

Using SystemVerilog Assertions for Creating Property-Based Checkers

Eduard Cerny
Synopsys, Inc.
Marlborough, USA
edcerny@synopsys.com

Dmitry Korchemny
Intel Corp.
Haifa, Israel
dmitry.korchemny@intel.com

Angshuman Saha
Synopsys, Inc.
Mountain View, CA, USA
angshuman.saha@synopsys.com

Abstract—The paper describes a method for constructing property-based checkers using SystemVerilog assertions. The checker is defined using a property with default parameters for clock and reset. The property is instantiated in a verification statement using a macro. Macro definitions are also provided for message generation in the fail action block. In this way all checkers have the same format. Small extensions to the SystemVerilog language allow to provide flexible clock inference and checker instantiation in structural and procedural contexts.

I. INTRODUCTION

There is an increasing interest in the application of Assertion-Based and formal verification. Even though the appearance of standard assertion languages such as SystemVerilog (SV) assertions and PSL makes the deployment much easier, the use of a library of canned checkers is still the preferred method for designers and novice users. The Accellera standard OVL SystemVerilog Assertions checker library [2] is a prime example. This library is very powerful in the types of behaviors that can be verified and the coverage information that can be collected. The OVL checkers are encapsulated in Verilog modules or SystemVerilog interfaces and fully parameterized. They are associated with a design the same way as any other Verilog module, by direct instantiation or using the SystemVerilog bind statement. This form of association does not allow inferring clocks and enabling conditions from the instantiation context.

The current OVL version is implemented on top of several different platforms: Verilog, SV, VHDL and PSL and to be consistent these implementations use the common denominator of these languages. While this approach ensures the same behavior in different language environments, it

does not allow using the advantages of more advanced languages such as SVA and PSL. For instance, only safety and co-safety properties may be implemented in Verilog or in VHDL; other liveness properties important for formal verification may not be expressed in these languages (e.g., such properties as “A *ready* bit should be asserted infinitely often” or “A reset signal should be eventually deasserted and remain inactive forever”). But building OVL checkers using the full SVA power reveals important problems that we are going to discuss in the rest of this paper.

One of the central problems is the checker packaging: OVL checkers built on top of SV are packaged into modules or interfaces. While this packaging allows incorporating modeling code into the checker, it limits the checker activation scope and does not support context sensitivity. Therefore an alternative solution proved to be useful for some usage paradigms: to package checkers into properties with an optional macro wrapper. This alternative approach complements the existing one: the checkers implemented as SVA properties are more flexible, but they cannot contain any modeling code. Unfortunately, currently available SV constructs are not sufficient to make the property-based checkers fully satisfactory, but even minimal SV extensions described below allow developing a powerful checker library. Note however that the paper is not describing the exact contents of such a library, but rather describes a methodology for building checkers for a library.

The structure of the paper is as follows: Section II describes the motivation of introducing the property based checkers, Section III describes the checker structure and the SV language extensions necessary for checker implementation, Section IV provides usage examples and Section V contains conclusions and future directions.

II. MOTIVATION

This section describes the motivation for introduction of property-based checkers.

A. Extendibility

The arguments of modules cannot be sequences or properties. Therefore when the checkers are packaged into modules their extendibility is very limited.

Consider the following example: whenever `start` signal is asserted, `ready` bit should be set. To express this assertion the following OVL checker may be used:

```
assert_implication ready_ok
  (clk, reset_n, start, ready);
```

Let's modify the assertion slightly: whenever `start` signal is asserted, `ready` bit should be after two clock ticks. The same OVL checker cannot be easily used anymore. One should use another checker, e.g., `assert_cycle_sequence` for this purpose.

Yet, in both cases the same property-based checker may be used: in the former case `ready` signal would be passed as a property argument, in the latter case a sequence expression `##2 ready`. The latter is not possible with a module-based checker.

B. Uniformity

Module-based checkers have two types of arguments: ports and parameters. E.g., in the following OVL checker

```
assert_delta
#(OVL_ERROR, 16, 0, 8,
OVL_ASSERT, "Error: large delta on y ",
OVL_COVER_ALL)
  valid_smooth (clk, reset_n, y);
```

the window bounds (0..8) are passed as parameters, while the signal `y` itself is a port. The syntax of module-based checker instantiation is not uniform, which leads sometimes to readability problems. In contrast, the property-based checkers have uniform instantiation syntax.

