IEEE 1800-2009 SystemVerilog: Assertion-based Checker Libraries

Eduard Cerny
and Surrendra Dudani
Synopsys Inc.
Email: eduard.cerny@synopsys.com
and surrendra.dudani@synopsys.com

Dmitry Korchemny
Intel Corp.
Email: dmitry.korchemny@intel.com

***Abstract*— The enhancements to the IEEE SystemVerilog language in the 2009 standard and in particular to the SystemVerilog Assertions (SVA) allow us to create much more useful and versatile checker libraries. They benefit primarily from the following features: checker encapsulation, let declarations, clock and disable inference, deferred assertions, elaboration error tasks, and enhanced property operators. In this paper we first identify the weaknesses of the current checker libraries by examining an example from the OVL library. We then provide a classification of *checkers*, and show how various forms of effective checker libraries can be created using the new constructs.**

I. Introduction

In the paper we refer to the SystemVerilog 2009 [1] **`checker`** language construct using bold face as a keyword, while the common usage of a checker (of any kind) as a verification unit in normal text font.

There are many functional properties common to any design that are reusable modulo some expression changes. Therefore, to speed up the deployment of assertions without requiring extensive knowledge of the syntax and semantics of the SystemVerilog assertions language, it is essential to create libraries of checkers. Such checker libraries have been around for some time, such as the Accellera Open Verification Library (OVL) [3], and other checker libraries from EDA vendors. A similar approach was used even before the arrival of assertion languages, by hiding procedural or RTL implementation of assertions in modules used as checkers. The initial implementation as well as the Verilog'95 implementation of OVL is in this form.

The enhancements to the SystemVerilog language in the 2009 standard and in particular to the assertion features allow us to create much more useful and versatile checker libraries. They benefit primarily from the following features: New encapsulation, **`let`** declarations, clock and disable inference, deferred assertions, elaboration error tasks, and enhanced property operators. The new **`checker`** encapsulation can be used to replace the module. These enhancements in SVA provide a solution to many problems faced today when designing a checker library. Let us recall the main new SVA features that help checker library development and deployment [2]:

- **`checker`** encapsulation is versatile for assertion libraries.
  - Argument specification is similar to that of properties.
  - **`checker`** can be instantiated in procedural code.
  - Inference of clocking event, disable condition on the ports of the checker is possible.
  - In an **`always`** and **`initial`** procedure, evaluation is triggered by control reaching the checker instance.
- Inference functions `$inferred_clock` and `$inferred_disable` can be used as default values on formal ports of checker, sequence and property declarations.
- **`global clocking`** and **`default disable iff`** declarations are possible.
- Free variables and modeling code in checkers is available with some restrictions:
  - Restricted assignment forms, but easier to analyze and synthesize.
  - Only non-blocking (NBA) assignments in clocked always procedures.
  - Must respect single assignment rule (SAR).
- **`let`** construct allows for making abstractions from expressions.
- Checking of configuration parameters at elaboration time.

The paper is organized as follows: We briefly discuss typical module-based checkers and their weaknesses in Section II. In the subsequent section III we offer a set of characteristics that checkers should possess and according to which they can be classified. In Section IV we show typical kinds of checkers that follow the classification. We conclude in Section V by summarizing the changes that must be done to module-based libraries to convert them to **`checker`** based libraries, and by alluding to several enhancements to the SVA language that would further simplify development and deployment of checker libraries.

