

Checker Constructs: Special Containers for Assertions

Objectives

One design goal for the SVA language is to enable the creation of assertion libraries, collections of higher-level checkers that design and validation engineers can use without having to learn the full syntax of SVA, or that can be used to more concisely instantiate common cases of assertion checking. A typical example of an assertion library is the Open Verification Library (OVL), which contains a set of defined checker modules that describe assertions or bundles of assertions: `assert_always(...)`, `assert_one_hot(...)`, etc.

In addition, formal verification engineers have a need for a convenient way to encapsulate modelling code, which is different from design code in some fundamental ways. In particular, rather than current SystemVerilog data types, formal modellers have some need for “free variables” (described below).

Currently, OVL and similar assertion libraries, and formal modelling code implemented as modules or interfaces, have some major limitations:

- **Lack of context sensitivity:** OVL checker modules can only be used where a module can be instantiated, and cannot be placed inside always blocks to inherit clock/enabling conditions from the context.
- **Confusion between synthesis and checking constructs:** Modules are always assumed to contain design code that is synthesizable into silicon, but we often need checker modelling code to support assertions. The language does not provide any mechanism to group this modeling code and isolate it from the design code. (Some design teams also use specialized methods where assertions are synthesized into emulation or silicon, but even in these cases we want to clearly separate these from real design elements.)
- **Default values for ports are not available:** While parameters can have default values, all ports in a module must be specified. But often we want to allow default values for some ports, such as a clock or reset defaulting to `$inferred_*` values (see Mantis ticket 1674.)
- **Limited and inflexible argument types:** An assertion checker could be made much more powerful by being able to, for example, pass in a sequence rather than a boolean for some ports. But due to the limitations of module interfaces, this is currently not possible.
- **Lack of free variables:** In a formal verification context, it is often desirable to utilize “free variables”, variables which are able to assume an arbitrary value, representing unpredictable input from the environment. Variables suited to formal modeling also have slightly different sets of assignment rules than standard verilog signals, which we want to be able to enforce. Free variables also simplify modeling, since they always use sampled values of non-free variables, and thus avoid race conditions.

To solve these problems, we are introducing a new type of SV container called a *checker*, somewhat similar to a PSL *vunit*. A checker is a hierarchical entity similar to a module, but intended to contain properties along with related modeling code. It is an observer, able to examine its inputs but not able to assign any values which might affect the actual design. This type of construct should enable powerful and general next-generation OVL libraries.

Definition of Checkers

A checker is declared similarly to a module, using the keyword `checker` followed by a name and formal argument list, and ending with the keyword `endchecker`. Formal arguments may be typed or untyped. In a formal argument declaration, the keyword `parameter` is used for any formal argument that is an

elaboration-time constant. Default values may be provided for formal arguments. Checkers may be declared inside modules or interfaces, but nested checkers inside checkers are not permitted.

Here is a simple example:

```
checker ovl_width (clk = $inferred_clock, test_sig, parameter width);
    ...
endchecker : ovl_width // `: ovl_width' is optional
```

Checkers may contain the following elements:

- Properties, sequences, and assertions.
- Free variables (defined in the next section) and their assignments. Assignments may be continuous *assign* statements or nonblocking clocked assignments. The right-hand side of an assignment may contain pure function calls, with no side effects and only internal automatic variables.
- Functions. While procedural statements (if, case, ...) cannot be placed directly in checkers, they can be used in functions that are inside or called from checkers.

Checkers may not include module, interface, or cell declarations or instantiations, and may not contain signal declarations or assignments. Also, a checker may not contain nested definitions of other checkers, though it may instantiate other checkers.

Other important features of checkers include:

- All formal arguments are inputs. A checker cannot modify the values of any of its formal arguments. The allowed types of checker formal arguments are constants, strings, nets, variables, sequences, properties, or events. In general, the semantics of checker formal arguments are similar to those of a property construct.
- Inheritance of design variables from scope of definition: Design variables are visible in a checker if and only if the checker was defined within the same scope. There is no visibility (other than through formal arguments) to local signals when a checker is instantiated or bound.
- Inheritance of clock/enable/disable context from scope of definition: These elements of context are similarly inherited from the scope of the definition. However, if a formal argument is assigned a default value of *\$inferred_clock*, *\$inferred_enable*, or *\$inferred_disable*, that value is inferred from the point of instantiation, as in a normal module. Also, these inherited conditions do not affect local continuous assigns, *initial*, or *always* blocks, and can be overridden by local default clock, enable, or disable declarations.
- Assertions within a checker may contain action blocks, but these cannot write into free variables. (See description of free variables below.)

