SUPERLOG® Design Assertion Subset

March 19~~February 27~~, 2002

Revision 1.6~~5~~

Copyright © 2001-2 Co-Design Automation Inc.

This document has been submitted to Accellera under the agreed terms of the "Co-Design Accellera SUPERLOG DAS Donation Agreement" of 27th February 2002. Usage is only permitted under the terms of that agreement.

Do not copy, fax, reproduce, or distribute without written permission.

Introduction 3
Procedural Assertions 3
  Syntax 4
  Immediate Assertions .... 4~~5~~
  Strobed Assertions .... 6~~5~~
  Clocked Immediate Assertions .... 7~~6~~
  Clocked Strobed Assertions .... 9~~7~~
  More Expression Sequences .... 9~~7~~
  Antecedent and Consequent .... 10~~8~~
  Resetting Assertions .... 10~~8~~
Controlling Assertions .... 11~~9~~
  Controlling the Steps .... 12~~9~~
Further Enhancements .... 12~~9~~

© 2001-2 Co-Design Automation, Inc. Accellera Copy

Change History

Version 1.6 of this document has been updated from version 1.5 to reflect the results of the Accellera SystemVerilog Assertions committee meetings held on

Friday, 3/8/02 (phone)
Wednesday, 3/13/02 (phone and in-person at Verplex in Milpitas, CA)

Introduction

An assertion is a statement that a property must be true. There are two kinds of assertions: concurrent assertions which state that the property must be always be true, e.g. throughout a simulation, and procedural assertions which are incorporated in procedural code and apply only for a limited time or under limited conditions.

There are various applications of assertions. They can be included in the design, to document the assumptions made by the designer and to facilitate "white box" testing. They can be outside the design, either in a testbench to check the response of the design to the stimulus, or to control a tool such as a stimulus generator or a model checker.

Concurrent assertions can be coded as modules in a library, but this limits the complexity of the property that can be expressed easily. It is more difficult to code procedural assertions as a library of tasks in Verilog, because events cannot be arguments, each assertion may need static data, and tasks block.

Procedural Assertions

SUPERLOG Design Assertions Subset provides the following kinds of procedural assertions:

Immediate assertions
Strobed assertions
Clocked immediate assertions
Clocked strobed assertions

These are statements in an initial or always block, or in a task. Immediate assertions can also be in a function.

© 2001-2 Co-Design Automation, Inc. Accellera Copy

Syntax

```
<proc_assertion>::= [<identifier> ':'] <immediate_assert>
                  | [<identifier> ':'] <strobed_assert>
                  | [<identifier> ':'] <clocked_immediate_assert>
                  | [<identifier> ':'] <clocked_strobed_assert>

<immediate_assert>::= 'assert' '('<expression>')'
                      <statement_or_null> //pass
                      ['else' <statement_or_null>] //fail

<strobed_assert>::=   'assert_strobe' '('<expression>')'
                      <restricted_statement_or_null> //pass
                      ['else' <restricted_statement_or_null>] //fail

<clocked_immediate_assert> ::= 'assert' [<event_control>] [<reset>]
                      '(' <formula_expr> ')' <step_control>
                                     <statement_or_null> //pass
                      ['else' <statement_or_null>] //fail
                  | 'assert' '(' <expr_sequence> ')' <step_control>
                      <statement_or_null> //pass
                      ['else' <statement_or_null>] //fail

<clocked_strobed_assert> ::= 'assert_strobe' [<event_control>] [<reset>]
                      '(' <formula_expr> ')' <step_control
                                <restricted_statement_or_null>
                      ['else' <restricted_statement_or_null>] //fail
                  | 'assert_strobe' '(' <expr_sequence> ')' <step_control>
                      < restricted_statement_or_null> //pass
                      ['else' < restricted_statement_or_null>] //fail

<formula_expr> ::=  <expr_sequence>
                  | <expr_sequence> 'triggers' <formula_expr>

<expr_sequence> ::=  <expression>
                  | '[' <constant_expression> ']' // skip n steps
                  | <range> // skip m to n steps
                  | <expr_sequence> ';' <expr_sequence> // sequence
                  | <expr_sequence> '*' '['<constant_expression>']'
                                                    //repetition
                  | <expr_sequence> '*' <range> // bounded repetition
                  | '(' <expr_sequence> ')'

<range> ::=         '[' <constant_expression> ':' <constant_expression> ']'

<step_control> ::=  '@@' <name>
                  | '@@' '(' <event_expressions> ')'

<reset> ::=         'accept' '(' <expression ')' // sync
                  | 'accept' <event_control>  // async
                  | 'reject' '(' <expression ')' // sync
                  | 'reject' <event_control>  // async
```