II. Weaknesses of Module-based Checker Libraries

To explain the current weaknesses, let us consider a simple checker `assert_handshake` inspired by its equivalent in the OVL library. The checker is reduced to include only its important excerpts. Details of included files are omitted. The user may wish to consult the OVL library for further details [3]. First, let us consider the checker interface.

```
// Accellera Standard V2.4 Open Verification
// Library (OVL).
// Accellera Copyright (c) 2005-2009.
// All rights reserved.
`module ovl_handshake (
  clock, reset, enable, req, ack, fire);
  parameter severity_level =
      `OVL_SEVERITY_DEFAULT;
  parameter min_ack_cycle = 0;
  parameter max_ack_cycle = 0;
  parameter req_drop = 0;
  parameter deassert_count = 0;
  parameter max_ack_length = 0;
  parameter property_type =
      `OVL_PROPERTY_DEFAULT;
  parameter msg =`OVL_MSG_DEFAULT;
  parameter coverage_level= `OVL_COVER_DEFAULT
      ;
  parameter clock_edge =
      `OVL_CLOCK_EDGE_DEFAULT;
  parameter reset_polarity =`
      OVL_RESET_POLARITY_DEFAULT;
  parameter gating_type =
      `OVL_GATING_TYPE_DEFAULT;

  input clock, reset, enable;
  input req;
  input ack;
  output [`OVL_FIRE_WIDTH-1:0] fire;
  //...
`endmodule // ovl_handshake
```

The macros `` `module `` and `` `endmodule `` resolve to either **module** and **endmodule** or **interface** and **endinterface**. This distinction is made so that the checker could also be instantiated in SV interfaces. In either case, the kinds of ports such checkers are allowed to have impose severe constraints on the deployment of the checker in a design:

- Clock port `clock` cannot be an event such as **edge** `clk` **iff** `en`.
- Clock, disabling condition `reset`, and the enabling condition cannot be inferred from the instantiation context.
- The checker cannot be instantiated inside a procedure.
- The ports `req` and `ack` must be integral expressions, they cannot be of type **sequence** or **property**.

The restrictions make the usage of the checker tedious. In particular, the last item makes the checker less flexible to use because if either the request or the acknowledgment are more complex temporal sequences of signal values, additional modeling code must be added on the outside of the checker instance to detect such sequences. This code and the checker instance are usually not to be included in the synthesized code, hence enclosing them between **`ifndef - `endif** compilation controls becomes necessary.

III. Kinds of Checkers and Their Characteristics

Checkers can be classified according four criteria:

1) Temporality: combinational (has no clock) vs. concurrent (requires a clock).
2) Encapsulation: **checker** vs. **property** (or **let**) based.
3) Packaging: in a Verilog library vs. in a SystemVerilog package.
4) Configurability: local per-instance vs. global for all instances.

A. Temporality

Many interesting checkers can be stated as unclocked Boolean expressions. Often clock is not needed and the user may be interested in instantiating the checker in combinational **always** procedures or design modules that do not have access to any clock. Such checkers cannot use concurrent assertions because they would require a clocking event. For this purpose deferred immediate assertions [1] are the best candidates. When it is required to verify behaviors that are synchronous to some clock, concurrent assertions need to be used. They can be Boolean expressions evaluating each attempt at a single clock tick or temporal properties evaluating attempts over several clock ticks.

B. Encapsulation

property-based encapsulation for temporal checkers, and **let**-based encapsulation for combinational checkers are the simplest ones. They are easy to use, but they allow no modeling code and can encompass only a single assertion. They are usually part of relatively simple checker libraries. More complex checkers that consist of several assertions, modeling code and coverage items need encapsulation in **module**, **interface** or more importantly now in **checker** constructs.

C. Packaging

Packaging checker libraries as a series of files, one per checker, in a ”library” directory that is included automatically during compilation is the most typical usage. This mechanism has been used with the various existing **module**-based checker libraries. **checker**, **property** and **let** encapsulations allow for a more robust use model by packaging them in the SystemVerilog **package** enclosure. In this way, the appropriate library can be ”imported” only where it is needed. Therefore, even different checkers with the same names can be deployed in different parts of the design. Of course, there is always the third possibility by accessing checker definitions that are brought into the source code using the **`include** directive. However, this method provides the least flexibility and we do not consider it further.

D. Configurability

Global configuration is achieved best by macros, for example, that includes the following:

- Enabling all assertions or all functional coverage or both.
- Exclusion of non-synthesizable code like action block reporting tasks, `covergroups` or test-bench related items.

