4 lists the class types we have defined to implement assertions. Figure 1 shows a simplified version of the class structure. We describe the functionality of the class types below.

3.1. Occurrence handlers

We can define an occurrence of a condition in many different ways. However, we assume that a condition can be represented by the value of one or more variables within the system design. If a condition depends on more

Table 3. Subcondition states.

Subcondition state	Description
<code>not_trig</code>	The initial state, indicating that no part of the subcondition has occurred
<code>pass</code>	Indicates a complete and valid occurrence of the subcondition
<code>fail</code>	Some part of the subcondition has violated the requirements of the assertion
<code>waiting</code>	Some part of the subcondition is expected and has yet to occur

Table 4. Class types.

Name	Description
occ_handler	Occurrence handler – detects occurrences of a condition
assertion	Assertion base class – monitors the triggering condition of an assertion
always, never, implies, until, before, before_, within, match	Assertion classes – monitor conditions through occ_handler objects and verify the relationships between the conditions
subcond_list	Subcondition list – maintains lists of occurrences as needed
occurrence	Represents one recorded occurrence of an event, instantiated in a subcondition list
assertion checker	Instantiates objects of the occ_handler class and the assertion classes, and calls methods within these objects to check conditions and assertions

than one variable, then an expression must be written to represent it. The expression can be evaluated once per cycle or iteration in a simulation to determine whether the condition occurred. An occurrence can then be defined as a point in time when the design variable or expression has a specified value, which we call the *trigger value*. By this definition, every simulation cycle in which the variable or expression has the trigger value would count as an occurrence of the condition. We refer to this as a *cycle occurrence*. Alternatively, we could restrict the definition

to include only those times when the variable or expression changes to the trigger value from some other value. We refer to this as a *transition occurrence*. Therefore, since we have at least two possible definitions for an occurrence, we chose to create a parameterized occurrence handler class (occ_handler) for the sole purpose of identifying occurrences of conditions. This class gives the assertions the flexibility to detect and respond to any type of occurrence we define.

The occ_handler class contains a *check()* function which determines whether the specified condition has occurred. The function references a variable containing the value of the expression that represents the monitored condition. The function reads the value of the variable and compares it with the trigger value. If the occ_handler object was instructed to search for transitions to the trigger value, then the function also compares the variable with its previous value. If the function notes an occurrence of the condition, it returns a true result; otherwise it returns false. The result of the function is also assigned to the public variable *occurred* within the occ_handler object.

3.2. Assertion base class

The class *assertion* serves as a base class for all of the assertion types we have defined. The variable *status* is the state of the assertion state machine, while *subc_stat* is the state of the subcondition state machine, if the assertion includes a subcondition. The pointer *condA* points to the occurrence handler object for condition A. The *check_base()* function reads *subc_stat* and the *occurred* variable in the occurrence handler for A. The function then updates *status* accordingly.

3.3. Assertion derived classes

A class is derived from *assertion* for each assertion type we have defined. The derived classes have pointers *condB* and *condC* as needed to address occurrence handlers for conditions B and C. In Figure 1, the *always* class is shown as an example of a derived class which does not add *condB* or *condC* pointers to the base class, since the *always* assertion refers only to condition A. The *before* class is an example of a derived class which adds both *condB* and *condC* to the base class. The *within* class adds only *condB*, but it also has a pointer to a *subcond_list* object which maintains a linked list of occurrences of condition A. Every derived class implements a *check()* function. For assertions with a subcondition, *check()* updates the subcondition state and then calls *check_base()* in the base class to update the *status* variable. If there is no subcondition to consider (as is the case for the *always* and *never* assertions), the

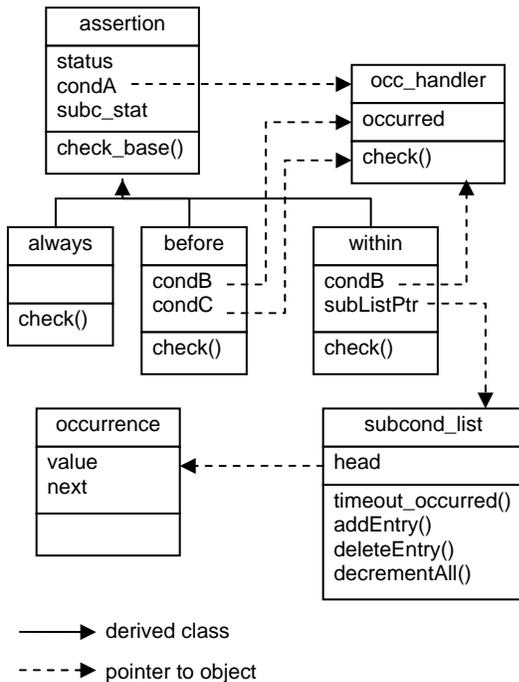


Figure 1. Class structure.

check() function can update *status* directly without calling *check_base()*.

