SystemVerilog Assertion Linting: Closing Potentially Critical Verification Holes

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

***Abstract*—When developing a new system, it is important to confirm that the system conforms to documented requirements and supplies specific features. In verifying this, reliance on SystemVerilog Assertions (SVA), the assertion specification subset of the SystemVerilog (SV) language, has grown in recent years. There are many advantages to using SVA in design and verification: they are natively integrated into the language, they may be checked in both simulation and formal verification, and they are convenient for designers to use while coding. However, misuse of SVA and/or failure to express a requested behavior properly may lead to verification problems. This is not just a theoretical hazard; in recent Intel projects there have been numerous cases in which SVA assertions were written that either failed to match user intent, failed to be checked at all, or caused major performance degradation in simulation or formal verification. Many of these cases could have been detected early by "linting", that is, performing preprocessing or compile-time checks to detect constructs that, while formally legal, might cause correctness or performance issues. We present here lint rules we helped develop to minimize the number of verification holes caused by common mistakes.**

Keywords- SystemVerilog, Assertions, Lint, Formal Verification, Simulation, Validation

I. INTRODUCTION

As an early adopter of SystemVerilog (SV) [6], Intel was one of the pioneering users of SystemVerliog Assertions (SVA) and has benefited from the advantages SVA offers in design and verification. To enable wider usage among designers and validators, Intel-internal usage was enhanced with the SVA Checker Library (a wrapper library, recently donated to Accellera) which further enables designers to efficiently include many commonly-used assertions in their RTL to aid in dynamic simulation and formal verification (FV). Overall, SVA has been critical to the effective and timely validation of many recent Intel projects.

When SVA was first deployed with the wrapper library, it was thought that enclosing assertions in a library would prevent most potentially dangerous misuse. However, as increasing numbers of engineers integrated SVA into their design processes, it became apparent that in many real-life cases assertions were written that did not behave as the designer intended (see [7]). All too often these types of misuse led to many wasted hours of debugging, bad performance hits, or even “false positives”, where incorrect simulation values that should have been detected were missed and only caught by chance much later on in the design and validation flows. While the library prevented some types of language misuse, other cases needed a different solution.

To help ensure the correctness of RTL code, most modern design projects throughout the industry use "linting", that is, preprocessing checks to detect constructs that are technically legal but may cause correctness or performance issues. By carefully examining and understanding escapees observed in real projects, we were able to develop a number of new lint rules for future projects. While a number of commercial tools provide SV lint features, most do not supply rules for detailed coverage of subtle SVA usage issues. We found that by taking into consideration real errors found by real projects, we could identify additional, potentially very effective, rules for preventing the recurrence of such errors.

The rules span a number categories, including assertions that do not correctly state the user’s intent, “void” assertions or ineffective coverage specifications that have no actual effect, and efficiency impacts on simulation or formal verification. Most of these do not represent truly illegal use of the language, and therefore all these lint rules should be waiveable, as they flag code which is standard-compliant and in rare cases can match user intent. But in each of these cases it is much more likely that the assertion code does not function as intended, and it is important to review it in order to reduce the danger of false positives, performance issues, coverage holes, and related issues.

Ultimately, the integration of SVA has been a strong win for Intel. It has dramatically improved our validation environment as well as strengthened integration between validation and design. With careful linting as described in this paper, we hope to continue strengthening our validation coverage and making our SVA usage even more successful.

In the examples of concurrent assertions given in this paper, the clock and reset are often omitted for the sake of clarity. In such cases it is assumed that the assertion belongs to a scope of some `default clocking` and `default disable iff`.

II. LINT RULES

The following table summarizes the major lint rules we have added based on project experience. The rules fall into three major categories:

1. **Wrong functionality**: This category covers cases where the logical conditions checked by the assertion probably don't match user intent.
2. **Possibly ignored assertions**: These are assertions that always pass when checked, regardless of signal values, or cover a condition in a trivial way.
3. **Performance Hazards**: These assertions are legally stated and do match user intent, but can cause major performance degradation in a simulation or formal environment.