© 2001-2 Co-Design Automation, Inc. Accellera Copy

Immediate Assertions

The immediate assert statement is a test of an expression performed when the statement is executed in the procedural code. The expression ~~which~~ is treated as a condition like in an if statement:

[<identifier> ':'] **assert** **(**<expression>**)** [<pass_statement>] [**else** <fail_statement>]

The pass statement is executed if the assertion succeeds, i.e. the expression evaluates to 'true'. As with the 'if' statement, if the expression evaluates to 'X', 'Z' or '0', then the assertion fails.~~.~~ The~~is~~ pass statement may, for example, record the number of successes for a coverage log, but ~~is normally~~may be omitted altogether. If the pass statement is omitted, then no action is taken if the assert expression is true. The fail statement is executed if the assertion fails (i.e. the expression ~~is false (0)),~~does not evaluate to a known, non-zero value) and can be omitted ~~and is also normally omitted~~. The optional assertion label (identifier and colon) creates a notional named block around the assertion statement (or any other ~~SUPERLOG~~ SystemVerilog statement) and can be displayed using the %m format code.

assert_foo : **assert** (foo) $display("%m passed"); **else** $display("%m failed");

If the **else** is omitted a default error message is written if the assertion fails.~~,~~ The contents of the message may be tool-specific. For simulation tools, it is recommended that the message include basic information about the assertion statement and when it failed to facilitate debug, such as

Run-time Warning: myfile.slg:7 Assertion assert_foo failed at time 105

If the **else** is present, the default message is still printed when the assertion fails. Additional information may be displayed after the default message by including ~~no message is written unless the statement contains a~~ a system task to do so, such as $fatal, $error, $warning, $info. These system tasks indicate the severity, and can contain additional information in the format of $display.

assert (foo); **else** $fatal("finish simulation because foo is false");

$fatal displays a Run-time Fatal, which terminates the simulation with an error code. The first argument passed to $fatal shall be consistent with the argument to $finish.

$error displays a Run-time Error, which cannot be turned off. This shall be the default.
$warning displays a Run-time Warning, which can be turned off. ~~This should be the default.~~
$info: displays an information message, which can be turned off by "quiet mode".
~~The simulator could count the number of errors and warnings.~~

```
<assert_msg> ::= <fatal_func> | <msg_func>;

<fatal_func> ::= $fatal'('<finish_num>',' // finish arg
                        [<format_string>]','[<expr>...]')';
;

<msg_func> ::= <msg_name>'('[<format_string>]','[<expr>...]')'';'

<msg_name> ::= $error | $warning | $info
```

The display of the default message and additional messages of specific types may be controlled by a command-line option.

Since the fail statement, like the pass statement, is any legal SystemVerilog procedural statement, it~~The **else** statement~~ may also be used to signal a failure to another part of the testbench:

assert (myfunc(a,b)) count1 = count + 1; **else** ->event1;
assert (y == 0); **else** flag = 1;

© 2001-2 Co-Design Automation, Inc. Accellera Copy

The assert statement serves as guidance to non-simulation tools that the condition should be true. The second statement above is equivalent to:

```
if ( y!=0) begin flag = 1; end
```

Strobed Assertions

If an immediate assertion is in code triggered by a timing control that happens at the same time as a blocking assignment to the data being tested, there is a risk of the wrong value being sampled. For example:

```
always @(posedge clock) a = a + 1; // blocking assignment
always @(posedge clock) begin
   ....
   assert (a < b);
end
```

This can be solved by using a strobed assertion, which waits in the background until the end of the time slot, like the $strobe system task.

```
always @(posedge clock) begin
   ....
   cas:assert_strobe (a < b);
end
```

Strobed assertions can have pass or fail statements like immediate assertions. However, they are restricted to another assertion statement, a system task call, a statement preceded by a delay control or an event control, or sequential block containing them. This is because the statement happens after the assertion is evaluated, at the end of the time slot, and hence they must not create more events at that time slot or change values. Statements which cause additional events to occur at the current time shall be an error.

