SVA4T: SystemVerilog Assertions - Techniques, Tips, Tricks, and Traps

Wolfgang Ecker, Volkan Esen, Thomas Kruse, Thomas Steininger
Infineon Technologies

Peter Jensen
Syosil Consulting

Abstract

ABV (Assertion Based Verification) is a very promising approach to cope with the continuously increasing verification gap. ABV works on the interface between the designer and the verification engineer. This paper gives an overview of coding and applying SVA (SystemVerilog Assertions) and gives some hints on language usage.

First, a technical introduction on SVA is given in the reverse order of the LRM (i.e., starting with assertions, and then going on with properties, sequences and Boolean expressions). Next, SystemVerilog constructs and features that support the application of SVA are presented. Additionally, further SystemVerilog constructs are shown, which ease the use of SVA. They are mainly the system calls for controlling the execution of assertions and for displaying assertion messages. The next part copes with building assertions. Issues such as 'X'-handling, consideration of reset, and pipelined vs. sequential behavior is discussed. SystemVerilog implementations of the OVL, as distributed with *vcs* in source code are discussed under those aspects. A set of tips and tricks resulting from our application of SVA and a set of coding rules derived from that and potentially already implemented in LEDA are shown afterwards.

The paper concludes with a summary of benefits that arise from the joint design and assertion language SystemVerilog.

Introduction of SystemVerilog Assertions

Assertions

Concurrent assertions are the work horses of the assertion notation. They must be clocked, either by specifying a clock edge with the assertion or by deriving a clock edge specification from a surrounding statement. Clocking is the key, because the evaluation of concurrent assertions starts every clock cycle with a new incarnation of a checker. Assertions can be suspended during asynchronous resets using the **`disable-iff`** construct. Below, a first assertion is shown. It requires that when **`a`** gets the value **`1`** that **`b`** takes the value **`1`** one cycle later and that **`c`** takes the value **`1`** one additional cycle later or that **`reset`** got the value **`0`**:

```
assert property(@(posedge clk)
   disable iff (!reset)
   a |=> b ##1 c);
```

Code 1: First Assertion

Concurrent assertions can also occur in sequential code. Here, they derive the synchronization and activation from the surrounding always block. An example is shown in Code 2. Here, the concurrent assertion is placed inside a case statement. The check of the assertion is performed every rising edge of the clock and is started only if the branch **`s0`** of the case statement is executed.

```
always_ff @(posedge clk)
   case (state)
    s0 :
      state <= s1;
      assert property
         ($past(state) = s4);
```

Code 2: In-lined assertion (code fragment)

Besides concurrent assertions, SystemVerilog also supports immediate assertions. They must be placed in sequential (procedural) code. Their execution is driven by the control flow of the surrounding procedural statements. Because of their sequential execution, sequential assertions are often seen as a small counterpart of concurrent assertions. However, sequential assertions are applied in several areas:

- In initial blocks to check module consistency and
- to model complex assertions, which cannot be modeled using concurrent assertions

A code fragment showing the application of immediate assertions is presented in Code 3. The need for immediate assertions to model specific sequential properties is discussed later.

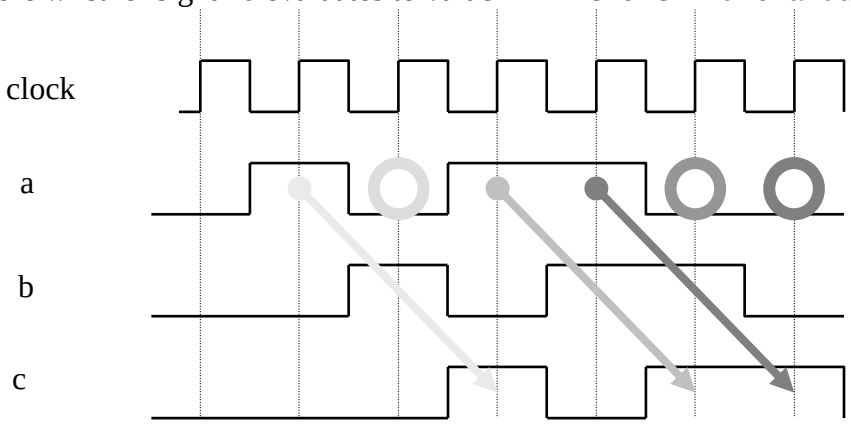
```
initial
   assert (N == M+1) else
      $error("Parameter ...");
```

