SystemVerilog Checkers: Key Building Blocks for Verification IP

Laurence Bisht, Dmitry Korchemny, Erik Seligman
Intel Corporation
{laurence.s.bisht@intel.com, dmitry.korchemny, erik.seligman}@intel.com

***Abstract*—This paper describes checkers – a SystemVerilog construct for packaging verification library entities and verification IP. An important example of a checker application is a checker-based verification library recently donated to Accellera. We provide a brief overview of checkers and their main applications, and describe new checker features expected to become part of the emerging SVA 2012 standard, the motivations behind them, and the challenges posed by their definition. We conclude with a consideration of other checker enhancements that fell beyond of the scope of SVA 2012.**

Keywords- SystemVerilog Assertions; assertion-based verification; RTL simulation; formal verification.

I. INTRODUCTION

Assertion-based verification (ABV) [1] has proven itself to be an important means of RTL validation. To natively support ABV, SystemVerilog has a sublanguage for assertion specification, called SystemVerilog Assertions (SVA) [2]. One of the major challenges for ABV support is packaging several assertions and the auxiliary instrumental code required for assertion verification together. Such packaging is required in order to organize standard assertion libraries, such as OVL [3], and to organize modeling code for formal verification. Early versions of SystemVerilog provided some container types that might seem to serve this purpose, such as modules and interfaces. This packaging is still used for OVL. However, none of these options is sufficient for organizing truly flexible verification libraries and modeling code. SystemVerilog 2009 [2] introduced checkers for this purpose. Checkers have numerous advantages over modules and interfaces: they are race-free, they allow sequences and properties as arguments, their invocation syntax is more intuitive and concise, they may be instantiated both inside and outside procedural code, they may infer their clock and reset contexts, they have free variables, etc. An important example of a checker application is a checker-based verification library recently donated to Accellera [4].

The work on the SVA 2012 standard has aimed to make checkers even more powerful by enriching their modeling capabilities, adding support for output ports, etc. These new features will make checkers better suited for writing large verification IP units and enable new uses for checkers. For example, checkers will be able to feed their assertion completion status to scan latches, thus opening the door for automatic assertion synthesis on chip; it will be possible to use checkers in testbenches as input stimuli generators that benefit from the checker randomization capability based on assumptions, etc. The introduction of new checker modeling constructs and the deprecation of general purpose always procedures in checkers make it possible to remove several annoying restrictions imposed on checker modeling in SystemVerilog 2009. For example, it should become possible to use continuous and blocking assignments to implement combinational logic, and the single assignment rule will be removed.

However, extending checker capabilities has turned out to be a challenging task. Unlike modules, checkers as building blocks for verification IP must have stable race-free behavior in any instantiation context. This requires careful definition of sampling in different checker constructs. Another problem to be addressed is the definition of the simulation semantics of the new constructs in the presence of free variables.

Time constraints prevent several important aspects of checkers from being addressed in this work. One such potential feature is checker instantiation in classes, which would enable, for example, implementing UVM [5] monitor classes as checkers. This would make it possible to reuse the same checkers at different levels of abstraction. Other issues include forcing in checkers and support of a variable number of checker ports.

We begin our paper with a brief overview of checkers and discuss their main use cases belonging to verification libraries and modeling. We then describe the new checker features of SVA 2012 and analyze the main challenges in their definition. We conclude with a consideration of other checker enhancements that fell beyond the scope of SVA 2012.

II. CHECKER OVERVIEW

A. High level description of checker construct

The checker construct is one of the major building blocks that make up a SystemVerilog design and verification environment. Checkers were designed and added to answer several needs: to encapsulate several assertions and their related modeling into bigger blocks having a well-defined functionality, to inherit context clock and enabling conditions, and to enable flexible argument types. This type of construct makes it possible to create powerful and general collections of higher-level checks 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.