C. Conciseness

Module parameters may have default values, while module ports cannot. Therefore it is not possible to specify default values for clock and reset values for module-based checkers. This is possible for the property-based checkers. Another problem with the module-based checkers is a necessity to explicitly specify the data size. Thus, in the above checker it should be explicitly stated that `y` is 16-bit long. This requirement is redundant because `y`'s size is known at the compile time. The property-based checkers accept untyped arguments and make type inference implicitly.

D. Context sensitivity

Since modules cannot be instantiated within the procedural code, the module-based checkers cannot be written within `always` blocks. Therefore module-based checkers cannot always be written next to statements assigning value to their data, nor can it be placed within a conditional operator to ensure conditional check of the assertion. The clock controlling the `always` block obviously cannot be inferred for module-based checkers, either. The property-based checkers may be instantiated both inside and outside of the procedural code and thus enjoy from SVA context inference.

E. Clock edge support

Events cannot be passed directly to modules, therefore the clock is passed to the module-based checker as a signal. Here is a typical (simplified) implementation of an OVL checker:

```
module assert_implication(logic clk,
  reset_n, antecedent_expr,
  consequent_expr);

  assert property @(posedge clk)
  disable iff (!reset_n) antecedent_expr
  |-> consequent_expr);

endmodule
```

Thus, when passing a clock to this checker, the assertion will be controlled by the rising edge of the clock. It is easy to mimic the control by the falling clock edge – it is enough to pass the negated value of the clock to the checker. Unfortunately, it is impossible to specify in this way the control by both edges of the clock. It is straightforward for the property-based checkers – the clocking event may be directly passed to them as `@(posedge clk)`, `@(negedge clk)` or `@clk`.

F. Efficiency

When an actual argument of a module-based checker has a 2-valued type, while the formal argument has a 4-valued type or vice-versa, there is a penalty during the simulation introduced by the run-time type casting. There is no such penalty for the property-based checkers. The workaround for the module-based checkers is to explicitly pass the argument type as a module parameter, but it is clumsy and the existing checkers do not support this feature.

III. CHECKER ARCHITECTURE

Each checker consists of a parameterized SystemVerilog property (or properties) and a macro definition. The macro definition provides the appropriate instantiation of the property(ies) in `assert`, `cover` or `assume` property statements. The reason for using a macro is that it also allows using the same form of instantiation when the assertion is purely combinational implemented using an *immediate* `assert` statement. The standard arguments of each checker are clock

(clk) and reset (rst); they are always the last two arguments and in that order. There is a default value of 1'b1 assigned to reset and a default symbolic value ``default_clk` assigned to the clock argument. This macro is predefined and currently set to `$inferred_clock`. The role of the default value will be explained shortly.

For example, the following property verifies that whenever `en` is true then at the occurrence of `ev` being true the property `prop` must hold, provided that `rst` is not asserted during the evaluation attempt.

```
`define default_clk $inferred_clock
property next_event(en, ev, prop,
    clk=`default_clk, rst=1'b0);
    @(clk) disable iff(rst)
        en ##0 ev[->1] |-> prop;
endproperty : next_event
```

The associated macro definitions are:

For an assertion,

```
`define ASSERT_NEXT_EVENT \
    (en, ev, prop, clk, rst) \
    assert property \
    (next_event(en, ev, prop, clk, rst))
```

and for an assumption,

```
`define ASSUME_NEXT_EVENT \
    (en, ev, prop, clk, rst) \
    assume property \
    (next_event(en, ev, prop, clk, rst))
```

The property uses the `clk` argument as the sampling clock. It receives it from the macro. The macro processor was modified in such a way that when an actual argument is empty, the macro instance receives an empty string in its place, rather than issuing an error. The processing of property arguments was also modified in such a way that if the `$inferred_clock` actual argument is received then clock inference from the enclosing always block or from default clocking (if any) is carried out. Therefore, if an empty string is received as the actual argument from the macro for the property clock argument, the default value `$inferred_clock` is received by the property which in turn triggers inference of the clocking event.