A.	**Wrong Functionality**
1.	Assertion active at both clock edges
2.	Sequence used as clocking event
3.	Complex Boolean expression used for clock
4.	Wrong argument type or size
5.	`$stable(sig[index]))` with variable index
6.	Non-sampled value in action message
7.	Property uses negated implication
B.	**Possibly ignored assertions**
1.	Short-circuitable function has assertion
2.	Action block with no system function
3.	Unbounded assertion always true due to weakness
4.	Implication (|->,|=>) in cover property
5.	Bad comparison to unknown
6.	Assertion with constant clock
C.	**Performance Hazards**
1.	Many instances of single assertion
2.	Assertion in loop not using index
3.	Large or distant time windows
4.	Unbounded time/repetition operator in antecedent
5.	Using cover sequence rather than cover property
6.	Applying `$past` to multiple terms of expression
7.	Antecedents with empty match

The interesting cases we observed in Intel projects that inspired each type of lint rule are summarized in the following subsections.

A. Wrong Functionality

These rules check cases where the logical conditions checked by the assertion most likely do not match user intent. In general, such cases arise from the complexity and flexibility of the SVA language: it is relatively easy to write expressions whose semantics do not match user intent. If these assertions go uncorrected, a dangerous verification gap may result.

1) Assertion active at both clock edges

This rule checks that an assertion always specifies a proper clock edge, as in `@(posedge clk)`, rather than omitting the clock edge qualifier and operating unintentionally on both edges. This might seem like a non-problem or a minor performance issue: if a logical condition needs to be checked, why is it bad to check it during some extra phases? We need to keep in mind, however, that many SVA assertions are sequential, spanning multiple clock cycles and many time steps. For example, one of our designers wrote an assertion similar to this:

```
a1: assert property (@(clk) a|->b[*8]);
```

Because there was no edge qualifier, the repetition operator counted 8 phases, which is 4 cycles, of the clock involved. Actual user intent, however, was this:

```
a2: assert property(@(posedge clk)a|->b[*8]);
```

In this version, the repetition operator only counts positive clock edges, and thus counts 8 cycles. In other words, before this issue was detected, the assertion was checking its condition only half the time intended.

2) Sequence used as clocking event

This rule has implications similar to those of the one above, but covers a rarer situation: using a sequence as a main clocking event of an assertion. SVA allows the use of a sequence as a clocking event. The timing issues, however, are tricky, since, for example, any delayed events will count occurrences of the sequence instead of clock cycles, which is very rarely what the user really wants. For example, in the following code, `a3` checks that `c` occurs after 3 further occurrences of the clocking sequence `a[*5]`, rather than 3 clock cycles later as the user intended:

```
sequence s;
   @(posedge clk) a[*5];
endsequence
a3:  assert property (@(s) b |-> ##3 c);
```

This use of a sequence to clock a property is a very unusual corner of the language, and we have yet to see a case in project RTL where this feature is used correctly. Thus it is fully reasonable to disallow it through lint rules.

3) Complex Boolean expression used for clock

This rule is inserted mainly to prevent glitch dangers. In one user example, an assertion in a block with an enabled clock was clocked using an expression `@(clk && en)`.

```
a4: assert property (@(clk && en) p1);
```

While this seemed fine logically, `en` was vulnerable to glitching while `clk` was 1: it could go from 0 to 1 for an instant during a time step, then back to 0 by the end of the time step. This resulted in a bogus edge of the assertion clock, causing a sudden evaluation and failure that was very difficult to debug. Such enabled clocks should be specified using `@(clk iff en)`, which is defined as being checked only when `clk` changes.

```
a5: assert property (@(clk iff en) p1);
```

4) Wrong argument type or size

Argument type or size seem like items clearly worth checking, but they can be difficult to check due to the

flexibility offered by SVA. This is a case where more detailed information is known about a library checker than can be directly inferred from its syntax. For example, one common misuse we have seen is a library invocation such as the following:

```
`ASSERTS_ONE_HOT(a6, sig1||sig2||sig3, …);
```

This assertion was intended to check that there is precisely one high bit among `sig1`, `sig2`, and `sig3`. But as written, this assertion merely looks at the one-bit value attained by ORing all the signals together, and checks that the bit is 1. As a result, errors due to multiple signals being 1 at the same time, the major intent of the assertion, could be missed. The second argument should have been a concatenated vector, as in

```
`ASSERTS_ONE_HOT(a7, {sig1, sig2, sig3} …);
```

By making lint tools aware of our assertion library, this and similar cases are easy to detect. In this particular case, we know that a ONE_HOT assertion does not make sense unless its data argument has more than one bit, so we can flag the erroneous case `a6` above. Even without a checker library, a limited version of this lint rule can be implemented by looking at the use of system functions such as `$onehot`.