SVA 2009 allowed checker formal ports to be defined only as inputs, and the checker modeling mechanism was limited to nonblocking assignments only. The following example illustrates checker usage:

```
checker follows(sequence first,
 property second,
 event clock = $inferred_clock,
 untyped reset = $inferred_disable,
 bit report_coverage = 1'b0);

 default clocking @clock; endclocking
 default disable iff reset;

 a: assert property (first |=> second);
 if (report_coverage)
  c: cover property (first #=# second);

endchecker : follows

module m(logic rst, clk, en, ...);
 default disable iff rst;
 logic req, gnt;

 always @(posedge clk) begin
  req <= ...;
  gnt <= ...;
   if (en)
    follows req_granted(req, gnt);
  ...
 end
endmodule : m
```

Here the checker `follows` checks that `first` is followed by `second`. In a simple case, `first` and `second` are Boolean, but in the general case `first` may be any sequence, and `second` may be any property. The checker also has an optional coverage statement that registers all occurrences of `first` followed by `second`. This check is only performed when `report_coverage` is a non-zero elaboration time constant.

The checker invocation in the module illustrates the checker's ability to be instantiated in procedural code and to infer its clock and disable condition from its invocation context [6].

B. Checkers and other containers of SystemVerilog

Though SystemVerilog offers a rich set of container objects, including modules, interfaces, programs, and properties, these containers are not satisfactory for representing the building blocks of verification libraries and formal verification IP:

Modules: These are the main building blocks of SystemVerilog-based design. They are well suited for general modeling code, and can hierarchically instantiate submodules. But they have many limitations, including the lack of clock/reset inference from context, the lack of untyped or sequence/property inputs, sensitivity to input races, and the inability to be instantiated in procedural code.

Interfaces: These are designed to package a bundle of common inputs and outputs in a reusable way for multiple modules. They share the same limitations mentioned above for modules, and in addition are unsupported by many formal verification tools.

Programs: These are more software-oriented containers, designed for packaging testbench code. They share the limitations of modules and interfaces.

Properties: These are the containers originally supplied by the SVA portion of the SystemVerilog language. They provide advantages including clock and reset inference from the instantiation context, untyped input arguments, and support for instantiation in the form of concurrent assertions in any of the other containers described above, including their procedural code. However, they can neither package several assertions nor contain general modeling code, facts which severely limit their usefulness for building libraries or complex verification IP.

Like most other containers in SystemVerilog, checkers may incorporate covergroups. This makes checkers well suited not only for correctness, but also for coverage checks. Checkers also have another important feature, *free variables*, not found in any of the above container types. Free variables are briefly described in the next section.

C. Free Variables

In a formal verification context, it is often desirable to utilize variables that are able to assume an arbitrary value, representing unpredictable input from the environment, possibly constrained with some temporal condition specified as an assumption. Free variables also simplify modeling, since like assertions they always use sampled values of non-free variables, and thus avoid race conditions. Another advantage of free variables is that unlike free inputs, they are randomized in simulation. This randomization respects constraints imposed on free variables by assignments and assumptions. Syntactically, free variables are distinguished from other checker variables by the qualifier **`rand`**. The following example[1] shows how free variables may be used to define a clock non-deterministically, i.e., a clock that matches both 0101… and 1010… time sequences.

```
rand bit clk;
always_ff @$global_clock clk <= !clk;
```

Refer to [7] for more information on checker free variables.

D. Checkers and PSL vunits

Besides SVA, PSL (Property Specification Language) [8] is another property specification language widely used in the industry. PSL also has a construct for packaging verification code, called vunit. It is instructive to compare SVA checkers to PSL vunits.

Both checkers and vunits may incorporate assertion directives and modeling code. Here are the most important differences between the two:

- Checkers may be either instantiated or bound to a device under test (DUT); vunits may only be bound.

[1] The example uses the checker procedure syntax as defined in the emerging standard (SVA 2012).

- Checkers resolve names using their declaration context, vunits resolve names using their instantiation context. Note, however, that it is possible to pass to a checker the clock and reset context at its instantiation point.
- Data declared in vunits override corresponding data in the design. Data declared in checkers have a different scope and do not directly interfere with the design data.

These peculiarities of checkers and vunits imply their main usability advantages and disadvantages. Checkers are better suited to serve as library units because of their ability to be instantiated. Vunits are rather clumsy for this purpose: they should be bound to shadow modules or interfaces as done in the PSL-based implementation of OVL [3].

On the other hand, it is easy to use vunits for writing formal verification IP: because vunits resolve names in the instantiation context, there is no need to pass all required design signals to a vunit as is required for checkers. Also, the ability of vunits to override allows seamless design pruning for verification needs.

It is worth mentioning that checkers have well-defined simulation semantics, whereas the simulation semantics of vunits is implementation dependent.