The default value `$inferred_clock` was defined as a system function so as not to extend the set of existing SystemVerilog keywords; a symbolic name was associated with it to facilitate a change of the keyword if and when the proposal for clock inference such as described in [1] is accepted by IEEE.

Another set of macro definitions is provided for issuing different error messages in the action block of the verification statements. For example,

```
`define FATAL_MSG(t) else $fatal(t)
`define ERR_MSG(t) else $error(t)
`define WARNING_MSG(t) else $warning(t)
`define INFO_MSG(t) else $info(t)
`define COVER_MSG(t) $info(t)
```

An instance of the assertion may appear as follows:

```
default clocking ck @(posedge clk);
endclocking
```

```
my_assert_next:
    `ASSERT_NEXT_EVENT(
        enable, trig, a&&b, , !rst_n)
    `ERR_MSG("failed") ;
```

Notice that the clock argument in the macro instance is omitted, indicating that the clocking event must be inferred from the context (always block or default clocking). Once the macro is expanded and the property inlined and the clock inferred from the default clocking block, it results in the following verification statement:

```
my_assert_next: assert property (
    @(posedge clk) disable iff(!rst_n)
    enable ##0 trig[->1] |-> a&&b
    else $error("failed");
```

Notice that since there is no module encapsulation of the property, the checker can be placed either in the structural context and can infer a clock from a default clocking statement, or it can be placed in a procedural context in which case it can also infer the clock and the enabling condition from the (synchronous) always block. However, unlike the standard OVL checkers, such property-based checkers cannot contain any modeling code. This missing feature would be obtained if and when the proposal [1] for encapsulating verification statements using the checker - `endchecker` unit is accepted. It could then replace the macro encapsulation used in this version of the library.

To achieve the necessary flexibility for checker instantiation and clock inference, the following small enhancements to SystemVerilog were introduced:

- A special default value (`$inferred_clock`) for arguments to properties was added, indicating that inference should take place.
- Optional argument specification is allowed in macro instances in which case an empty string is used as the actual argument.
- Placement of assertions into combinational always blocks is allowed provided that explicit or default clocking is specified. This is useful for inferring enabling conditions even from combinational always blocks.

- Explicit clock specification in the checker can differ from the inferred clock from synchronous always blocks, and the assertion can have several clocks (i.e., be a multi-clock property).

The latter two extensions make the checkers suitable for verifying designs in which the behavior is expressed in combinational always blocks and the synchronous always blocks only assign registers.

The checker structure described so far uses a specific design clock provided by the user. When using checkers with formal or hybrid verification tools there is usually a master system clock to which all the different design clocks are synchronized. Signal updates (if any) happen at the ticks of the system clock. This means, however, that design inputs constrained using "assume" checkers will only respect the checker constraint at the ticks of the design clock. The signals may thus take any value permitted by their type at the system clock ticks that do not coincide with the design clock ticks. Although this may or may not affect the synchronous behavior of the design, it makes visualizing and debugging failures more difficult for the verification engineer. Therefore, in situations where such signal changes are not desired, there must be additional assumptions imposed to stabilize the signals.

We do not propose here specific behaviors that such assumptions should take, but rather explain how a property-based checker can be implemented to operate on system clock. The property samples design signals and design clocks using the system clock. In formal tools, the system clock is usually predefined, but in simulation the user must provide the system clock driver and make sure that all other clocks are synchronous with it. Therefore, the name of the system clock must not be fixed in the property. In our implementation it is defined using a macro because the system clock is global to the entire system. For example:

```
`define SYS_CLK system_clk
```

A simple checker that may be needed to stabilize signals could have the following specification:

Signal sig is stable between every two clk ticks. The value of sig can only change at a clk tick (i.e., when sig is sampled.) The value of sig at tick i should hold until tick i+1 (exclusively).

