9/6/2010

Usability Enhancements in Checkers

Proposal Sketch

Dmitry Korchemny

Usability Enhancements in Checkers

Proposal Sketch

This document describes a sketch of the proposal regarding usability enhancements in checkers. The goal of this document is to identify and address the issues in their entirety so that proposals for different features be in agreement.

Enhancements Addressed

In this document the following mantis items are addressed:

- 2093: Checker construct (Mantis 1900) should permit output arguments
- 3034: Allow continuous and blocking assignments in checkers
- 3033: Allow procedural control statements is checkers
- 2743: Allow subroutine_call_statement in a checker
- 3035: More flexible definition of checker argument sampling

These Mantis items cover the following enhancements:

- Continuous assignments in checkers
- New checker procedures
- Blocking assignments in checkers
- Procedural control statements in checkers
- Subroutine calls within checkers
- Output checker arguments

The following issues are covered:

- Definition of sampling
- Value sampling in checkers
- Simulation semantics of continuous and blocking assignments
- Single Assignment Rule (SAR)
- Semantics of checker output arguments and usage models
- Limitations imposed on checker tasks

Motivation

The main goal of the proposed enhancements is to make checker coding more natural and similar to the coding of other SV constructs, especially to module coding. This includes continuous assignments, blocking assignments, and procedural conditional and looping statements. Subroutine calls within

checkers are important for tracing the checker execution, e.g., `$display`, `$monitor`, etc. Subroutines may also be a convenient instrument for bookkeeping, for interaction with a coverage database, etc.

The motivation of introducing checker output arguments is threefold:

- Return assertion status as a bit to a module. This bit can then be fed into a scan latch.
  - **Open question: assertion status is evaluated in the Reactive region. It will miss the clock edge of the scan latch. This is not a problem specific to checkers, but it relates to assertions.**
- Do assertion modeling in a sub-checker. Should be smooth.
- Feed DUT: checker as generator. Checker acts as a TB, also from the point of view of the simulation semantics.

Checker argument sampling definition plays an auxiliary, but an important role to define simulation semantics for the constructs listed above.

Checker Argument Sampling

According to the LRM checker arguments are always sampled when sampling of the argument types makes sense (e.g., for sequences and events sampling does not make sense), except for arguments having explicit **`const'`** cast. The motivation section of Mantis 3035 proposal explains why unconditional sampling of checker arguments is problematic. Our proposal is not to sample checker arguments automatically, but to decide on expression sampling according their context in a checker. Classical example is expressions in concurrent assertions, other examples are discussed below. I.e.,

Checker argument values are not sampled automatically. Checker formal arguments in expressions have the same semantics as their direct substitution (=rewriting) result, and the decision on the expression sampling is made uniformly in checkers for all expressions in a give context, regardless whether these expressions contain checker formal arguments or not.

Definition of Sampling

The LRM definition of a sampled value of an expression is the value of this expression in the Preponed region. This definition has an important drawback, however.

According to the definition of the sampled value function `$past`, `$past` returns the value of expression1 that was sampled in the Preponed region of a particular time step strictly prior to the one in which `$past` is evaluated… This definition does not work for free checker variables because free checker variables are in assertions not sampled. This means that the sampled value functions, both past and future cannot be applied to free checker variables or to expressions containing them, which is hardly acceptable in FV. To support this case, the definition of sampling could be modified as follows:

Sampled value of an expression e is the value returned by system function $sample d(e). System $sample d function is defined recursively on its argument; e.g., $sampled(e1 || e2) = $sampled(e1) || $sampled(e2). If e is a free checker variable then in time t $sampled(e) = e. When sampled value of e in time t for any expression e is referenced from a different simulation time-step then the final sample value in the time-step t is used, i.e., the result of application of function $sampled in the Postponed region of time-step t.

The definition $sampled(e) = e should be applied to sequence.triggered and to automatic variables.

These definitions will allow us to define sampling rules in checkers uniformly.

In Example 1 the free variable `a` in the RHS of the NBA is sampled. According to our definition the sampled value of the free variable is its current value. Since checker variable NBA is performed in the Re-NBA region the new value of `a` will be used as expected. To define the value of `$past(a)` the sampled value of a in the strictly previous clock tick (if it exists). This sampled value is the new value of a in that simulation time-step.

```
checker check(event clk);
   rand bit a;
   bit b;
   always @clk
      b <= a & $past(a);
endchecker : check
```

Example 1. Free variable sampling

Checker Procedures and Sampling

It is proposed to allow the following checker procedures in addition to the existing ones:

- **`always_comb`** procedure to model combinational logic. No timing controls should be allowed in an **`always_comb`** procedure in checkers.
- **`always_ff`** to model sequential logic. Essentially an always procedure currently allowed in checkers acts as **`always_ff`**

It is proposed to define variable sampling in checkers according to the following guidelines:

- Procedure control is not sampled.
- Sampling in concurrent assertions is defined as usual.
- Code in **`always`** and **`always_ff`** procedures uses sampled values.
- Code in all other locations uses non-sampled values.