III. Checker Use Case: Verification Libraries and IP

The most important motivation for introducing checkers was to enable flexible, reusable verification library development. Verification libraries such as OVL [3] already existed prior to checkers, but had to make use of the available container types, accepting the disadvantages described in Section IIC above. Checkers have enabled the development of cleaner and more flexible libraries, such as the SVA Checker Library [4] recently donated to Accellera. The following table [9] describes some major differences between the two libraries in terms of the library requirements we identified from project usage:

Library Requirement	OVL Issues	SVA Checker Library Solution
Extensibility: parameterizing library entities with sequences/properties.	**Not possible**: a module cannot take a sequence or property as input.	**Solved**: library entities may take sequences or properties as arguments.
Inference: infer clock and reset from context.	**Not possible**: no inference mechanism for modules.	**Solved**: checkers may infer clock and reset from their instantiation context.
Locality: entities instantiated as close as possible to target code.	**Limited**: modules cannot be instantiated in procedural code.	**Improved**: checkers may be instantiated inside or outside procedural code.
Efficiency: entity implementation should be as efficient as possible.	**Some problems**: OVL entities are modules—having many of them results in notable simulation cost.	**Improved**: checkers are more efficient than modules due to substitution semantics and specialization.
Conciseness: entity invocation should be concise.	**Not possible**: parameterized module instantiation is verbose.	**Solved**: checker parameterization is implicit and its arguments may be untyped.
Determinism: result should be independent of input evaluation order	**Poor:** races between data and clocks are possible	**Solved**: sampling prevents races

Table 1: OVL vs. SVA Checker Library

See [9] and [10] for detailed discussion about verification libraries.

Related to the problem of supplying verification libraries is the more general problem of verification IP. Reusable IP is becoming more and more important throughout the industry [11] [12] [13] [14] [15]. Due to the ability to customize and parameterize designs based on common IP, equally flexible verification libraries are increasingly desirable. We believe that checkers will become a key enabler for the development of these libraries in the coming years.

IV. New Checker Features Targeted for SVA2012

As we have seen, the checker construct in its current definition brings clear added value; however, users already see several limitations and have identified usability enhancements needed for making checkers even more powerful and using them in common practice. In this section, we will describe the major checker features targeted for SVA 2012.[2] These features make checker modeling capabilities similar to those of modules and thus more familiar to design and validation engineers.

A. Assignment statements

In SVA 2009 the only assignment statements legal in checkers were non-blocking assignments (NBA). In addition to non-blocking assignments, SVA 2012 introduces other kinds of assignments that exist in modules: continuous and blocking assignments. There are, however several subtle differences between checker and module assignments: in checkers continuous assignments may not be procedural, blocking assignments may not be placed in an **`always_ff`** procedure, the right-hand side of non-blocking assignments in an **`always_ff`** procedure is sampled, and no assignment is legal in an **`initial`** procedure. The reason for these differences is explained in Section G.

B. Always procedures

In SVA 2009 the only always procedure allowed in checkers was the general-purpose **`always`** procedure. This procedure allowed a single timing control statement and non-blocking assignments as the only modeling statements. Therefore, this procedure in checkers essentially played the role of an **`always_ff`** procedure. For this reason the general-purpose always procedure will be deprecated in SVA 2012, and the **`always_ff`** procedure will be introduced instead. SVA 2012 will also allow **`always_comb`** and **`always_latch`** procedures to model combinational and latched logic as it is done in modules.

[2] SVA 2012 has not been officially approved as of the time of writing this paper. Therefore, though we believe the information to be accurate, there is no guarantee that all the features will become part of SVA 2012 in the form they are described in this paper.

C. Procedural control statements

In SVA 2009 there were two different coding styles in modules and checkers: in modules, continuous assignments and procedural control statements were used, whereas in checkers functions and let statements were used instead. To allow in checkers the RTL coding style used in modules, SVA 2012 will make procedural control statements and looping statements legal in checkers. For example, SVA 2009 required the following coding style to compute the new value of a variable window in a checker:

```
function bit next_window (bit win);
 if (reset || win && end_event) return 1'b0;
 if (!win && start_event) return 1'b1;
 return win;
endfunction
always @clock
 window <= next_window(window);
```