All signals are sampled using the system clock. Furthermore, design signal values used in the property must be those as sampled before the design clock tick, as sampled by the system clock. Assuming for the moment that a tick of clk is its rising edge, the above property could be written as follows:

```
property stable_between_ticks
    (sig, clk, rst=1'b0);
    @(`SYS_CLK) disable iff(rst)
```

```
    $rose(clk) ##1
    (!$rose(clk)[*1:$])
    |-> $stable(sig) ;
endproperty : stable_between_ticks
```

The associated macro definition for rising edge of clk is

```
`define ASSERT_STABLE_BTWN_TICKS_PEDGE \
    (sig, clk, rst) assert property \
    (stable_between_ticks(sig, clk, rst))
```

A similar macro definition can be provided for sampling by the falling edge of clk, simply by passing the negated value of clk to the same property:

```
`define ASSERT_STABLE_BTWN_TICKS_NEDGE \
    sig, clk, rst) assert property \
    (stable_between_ticks(sig, !clk, rst))
```

For design clocks that sample on both edges, a separate property must be provided because \$rose must be replaced by \$stable as follows:

```
property stable_between_ticks_e
    (sig, clk=$inferred_clock, rst=1'b0);
    @(`SYS_CLK) disable iff(rst)
    !$stable(clk) ##1
    ($stable(clk)[*1:$])
    |-> $stable(sig);
endproperty : stable_between_ticks_e
```

The corresponding macro is

```
`define ASSERT_STABLE_BTWN_TICKS_EDGE \
    (sig, clk, rst) assert property \
    (stable_between_ticks_e(sig, clk, rst))
```

Notice how the macros hide from the user any differences in the usage of the properties. In the next section we illustrate the use of the checkers on a small but complete example.

IV. EXAMPLE

The following FIFO example shows how the property-based checkers can be used inside a combinational always block. This is particularly useful in the coding style where the behavior is described using combinational always_comb blocks. The state variables are assigned in synchronous always_ff blocks using their next-state values computed in the combination always_comb. The assertions infer the enabling condition which is added (&&) in the antecedent of the implication in the property (see SV LRM [3].) The same checkers can also be instantiated inside synchronous always blocks for clock and enabling condition inference from the always block, and in structural context where only default clock inference can be made if the clock is not explicitly specified. The checkers used in the example are the following:

```

`define ASSERT_STABLE_POSEDGE \
(sig, start_ev, end_ev, clk, rst) \
assert property(stable \
(sig, start_ev, end_ev, clk, rst))

property stable(sig, start_ev, end_ev,
clk, rst=1'b0);
@(`SYS_CLK) disable iff(rst)
$rose(clk) && $past(start_ev)
##1
(!($rose(clk) && $past(end_ev) &&
($past(sig)==$past(sig,2)))[*1:$])
|->
($past(sig)==$past(sig,2));
endproperty : stable

`define ASSERT_TRIGGER \
(trig, prop, clk, rst) \
assert property(trigger \
(trig, prop, clk, rst))

property trigger(trig, prop,
clk=`default_clk, rst=1'b0);
@(clk) disable iff(rst)
trig |-> prop;
endproperty : trigger

`define ASSERT_NEVER \
(prop, clk, rst) \
assert property(never \
(prop, clk, rst))

property never(prop,
clk=`default_clk, rst=1'b0);
@(clk) disable iff(rst) not(prop);
endproperty : never

```

The checkers are then instantiated as shown next:

```

`define SYS_CLK sysclk
bit sysclk ;
`include "property_checkers.sv"
module fifo ;

`include "sysclk.v"
`include "reg.defines"
`include "param.defines"

// read sm
// two cycles to read data
always_comb begin
rd_mem = 0 ;
rd_nxtst = IDLE ;
incr_rd_ptr = 0 ;
case(rd_st)
IDLE : if (rd_fifo && !empty) begin
rd_nxtst = RD_MEM ;

```

```

rd_mem = 1 ;

rd_mem_stable :
`ASSERT_STABLE_POSEDGE(rd_mem,
(rd_st == IDLE),(rd_st == RD_MEM), clk);
// rd_mem stability check

end
RD_MEM : begin
rd_mem = 1 ;
rd_nxtst = IDLE;
incr_rd_ptr = 1 ;
end
endcase
end

always_ff @(posedge clk or negedge
rst_n)
begin
if (!rst_n) begin
rd_st <= IDLE ;
wr_st <= IDLE ;
end
else begin
rd_st <= rd_nxtst ;
wr_st <= wr_nxtst ;

rd_fifo_chk: `ASSERT_TRIGGER(
(rd_st == RD_MEM), !rd_fifo) ;
// ensure read request should not occur
in RD_MEM state, clk inferred

wr_fifo_chk: `ASSERT_TRIGGER(
(wr_st == WR_MEM), !wr_fifo) ;
// ensure write request should not occur
in WR_MEM state, clk inferred
end
end

fifo_underrun_chk: `ASSERT_NEVER
(((wr_toggle == rd_toggle) &
(rd_ptr > wr_ptr)), clk) ;
fifo_overrun_chk: `ASSERT_NEVER
(((wr_toggle != rd_toggle) &
(wr_ptr > rd_ptr)), clk) ;

always_ff @(posedge clk or negedge
rst_n) begin
if (!rst_n) begin
wr_toggle <= 0 ;
rd_toggle <= 0 ;
wr_ptr <= 0 ;
rd_ptr <= 0 ;
end
else begin
if (incr_wr_ptr)
begin