Local configuration on a per-instance basis is best achieved by elaboration-time constants and conditional generate blocks. This includes:

- Selection of specific functional coverage items or levels, from a combination of **cover property** and **covergroup** constructs.
- Selection of **assert**, **restrict** or **assume** forms of assertions.
- Configuration and selection of subsets of assertions that should be active in a checker instance.
- Specification of minimal and maximal delay latencies and repetition counts in clock cycles.
- Specific user failure and success messages.
- Specification of severity level of assertion failure.

Elaboration-time constants must provide default values. For example,

- Most typical assertion usage (kind, delays, repetitions).
- Default failure message.
- Minimal useful functional coverage.

Functional coverage should provide several levels of detail whenever practically useful. Some may or may not be suitable for formal and synthesis tools and these should also be under global control. Here is a typical gradation:

- Minimal — "did the checked behavior ever happen?"
- More detailed — "which specific delay, data, etc., values were observed?"
- Corner cases — "were min and max delays, and extreme data points ever encountered?"

It is often desirable to perform `X/Z` value checks on signals used in a checker. There may be separate checkers that perform just that task and report a failure when an `X` or `Z` is detected. However, even "regular" assertions may have to include such checks, either to disable the assertion from failing or forcing a failure. The choice depends on whether separate checks for these values are used. If yes, then there is no point reporting a failure in regular assertions, they should just report a vacuous or disabled success. Usually the detection of `X/Z` is done using the system function `$isunknown` that returns true if an `X` or `Z` is detected in the argument expression value.

The ease of using configuration capability is important when different test environments are used. For example, control may be provided over the following features:

- Choice of failure, success and information reporting integrated with the SystemVerilog test-bench verification methodologies (e.g., VMM [4] or OVM [5]), or only reporting using `$display`, or using run-time error tasks such as `$error`, etc.
- Macro encapsulation over the checker such that it would automatically provide some of the keywords, thus simplifying instantiation (e.g., see IV-B). It may also hide differences between **checker**, **property**, and **module**-based checkers.
- Validation of values of arguments used as elaboration-time constants at elaboration time. Using conditional generate statements to test the constant arguments, elaboration tasks can issue error messages at elaboration, rather than at a later time during simulation (possibly many hours after the start of the compilation of the design).

The checker instances should also be easily identifiable by synthesis and formal tools, without the need of **`ifndef** - **`endif** enclosures around the checker instances. It is then left up to the tool to specify whether checkers should or should not be included in the process.

IV. Examples of Typical Checker Kinds

We now examine examples of different forms of checkers, illustrating the various characteristics and limitations.

A. Simple Combinational Checker

The combinational checker shown in the following example is defined in a **let** declaration, which is then used in a deferred immediate assertion. Notice the configuration mechanism using **`ifdef** `SYNTHESIS` for selecting a form that is suitable for synthesis and formal tools.

```
// in a package or an `include file
let onehot0 (sig, reset_n = 1'b1) =
`ifdef SYNTHESIS
 // Selected for synthesis or formal
   !|reset_n || $onehot0(sig);
`else
// Selected for 4-valued simulation
  |reset_n === 0 ||
    ($onehot0(sig) && !$isunknown(reset_n);
`endif
```

Such a checker can be instantiated in a module (program or interface), and procedural scope as follows.

```
module m(input logic [3:0] r1,
         output logic [3:0] r2);
  A1: assert #0 (onehot0(r1))
  else $error("A1 failed"); //check input
  always_comb begin
    r2 = r1;
    A2: assert #0 (onehot0(.sig(r2)))
    else $error("A2 failed"); //check output
  end
endmodule
```

The example illustrates several points:

- A macro definition `SYNTHESIS` selects between two forms of let, one suitable for formal tools and synthesizable checkers, the other one for 4-valued simulation. In the latter case the assertion is enabled when `reset_n` is `1`, disabled (success) when `reset_n` is `0`, and it is forced to fail when `reset_n` is `X`, or `Z`.
- Both positional as in `A1` and named as in `A2` argument associations can be used.
- The system function `$onehot0` could be used directly in the assertion, however, it would not provide for a disabling condition.
- The `reset_n` argument has a default actual argument `1'b1`, meaning that when the actual is not provided in an instance, the resetting condition is false by default as shown in both `A1` and `A2`.
- The assertion is in the deferred form, hence it filters out 0-width glitches on both `reset_n` and `sig` actual arguments.