3.4. Subcondition list and occurrence classes

The class *subcond_list* maintains a linked list of occurrences of a condition. The list is composed of objects of the *occurrence* class. The *head* pointer references the first entry in the list; the *next* pointer in each *occurrence* object references the next entry. Such a list is needed if an assertion has to store unique data about each occurrence. For example, the *within()* assertion requires that condition B occur once within a specified number of cycles after each occurrence of condition A. The *within* class uses *subcond_list* to create a list of occurrences of A that have not yet been followed by an occurrence of B. Upon each occurrence of A, *subcond_list* calls the *addEntry()* function, which creates a new *occurrence* object and sets its *value* variable to the cycle limit of the *within* assertion. Then, while the assertion waits for occurrences of B, the *decrementAll()* function subtracts one from the *value* in each *occurrence* object once per cycle. The *timeout_occurred()* function tests the head entry (the first object in the list) for a negative value, which would indicate that B failed to occur within the specified cycle limit. If this is the case, the *deleteEntry()* function removes the head entry from the list. The entry can also be deleted if and when B occurs within the cycle limit. The *occurrence* class is defined as a template so that its *value* can be of any type.

3.5. Assertion checker

An assertion checker instantiates objects from the class structure and calls functions within the objects, as shown in Figure 2. The assertion checker creates an occurrence handler for each condition that needs to be monitored, and it instantiates an object of the appropriate derived class for each assertion. Each derived class object automatically creates a corresponding object of the assertion base class. The checker then begins reading the values of any variables in the design that appear in the expression for any of the conditions being monitored. The design variables can be read either from a trace file or directly from a running simulation. The checker must process the design variables through the occurrence handlers and assertion classes one simulation cycle at a time.

After the checker reads the design variables for a given cycle, it evaluates the expression for each condition and assigns the result to a condition variable. The value of the condition variable can also be copied directly from a design variable if only one design variable represents the condition. Next, the checker calls the *check()* function in each derived class object. The derived class calls *check()*

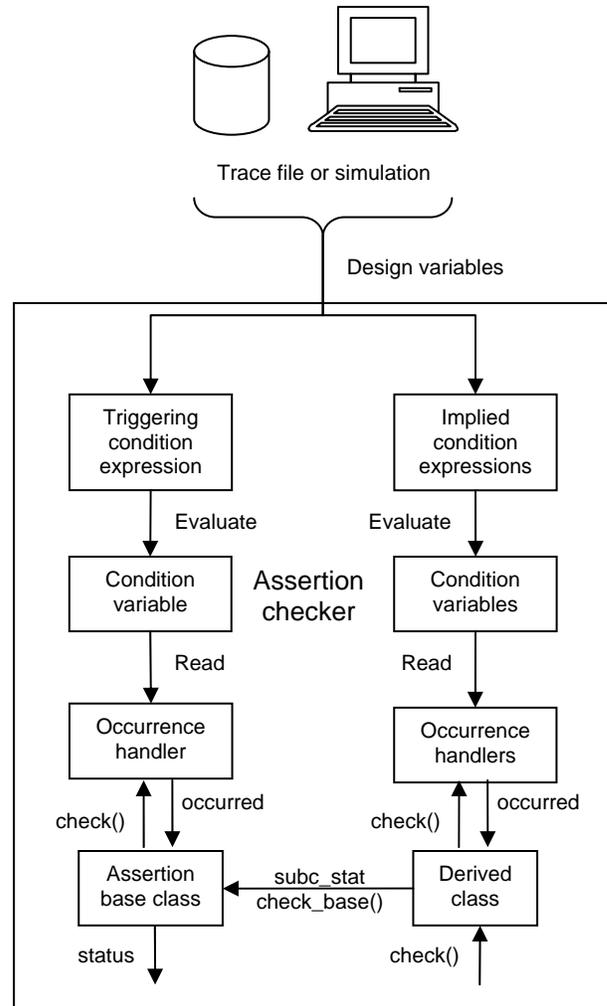


Figure 2. Assertion checker structure.