Code 3: Immediate assertion (code fragment)

It will be shown later that assertions can be also built iteratively on properties, sequences, and Boolean expressions, either by direct use or by instance.

Assertion Semantic

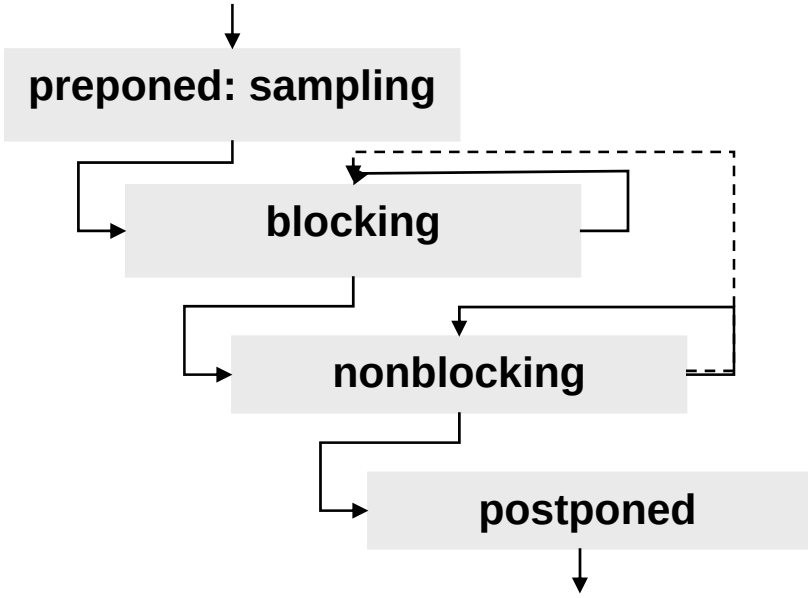
As already stated, the execution semantic of assertions is defined in the way that evaluation of assertions starts every clock cycle with a new incarnation of a check. From the start point of a check, the check is postponed into the future until a definite conclusion on pass or fail can be made.
This is shown in Waveform 1 illustrating the analysis of the assertion from Code1.
As shown, the analysis starts every clock cycle. Every time when signal **`a`** shows the value **`0`**, the evaluation of the check passes successfully already in the first cycle. The implication is true in this case. Only if signal **`a`** shows the value **`1`**, must the evaluation be continued and thus be checked whether signal **`b`** has the value **`1 in the second evaluation cycle`** and furthermore whether signal **`c`** evaluates to value **`1 in the third evaluation cycle`**.



`Waveform 1: Concurrent assertion (code fragment)`

In addition, a sampling semantic is defined to avoid glitches in checking signals. In contrast to VHDL , which provides or at least allows postponed processes[1], SystemVerilog requires sampling of values that are sampled with the clock associated with the assertion[2]. To avoid signal races, the Verilog simulation algorithm has been extended slightly. A simplified SystemVerilog simulation algorithm is shown in Picture 1. Two major extensions are shown: Values evaluated in an assertion, or used in a **`$sampled()`** construct, or used in a clocking construct are sampled before blocking and nonblocking statements. Assertion checks are executed after blocking and nonblocking statements and use sampled values only. In this way, hazards or races, where clock active edge and the signals used in an assertion change in the same cycle, can be avoided; however, the current value of a signal may have a different value than used during the assertion check.

[1] Postponed processes are processes that are executed after all delta cycles (0-delay cycles) have been simulated.
[2] In the simple case, a clock is immediately associated with an assertion. However, also other possibilities exist.