For scheduling execution of checkers, we can sort all local assignments topologically (see the ‘SAR’ and ‘AAR’ rules described below) and execute each only once in a simulation tick. The main checker code is executed in the observed region, though action blocks of checker assertions are executed in the reactive region, as usual. In the observed region, first continuous assignments to free variables are executed in topological order; then all assertions in the checkers are evaluated; and finally next-state assignments in the checker are executed.

Free Variables

As mentioned above, any variable declared within a checker is a *free variable*, a variable object introduced into SV to support formal modeling. A free variable is declared using the same syntax as a signal declaration outside a checker. Free variables may be assigned using a continuous *assign*, or a nonblocking clocked assignment statement. They may be initialized in an *initial* block, or inline when the variable is declared. They differ from variables in several ways:

- A free variable appears only in checkers, and thus should never be synthesized into silicon.
- If not assigned, a free variable is assumed to take an arbitrary value, rather than assuming its default value. So a free variable can represent unpredictable inputs from the environment or be used to write abstract non-deterministic models.
- A clocking event can be specified as a trigger for a nonblocking assignment. This means that the variable takes on the new value when the event occurs, and preserves its previous value until then.
- A free variable is required to obey the Single Assignment Rule, or SAR. This means that a free variable can only be assigned once during any evaluation phase.
- A free variable is required to obey the Acyclic Assignment Rule, or AAR. This means that a free variable cannot participate in a combinational loop of assignments during a single evaluation phase. Note that if a variable affects the relevant cycle of a sequence that uses `‘.ended’`, `‘.matched’`, or `‘.triggered’` on the right-hand side of an assignment, this rule means it cannot depend on the variable on the left-hand side.
- The RHS of a free variable assignment may contain a call to the new system function `$nondet`, which allows a nondeterministic choice from its arguments.
- Any expression containing free variables may be used only at the right-hand side of free variable assignments or in concurrent assertions. This also means that if there is an external module reference (XMR) on the LHS of an assignment, no free variables are allowed in the RHS. The opposite is not true: the right-hand side of an assignment to a free variable may contain arbitrary expressions (within the rules described above).

In simulation, when the values are nondeterministic, a tool implementation-dependent decision shall be made to use one of the following simulation methods:

- Symbolic simulation, where all possibilities are represented.
- Random values chosen at simulation time.
- Default values for their type used at simulation time.

Free variables can also be used to define clocking events. Let `ev` be an expression containing free variables. We introduce a virtual free variable `v` for this purpose as

```
assign v = ev
```

We say that `ev` happens iff `value(v)` before free variable assignment differs from its value after free variable assignment. `@(posedge ev)` and `@(negedge v)` are defined in a similar manner.

For simulation scheduling, the values of nets and variables used in a free variable assignment are sampled in the preponed region, and then assigned in the observed region. Due to the SAR and AAR rules, a topological sort should always be possible, to order the variables so that each need only be assigned once. Both blocking and non-blocking assignments are scheduled together in the beginning of the observed region as part of the common topological sort.

Assignments of free variables to expressions containing sequences with `‘.ended’`, `‘.matched’`, or `‘.triggered’` are also supported, due to this assignment taking place in the observed region.

Examples

1. Simple checker.

```
// If clock missing, get value from context
checker error_free (efclk = $inferred_clk,
                  test_cond, err_cond);
    default clocking efclk;
    al: assert property (test_cond |-> !err_cond);
endchecker
```

2. Testing a complex condition with above simple checker, by passing sequences and properties.

```
property p_bad; ... endproperty

// Can do this with checkers, but not modules
// Check: foo followed by bar does not cause p_bad.
error_free efirst1(
    .test_cond(foo ##1 bar), .err_cond(p_bad));
```

3. Context inference example.

```
default disable my_disable;
always @(posedge clk) begin
    ...
    // checker will inherit posedge clk, my_disable
    mycheck mycheck1(.a(a), .b(b));
    ...
end
...
checker mycheck(a,b);
...
    // p1 inherits clock and disable from calling context
    p1: assert property (a);
    always @(posedge clk2)
        // No inferences here, since inside new always block.
        p2: assert property (b);
...
endchecker
```

4. Checker with parameter.

```
// Checker says that test_sig must be held at least <width> cycles,
// with <width> available at compile time
checker ovl_width (clk = $inferred_clock,
    test_sig, parameter width);

    int t = 0;
    bit checking = 0;
    always @(clk) begin
        t<= (!test_sig) ? 0 : t+1;
        checking <= (checking | test_sig);
        // clock 'clk' inherited
        ovl_width_al:
            assert always (
                (checking & !test_sig) |-> (t >= width));
    end
endchecker : ovl_width // ': ovl_width' is optional
```