in each of its occurrence handlers to find out if any of the implied conditions occurred. The occurrence handlers return the value of the variable *occurred* for each condition. The implied conditions are checked first in case they occur before the triggering condition (an example of what we call an “out of sequence” error). The derived class *check()* function also uses the implied conditions to determine the subcondition state (*subc_stat*), which in turn is factored into the assertion state (*status*). To determine the assertion state, the derived class calls *check_base()* in the assertion base class. The *check_base()* function calls *check()* in the occurrence handler associated with the base class. This handler returns the value of *occurred* for the triggering condition. The assertion state is then updated according to the previous assertion state, the subcondition state and the value of *occurred* for the triggering condition. The derived class can also update *status* directly if there is no

subcondition. The checker reads *status* and reports whether or not the assertion failed. The checker then reads the design variables for the next simulation cycle and repeats the process in Figure 2.

At the end of a simulation, the assertion checker must confirm the state of each assertion for the simulation as a whole. We refer to this as the *final state* of the assertion. Previously we pointed out that the fail state in our assertion state machines is not a trap state. Rather, an assertion remains in a fail state only for those simulation cycles where violations actually occur. This implies that at one point in a simulation, an unsatisfactory series of occurrences could cause an assertion to enter its fail state, and then at some later point a satisfactory series of occurrences could cause it to enter its pass state. Thus, to determine the final status of an assertion, the assertion checker must record the number of cycles the assertion spent in each of its four possible states, along with the state after the last cycle of the simulation.

Table 5 gives the final state for an assertion based on the recorded state information. If the triggering condition A never occurs, then the assertion remains in the *not_trig* state for the entire simulation, and the final state is *not_trig*. The test engineer should then note the assertion as *untested* and run other tests that will produce the triggering condition in the system. If the triggering condition occurs at least once but the subcondition does not occur an equal number of times, then the assertion will be in the *waiting* state at the end of the simulation. In that case, the final state of the assertion is *fail*, because the subcondition did not occur the expected number of times. If the assertion spends at least one cycle in its fail state, its final state is also *fail*. The final state is *pass* only if the assertion spent at least one cycle in its pass state, no cycles in the fail state, and the last cycle in either the *not_trig* or *pass* state.

4. Design examples

This section describes two simple designs to which we applied our assertion classes: a producer-consumer model and a shared-resource model known as the Ornamental Garden.

Table 5. Final state of an assertion.

Number of cycles			Last cycle	Final state
waiting	pass	fail		
0	0	0	not_trig	not_trig
> 0	don't care	don't care	waiting	fail
don't care	don't care	> 0	don't care	fail
don't care	> 0	0	not_trig or pass	pass

4.1. Producer-consumer model

The producer-consumer model is pictured in Figure 3. The producer feeds integers one by one to a FIFO at random time intervals up to a specified maximum interval length. The consumer reads each integer from the FIFO at random time intervals which are independent of the producer's intervals. The variable *first* contains the index in the FIFO of the next integer that the consumer will read. The variable *num_elements* contains the total number of spaces occupied by data in the FIFO memory. The parameter *size* is the capacity of the FIFO. We wrote the following assertions to monitor the operation of the FIFO:

```

always(0 ≤ first < size)
always(0 ≤ num_elements ≤ size)
match(in, out)

```

The first two assertions assume that the elements of the FIFO array are indexed from zero to (size-1). The third assertion simply requires that the sequence of numbers observed at the output of the FIFO must match the sequence observed at the input, with an arbitrary delay between the two.