Picture 1: SystemVerilog Simulation Algorithm (simplified)

Properties

Assertions and Properties

In Code 1, an assertion is shown that directly includes the property specification, which shall be asserted. SystemVerilog also allows declaring a property separately. Furthermore, an assertion can instantiate the property. The assertion shown in Code 4 is split into an assertion with separate property declaration. In this case, the clock is defined with the assertion and passed to the property for the definition of clocked delay (**|=>, b##1**).

```
assert property(@(posedge clk) my_prop);

property my_prop; disable iff( ! reset)
      a |=> b##1 c; endproperty
```

Code 4: Assertion instantiating a property (clock by assertion)

Alternatively, as shown in Code 5, a clock may be defined with the property. In this case, the assertion need not have its own clock specification. In case, the assertion would have its own clock specification also, it must be the same as the clock specification of the property (same edge, same signal).

```
assert property( my_prop );
property my_prop;
   @(posedge clk)
      disable iff (! reset)
      a |=> b ##1 c; endproperty
```

Code 5: Assertion instantiating a property (clock by property)

Summarizing, concurrent assertions can include properties and sequences directly or can instantiate them.

Properties
Properties must be clocked either by a separate clock specification or by a clock specification that is passed to the property. A property can be reset asynchronously using the disable-iff construct. To express behavior, properties can include sequences directly or instantiate properties and sequences, which is shown later. But properties are not executed *autonomously*! They need an assertion, cover, assume, or expect statement, which starts evaluation of the property and which gives semantic to the result of the evaluated property.

Properties Using Properties
Properties have several alternatives for structuring. One of them is the use of properties in properties. Here also a clock specification can be passed from the instantiating property to the instantiated property.
Properties instantiating properties allow specification of recursive (i.e., potentially infinite) properties.
An additional possibility for structuring properties is to make use of different composition operators (see Code 6). Here, an enabling sequence **sx** implies a property **py** to be checked for true. The non-overlapping implication operator **|=>** defines that the enabling sequence and checked property do not overlap in time but follow subsequently.

```
property my_prop; @(posedge clk)
   disable iff( ! reset )
      sx |=> py; endproperty
sequence sx; a; endsequence
property py; b ##1 c; endproperty
```

Code 6: Properties instantiating a property

Implications
Not only one but several implications are supported in SystemVerilog. Three of them are shown in Code 7. All of them have at least one operand, which is a property.
First, the non-overlapping implication operator **|=>** is shown in property **my_prop1**. Next, the overlapping implication operator **|->** is shown in property **my_prop2**. Here, the last clock cycle of the enabling sequence **sx** and the first cycle of the checked property **py** overlap. If more or full overlap is needed, the implication can be composed according to Boolean algebra from **not** and **or**. This is shown in property **my_prop3**. Here, sequence **sx** can also be a property (e.g., **px**) in **my_prop3**.
Generally speaking, **not**, **and**, and **or** operators[3] can be applied to properties in any order.
Properties can also be multiplexed using an **if**-construct. In this case the condition must be a Boolean expression and cannot be either a sequence or a property.

```
property my_prop1; @(posedge clk)
   disable iff( ! reset )
      sx |=> py; endproperty

property my_prop2; @(posedge clk)
   disable iff (! reset)
```

[3] Note the notation **and**, **or**, … ! It is used to clearly separate from Boolean operations (**&&, ||**) and to avoid ambiguity.

```
      sx |-> py; endproperty

property my_prop3; @(posedge clk)
   disable iff ( ! reset)
      (not sx) or py; endproperty
```

Code 7: Implications

In SystemVerilog, like in logic implications, the enabling sequence is also called antecedent and the checked property consequent.