The example below illustrates the effect of blocking and non-blocking assignments on immediate and strobed assertions. The immediate assertions are like $display statements and the strobed assertions are like $strobe statements:

```
module test;
reg [3:0] a=0; c=0, d=0;
reg clk = 0;
wire b;

initial begin
   #10 clk = 1;
   forever (#5 clk = !clk); // posedge clk at 10,20,30,40...
end

assign b = a+1;

always @(posedge clk) begin
   a1: assert(c<3); // fails at time 40
   c = c+1;
   a2: assert(c<3); // fails at time 30
   a <= a+1;
   a3: assert(a<3); // fails at time 40
   a4: assert(b<3); // fails at time 40
   a5: assert_strobe(a<3); // fails at time 30
   a6: assert_strobe(b<3); // fails at time 30
end
```


© 2001-2 Co-Design Automation, Inc. Accellera Copy

```
always @(a) begin // models transient behavior on comb. nets
   d = a+2; // spikes to 2 at t0, 3 at t10, 4 at t20
   assert(d<3); // fails at time 10
   d = d-1; // settles to 1 at t0, 2 at t10, 3 at t20
   assert(d<3); // fails at time 20
end

always @(d) assert_strobe (d<3); // fails at time 20

endmodule
```

Clocked Immediate Assertions

~~A sequence of Boolean expressions can be asserted, with the steps (i.e. the timing between them) provided by the enclosing always block. This is called an implicit clock:~~

~~**always** @(**posedge** clock) **begin**~~
~~....~~
~~**assert** (a; b; c); // a is sampled now, b at next posedge, c at following posedge~~
~~....~~
~~**end**~~

A sequence of expressions can be asserted by specifying a *sequential regular expression* in the assert statement, along with a step control to specify the timing between evaluations of the sequential regular expression.

```
always @(posedge clock or negedge reset)
 assert (a; b; c) @@ (posedge clock); // note the @@ token to distinguish the step control
                                      // from the pass statement
```

A sequential regular expression is a semicolon-delimited list of expressions. The first expression in the list is evaluated immediately when the assert statement is executed. The other subsequent expressions are evaluated one at a time on successive occurrences of the step control event expression. In the above example, the expression 'a' is evaluated immediately, just as for an immediate assertion, with 'b' being evaluated on the next posedge clock and 'c' evaluated on the posedge after that. The immediate clocked assertion expressions are evaluated among the first active events in the timestep in which the explicit event is triggered.

~~Note that there must be a unique event control to use this construct. Always blocks with various event controls make the steps ambiguous. In such a case the steps should be explicitly given using a step control after the sequence of the assertion. For example, the above assertion is equivalent to:~~

~~**assert** (a; b; c) @@ (**posedge** clock); // note the @@ token to distinguish it from the pass statement~~

The '@@' token is introduced to distinguish the step control from an ordinary event control at the start of the pass statement. Consider the following:

```
always @(posedge clock)
  assert (a)
    @(posedge clock) // This is an event control in the pass statement
      $display("Hello at time %t",$time);
```

In this example, the "@(posedge clock)" in the pass statement causes the display action to occur on the next posedge clock after the assertion succeeds. Therefore, a new token is required to distinguish the assertion sequence step control from the pass statement. Specifying an explicit step control for a sequence makes it possible to use clocked assertions in combinational always blocks:

```
always @(foo,bar)
   assert (a;b;c) @@(posedge clk); // look for a when foo or bar change,
```


© 2001-2 Co-Design Automation, Inc. Accellera Copy

```
                                    // then look for b on next posedge clk
```

The assertion

```
        always @(posedge clk)
          a1: assert (a;b;c) @@(posedge clk);
```

is nearly equivalent to

```
        always @(posedge clk)
           a2: assert (a) @@(posedge clk) // no semicolon
                @(posedge clk) a3: assert (b;c) @@(posedge clk);
```

The assertion **a1** and the combination of assertions **a2** and **a3** both evaluate the same sequence, but splitting the sequence between two assertions allows the user to execute different fail statements depending on where the assertion fails. For example, it may be deemed an error if 'a' is not true now, but only a warning if 'b' is not true on the next clock edge. Another way of expressing the same behavior is

```
        always @(posedge clk)
          a4: assert (a)  // no semicolon
            a5: assert(1;b;c) @@(posedge clk); // b evaluated on next posedge clk, then c on next posedge clk
                else $warning("b or c not found"); // else clause for a5
             else $error("a not found immediately"); // else clause for a4
```

In this example, assertion a5 evaluates the first expression in the sequence immediately. The value '1' is always true, so on the next posedge of clk, it will evaluate 'b'. This is a convenient notation for specifying a sequence that begins on the next clock.