5) $stable(sig[index]) with variable index

This rule is an interesting, subtle case that was discovered by a user after many confused hours of debug. The problem was that sampled-value functions like `$stable` (which checks that a signal matches its previous value) look into the past of all their arguments, including array indices. So if `index` just changed from 5 to 6, instead of comparing `sig[6]` to its previous value as the user intended, `$stable(sig[index])` would compare the current value of `sig[6]` to the previous value of `sig[5]`. To prevent this, we needed to add a rule checking that the index of a vector signal used in a sampled-value function is an elaboration-time constant or an automatic variable. This applies to all the sampled-value functions, including `$past`, `$stable`, `$changed`, `$rose`, and `$fell`. While this rule might initially sound very limiting, in the majority of cases a simple rewrite with a generate block solves the problem. For example:

```
int q_pos;
a8: assert property ($stable(sig[q_pos]));

generate
for (genvar i = min; i < max; i++) begin
   a9: assert property (
      (i == q_pos) |-> $stable(sig[i]));
end

   endgenerate
```

In the above code, suppose `q_pos` transitions from 0 to 1. When calculating stability, `a8` will compare the current `sig[1]` to the previous `sig[0]`, while `a9` will compare the current `sig[1]` to the previous `sig[1]`, since **`genvar`** `i` is not sampled. The latter case is much more likely to match user intent.

6) Non-sampled value in action message

This rule does not impact assertion correctness, but can save many wasted hours of debug time. The problem is that the way SVA is defined, most signals in assertions use values sampled at the beginning of the time step, while the action block, where messages are reported, uses current values when it is executed. So, in the following assertion, the value of sig used for evaluation will be from one cycle earlier than the value printed:

```
a10: assert property (sig) else
       $error("Bad: %d", sig);
```

This can mislead the user during debug. The language offers a simple fix, in the `$sampled` function:

```
a11: assert property (sig) else
       $error("Bad: %d",$sampled(sig));
```

Thus linting should check that concurrent assertions report sampled values when applicable.

7) Property using negated implication

We have seen some designers create properties using negated implication operators, such as the following:

```
a18:  assert property (not (a |-> b));
```

Using the DeMorgan-like laws that apply to formally rewriting SVA properties, this is equivalent to

```
a18:  assert property (a && !b);
```

Thus `a18` actually means that `a` must be true every cycle, and must always occur with `!b`. However, in the vast majority of these cases, the actual user intention was that **if** `a` occurs, it always results in `!b`:

```
a19:  assert property (a |-> !b);
```

We therefore have a lint rule to flag any cases of negated implication.

B. Possibly ignored assertions

The cases in this section are similar to those above, except that rather than checking the wrong conditions, an assertion may appear to be valid without actually checking anything at all. Such cases are especially dangerous because they are likely to lead to "false positives", where the designer thinks some aspect of the design has been validated when in reality it has actually been ignored.

1) Short-circuitable function has assertion

This rule checks that the user-instantiated functions are in a place where the assertion is evaluated constantly. SystemVerilog's short-circuiting feature causes some terms of a Boolean expression to be ignored if the value is fully determined by earlier terms, so there are some code locations where an assertion will not always be evaluated. For example, in a `||` expression, if the first term is 1, the overall

value is 1, and any further terms are irrelevant. Consider the following snippet:

```
function bit legal_state(
    bit [0:3] current, bit valid);
    a12: assert #0 (valid |-> current != '0);
    legal_state = valid && $onehot(current);
endfunction
…
if (status || legal_state(valid, state))
…
```

Looking at the **if** condition, since the main operator is a logical one, users usually order the arguments by complexity, seeing performance as an important factor. In the above example, each time `status` is equal to `true`, given that the logical or operator `||` implies short-circuiting, the assertion `a12` will not get checked. This is a typical case that causes missed checks which are hard to identify.