5. Checker instantiation in a loop.

```
always @(posedge clk)
begin
    // 15 checker instantiations of the form B1[i].B2[j].mycheck1
    for (int i = 0; i < 5; i++) begin : B1
        for (int j = i; j < 5; j++) begin : B2
```

```

        mycheck mycheck1 (...);
    end
end
end

```

6. Checker instantiating another checker, & referencing variables

```

// This checker contains no properties, but is used
// for modelling
checker resource_counter (parameter NumRsce);
    ...
    bit rsCtr [log2(NumRsce)+1] = NumRsce;
    rsCtr <= ...;
    bit foobar;
endchecker

checker dispenser (...);
    ...
    resource_counter cnt ($bits(grant));
    // LEGAL: uses free var from checker inst
    let ctr = cnt.rsCtr;
FAIR_DISPENSER:
    assert property(ctr>0 && request|=>grant != 0);
    ...

    // ILLEGAL: can't assign free var of
    // instantiated checker
    cnt.foobar <= 1'b0;
endchecker

```

7. A more complex checker: implementation of ovl's assert_window.

```

typedef enum { cover_none, cover_all } coverage_level;
// Uniform interface, no separate interfaces & ports.
// Arguments 'msg' and 'clevel' are compile-time,
// others are runtime inputs.
checker assert_window (
    logic test_expr, start_event, end_event,
    event clk = $default_clock,
    logic reset_n = !$default_reset,
    parameter string msg = "VIOLATION";
    parameter coverage_level clevel = cover_all;
);
default reset !reset_n;
bit window = 0;

// function used here to enclose procedural code.
function bit next_window;
    if (reset_n == 1'b0 ||
        window && end_event == 1'b1)
        return 1'b0;
    if (!window && start_event == 1'b1)
        return 1'b1;
    return window;
endfunction

window <= @(clk) next_window;

```

```

property ASSERT_WINDOW_P;
    @(clk) (start_event && !window) | =>
        (test_expr) [*0:$] ##1 (end_event && test_expr);
endproperty

A_ASSERT_WINDOW_P:
    assert property (ASSERT_WINDOW_P)
        else $error(msg);

if (coverage_level != ovl_cover_none) begin
cover_window_open: cover property (@(clk)
    ( start_event && !window) )
    $display("win_open_covered");
cover_window: cover property (@(clk)
    ((start_event && !window) ##1
    (!end_event && window) [*0:$] ##1
    (end_event && window)) )
    $display("window covered");

endchecker : assert_window

```

8. More legal and illegal uses of free variables

```

checker demo (bit sig1, bit sig2);
    bit x, y, z;

    // ILLEGAL: can't use next value of var in <=
    x <= y & $next_gclk(z);
    // LEGAL: can use next value of RTL signal
    x <= y & $next_gclk(sig);

    // LEGAL: can use clocking event with <=
    y <= @(clk) z;

    // ILLEGAL: violate SAR (single-assign rule)
    bit [3] ctr;
    assign ctr[1:0] = 2'b00;
    assign ctr[2:1] = 2'b00;
    // ILLEGAL: SAR violation starting at cycle 1
    bit [3] ctr;
    assign ctr[1:0] = 2'b11;
    ctr[0] <= 1'b0;
    // ILLEGAL: violates AAR (acyclic-assign rule)
    bit [3] ctr;
    assign ctr = ctr + 1;
    // LEGAL: initial at cycle 0, increment later
    bit [3] ctr = 3'b000;
    ctr <= ctr + 1;
    ...

```

9. Functions and free variables.

```

checker qcheck;
...
// queueCounter is a free variable, since declared in a checker
bit [5] queueCounter = 0;

```

```

queueCounter <= next_counter(queueCounter);

// qc is NOT a free variable within the function
function bit next_counter(bit [5] qc);
  priority if (reset == 0) next_counter = 0;
  else if (validWrite && !validRead)
    next_counter = qc + 1;
  else if (!validWrite && validRead)
    next_counter = qc - 1;
  else next_counter = qc;
endfunction
...
endchecker : qcheck

```

10. Assigning a free variable using \$nondet(type, val1, val2, ...)

```

typedef enum {OP_NOP, OP_READ, OP_WRITE, OP_FUBAR} opcode;
checker mycheck;
  bit [3:0] opcode = OP_NOP;
  // Each cycle one of these opcodes may arrive
  opcode <=
  $nondet(opcode, OP_NOP, OP_READ, OP_WRITE);
  ...

```