- The assertions can be instantiated in the module scope like A1, or in a procedure like A2.
- The disabling condition cannot be inferred in **let** instances. That is, $inferred_disable may not be used as a default actual argument.

*B. A **checker**-Based Combinational Checker*

Next we examine a more flexible combinational checker. We assume that the convention is that **default disable iff** provides an active low reset.

```
typedef enum {ASSERT, ASSUME, NONE}
    assert_type;
typedef bit [15:0] cover_type
checker onehot0( sig,
   assert_type usage_kind = ASSERT,
   cover_type cover_level = 1,
   reset_n = $inferred_disable,
   string msg = "", synthesis = `SYNTHESIS);
if (cover_level<16'b0 || cover_level>16'b11)
// check valid coverage selection
  $error("Coverage level is invalid %d",
    cover_level,
    "\nonly 1(level 0), 2(level 2), 3(both)",
    " or 0 (disabled) are allowed");
if (usage_kind != ASSERT || usage_kind !=
    ASSUME)
  $warning("No assert or assume selected");
if (synthesis) begin : SYNTH
  let check_onehot0 (sig, reset_n=1'b1) =
    ((!|reset_n) || $onehot0(sig));
  let cover_onehot0 (sig, reset_n=1'b1) =
    ((|reset_n) && $onehot0(sig));
end : SYNTH
else begin : NO_SYNTH
  let check_onehot0 (sig, reset_n=1'b1) =
    ((|reset_n === 0) || $onehot0(sig) &&
      !$isunknown(reset_n));
  let cover_onehot0 (sig, reset_n=1'b1) =
    ((|reset_n === 1) && $onehot0(sig));
end : NO_SYNTH
`ifdef ASSERT_ON
if (usage_kind == ASSERT) begin : ASSERT
  Assert_onehot0:
    assert #0 (check_onehot0(reset_n, sig))
      else $error(msg);
end : ASSERT
else if (usage_kind==ASSUME) begin : ASSUME
  Assume_onehot0:
    assume #0 (check_onehot0(reset_n, sig))
      else $error(msg);
end : ASSUME
`endif
`ifdef COVER_ON
if (cover_level & 1) begin : COVER_L1
  Cover_onehot0_1:
    cover #0 (cover_onehot0(reset_n, sig));
end : COVER_L1
if (!synthesis && (cover_level & 2))
  begin : COVER_L2
  function int position(logic $bits(sig) arg);
    for (int i=0; i<$bits(sig); i++)
          if (sig[i] === 1) return i;
    return 0;
  endfunction // position
  covergroup cg_onehot0_2 with
    function sample(int index);
    coverpoint index;
  endgroup
  cg_onehot0_2 onehot0_2_index = new();
  Cover_onehot0_2:
    cover #0 (cover_onehot0(reset_n, sig))
      onehot0_2_index.sample(position(sig));
end : COVER_L2
`endif
endchecker : onehot0
```

This combinational checker illustrates many of the features that the **checker** encapsulation provides over the simpler **let**-based form:

- Coverage can be enabled globally for all checker instances by defining the symbol COVER_ON. Similarly, verification statements (**assert** or **assume**) can be globally enabled by defining ASSERT_ON.
- Synthesizable form is selected by a conditional generate block controlled by the argument synthesis that has as default actual value the macro symbol SYNTHESIS. This allows overriding the global selection if so required.
- The reset condition may be inferred from a **default disable iff** declaration.
- Deferred **assert** (usage_kind == ASSERT) or **assume** (usage_kind == ASSUME) statement or none can be selected using the argument usage_kind.
- When an invalid value is provided for cover_level, no coverage is enabled in this instance and an error message is issued.
- When an invalid value (or NONE) is provided for usage_kind, no verification statement (**assert** or **assume**) is enabled in this instance and a warning message is issued.
- Two levels of functional coverage are provided, they can be individually enabled or disabled:
  - Level 1 — when cover_level == 1 is selected, it collects information on how many times a one hot or 0 condition was encountered while not disabled by reset.
  - Level 2 — when cover_level == 2 is selected, the **covergroup** classifies the bit positions that are set to 1 when the one hot condition holds. A deferred **cover** statement is used to trigger sampling of the bit position index by calling the sample method of the **covergroup** in the pass action statement of the deferred **cover** statement.
  - Both levels can be selected by setting cover_level == 3.