Properties Using Sequences

Properties using sequences have already been shown above. Properties that use in-lined sequences are basic building blocks of temporal relations. This is shown in Code 8. The property **my_prop** in Code 8 is equivalent to the property **my_prop** in Code 4.

```
property my_prop; @(posedge clk)
   disable (iff ! reset)
      sx |=> sy; endproperty
sequence sx; a; endsequence
sequence sy; b ##1 c; endsequence
```

Code 8: Properties instantiating a sequence

Sequences

Simple Sequences

Sequences can themselves be built from sequences or from Boolean expressions (abbreviated with **bx**, **by**, and **bz**, subsequently). In the simplest case, sequences can be composed from a list of Boolean expressions ordered increasingly according to clock cycles. An example would be **by ##1 bz**.

##1 or generally **##N** (whereas **N** is a positive finite number) is used to specify relative distance in terms of clock cycles. So **by ##5 bz** specifies **by** to be true and five cycles later **bz** to be true. Also cycle ranges can be specified in the form of **##[N:M]** . Here, **N** must be also a positive finite number and **M** must be a positive potentially infinite number (described with **$**). Examples are **by ##[1:5] bz** or **by ##[1:$] bz.**

Also several kinds of repetition can be specified, which however is not discussed in this paper.

Sequences Using Sequences

As already mentioned, sequences can be composed also from sequences. The simplest way is the concatenation, which for example can be described as **sx ##1 sy.** In addition, sequences can be and-connected in several ways.

- They can be and-ed by using **sx and sy.** Here, the longest match of both sequences is checked.
- They can be and-ed by using **sx intersect sy**. Here, **sx** and **sy** must have exact the same time window length.

- They can be and-ed by using **`sx within sy`**. Here, the time window of the second sequence **`sy`** completely covers the time window of the first sequence **`sx`**.

Sequences can also be or-ed. This can be specified using **`sx or sy`**. Here, the shortest match is checked. Finally, sequences can be combined with Boolean expressions. For **`bx throughout sx`** the Boolean expression must be true through sequence **`sx`**.

Boolean Expressions

All Boolean expressions can be used in sequences, properties, and assertions. They include Boolean/logical operations, shift/rotate operations, arithmetic operations, slices/indexed values, hierarchical and selected names. Boolean interpretation is taken from any value. This interpretation is mostly straight forward; special care must be taken, when an **X** occurs. This **X** is implicitly mapped to **false**.

Inside Boolean expressions, some temporal operators can be used to allow analysis of values from previous samples. They are: **$past(b)**, **$past(b , N)**, and **$sampled(b).** Also special functions are provided. They are **$onehot(b)**, **$onehot0(b)**, **$isunknown(b**). The first function **$onehot(b)** returns true, if **b** is one-hot encoded. The second function **$onehot0(b)** also checks for one-hot encoding or the value **0**. The third function returns **true** (respectively **1**), if **b** contains at least one unknown value.

SystemVerilog Support for Application of Assertions

Design Assertions

The applicability of assertions is of importance for their use and usability. As already mentioned, concurrent assertions can be specified inside modules and inside always blocks, this means, in concurrent and in procedural environments.

This kind of notation supports the specification of design-related assertions. Unintended behavior, assumptions on interface signals, design invariants, corner cases (max-and-min values as well as corner case behavior such as full or empty FIFOs), and internal interfaces shall be asserted. Repeating the RTL functionality in assertions does not provide any additional benefit. Subsequently, the application of assertions in a concurrent environment is shown.

```
module my_fsm
(  input bit clk,
   input bit ret,
   input bit[1:0] status,
   output bit [3:0] control)
bit [2:0] state;

assert property(@(posedge clk) // invariant
   $onehot( state ));
assert property(@(posedge clk)
   (state==3'b001 && status==2'b10) |=> state==3'b010); // bad style
assert property(@(posedge clk)
   (state==3'b001 && status!=2'b10) |=> state==3'b001); // bad style

always_ff @(posedge clk)
begin
   if( state==3'b001 && status==2'b10)
```

```
      state <= 3'b010;
   else
      state <= 3'b001;
end
endmodule
```

Code 8: Concurrent assertions (incomplete model)

The first property specifies a design invariant (i.e., something that holds in any case). Here, the one-hot encoding of the state signal is asserted. It is important to note that the invariant cannot be directly seen from the implementation.
The second and third assertions specify design properties that are shown almost identically in the implementation below. These assertions show non-recommended applications.