We ran two tests of the model with the three assertions. In the first test, the random time intervals for the producer and consumer were uniformly distributed over the same range. The producer and consumer ran concurrently for the full length of the simulation. As expected, near the end of the simulation, the producer generated integers which the consumer did not have time to read. This left the *match(in, out)* assertion in its *waiting* state at the end of the simulation, causing that assertion to fail. In the second test, the range of random

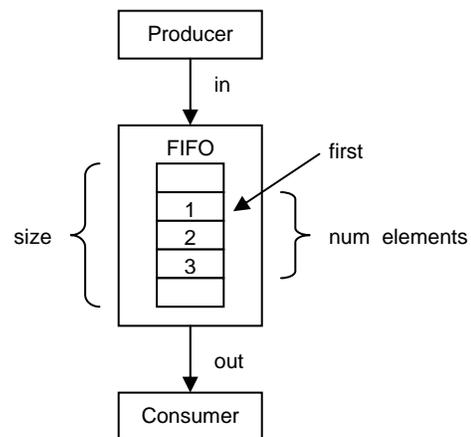


Figure 3. Producer-consumer model.

time intervals for the consumer was reduced to half the range for the producer. The consumer then read integers in half the time on average that the producer took to generate them. This enabled the consumer to empty the FIFO before the end of the simulation, although theoretically it still could have left integers in the FIFO. The match(in, out) assertion confirmed that the consumer read all the integers successfully. The other assertions verified that the *first* and *num_elements* variables remained within their required ranges at all times.

4.2. Ornamental Garden model

The Ornamental Garden, shown in Figure 4, is a design example given in [2] to illustrate the importance of controlling access to shared resources in any system. The garden is a fictitious tourist attraction with entrances on the west and east sides. A turnstile at each entrance limits access to the garden to 20 visitors at a time. A shared main counter keeps a running total of the number of visitors that have entered from both sides. When a turnstile admits a visitor, the turnstile increments its own counter (“West count” or “East count”). The turnstile then reads the count value from the main counter, increments it and updates the main counter with the new value. To simplify the example, the system does not count visitors as they leave. The main counter can maintain a correct total count only when the system includes a mutual exclusion mechanism (*mutex* for short) to control which turnstile can access the main counter at any given time. Without the mutex, the turnstiles may both update the main counter at the same time, causing the total count to be incorrect.

To verify the operation of the main counter, we needed to define and monitor a set of conditions which were more complex than the conditions for the producer-consumer model. We defined the conditions listed in Table 6. Using the abbreviations for these conditions, we wrote the following assertions:

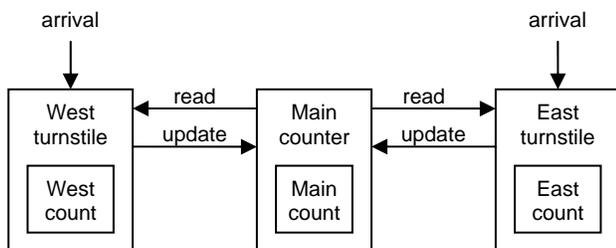


Figure 4. Ornamental Garden model.

Table 6. Conditions for Ornamental Garden.

Condition (Abbreviation)	Description
West Arrival (WA)	West turnstile receives a visitor and updates West count
West Update (WU)	West turnstile updates Main count
East Arrival (EA)	East turnstile receives a visitor and updates East count
East Update (EU)	East turnstile updates Main count
Main Count Updated (MCU)	Main count == West count + East count
Main Count Not Updated (MCNU)	Main count != West count + East count

```

always(((WU || EU) && MCU) ||
        ((WA || EA) && MCNU))
within(WA, WU, 1)
within(EA, EU, 1)
  
```

The first assertion relates the updating of the turnstile and main counters to the values present in the counters. When a turnstile receives an arrival (WA or EA), it increments its own counter first and updates the main counter after a delay. During this delay, the main count is not equal to the sum of the turnstile counts (MCNU). Then, when the turnstile updates the main counter (WU or EU), the main count should become equal to the sum of the turnstile counts (MCU). The second and third assertions require that an arrival at a turnstile be followed on the next cycle by an update of the main counter from that turnstile. After an arrival at one turnstile, the other turnstile should not be allowed to read the main counter until the first turnstile has updated the main count value.

We simulated this model with and without a mutex. In the version without a mutex, the *within* assertions easily confirmed that both turnstiles were being allowed to read the main counter before either one could update the main count. The *always* assertion verified that the main count was incorrect after almost every update because of the unrestricted access to the main counter. In the version with a mutex, the *within* assertions showed that each arrival at a turnstile was immediately followed by an update from the same turnstile. The *always* assertion confirmed that the main count value was correct after each update.