The **checker** can be instantiated in a simpler way than the one using a **let** declaration because the disable condition can be inferred:

```
`define ASSERT_ON
// Instantiations of a combinational checker
module m(input logic [3:0] r1,
         output logic [3:0] r2);
```

```
  default disable iff 1'b1;
  onehot0 A1(r1); // check input
  always_comb begin
    r2 = r1;
    onehot0 A2(r1); // check output
  end
endmodule
```

There is however a difference between using this checker and the simple checker in Section IV-A in that according to the **checker** specification in the LRM, all the variable inputs to the checker are sampled (the values are taken from the *Preponed* scheduling region), while the simple checker uses only the current values. This may make a difference when the **checker** is used in procedural code like the instance A2. The problem can be avoided by **const** casting of the actual arguments r1. The instance in the **always** procedure thus becomes:

```
  always_comb begin
    r2 = r1;
    onehot0 A2(const'(r1)); // check output
  end
```

The first instance, A1, may use sampled values because it does not depend on any current time condition in the module. It would report any violation one simulation time step later than if it used the current value, but that is all. If we remove sampling even in instance A1 then we can encapsulate the checker instantiation statement in a macro and hide the **const** cast as follows:

```
`define ASSERT_ONEHOT0 \
  (name = "", sig, usage_kind = ASSERT, \
  cover_level = 1, reset_n = 1'b1, msg = "", \
  synthesis = `SYNTHESIS) \
  onehot0 `name (const'(sig), usage_kind, \
                 cover_level, const'(reset_n), \
                 \
                 msg, synthesis)
```

Unfortunately, due to the automatic sampling of checker arguments, we cannot use $inferred_disable as a default value for reset_n. There is no way to place the **const** cast on the function. Therefore, by using the macro encapsulation we have lost the ability to infer the reset condition.

As with the previous simple combinational checker, this more complex one can still be instantiated both inside (instance A1) and outside (instance A2) a procedure. Both checker instances use default values for configuration constants and enable verification statement **assert** #0 because ASSERT_ON is defined and usage_kind default value of ASSERT is used. Coverage is globally disabled because COVER_ON is not defined.

C. A Simple Property-Based Temporal Checker

We now turn our attention to checkers that verify behavior over time — temporal checkers.

Similarly as with the simple combinational checker and **let** declarations, we can define a simple temporal, clocked, checker using **property** declarations. As before, we assume that **default disable iff** defines an active low reset.

```
property time_interval_p
  (sequence trig, property cond,
   start_tick=1, end_tick=1,
   event clk = $inferred_clock,
   untyped rst_n = $inferred_disable);
     @clk disable iff (!bit'(|rst_n))
     trig |->
        always [start_tick:end_tick] cond;
endproperty : time_interval_p
```

Note that in the consequent of |-> we used the **always** operator instead of using the consecutive repetition cond[*start_tick:end_tick]. The reason is that we obtain maximum generality as to the actual argument for the formal cond. It can be not only a Boolean or a **sequence**, but also any **property** expression.

The property verifies that when trig occurs cond holds true in the interval start_tick to end_tick clock ticks, unless it is disabled by rst_n being 1'b0. The property has the following characteristics:

- The actual argument for trig is restricted to the type **sequence** because it is used in the antecedent of |->. The actual argument for cond can be any **property** expression (Boolean, sequence or property). The actual argument for clk must be a clocking event, while rst_n is left **untyped** for the user to be able to pass any valid expression.
- trig and cond do not have default actual arguments, hence the user must supply valid arguments there.
- Both clk and rst can be inferred from the context because the inference functions are used as default actual arguments.
- The arguments start_tick and end_tick have the typical default value of 1. If used as in the following instantiation example, the property will check that cond holds true at the next clock tick after trig holds true.