For the above example, there are simple solutions for forcing evaluation of the function: either replace the binary logical "or" operator with the binary bit-wise "or" operator `|`, which does not allow short-circuiting, or switch the order of the operator's arguments (care must be taken in choosing an appropriate solution for the situation, however; for example, indiscriminate use of bitwise rather than logical operators can degrade performance). To prevent situations where the user-instantiated functions will not be evaluated correctly due to short-circuiting, we added a rule that checks that assertions always exist on a path where short-circuiting will not affect their evaluation.

2) Action block with no system function

This rule checks a tricky class of user typos. When using assertion blocks, the user usually wants to display data regarding the assertion violation or to add an indication as to what went wrong. It is unlikely that the user will add a statement that does not contain a system function. This becomes a serious issue when the user uses assertions wrapped in macros, and forgets whether the macro is responsible for adding the semicolon. See the following example:

```
`define MY_MUTEX(sig) \
  assert #0($onehot0(sig))

always @(posedge clk) begin
  ...
  `MY_MUTEX(fsm_1_state)
  `MY_MUTEX(fsm_2_state);
end
```

Due to the missing semicolon after the first assertion, the second assertion is treated as its action block, rather than as a normal assertion which is continuously checked. While this pitfall is easier to fall into when assertion macros are used, a similar situation is possible using standard SVA assertions. To prevent such cases, we require that each action block include a system function.

3) Unbounded assertion always true due to weakness

This rule checks for possibly vacuous assertions. Starting in the P1800-2009 standard of SystemVerilog, assertions were defined as *weak* by default, which means that they are considered true unless a finite simulation trace exists that can disprove them. In contrast, a *strong* assertion can be disproved through formal analysis on infinite traces. When checking a protocol, the user usually adds assertions to check its behavior, for example, that following a request it is guaranteed that a grant will arrive at some point. When using the open bound operators, the user must take care not to write an assertion that might be vacuous due to the definition of weakness. A typical case is the following:

```
a13: assert property
       (@clk req |-> ##[1:$] gnt);
```

This assertion will pass even if no grant can ever arrive since the operator `##` appears in a weak context: no finite trace can show that this is false. A strong version of the property (using the SVA 2009 **strong** operator) can be disproven in formal verification by demonstrating that infinite traces exist where a `req` arrives and the system can then enter an infinite loop where a `gnt` never does:

```
a14: assert property
       (@clk req |-> strong(##[1:$] gnt));
```

In simulation, care must also be taken to ensure that proper simulator options are used to report unbounded assertions still incomplete at the end of simulation. These may be treated as passes or failures on a case-by-case basis.

This issue is further complicated by the fact that before the P1800-2009 standard ([4]), the strength and weakness of properties in the language was not defined in a useful way, and properties such as the one in `a13` were strong. In the past year we have seen code like this treated both as weak and strong by different FV tools. Thus during linting it is especially important to flag weak properties spanning unbounded time windows which may be usefully handled by some tools and treated as trivially true by others.

4) Implication in cover property

This rule checks for possibly vacuous coverage properties. The implication operators are commonly used to define assertions stating that an antecedent implies a consequent. The result of the implication is either true or false. A common mistake users make is to use the implication in a **cover property** expression when the intent is to cover the case when the antecedent is true and the consequent follows. However, with these operators, if there is no match of the antecedent, the implication evaluation returns true by definition, creating a coverage report that is not useful. In general this stems from the basic Boolean logic rule that if `A->B`, and `A` is false, then the implication is trivially true. Some tools may solve this by omitting such vacuous coverage, but this is not required by the language and so cannot be relied upon in all cases.

Consider the following example:

```
property next_state(state, clk);
     (@clk (state==REQ) |=> (state==SEND));
endproperty
```

```
a15: assert property (next_state(st, clk));
c15: cover property (next_state(st, clk));
```

The user wrote an assertion for a specific state in an FSM along with a cover property. The assertion will fulfill the intention, but the cover property will collect coverage each time `st` is not equal to `REQ`, likely resulting in trivial reports that all tests cover this case. A much more useful cover point, probably what the user intended, would have been:

```
c16: cover property (@clk
       (state==REQ)##1 (state==SEND));
```

To prevent cases such as that described above, we disallow use of implication in cover properties.