SVA 2012 allows rewriting this in a more conventional way:

```
always_ff @clock begin
 if (reset || win && end_event)
  window <= 1'b0;
 if (!win && start_event) window <= 1'b1;
end
```

D. Cancellation of single assignment rule

SVA 2009 enforced a Single Assignment Rule (SAR): it was illegal to use the same bit of a checker variable in several assignment-like contexts. SAR was required in order to prevent assignment races in checkers. Because of the deprecation of the general-purpose **always** procedure in checkers, the explicit SAR became redundant: the interprocess single assignment is imposed implicitly by special always procedures: **always_comb**, **always_latch** and **always_ff**; multiple assignment to the same variable in the same process has well defined semantics. For this reason, SAR will be cancelled in SVA 2012.

E. New sampling rules

SVA 2009 defined all checker arguments to be sampled in the Preponed region, i.e., at the beginning of the simulation time step. This is problematic when a checker is used to wrap deferred assertions that check current values of signals. For example, in the following assertion

```
assert #0 (a);
```

the variable `a` is not sampled. However, if the same assertion was contained in a checker, its expression was sampled:

```
checker check1(x); assert #0 (x); endchecker
…
check1 c(a);
```

The ugly workaround was to **const** cast all the actual arguments of a checker instance. However, this workaround does not work to infer the reset value. The following example illustrates the problem:

```
checker check3(a, rst = $inferred_disable);
 a1: assert #0 (rst || $onehot0(a));
endchecker : check3

module m(...);
 default disable iff reset;
 always_comb begin
  x = ...;
  y = ...;
// x and y are not sampled, reset is sampled
  check3 c3(const'({x, y}));
 end
endmodule
```

A nastier problem occurred when a clock was passed to a checker as a signal and not as an event:

```
checker check2(logic clk, a);
 assert property (@clk a);
endchecker
```

Sampling of the clock signal `clk` resulted in a non-intuitive (and ill-defined) behavior of the assertion.

Therefore, in SVA 2012 checker arguments will not be sampled. Instead, to make checkers behave deterministically, all expressions in **always_ff** procedures except free variables and variables in the event control (see Section V.C) will be sampled to enforce variable sampling in NBA. Consider the following example:

```
always_ff @clk a <= b;
```

In modules, if `b` changes before `clk`, `a` is assigned the new value of `b` instead of the old one. In well-formed RTL this situation does not happen. This situation is not acceptable in checkers because a checker may use instrumental code which does not have to be well-formed RTL, and may even relate to testbench variables. Therefore, in a checker `b` must be sampled.

F. Output ports

SVA 2012 will introduce output ports in checkers. The types of output ports may be the same as the types of input ports, except for **untyped**, **sequence**, or **property**. Output ports may have optional initializers. Introduction of output ports will significantly extend checker applicability and modularity. The main use cases for checker output ports are described below.

1) Feeding checker status into RTL

If a checker is synthesized on the chip, it is required to feed its status into RTL. This capability exists in OVL [3], and SVA 2012 will make it possible in checkers too, as shown in the following example:

```
checker mutex(input sig,
              event clk = $inferred_clock,
              output logic res);
   a1: assert property (@clk $onehot0(sig))
         res = 1'b1 else res = 1'b0;
endchecker

module m(logic clock, …);
   logic read, write, scan;
   always @(posedge clock) begin
      read <= …; write <= …;
      mutex check_mutex(
         {read, write},, scan);
```

```
      //…
    end
  endmodule
```

2) Modular assertion modeling

It is desirable to construct complex checkers in a modular way by combining several subcheckers. If a subchecker does assertion modeling, it needs output ports to return the modeling results, as shown in the following example.

```
checker check_fsm (
 // Concrete state
 logic [1:0] state,
 event clk = $inferred_clock);
 // Abstract state
 logic [1:0] astate;
 model_fsm c1(state, clk, astate);
 check_assertions c2(state, astate, clk);
endchecker

checker model_fsm (input logic [1:0] state,
 event clk,
output logic [1:0] astate = IDLE);
 always @clk
  case (state)
   IDLE: astate <= …;
   // …
   default: astate <= ERR;
 endcase
endchecker

checker check_assertions (
 state, astate, event clk);
 default clocking @clk; endclocking
 a1: assert property (astate == IDLE
   <-> state inside {IDLE1, IDLE2});
 //…
endchecker
```

In this example the top level checker `check_fsm` verifies the correctness of a finite state machine (FSM). It gets a concrete state from RTL and builds an abstract state machine and then verifies that the concrete FSM complies with this abstract FSM. The output port feature allows splitting this checker into two subcheckers: `model_fsm` that computes the abstract state and returns it via the output port `astate`, and `check_assertions` that checks compliance of the concrete state to the abstract one.