A simple instantiation of the time_interval_p property is illustrated in the next example.

```
module m(input logic clk, reset_n, load,
         input logic [3:0] r1,
         output logic [3:0] r2);
default disable iff reset_n;
always @(posedge clk) begin
  if (!reset_n) r2 <= 'b0;
  else if (load) r2 <= r1;
  Loaded_r2: assert property(time_interval_p(
    $past(load), r2 == $past(r1))) else
    $error("r2 not loaded correctly by r1");
end
endmodule
```

Except for the arguments that are used in the actual verification, all other ones use default values. The clock and the disabling condition are inferred from the **always** procedure and from the **default disable iff** declaration, respectively.

D. A *checker*-Based Temporal Checker

The final example illustrates the full power of a **checker**-based temporal checker definition. We show a modified form of the of `assert_handshake` checker discussed at the beginning of our paper (Section II), but for reasons of brevity we include only those portions of the code that will illustrate the differences.

The interface of the new checker is now as follows:

```
//Modified assert_handshake checker
import std_ovl_defines::*;
checker assert_handshake (
  sequence req, sequence ack,
  event clk           = $inferred_clock,
  untyped reset_n     = $inferred_disable,
  //elaboration-time constants:
  int severity_level = OVL_SEVERITY_DEFAULT,
  int min_ack_cycle  = 0,
  int max_ack_cycle  = 0,
  int req_drop       = 0,
  int deassert_count = 0,
  int max_ack_length = 0,
  int property_type  = OVL_PROPERTY_DEFAULT,
  string msg         = OVL_MSG_DEFAULT,
  int coverage_level = OVL_COVER_DEFAULT,
  int synthesis      = SYNTHESIS
);
//...
generate // elaboration-time constant checks
    at compile time
  if (min_ack_cycle < 0)
    $error("min_ack_cycle is negative");
  if (max_ack_cycle < min_ack_cycle) $error(
    "max_ack_cycle is less than min_ack_cycle"
        );
  if (req_drop < 0 || req_drop > 1) $warning(
    "req_drop \%0d is not 0 or 1",
    req_drop, " positive assumed 1, "
    "anything less than 1 assumed 0");
  // ...  checks for other arguments ...
endgenerate

default clocking checker_clk @clk; endclocking
default disable iff (reset_n);

//... Body of the checker ...

endchecker : assert_handshake
```

The parameters from the original checker became regular arguments of the **checker**-based checker. It simplifies instantiation, although the user should be aware that these arguments must be elaboration-time constants. The argument values of constants are verified at elaboration time using a conditional **generate** and elaboration time error tasks. If the values are illegal then an error message is issued, or if a reasonable alternative exists then that value is used and a warning is issued.

The following parameters from the original checker are missing: `clock_edge`, `reset_polarity` and `gating_type`.

This is because

- `clock_edge` is not needed as the argument `clk` can be an event expression.
- `reset_polarity` is not needed because we can pass any expression to the checker and it can infer the appropriate default expression from the contextual **default disable iff** declaration.
- `gating_type` is omitted for the same reason as `clock_edge`, i.e., the actual clocking event provided for `clk` can contain **iff** enabling condition.

The formal arguments `clk` and `reset_n` were placed after the arguments that do not have defaults. This simplifies instantiation of the checker when all arguments use default values. The type of `reset_n` is left unspecified (the keyword **untyped**) to provide more flexibility as to the kind of the actual reset expression. As in the case of the **checker** based combinational checker, all variable arguments are sampled. This may not be desirable for `reset_n`, hence the actual argument for `reset_n` should have the **const** cast. Unfortunately, if the default is inferred, it will be sampled.

The default value constants for the arguments are no more `` `defines ``, but instead they are constants picked up from package `std_ovl_defines` as **enum** type values.

The type for `req` and `ack` is specified as **sequence**, to allow Booleans and sequences, but prohibit supplying a property expression as the actual argument. This makes the checker more general, eliminating the need for modeling code to reduce a complex temporal behavior to a Boolean expression.

