Mantis 2088: Allow checker construct to include covergroups

Motivation

Mantis 1900 added a new construct called a checker to encapsulate related verification constructs like assertions into one entity. This proposal extends the checker to allow covergroup constructs to be included. This is required to make the checker more usable in a coverage-driven methodology.

Change Notes

2/21/2008: Modified the proposal in response to feedback from Gord Vreugdenhil. Now, only instances of covergroups are allowed within the checker. Changes no longer are required to the BNF, so I removed these changes.

Note to editor: This proposal relies on the following other language-extending Mantis items:

- 1900: checker
- 2182: VPI Diagrams for checkers
- 2149: Covergroups sample() method with arguments (for last example)

16.18.1 Overview

Note to editor: from Mantis 1900 Text

REPLACE

The checkers may contain concurrent assertions, free variable declarations and assignments, structural procedures, function declarations, **`let`** declarations, sequences and properties, and generate regions. The following procedures are allowed in a checker (see 16.18.4):

WITH

The checkers may contain concurrent assertions, free variable declarations and assignments, structural procedures, function declarations, **`let`** declarations, sequences and properties, covergroup instances, and generate regions. The following procedures are allowed in a checker (see 16.18.4):

16.18.2 Checker declaration

Mantis 1900

REPLACE

A checker body may contain the following elements:

— Declaration of **`let`**, sequences, properties and functions.

— Concurrent assertions.

— Free variables and their assignments (see 16.18.5).

— Default clocking and disable declarations.

— **`initial_check`** and **`always_check`** procedures (see 16.18.4).

— Generate blocks, containing any of the above elements.

WITH

A checker body may contain the following elements:

— Declaration of **let**, sequences, properties and functions.

— Concurrent assertions.

— Covergroup instances.

— Free variables and their assignments (see 16.18.5).

— Default clocking and disable declarations.

— **initial_check** and **always_check** procedures (see 16.18.4).

— Generate blocks, containing any of the above elements.

ADD NEW SECTION Note to editor: Shift subsequent section numbers

16.18.6 Covergroups in checkers

One or more **covergroup** instances (see 18.3) are permitted within a checker. Any **covergroup** declarations, however, must occur external to the checker. These **covergroup** instances shall not appear in any procedural block in the checker. A covergroup will not be able to reference checker variables directly, since it is defined externally to the checker. However, any variable visible in the checker, including checker formal arguments and checker variables may be passed as an argument to the covergroup **new()** constructor. For example:

```
covergroup cg_active (ref logic clk, ref logic active, ref bit active_d1)
         @(posedge clk);
   cp_active : coverpoint active
   {
      bins idle = { 1'b0 };
      bins active = { 1'b1 };
   }

   cp_active_d1 : coverpoint active_d1
   {
      bins idle = { 1'b0 };
      bins active = { 1'b0 };
   }
endgroup

checker my_check(logic clk, active);
   checkvar bit active_d1 = 1'b0;

   always_check @(posedge clk) begin
      active_d1 <= active;
   end

   cg_active cg_active_1 = new(clk, active, active_d1);
endchecker : my_check
```

The clocking event which controls the covergroup sampling may not reference or depend on a checker variable. For example:

```
covergroup cg_active (ref bit clk, ref logic active) @(posedge clk);
. . .
endgroup

checker my_check_w_clock_div(logic clk, active);
   checkvar bit clk_div2 = 1'b0;
   checkvar bit a;

   assign a = active;

   always_check @clk clk_div2 <= !clk_div2;

   // Illegal: covergroup may not reference checker variables in clocking
   //          event.
   cg_active cg_active_1 (clk_div2, active)

endchecker : my_check_w_clock_div
```

A covergroup may also be triggered by a procedural call to its `sample()` method (see 18.8). The following examples show how the `sample()` method may be called from a sequence match item to trigger a covergoup.

```
covergroup cg_op (ref bit [3:0] opcode_d1);
   cp_op : coverpoint opcode_d1;
endgroup:cg_op

checker op_test (logic clk, vld_1, vld_2, logic [3:0] opcode)
   checkvar bit [3:0] opcode_d1;

   always_check @(posedge clk) opcode_d1 <= opcode;

   // Covergroup instance defined here
   cg_op cg_op_1 = new(opcode_d1);


   sequence op_accept;
      @(posedge clk) vld_1 ##1 (vld2, cg_op_1.sample());
   endsequence
   cover property (op_accept);
endchecker
```

In this example, the coverpoint `cp_op_1` refers to the checker variable `opcode_d1` directly. It is triggered by a call to the default `sample()` method from a sequence match item. This function call occurs in the Reactive region, which is after all checker variables have been updated. As a result the covergroup will sample the updated value of the checker variable `opcode_d1`.

Note to editor: The following example relies on Mantis 2149

It is also possible to define a custom `sample()` method for a covergroup (see 18.8.1). An example of this is shown below:

```
covergroup cg_op with function sample(bit [3:0] opcode_d1);
   cp_op : coverpoint opcode_d1;
endgroup:cg_op

checker op_test (logic clk, vld_1, vld_2, logic [3:0] opcode)
   checkvar bit [3:0] opcode_d1;

   always_check @(posedge clk) opcode_d1 <= opcode;
```

```
    cg_op cg_op_1 = new();

    sequence op_accept;
       @(posedge clk) vld_1 ##1 (vld2, cg_op_1.sample(opcode_d1));
    endsequence
    cover property (op_accept);
endchecker
```

In this example, a custom `sample()` method has been defined for the covergroup `cg_op`, and the coverpoint `cp_op` references the formal argument of the custom `sample()` method. This custom method will be called in the Reactive region upon a sequence match, but the sampled value of the sequential checker variable `opcode_d1` will be passed to the `sample()` method. As a result the covergroup will sample the value from the Preponed region.