5. Discussion

Assertion-based verification is already recognized in industry as an effective technique for clarifying design intent, increasing test coverage, facilitating integration and enhancing reuse. System-level design desperately needs the benefits of assertions, particularly when the

design process includes integration and reuse of existing intellectual property (IP) design blocks. As Anderson [3] points out, the increasing reuse of design code necessitates more thorough functional verification to ensure quality, which in turn tends to increase the portion of development time spent on verification. When only the outputs of a system are analyzed after a simulation, the only way to make testing more comprehensive is to increase the number of functional tests. But when the activity at interfaces within a design is captured, each simulation can reveal the actual behavior of the system in greater detail. Assertions can then verify correct internal behavior of the system, which increases the level of confidence in the design, and errors become easier to detect and trace to their sources. Assertions can also ease integration by specifying how each interface of an IP block should function. This helps avoid integration errors due to misinterpretation of functionality. Scherer [4] adds that assertions can indicate the functional coverage of a test, if the testbench notes any combination of conditions that is not tested or that causes an assertion to fail. Other experts on verification methodology also cited many of these benefits of assertion-based verification at a panel discussion hosted by the Accellera standards group last year [5].

Our assertion class structure provides the flexibility and scalability needed to capture numerous types of relationships between different conditions in a system. The occurrence handlers can monitor any condition that can be described by a numeric or Boolean-valued expression. The base assertion class watches for occurrences of the triggering condition and determines the overall state of a given assertion. A class derived from the base class can track any number of other conditions and relate them to the triggering condition by means of a subcondition state machine. Since the class structure is defined in C++, our assertions are directly compatible with C++ software as well as SystemC [6] hardware designs. The assertion checker can also conceivably post-process text-based trace files generated by simulations in any design language. The structure therefore shows promise in a wide range of applications.

By contrast, Habibi and Tahar [7] have extended SystemC using SystemVerilog assertions. This approach requires the user to learn how to write SystemVerilog assertions, whereas our C++ structure avoids the need to learn a separate assertion language. Chang et al [8] have added a temporal assertion capability to Verilog using the programming language interface (PLI) which many Verilog simulators support. However, we know of no similar effort to extend SystemC using assertions based on C++.

At present, in our ABV methodology, a verification engineer must write the assertion checker for a design manually and instantiate objects of the proper assertion

classes within the checker. This requires the verification engineer to understand the behavior of the design variables being monitored and the intended relationships between them. In the future, we plan to develop techniques for automated generation of assertion checkers. These techniques would extract requirements from an abstract system model written in a graphical language such as Unified Modeling Language (UML) [9] and instantiate objects of our assertion classes in the checker based on such requirements.

6. Conclusions

We have presented a very flexible object-oriented class structure for defining assertions in system-level simulations. Assertions are widely recognized as an effective tool for automating the analysis of test results and pinpointing the source of design failures. Since there are currently few provisions for assertion-based verification of system-level designs, our structure should help to fill a gap in the verification process.

7. References

- [1] *Sugar Formal Property Language Reference Manual* (draft), version 0.8, Sept. 12, 2002. www.haifa.il.ibm.com/projects/verification/sugar/fp_lrm_0912.pdf.
- [2] A. Burns and G. L. Davies, *Concurrent Programming*, Addison-Wesley, Boston, 1993.
- [3] T. L. Anderson, "Design Intents Raise Verification Level," *EE Times*, June 15, 2001.
- [4] A. S. Scherer, "Integrating IP with Assertion-Based Verification," *EE Times*, May 29, 2002.
- [5] R. Goering, "Methodology Sought for Assertion-Based Verification," *EE Times*, October 6, 2004.
- [6] T. Grotker et al, *SystemC for System Designs*, Kluwer Academic Publishers, Boston, 2002.
- [7] A. Habibi and S. Tahar, "On the Extension of SystemC by SystemVerilog Assertions," *Canadian Conf. on Electrical and Computer Engineering*, 4: 1869 - 1872, 2004.
- [8] K. H. Chang, W. T. Tu, Y. J. Yeh and S. Y. Kuo, "A Temporal Assertion Extension to Verilog," *Lecture Notes in Computer Science*, 3299: 499-504, 2004.
- [9] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, 2nd ed., Addison-Wesley, Boston, 2004.