~~The steps can also be explicitly given using an event control at the beginning of the assertion. This delays the first expression in the sequence until the event occurs. For example, the above assertion is equivalent to:~~

```
        assert (a) // no semicolon
          assert @(posedge clock) (b; c); // pass statement executes if first assertion OK
```

~~This is also equivalent to:~~

```
        assert  (a) // no semicolon
          assert  @(posedge clock) (b) // pass statement, if first one OK
            assert  @(posedge clock) (c); // pass statement, if second one OK
```

Note that to avoid races, the variables read in clocked immediate assertions should be written by non-blocking assignments.

A clocked assertion can be triggered twice at the same ~~clock cycle~~timestep, and the first expression in the sequence will be re-evaluated ~~if it is not delayed~~. This could happen in a zero-delay loop, but is not recommended practice.

A clocked assertion may also be re-triggered at the next ~~clock cycle~~timestep, before the sequence has expired. Any single assertion shall only be spawned once at a particular timestep.

If the step control event occurs multiple times at the same timestep the evaluation shall not progress through the sequence. Instead, the current expression in the sequence is re-evaluated. Consider:

```
        module top;
        reg clk = 0;
        reg a,b,c;

        initial begin
          #10 clk = 1;
```


© 2001-2 Co-Design Automation, Inc. Accellera Copy

```
        forever begin
          clk = 0;
          clk = 1; // 2 posedges clk at 10,20,30,40...
          #5 clk = 0;
          #5 clk = 1;
        end
      end

      always @(posedge clk)
        assert(a;b;c) @@(posedge clk); // 'a' is evaluated only once at 10, 'b' once at 20, 'c' once at 30
```

Note that the step control expression may be any valid event expression in SystemVerilog. The following assertions all use valid step control expressions:

```
      bit clk;
      event ev1;

      always @(posedge clk or negedge reset) begin
        assert (a;b;c) @@(negedge clk); // sequence sampled on negedge clk
        assert (a;b;c) @@(clk); // sequence sampled on any edge of clk
        assert (a;b;c) @@(ev1); // sequence sampled when event ev1 fires
      end
```

This flexibility also allows nested assertions to use different clocks:

```
      always @(posedge clk) begin
       assert (a;b) @@(posedge clk) // on posedge clk
         assert (1;c;d) @@(negedge clk); // look for c and d on negedge clk
       assert (e;f) @@(posedge clk2)
         assert (1;g;h) @@(ev1);
       end
```

Clocked Strobed Assertions

Another way to avoid races is to use the same constructs with strobed assertions:

```
      always @(posedge clock) begin
        ....
        assert_strobe (a; b; c) @@(posedge clk); // a is sampled at end of current time step, b at end of next
clock time step
        ....
      end
```

~~An explicit step control can be used after the sequence:~~

~~**assert_strobe** (a;b;c) @@ (**posedge** clock);~~

This is also equivalent to

```
      assert_strobe (a) // no semicolon
        assert_strobe @(posedge clock) (1;b; c) @@(posedge clk); // pass statement, if first one OK
```

This is also equivalent to:

```
      assert_strobe  (a) // no semicolon
        assert_strobe  @(posedge clock) (1;b) @@(posedge clk) // pass statement, if first one OK
          a6: assert_strobe  @(posedge clock) (1;c) @@(posedge clk); // pass statement, if second one OK
```


© 2001-2 Co-Design Automation, Inc. Accellera Copy

Note that assertion a6 will not evaluate until the successful completion of (a;b), which is why only a single '1' is required to specify that c be evaluated on the next posedge clk.

Strobed assertions with explicit clocks can be used in always blocks that model combinational logic and therefore do not have a clock and are subject to glitches.

Even if a clocked strobed assertion is triggered twice at the same clock cycle, it only executes once at the end of the time step. A clocked strobed assertion may be re-triggered at the next clock cycle, before the sequence has expired.