External Assertions

To specify functionality and assertions in a design-independent way, SystemVerilog introduces program blocks. Furthermore, SystemVerilog introduces clocked regions that specify sampling of signals that are accessed in the program blocks[4]. These clocked regions can also force sampling of values in the statement part of an assertion and make $sampled superfluous here. Clocked regions must specify a clock that can be passed to un-clocked assertions. In case assertions, properties, or sequences possess their own clock specification, they must be identical with the clock specification of the clocking block.
Assertions in program blocks specify either design assertions that check signals in different blocks (e.g., invariants between buffer pointers), or verification-related assertions (e.g., protocol checks or protocol parsers for testbenches).

```
program test;
default clocking clk1 @(posedge clk)
   input bit[3:0] control;
   output bit[1:0] status;
end clocking;

assert property
    ( (status==2'b11) |=>
    ( control==4'b1000 ##1
      control==4'b0100 ##1
      control==4'b0010 ##1
      control==4'b0001 ) ) else
   $warning("%m illegal c %b",
       control));
```

Code 9: Assertions in program blocks (incomplete code)

Code 9 shows an application of assertions in a program block. Here, the property checks a sequence of control signals that should follow after the **`2'b11`** status.

Interface Assertions

[4] Clocked blocks allow also to specify a delay between variable assignment in clocked blocks and visibility of that assignment outside the clocked block. However, this is not relevant for assertions.

Furthermore, SystemVerilog supports annotation of assertions in interfaces. This allows the specification of checks on signals and their temporal relationship in the interface. These checks are performed independently for all instantiations of the interface. Doing so, the protocol of the interface can be specified in a formal way.
Assertions in interfaces result from protocol specification. They support protocol-specific re-use of assertions. These assertions also protect protocol-specific implementation in modules connected to the interface.
Code 10 shows an invariant of an interface **bus**, which checks the exclusivity setting of signals of the array **sl_ack**.

```
interface bus ();
   parameter int no_slaves = 0;
   dataword ma_data;
   addrword ma_addr;
   bit ma_req, ma_ack;
   bit [no_slaves-1:0] sl_ack;
   bit [no_slaves-1:0] sl_req;

   assert property(@(posedge clk)
      $onehot0( sl_ack ));

endinterface
```

Code 10: Assertions in interface

Bound Assertions
Finally, SystemVerilog supports binding of assertions specified in program blocks, interfaces, or modules to other program blocks, interfaces, or modules. Doing so, assertions supporting verification of one module can directly be bound to that module. Hierarchical names can be avoided in that way. So assertions can be bound independently to the place in the module to be checked in the design hierarchy. Code 11 shows an example for binding.

```
bind my_fsm test test_my_fsm
   (clk, control, status);
```

Code 11: Binding assertions

Bindings could also be used to separately specify a protocol implementation and separate checks to that protocol implementation.

SystemVerilog Constructs Supporting Use of SVA

Dynamic Assertion Control
SystemVerilog includes system calls for controlling the execution of assertions during simulation. They allow execution to stop (**$assertoff**), re-start (**$asserton**), and cancel assertions (**$assertkill**). Assertions can be controlled generally or assertions can be controlled by specifying a hierarchy, or a specific assertion. An example would be **$assertoff(m1.a1,m2)**.

Dynamic control of assertions can be used to turn off assertions during reset and initialization or during simulation of erroneous protocols. Doing so, expected fail-checks can be turned off.