The body of the checker has to be modified to comply with restrictions on modeling code in **checker** constructs, and to use all the new features that help implementing and using checkers. The body of the **checker**-based checker is shown next. Please refer to the OVL library to compare with the original checker body [3].

Only those portions as in the example of the original checker are shown that illustrate the differences with the original checker. The following piece of code shows the transformation needed in the modeling code of the checker.

```
`ifdef ASSERT_ON
logic first_req = 1'b0;

sequence s_req;
  req;
endsequence

function logic setFirstReq();
  if (!reset_n) return 1'b0;
  if((first_req ^ first_req) == 1'b0)
    return s_req.triggered;
  return 1'b0;
endfunction : setFirstReq

always @(clk) first_req <= setFirstReq();
```

Since always procedures in **checker** cannot have any conditional statements, the function `setFirstReq` is defined to encapsulate the procedural code. The function is then called on the right-hand side of a non-blocking assignment. Variable `first_req` is used in a **property** in the following code fragment.

```
property ASSERT_HANDSHAKE_ACK_MIN_CYCLE_P;
  s_req |-> not s_eventually [0:min_ack_cycle]
       ack;
endproperty
property
  ASSERT_HANDSHAKE_ACK_WITHOUT_REQ_FIRST_REQ_P
      ;
  (##1 ack) implies
    (first_req or s_req.triggered);
endproperty
// other properties
// this remains as before
case (property_type)
  OVL_ASSERT_2STATE, // defined as enum types
  OVL_ASSERT: begin : ovl_assert
    if (min_ack_cycle > 0)
    begin : a_assert_handshake_ack_min_cycle
      A_ASSERT_HANDSHAKE_ACK_MIN_CYCLE_P:
        assert property (
            ASSERT_HANDSHAKE_ACK_MIN_CYCLE_P)
      else ovl_error_t("...as before...");
    end
// other assert and assume statements
endcase
`endif //ASSERT_ON

`ifdef COVER_ON
generate
  if (coverage_level != OVL_COVER_NONE) begin
      : ovl_cover
    if (OVL_COVER_BASIC_ON)
    begin : ovl_cover_basic
      cover_req_asserted:
      cover property
        (reset_n throughout s_req))
          ovl_cover_t("req_asserted covered");
    end
    //... other cover statement ...
  end
endgenerate
`endif // COVER_ON
```

Notice the following differences:

- Procedural conditional code is placed in a function to satisfy restrictions on assignments to checker variables. The **always** procedure contains a single assignment only.
- Case default values on parameters are removed since constant argument values are checked at compile time.
- Case item labels are predefined **enum** types rather than **`define** symbols.
- The property expressions use property operators to allow sequences as the arguments and to make the assertions more efficient for formal tools. For example, in property ASSERT_HANDSHAKE_ACK_MIN_CYCLE_P the sequence repetition is replaced by **not s_eventually** ... to accept a sequence expression for ack.
- A new sequence s_req is defined that instantiates req. This allows us to use the sequence method s_req.triggered in the function.
- Since both req and ack can be sequences, $rose had to be removed from both of these operands of the property. Thus, if the user wishes to use $rose on a Boolean, the appropriate expression has to be passed as the actual argument. [1]

There is currently a restriction imposed on this form of checkers: There are no **output** and **inout** arguments allowed in **checker** encapsulation. That is, the output port fire of the original OVL checker is missing here. Therefore, the only way to chain several **checker** based checkers is to access some internal variable of one checker instance in some other checker instance using a cross checker reference. This makes it less flexible.

Note that by extending the type of the arguments to **sequence**, it is now impossible to include checks for the presence of X/Z values in the variables involved in the actual arguments. If such checking is required, the best approach is to create specific checkers just for the purpose of verifying X/Z on variables. An open question remains how to disable existing assertions within the checker in such cases. It either requires an enhancement to the SystemVerilog language to provide a function that detects X/Z in sequences and properties, or an enhancement in the simulator to evaluate assertion in a pessimistic fashion.