This definition implies that the RHS of checker variable NBA in **`always`** and **`always_ff`** procedures is sampled. This includes both checker and non-checker variables.

Sampling of non-checker variables is required for consistency with concurrent assertions. For example, assertion `a1`

```
a1: assert property (@clk a |=> b);
```

is equivalent to assertion a2

```
always @clk c <= a;
a2: assert property (@clk c -> b);
```

only when `a` is sampled.

The reason why checker variables should be sampled in the RHS of checker variable NBA is as follows:

Since checker variables are assigned in the Re-NBA region, when the clock is generated in the Reactive region this assignment will be performed before entering the Observed region for the second time. Therefore, if there are two clocks, `clk1` driven from the Active region and `clk2` driven from the Reactive region. If the assignment is done on `clk1` and read on `clk2` in the same time step, then without sampling the value read would have been updated.

The following examples show the application of the above rules to emerging procedural control statements in checkers.

```
always @(posedge clk) begin
   if (en) // en is sampled
      v <= a;
end
```

Example 2. Conditional statement in sequential procedure

```
always_comb begin
   if (en) // en is not sampled
      v = b;
   else
      v = c;
end
```

Example 3, Conditional statement in combinational procedure

Open questions:

- **How well will this definition of sampling work for asynchronous resets of sequential logic? Can these resets be sampled in checkers?**
- **Should variables be sampled in checker covergroups?**
- **Should variables in action blocks of concurrent assertions be sampled?**

Continuous Assignments and always_comb Procedures

Continuous assignments in checkers will be introduced. The RHS of continuous assignment according to the sampling rules in checkers will use non-sampled values of variables.

The simulation semantics of the checker continuous assignment should comply with the rules defined in 10.3.2 The continuous assignment statement:

Assignments on nets or variables shall be continuous and automatic. In other words, whenever an operand in the right-hand expression changes value, the whole right-hand side shall be evaluated. If the new value is different from the previous value, then the new value shall be assigned to the left-hand side.

More specifically, if the RHS of a continuous assignment changes in the Active region set then it should be performed in the Active region; if it changes in the Reactive region set, then it should be performed in the Reactive region. If checker free variables are changed, the corresponding event will be inserted into the Re-Inactive region queue.

Same simulation semantics should be applicable for **always_comb** procedures: the evaluation of the procedure is scheduled in the same region where a change of a signal from the implicit sensitivity list is detected.

Option: Always perform continuous and blocking assignments of checker variables in the Reactive region.

Single Assignment Rule (SAR)

No SAR inside the same structural procedure. SAR for **always_comb** and **always_ff** is already in the LRM. General always procedure in checkers should be treated as now.

Checker Output Arguments

Rewrite semantics should work then the actual arguments are variables. The introduction should be straightforward if the checker variables are explicitly assigned to shadow variables should the need arise, as shown in Example 4.

```
checker outer(req, ack, clk = $inferred_clock);
   default clocking @clk; endclocking
   bit en1, en2;
   inner gen(en1, en2);
   a1: assert property (en1 & req |-> ##[1:5] en2 & gnt);
endchecker : outer

checker inner(output a, b, input clk = $inferred_clock);
   default clocking @clk; endclocking
   rand bit x, y;
   m1: assume #0 ($onehot0({x, y});
   assign {a, b} = {x, y}
endchecker : inner
```

Example 4. Checker output arguments

To eliminate shadow variables syntactical changes are required.

When the actual arguments are nets (Example 5), the semantics should include an implicit continuous assignment of the checker output argument to a net.

```
module outer_module(…);
   wire en1, en2;
   inner gen(en1, en2);
   //…
endcmodule : outer_module
```

Example 5. Nets as actual arguments

In this case we introduce two dummy variables, and the entire code becomes equivalent to:

```
module outer_module(…);
```

```
    wire en1, en2;
    logic en1_dummy, en2_dummy;
    inner gen(en1_dummy, en2_dummy);
    assign en1 = en1_dummy;
    assign en2 = en2_dummy;
    //…
endcmodule : outer_module
```

Checker tasks

It should be legal to call tasks from checkers. The tasks should not affect checker variables. Task arguments are not sampled.