5) Bad comparison to unknown

This rule involves assertions containing X values and used incorrectly in expressions. For example:

```
a17: assert property (@clk !(a == 1'bx));
```

The intention of the user is clear, but this is a constantly true assertion that will never fail. To properly compare 4-valued logic values, the `===` (triple equals sign) operator must be used. While this situation is not unique to SVA, we have found that this mistake is much more likely in practice to appear in assertions. Thus we defined a lint rule reporting any case of comparison to X or Z using the `==` operator.

6) Assertion with constant clock

We were surprised to see some RTL code in our projects where assertions were clocked by a constant.

```
parameter ALWAYS_ON = 1;
a20:  assert property
          (@(ALWAYS_ON) (a || b) );
```

Here the users have made a basic misinterpretation of the language, thinking that a constant 1 will ensure that the assertion will always be active. However, since assertions are triggered at the edge of their relevant clock, this actually had the opposite effect, permanently deactivating the assertion. The correct solution would have been to use an immediate assertion, since these are unclocked and thus are inherently active at every time step their inputs change:

```
a21:  assert #0 (a||b);
```

To prevent such errors, we defined a new lint rule to detect constant clocks in assertions.

C. Performance hazards

Assertion performance essentially depends on the way the assertion is written. Performance of two equivalent assertions may be absolutely different both in simulation and in formal verification. For example, the assertion

```
a22: assert property (a |=> b);
```

introduces minimal simulation overhead, whereas the equivalent[1] assertion

```
a23: assert property (##[*]a |=> b);
```

may slow down the simulation [1].

The situation with assertion performance is complicated by the fact that often simulation and formal verification have different and even contradictory requirements for assertion performance. This issue is discussed in detail in [2] (see also the comparison of different simulation algorithms there). For example, for many formal verification tools the performance of assertions `a22` and `a23` is exactly the same. It is worth noting that the requirements for assertion efficiency in emulation are usually more similar to those imposed by formal verification than to those imposed by simulation [2].

In this section we discuss lint rules related to performance hazards. Some rules are relevant only for simulation or formal verification; others are universal. While there is nothing logically incorrect about these cases, their cumulative performance costs may have a significant negative impact on project success.

1) Many instances of single assertion

The justification for this rule seems pretty obvious, but it is amazing how often it is violated in user code. The rule states that one should avoid writing many assertions for individual signal bits when it is possible to write one assertion for the entire signal. For example, if one wants to make sure that two vectors `a` and `b` have equal values, it is sufficient to check that `a == b` instead of checking in a procedural or in a generate loop that each bit of `a` is equal to the corresponding bit of `b`. In the following code, while `a24` and `a25` are logically equivalent (assuming that `a` and `b` are 128-bit vectors), it is very likely that the version in the generate loop is much less efficient for simulation:

```
generate for (i = 0; i < 128; i++) begin
  a24:  assert #0 (a[i] == b[i]);
end
endgenerate

a25: assert #0 (a==b);
```

2) Assertion in loop not using index

This rule addresses the situation where an assertion is in a loop, but the actual expression being checked does not involve the index of the loop. Thus, the loop is actually irrelevant to the assertion, and it could easily have been placed outside it, significantly reducing the number of executions. Instead of executing once per clock tick (for concurrent assertions) or when variables in their expression change (for deferred or immediate assertions), such assertions execute for each iteration of the loop. This means that many of the executions are wasted, with repeated execution of the same assertion on the same values. For example:

---

[1] Strictly speaking, these two assertions are not equivalent because they have different attempts, but logically they check exactly the same thing.

```
for (i = 0; i < 128; i++) begin
   a26: assert #0 (a == 1);
end
```

There is a slight complication here: to be fully general, this rule should take into account that an assertion may be using a loop index indirectly – i.e., it may contain a variable which depends on the loop index. However, in the common case it may be sufficient simply to examine the assertion expression, flag the assertion in lint if it does not use the index, and let the designer waive if necessary.