The following is an example of instantiation of the new checker. For simplicity all elaboration-time arguments take on default values.

```
module m;
bit clk; logic rst_n, request, acknowledgment,
     endtrans;

default clocking @(posedge clk iff enabled);
endclocking
default disable iff rst_n;
//... some design code ...
always @(posedge clk) begin
  assert_handshake chk_handshake_inst(
    .req($rose(request)),
    .ack(acknowledgment ##1 endtrans));
  if (!rst_n) begin
    //... some design procedure ...
  end
end
//... some design code ...
endmodule
```

The main points to notice are:

- The checker instantiation syntax is similar to that of a module, except that there is no parameter section.
- It can be instantiated in an **always** procedure.
- The reset argument is inferred from **default disable** declaration.
- The clocking event is inferred from the **always** procedure, hence even though **default clocking** is defined, the clock from the **always** procedure takes precedence.
- The actual argument for the formal argument req is $rose(request); its clock is obtained from the default clocking defined in the module.

[1] If a system function existed that allowed to distinguish Boolean expressions from temporal sequences and properties, a conditional generate could be used to construct different forms of properties depending on the actual argument.

- The actual argument for `ack` is a sequence expression.

V. Conclusions

In conclusion we summarize the transformations to consider when converting the old-style module-based checkers into the new format based on **`checker`** construct encapsulation, and suggest some further enhancements to the SVA language to ease the development and usage of checker libraries.

A. Transformations Needed to Convert Module-Based Checkers to the New Format

- Replace **`` `define ``** for various constants by **`typedef`** declarations using an **`enum`** type whenever possible.
- Change modeling code to respect Single Assignment Rule (SAR) by introducing functions for evaluating the right-hand side expressions of assignments.
- Replace all continuous assignments by references to **`let`** statements.
- Create compile-time checks on elaboration-time constant values.
- Use initial procedures only for indicating that the enclosed assertions should have only one evaluation attempt. Initialize variables in their declaration.
- Change interface definition to include original parameters as regular arguments.
- Provide inference functions as default arguments to clock and reset.
- Provide default actual arguments wherever appropriate.
- Generalize the type of arguments to **`sequence`** or **`property`** wherever the checker properties can admit such operands.
- Checker instance identification task calls in initial procedures should be replaced by initial and an immediate assert statement on true, with a pass action statement displaying the required identification message.
- Consider using **`covergroup`**s to provide more detailed coverage, selectable by an argument.
- Add **`default clocking`** and **`disable iff`** declarations and simplify assertions.
- Place the new checkers in a package for easy yet controlled access from a design unit.

B. SVA Language Enhancements

- The usage and development of checkers for libraries could be made easier if **`let`**, **`property`**, **`sequence`** and **`checker`** allowed passing variable number of arguments. This would reduce the number of checker variants and configuration arguments. The same enhancement should also be done for macro definitions so as to allow encapsulation of the generalized checkers in macros as we illustrated earlier.
- The enhancement for variable number of arguments should be supplemented with new elaboration-time system functions to manipulate the variable lists of arguments (e.g., similar to *map* and *reduce* functions in LISP).
- Sampling of checker arguments should be controlled by some qualifier on the formal argument declaration.
- Allow continuous assignment in **`checker`** encapsulation. It may be more efficient in some situations because of the substitution semantics of **`let`**.
- Allow the use of conditional statements in always procedures in checkers, possibly under some restrictions.
- Provide **`output`** and **`inout`** ports in **`checker`** encapsulation.

Acknowledgment

The authors would like to thank John Havlicek (Freescale) for helpful discussions.

References

[1] *IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language*, IEEE 1800-2009.

[2] E. Cerny, D. Korchemny, L. Piper, E. Selingman, S. Dudani, *Verification case studies: evolution from SVA 2005 to SVA 2009*, Proc. Design Verification Conference (DVCon) 2009.

[3] *Accellera Standard Open Verification Library* (OVL), Version V2.4, Accellera, 2009.

[4] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale *Verification Methodology Manual for SystemVerilog* (VMM), Springer, ISBN: 978-0-387-25538-5, 2005.

[5] M. Glasser, *Open Verification Methodology Cookbook*, Springer, ISBN: 978-1-4419-0967-1, 2009.