Static Assertion Control
One way to control assertions statically is to make use of SystemVerilog's pre-processing capabilities. An example is shown in Code 13. Assertions are already ignored during compilation. Different models can be built in this way. Pre-processing pragmas can be applied in any place in the code and thus allow for general assertion control.

```
'ifdef ASSERT_ON
assert property (@(posedge clk)
   disable iff (!reset)
   a |=> b ##1 c);
'endif;
```

Code 13: Assertion statements controlled by pre-processor

The other way to statically control assertions is to make use of the generate statement. Here, assertions are compiled but not considered during elaboration of the model. The Boolean expression of the if-generate statement can depend on parameters or constants. An example is given in Code 14.

```
generate
if( my_assertions_on )begin:my_label
assert property (@(posedge clk)
   disable iff (!reset)
   a |=> b ##1 c);
end;
endgenerate;
```

Code 14: Assertion statements controlled by generate statement

Generate statements can only be used to control assertions in a concurrent region. In a procedural region, the sequential if-statement must be applied.
Static control of assertions is used to speed-up simulation by turning off checks or to select the appropriate assertion model for a given problem.

Assertion Messages
Assertion violations are reported by the simulator automatically. If additional messages are desired, a set of system calls is provided. They give different kinds of messages:

- information (**$info**)
- warning (**$warning**)
- error (**$error**)
- failure (**$fatal**)

They also control the execution of simulation. To make their use more general, a task for general message handling is provided. This task is parameterized with the desired message. Even better would be the use of a message handler in the testbench environment.
Messages can be combined with assertions in the statement part that is associated with every assertion. Statements can be added for the pass and for the fail of an assertion check. Both code

parts are separated by an **else**. An example is given in Code 12. In case of passing the check, the message **property p1 is passed** is output by the simulator. In case of failing, **error** preceded by the hierarchical name of the assertion (specified by **%m**) is output

```
assert property(@(posedge clk)
   disable iff (!reset)
   a |=> b ##1 c )
   $info( "property p1 passed"); else
   $error( "%m error");
```

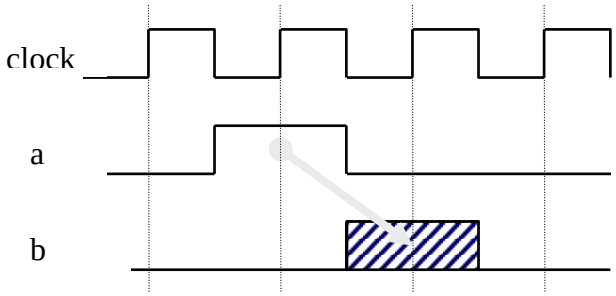
Code 12: Assertion statements messaging checking results

Issues Building Assertions

When building assertions, a set of basic principles must be considered. The most important ones are listed in this section.

'X'-Handling

Assertion checks must return a pass or fail. An un-decidable alternative, similar to un-defined values, is not possible. For that purpose, the value 'X' is interpreted as '0'.



Waveform 2: Assertion with 'X'-value

As shown in Waveform 2, the implicit **X** mapping can hide a potential assertion violation or can let pass a potential assertion failure. It is strongly recommended to check occurrence of **X** explicitly and print a message in that case. Two alternatives for doing this is shown in Code 15.

```
assert property(@(posedge clk)
   (a!='X' && b!='X'));

assert property(@(posedge clk)
   ! $isunknown( a ^ b );
```

Code 15: Assertion statements checking for undefined values

In case, only bit values are used in the model, then the check shall be turned off. If bit/logic value representation can be configured in the model, the **X** check control shall be combined with that configuration.
Furthermore, **X** checks shall be dynamically turned off before or during reset.