3) Implementing testbench environment with checkers

The mechanism of free variables and checker output ports make it possible to use checkers as testbench environments. The main advantages of these checker-based testbench environments are their high level of abstraction and their ability to be used in both simulation and formal verification. The classical testbench environments normally cannot be used in formal verification. The following example illustrates this concept.

```
checker env(event clk,
            output logic out1, out2);
 rand bit a, b;
 m: assume property (@clk $onehot0({a, b}));
 assign out1 = a;
 assign out2 = b;
endchecker : env

module m(input logic in1, in2, clock,
         output …);
 …
endmodule : m

module top();
 logic clock, n1, n2;
 …
 m m1(n1, n2, clock, …);
 evn env1(posedge clock, n1, n2);
endmodule : top
```

In this example the checker `env`, in simulation, generates random mutually exclusive inputs for module `m` at each tick of the clock. In formal verification, it constrains module inputs to be mutually exclusive.

G. Types of checker ports

Most limitations imposed on checker port types in SVA 2009 will be removed in SVA 2012. For example, in SVA 2012 it will be possible to declare checker arguments of real types or as dynamic arrays. The latter capability may provide a workaround for passing a variable number of arguments to a checker, as shown in the following example.

```
checker one_of (val, int values[]);
 a: assert #0 (val inside {values});
endchecker : one_of

module m(…);
 logic[2:0] state;
 …
 one_of state_legal(state,
                    '{IDLE, ACTIVE, WAIT});
 …
endmodule : m
```

In this example the legal states of an FSM are passed to the checker `one_of` using a dynamic array.

V. Main Challenges in Checker Definition

The new checker capabilities of SVA 2012 are similar to well-known features that have existed in modules for many years. However, introducing them in checkers was challenging because of the following checker peculiarities:

- Free variables
- Procedural instantiation
- Rewriting semantics for checker instantiation.

Below we describe the challenges of defining new checker constructs.

A. Checker NBA and always_ff procedures

SVA 2009 allowed only simple always procedures in checkers having an event control.[3] Aside from assertions, such

[3] Strictly speaking, it was possible for an always procedure not to have any control at all, but the semantics of such procedures was non-intuitive. In the emerging standard use of *always* procedures in checkers is deprecated, and

procedures could only have NBA statements. These NBA statements were executed in the Re-NBA region [2]. One of the main reasons to execute these assignments in the Re-NBA region was to allow having a sequence `triggered` method in the right-hand side (RHS) of the assignment. Indeed, the value of a sequence `triggered` method is only set in the Observed region, and an attempt to sample its value in the NBA region, as in modules, would result in the `triggered` value being identically false. This behavior of checker NBA will be preserved in SVA 2012.

B. Continuous assignments

Defining continuous assignments turned out to be a challenging task. Consider what happens if an RHS of a continuous assignment contains a free variable:

```
default clocking @clk; endclocking
rand bit r; bit a, b;
m1: assume property (@clk r == b);
assign a = r;
```

The free variable `r` gets the sampled value of `b` from the assumption `m1`. Therefore, the new value of `a` equals the old value of `b`. The only way to preserve the combinational behavior of a continuous assignment is to make its RHS sampled, and to sample checker variables in all contexts, including deferred assertions! This behavior is undesirable, as explained in Section IV.E. To break this vicious circle it was decided to disallow free checker variables in an RHS of a continuous assignment and thus to keep a conventional definition of a continuous assignment. There were two options: to perform continuous assignments in the Active region set, as in modules, or in the Reactive region set, as in programs. The latter option was selected for considerations of consistency and efficiency:

- Since checker NBA are performed in the Re-NBA region (see Section A), it is more consistent and efficient to perform continuous assignments in the Reactive region set.
- With the introduction of output ports (see Section IV.F), checkers may act as signal generators, playing the role of a testbench. Since SystemVerilog testbench constructs (programs) are executed in the Reactive region set, checker assignments should follow this rule.