```

```

    if (wr_ptr == MAX_RAM_SIZE) begin
        wr_toggle <= !wr_toggle ;
        wr_ptr <= 0 ;
    end else
        wr_ptr <= wr_ptr + 1 ;
    end
    if (incr_rd_ptr) begin
        if (rd_ptr == MAX_RAM_SIZE)
            begin
                rd_toggle <= ~rd_toggle ;
                rd_ptr <= 0 ;
            end else
                rd_ptr <= rd_ptr + 1 ;
        end
    end
end

assign empty = ((wr_toggle == rd_toggle)
&& (rd_ptr == wr_ptr));
assign full = ((wr_toggle != rd_toggle)
&& (rd_ptr == wr_ptr));

// wr sm, wr takes two cycles

always_comb begin
    wr_mem = 0 ;
    wr_nxtst = IDLE ;
    incr_wr_ptr = 1 ;

    case(wr_st)
        IDLE : if (wr_fifo & !full) begin
            wr_nxtst = WR_MEM ;
            wr_mem = 1 ;

            wr_mem_stable :
            `ASSERT_STABLE_POSEDGE(wr_mem,
(wr_st == IDLE),(wr_st == WR_MEM), clk);
            // wr_mem stability check

        end
        WR_MEM : begin
            wr_mem = 1 ;
            wr_nxtst = IDLE;
            incr_wr_ptr = 1 ;
        end
    endcase
end
endmodule

```

V. CONCLUSIONS AND FUTURE DIRECTIONS

We described a property-based checker library, which is flexible, concise and easy to use. Unfortunately, the property-based checkers don't solve all the problems: they

cannot bind several properties together and cannot contain modeling code. E.g., these checkers cannot collect the coverage information as OVL checkers do. Therefore we face a problem of SV language enhancement to achieve a better support of assertion libraries. Even to make our property-based library usable we had to implement several minor language enhancements.

Here is a brief list of additional enhancements useful for checker library support:

- Introduce *checker*, a new language entity to specify a kind of encapsulation that can be instantiated anywhere in the code (both inside and outside of the procedural code) and infer the instantiation context. In this way the need for macro encapsulation would not be needed. A checker can also contain modeling code which is impossible with the property-based assertions.
- Allow argument lists for checkers with a variable number of actual arguments having different types. (a classical example – a checker making sure that all its arguments have the same value.) Arrays are not sufficient since they require all their arguments to be of the same type.
- Extend `generate` constructs to properties and checkers to enable more efficient implementation based on types or values of the actual arguments.
- Extend SVA temporal logic with LTL operators to achieve better extendibility.
- Allow immediate assertion specification outside of the procedural code.

These enhancements will make checkers more generic and will allow combining the advantages of the module-based checkers – packaging several properties and modeling code together in one checker with the flexibility of the property-based checkers: context sensitivity, locality and conciseness.

ACKNOWLEDGMENTS

We would like to thank the following people for their contribution to this work: Roy Armoni, Shalom Bresticker Dan Carmi, Alon Flaisher, Gavri Gavriellov, Haim Kerem, and Ariel Nathanson from the Intel side, and Surrendra Dudani, Pallab Dasgupta and Tom Borgstrom from the Synopsys side.

- [1] ReferencesProposal for enhancement of IEEE 1800 SystemVerilog (<http://www.eda-stds.org/svdb/> Item number 1530)
- [2] Accellera OVL SVA library (<http://www.eda.org/ovl/>)
- [3] IEEE 1800 - 2005 standard for SystemVerilog.