Consideration of Reset,

Reset must be considered when modeling assertions. Unintended error messages before and during reset and synchronization of the sequences with the model after reset must be guaranteed. Reset must be considered in the assertion in the same way as it occurs in the design.
Asynchronous reset is supported directly using the **`disable iff`** construct. An example for the use of that construct is given in Code 1.
If synchronous reset must be considered, two alternatives can be used generally. One is shown in Code 16. Here, the asynchronous reset signal is synchronized, so that it applies at the **`disable iff`** synchronously with the clock, only.

```
always @(posedge clk)
   sync_reset <= reset;
assert property(@(posedge clk)
   disable iff !sync_reset
      a |=> b ##1 c);
```

Code 16: Assertion statements with synchronized reset

Another alternative is the explicit consideration of reset in the sequence. The re-coding of the assertion of Code 1 to cope with synchronous reset is shown in Code 17.

```
assert property (@(posedge clk)
  a && !sync_reset |=>
     (b ##1 c) or (##[0:1] sync_reset));
```

Code 17: Assertion statements with explicitly modeled reset

In case, properties do not need explicit synchronization with reset, they also can be turned off before and during the reset and turned on after reset. The required system functions were already presented above.

Kind of Pipelined Behavior

A key issue when modeling temporal relationships is the pipelined, partially pipelined, or non pipelined (sequential) nature of the behavior to be asserted.
Pure non pipelined behavior can either be described using concurrent assertions, as shown in Code 1, or using immediate assertions embedded in always blocks or tasks. The check of a non pipelined behavior using immediate assertions is shown in Code 18.

```
always forever begin
   @(posedge clk);
   if (!a) continue;
   @(posedge clk);
   assert b;
   @(posedge clk);
   assert c; end
```

Code 18: Immediate assertions checking an un-pipelined protocol

In case, the model showed pipelined behavior, this would be ignored by the check. If pipelining is illegal (i.e., the absence of pipelining must be checked also), procedural code with immediate

assertions can be used. Alternatively, some behavioral code can be added and several concurrent assertions can be formulated. The latter is often called a checker in the area of assertion based verification.
The check of the same temporal relationship is defined in Code 18. A check for illegal pipelining is shown in Code 19.

```
always forever begin
   @(posedge clk);
   if (!a) continue;
   @(posedge clk);
   assert b and !a;
   @(posedge clk);
   assert c and !a; end
```

Code 19: Immediate assertions checking an un-pipelined protocol and checking illegal pipelining

Not shown is that immediate assertions together with procedural code can specify more complex checkers than concurrent assertions alone. The potential mix of procedural code, immediate assertions, and concurrent assertions give SystemVerilog sufficient power to model all required checks.

Coding Rules

Several coding style rules were defined during our use of assertions. They are collected in this section. Some of them were already mentioned in the sections above.

- *Design code shall not be repeated in assertions!*
  This might help to decide on the use of concurrent assertions or immediate assertions for assertion checks. The latter often has the danger of repeating design code.
- *Explicit naming is needed to allow for fine-granular control!*
  This is needed to specifically apply the system tasks controlling execution of assertions. Later, the assertion name is used in the assertion message if available.
- *Occurrence of 'X' shall be checked explicitly!*
  This is because 'X' is implicitly mapped to '0', which might hide a failure of an assertion check.
- *Assertions must consider reset in the same way as it occurs in the design!*
  This is needed to synchronize assertions properly with the model. Otherwise, one clock-cycle offset occurs. This offset mostly produces illegal checks.
- *Assertion checks shall use the same clock as the design, which drives evaluated variables!*
  Otherwise races between design and assertions can occur. Synchronization does not work properly in this case.
- *It is recommended to avoid unlimited assertions that are valid for recursive and infinite intervals!*
  The reason for this rule is that there is different interpretation of (some) formal and simulation-based tools for this kind of assertion. Because the check might finalize or not, covering but asserting unlimited properties might be an alternative approach.

- *Make assertion messages as clear as possible and use message tasks to let the simulator output the assertion message!*
  This is mainly to improve debugging after failure of assertion checks.

Summary and Conclusion

A short tutorial introduction to SystemVerilog Assertion has been given. Key issues for using assertions have been shown.
Summarizing, SVA proved to be a powerful assertion language. It provides several ways to control assertions, several alternatives to annotate code with assertions or to add assertions to code, and several levels for notation: assertions, properties, sequences, and Boolean expressions. Combined with other SV-HDL constructs, SVA notation becomes even more powerful.