The issue remains: what to do with checker free variables? The SVA 2012 solution merely rules them out instead of addressing the problem! However, some provision has been made for the future. Currently, the target of a continuous assignment cannot be a free variable. Such assignment may be introduced in the future, and this assignment would allow free variables in the RHS as well. This definition would require all non-free variables in the RHS to be sampled. Such a definition will be consistent with the rest of the language: the free variables are not sampled, but they depend on sampled values of non-free variables.

always_ff is intended to be used instead. In addition, *always_comb* and *always_latch* procedures have been introduced.

C. Procedural control and looping statements. Sampling

Defining simulation semantics for continuous assignments paved the way for the introduction of **`always_comb`** and **`always_latch`**: all statements in **`always_comb`** and **`always_latch`** are executed in the Reactive region set.[4] A problem arises with the introduction of procedural control and looping statements: should the control expressions be sampled or not? On the one hand, expressions in procedural statements should not be sampled:

```
always_comb if (a) x = b; else x = c;
```

should be equivalent to

```
always_comb x = a ? b : c;
```

On the other hand, if NBA are in a scope of a conditional statement, the condition should be sampled:

```
always_ff if (a) x <= b;
```

should be equivalent to

```
always_ff x <= a ? b : x;
```

It follows that the variable sampling should depend not on the statement, but rather on the procedure: in **`always_ff`** procedures variables should be sampled, whereas in **`always_comb`** and **`always_latch`** they should not.

However, there are additional complications for **`always_ff`** procedures. Consider the following example:

```
always_ff @clk begin
  b = a; c = b; ...
end
```

Since variables in an **`always_ff`** procedure are sampled, the behavior of the blocking assignments above is non-intuitive: they are actually nonblocking because of the sampling of their RHS. To cope with this problem, blocking assignments have been made illegal **`always_ff`** procedures in checkers.

Another complication is caused by variables in an event control. Consider the following example:

```
always_ff @(posedge clk or posedge rst)begin
  if (rst) ...; ...
end
```

To make reset work as expected, the variable `rst` should not be sampled. The following rule has therefore been added: variables used in event control are not sampled.

VI. Checker Features Yet to Be Addressed

There are several important features still to be addressed: checkers in functions and tasks, checkers in classes, forcing in checkers, and checkers with a variable number of arguments. In this section we discuss these features and their usability.

[4] Except for concurrent assertions, which preserve their regular semantics.

A. Checkers in functions and tasks

Checkers currently cannot be instantiated in functions and tasks. This prevents generic usage of checkers as building blocks of verification libraries. For example, assume that there is a library checker checking mutual exclusiveness between two signals

```
checker mutex(logic a, b);
  assert #0 ($onehot0({a, b}));
endchecker
```

This checker may be used in modules (interfaces, etc.), but not in functions or tasks. If this check is required in a function or task, the assertion has to be instantiated there directly or wrapped into a macro.

The main issue preventing the introduction of checker instantiation in functions or tasks are statements that can appear in checkers, but not in functions or tasks. For practical needs it would be possible to limit the constructs allowed in a checker instantiated in the context of a function or task. However, a solution needs to be found for generate constructs which are essential for library checkers.

B. Checkers in classes

Checking execution correctness is an important part of UVM methodology [5]. In UVM, correctness checking is performed by special classes called monitors. However, classes do not have optimal infrastructure for correctness checking: most importantly, class methods cannot instantiate concurrent assertions. Checkers have everything needed for assertion checking, but they cannot be instantiated in classes. Therefore, it would be helpful to allow checkers to be class members, and to start their assertion checking at the time of class construction. This feature would make it possible to reuse the same checker in different contexts, both RTL and transaction level.[5]

C. Forcing in checkers

For the purposes of formal verification it is important to prune parts of the block under verification: to disconnect some logic or to hardwire it to some specific value. Pruning currently is done either using tool-specific directives or by manually changing a copy of a block. Both approaches have major drawbacks: pruning directives are non-standard and are not supported by simulators; changing the block manually requires effort and is error-prone.

Allowing forcing in checkers would solve this problem, as shown in the following example:

```
module m(...);
  logic a;
  assign a = ...;
  // ...
endmodule : m

checker c;
  initial force top.m1.a;
endchecker

module top;
  // ...
  m m1(...); c c1(...);
endmodule
```

[5] Transaction-level modeling also requires defining features for transaction-level assertions, see [16].