More Expression Sequences

A number of steps can be skipped either by writing expressions which are always true:

assert (a;1;1;c) @@(**posedge** clk); // two steps between a and c

or by using the notation [n] to count the number of steps:

assert (a;[2];c) @@(**posedge** clk); // two steps between a and c
assert (a;[1];[1];c) @@(**posedge** clk); // two steps between a and c

Note that in [n], the n must be a literal or a constant expression. The number of steps to be skipped may also be expressed using [min:max], where the minimum number of steps may be greater than or equal to zero. Both min and max must be a literal or constant expression.

assert (a;[0:10];b) @@(**posedge** clk); // b occurs between the next and 11th clock edges, inclusive

If an expression must be repeated a defined number of times, this can be expressed with a trailing *[n]. If it can be repeated a minimum or maximum number of times, this can be expressed with a trailing *[min : max]. These repetition counts must also be literals or constant expressions:

assert (a; b)*[5]; // a;b;a;b;a;b;a;b;a;b
assert (a; b)*[1:2]; // (a;b) or (a;b;a;b)

Note that (a*[0:3];b;c) is equivalent to (b;c) or (a;b;c) or (a;a;b;c) or (a;a;a;b;c). This means that a sequence a;ab;a;b;c; will pass. The expression sequence is not equivalent to ((a && !b)* [0:3];b;c), which would fail the same sequence.

Antecedent and Consequent

An assertion is usually only applicable under a certain condition, called an antecedent. One way of specifying a condition is to use procedural code - **if** or **case**. For example:

if (a) **assert** (1;b;c) @@ (posedge clk~~1~~); // only apply assertion if a is true

Another way is to use the **triggers** keyword in the assertion itself. The following is equivalent to the example above:

assert (a **triggers** (1;b;c)) @@ (**posedge** clk~~1~~); // only check b;c if a is true

If a is false when the assertion is applied, the assertion is aborted. It neither passes nor fails. If the antecedent a is true, the rest of the assertion, called the consequent, is applied.

The benefit of using the second form is that the antecedent can be a sequence. For example:

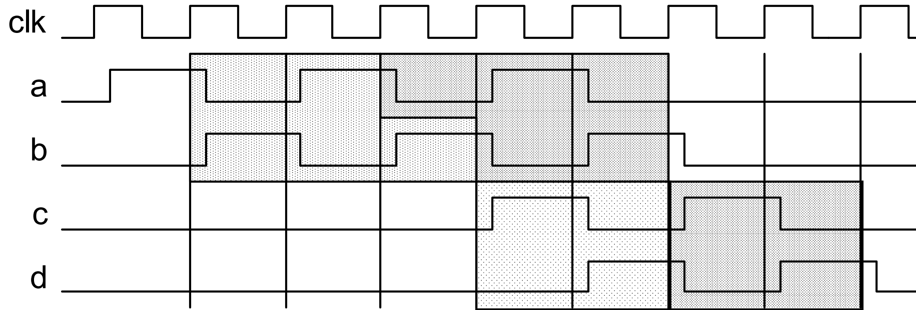
assert ((a;b) triggers (1;c)) @@ (**posedge** clk~~1~~); // only check c if a is true then b is true

This technique can conveniently used for assertions that are not embedded in procedural code, but are stand-alone concurrent statements

always @(**posedge** clock)
  assert_strobe ((a;b;a;b) **triggers** (c;d)) @@(**posedge** clk);~~// implicit clock~~

© 2001-2 Co-Design Automation, Inc. Accellera Copy

Note that the antecedent (a;b;a;b) is retriggerable, so that the input sequence ababab requires two instances of the consequent (c;d) i.e. cdcd.



Resetting Assertions

A named assertion can be disabled like any other named SUPERLOG block. If this is done before the expression sequence has finished, it means that neither the pass statement nor the fail statement is executed.

disable cas;

Note that if the disable is applied at the same simulation time step as the last clock step of a sequence, there is a race in the case of an immediate assertion, but a strobed assertion is always disabled.