D. Variable number of checker arguments

The number of checker arguments must be fixed, even though some arguments can have default values. To build flexible library checkers it would be useful to allow a variable number of checker arguments. As an example, consider a checker verifying that signal sequences `seq_1`, `seq_2`, …, `seq_n` come in a specific order. Currently one has to define different checkers for different numbers of sequences. It would be more natural to have one checker that admits a variable number of arguments. This capability can be partially implemented in SVA 2012 using dynamic arrays as explained in Section IV.G. However, dynamic arrays cannot contain, for example, sequence arguments, so that the case described in this section cannot be implemented in SVA 2012.

VII. Conclusion

The new checker constructs introduced in 2009 close some key gaps in the SystemVerilog standard. We have described numerous features of checkers that are well-suited for building flexible, maintainable verification libraries:

- Instantiation in module or procedural code
- Packaging of properties and modeling code
- Typed or untyped inputs that can be variables, events, sequences, or properties
- Context inference for clocks and resets
- Substitution semantics for flexibility and efficiency
- Insensitivity to races

Since SVA 2009 was issued, the design community has found additional improvements to be desirable. The upcoming 2012 revision of the standard will introduce features including more general procedures, relaxation of the single-assignment rule, non-sampled checker inputs, output arguments, etc. In future revisions of the language we hope to add additional features such as signal forcing, and to extend checkers to other SystemVerilog constructs such as functions, tasks, and classes.

Based on our experience supporting the validation of a variety of Intel CPUs and SoCs, we believe that the checker construct will play a key role in future simulation and formal verification environments. As their usage in the validation community continues to grow, we expect to further refine the definition of these new, integral building blocks of our verification infrastructure.

Acknowledgment

The authors would like to thank Paul Inbar for reviewing this paper.

REFERENCES

[1] Ping Yeung, "Four Pillars of Assertion-based Verification," in *Euro DesignCon*, 2004.

[2] "IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification," IEEE STD 1800-2009, 2009.

[3] "Accellera Standard Open Verification Library (OVL) V2.5," Accellera, 2010.

[4] Eduard Cerny, Surrendra Dudani, and Dmitry Korchemny. (2010) SVA Checker Library. [Online]. http://www.accellera.org/apps/org/workgroup/ovl/download.php/3512/SVACheckerLibrary_101006dk.pptx

[5] Universal Verification Methodology. [Online]. http://uvmworld.org/

[6] Eduard Cerny, Surrendra Dudani, Dmitry Korchemny, and Lisa Piper, "Verification case studies: evolution from SVA 2005 to SVA 2009," in *DVCon*, 2009.

[7] Eduard Cerny, Surrendra Dudani, John Havlicek, and Dmitry Korchemny, *The Power of Assertions in SystemVerilog*.: Springer, 2010.

[8] "IEEE Standard for Property Specification Language (PSL) ," IEEE 1850-2005 , 2005.

[9] Doron Bustan, Dmitry Korchemny, Erik Seligman, and Jin Yang, "SystemVerilog Assertions: Past, Present and Future," *IEEE Design and Test of Computers*, no. To appear, 2012.

[10] Eduard Cerny, Surrendra Dudani, and Dmitry Korchemny, "IEEE 1800-2009 SystemVerilog: Assertion-based Checker Libraries," in *DVCon*, 2010.

[11] Adrian J. Isles, Jeremy Sonander, and Mike Turpin, "AMBA Compliance Checking," in *DesignCon*, 2005.

[12] OCP web page. [Online]. http://www.ocpip.org

[13] AMBA Open Specifications. [Online]. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[14] Andrea Fedeli, Matteo Moriotti, Umberto Rossi, and Franco Toto. (2004, February) "Addressing IP Reuse With Formal Verification and Assertion Based Verification", Design and Reuse. [Online]. http://www.design-reuse.com/articles/9511/addressing-ip-reuse-with-formal-ver

[15] Robert Adler, Sava Krstic, Erik Seligman, and Jin Yang, "CompMon: Ensuring Rigorous Protocol Specification and IP Compliance," in *DVCon*, 2011.

[16] Wolfgang Ecker, Volkan Esen, Thomas Steininger, and Michael Velten, "Requirements and Concepts for Transaction Level Assertions," in *IESS*, 2007.