An alternative way to reset the assertions is to use the **accept** or **reject** keywords. Acceptance means that the assertion succeeds, and the pass statement is executed. Rejection means that the assertion fails and the fail statement is executed. These keywords can be used either with a Boolean expression for synchronous reset, or with an event control for asynchronous reset. The following examples use an explicit clock:

a7: assert ~~@(**posedge** clock)~~ **reject** (!busy) (a;b;c) @@(**posedge** clk); // busy must be 1
assert ~~@(**posedge** clock)~~ **accept** (sync_reset) (a;b;c) @@(**posedge** clk);
assert ~~@(**posedge** clock)~~ **reject** @(**negedge** busy) (a;b;c) @@(**posedge** clk);
assert ~~@(**posedge** clock)~~ **accept** @(**posedge** async_reset) (a;b;c) @@(**posedge** clk);
assert_strobe reject (!busy) (a;b;c) @@(**posedge** clock); // busy must be 1
assert_strobe accept (sync_reset) (a;b;c) @@(**posedge** clock);
assert_strobe reject @(**negedge** busy) (a;b;c) @@(**posedge** clock);
assert_strobe accept @(**posedge** async_reset) (a;b;c) @@(**posedge** clock);

The effect of a synchronous **reject** expression is to logically and the inverse of the reject expression with each element in the sequence. Therefore, assertion a7 above is equivalent to

assert (a **&&** busy; b **&&** busy; c **&&** busy) @@(**posedge** clk);

If an asynchronous reset occurs simultaneously with the last clock step, there is a race in the case of an immediate assertion and the reset action always happens in the case of a strobed assertion.

Other examples of synchronous accept/reject are shown here:

```
always @(posedge clk)
   if (fifoFsmState == READ)
      assert accept(ack) (dataValid * [15]) @@(posedge clk);
```

Once the fifo goes to READ state, the dataValid must be held high for 15 cycles or when the ack goes high, whichever comes first.

© 2001-2 Co-Design Automation, Inc. Accellera Copy

```
always @(arbiterState or request)
  if (arbiterState == GRANT)
    assert reject(~request) ((arbiterState==GRANT)*[5]) @@(posedge clk);
```

Once the arbiter goes to GRANT state, it must stay in that state for at least next 5 cycles during which the request signal cannot be lowered. If request goes low, the assertion fails.

Issue For Discussion:
*If an assertion is named, could it be accepted or rejected explicitly from a different block, similar to **disable**? Example:*

```
always @(posedge clk)
 if(state == READ)
  a8: assert (!as;!ds;rdy*[0:10];!rdy;rdy) @@(posedge clk);

always @(posedge clk)
 case(arbState)
 ...
 UNFAIR:  reject a8;
 endcase
```

Controlling Assertions

System functions are provided to limit assertion checking to part of the design and part of the simulation time.

```
$assertoff (hierarchical_names) // name of module instance or individual assertion
$asserton (hierarchical_names) // name of module instance or individual assertion
```

Assertions are on by default until turned off. If the list of names is empty, it is taken to refer to all assertions.

The effect of turning assertions off is to stop the check and both the pass and fail statements. Assertions already started are not affected. They can be turned off individually by **disable** statements.

Controlling the Steps

To control the stepping of always blocks, SystemVerilog has the following enhancement from Verilog:

```
@(posedge clk iff rst enable == 10)
```

The event expression only triggers if the expression after the **iff** is 1, in this case when rst enable ==10. Note that such an expression is evaluated when clk changes not when rst enable changes. Also note that **iff** has precedence over **or**, so that the **iff** needs to be repeated for each **or**ed event to which it applies.

Since Tthis enhancement feature is a valid event expression, it can be used in an event expression before or after the expression sequence.to control the stepping of an assertion sequence. In effect, this allows a "gated clock" to control the assertion without the user having to declare the gated clock explicitly. Because this could have significant impact on the ability of Formal Verification tools to evaluate the assertion successfully, it is recommended that this construct be used only for simulation.

Further EnhancementsSystem Functions

There is a need for system functions to be able to access the default messages so that they can be manipulated by user code.

© 2001-2 Co-Design Automation, Inc. Accellera Copy

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is "one-hot." The following ~~S~~system functions are included to facilitate such common assertion functionality:

~~as~~ $onehot (<expression>~~a & (a-1) == 0~~) ~~or~~ // only one bit of <expression> is high
$in (member, list) // member appears in the list
~~to assert common properties can be added.~~

Item for Discussion:
What other common actions are done in assertions that should have accompanying system functions?

Concern: Must be implemented in a way that does not impact simulation performance.
Orthogonal to the rest of the document.

© 2001-2 Co-Design Automation, Inc. Accellera Copy