3) Large or distant time windows

Large or distant time or repetition windows in sequences, and `$past` functions with a large cycle count are inefficient both in simulation and in formal verification. For example, all sequences and expressions similar to the following should be avoided: `a[*2:1000]`, `a[*999:1000]`, `##[2:1000]`, `##[999:1000]`, `$past(a, 1000)`. In formal verification and in emulation it is usually much more efficient to use unbounded time or repetition windows in this case, e.g. `a[*2:$]`.

4) Unbounded time or repetition window in antecedent

Unbounded time and repetition windows in an assertion antecedent may be costly in simulation, as in this example:

```
a27: assert property (a[*1:$] |=> b);
```

Most simulators will launch a new checking thread for this property every time `a` is active. This can lead to an inefficient explosion of threads. Thus any case of an unbounded time window in an assertion antecedent should be reported.

This rule forbids using unbounded repetition windows at the beginning or end of the antecedent, and unbounded time windows at any place in the antecedent. Unbounded repetition windows in the middle of the antecedent, though possibly presenting some performance risk, are actually quite common, as shown in the following example:

```
a28: assert property (
   req ##1 gnt[->1] |=> done);
```

Here it is checked that the `done` signal is asserted after the request has been granted. There is an implicit unbounded repetition in the antecedent, as `gnt[->1]` is equivalent to `!gnt[*] ##1 gnt`. This assertion is generally efficient when the system is behaving in a reasonable way, i.e., when a request is granted within several clock cycles.

5) Using cover sequence rather than cover property

SVA defines two versions of the concurrent cover statement: **`cover sequence`** and **`cover property`**. The **`cover sequence`** statement must report every match of the sequence for each attempt, while the **`cover property`** statement is only required to report one match per attempt. An *attempt* occurs whenever the sequence being covered begins evaluation. The **`cover sequence`** statement is very rarely needed in practice, and in many cases is extremely inefficient in simulation. For example, assume the reset signal falls once, and we have the following two cover points:

```
c29: cover sequence ($fell(rst) ##[*] b);
c30: cover property ($fell(rst) ##[*] b);
```

Cover `c29` will be active continuously after the fall of rst, reporting a coverage success every time `b` is true. Cover `c30` will report a success the first time `b` is true, and afterwards cause no further overhead in simulation.

6) Applying $past to every argument of an expression

The sampled value function `$past` imposes performance overhead in both simulation and formal verification. This rule flags cases of redundant `$past` usage: instead of applying `$past` to all arguments of an expression, it is more efficient to apply it to the expression result. For example, instead of `$past(a) * $past(b) == $past(c)` one should write `$past(a * b == c)`.

7) Antecedents with empty match

Properties with antecedents that admit an empty match are redundant and often do not correctly reflect user intent. In addition, they will be triggered every time step, which causes inefficiency. For example (see [2]), `a[*] |-> p` is equivalent to `a[+] |-> p`, and `a[*] |=> p` is equivalent to `p`. Therefore, all assertions with antecedents that admit an empty match should be reported.

III. Non-Lint Techniques and Future Additions

It is important to point out that we have not solved all SVA usage problems: some require different or more complex procedures than linting to detect, and others are more easily addressable through tool or language extensions. In this section we show examples of issues that fall into these categories.

A. Unsupported SVA 2009 constructs

We observed, in numerous cases, use of constructs that, while legally in the language, are unsupported by some of our EDA tools. For example, the following code makes use of a global clocking future value function, a nice feature in SVA 2009 that makes it possible to look ahead one tick of the fastest clock:

```
a31: assert property (
   a |-> $rising_gclk(b));
```

We need to find a way to enable our RTL to take advantage of these features when useful, since our main simulation and formal tools do support them, but to be able to turn them off when compiling for a flow that does not support them.

Our solution is based on leveraging our common assertion library. Assertions that use such constructs are required to protect them with an `` `ifdef `` flag: if SVA_LIB_SVA2005 is defined, we replace the assertion code with a constant 1. We lose the checking of that assertion in such cases, but are able to smoothly compile the model for non-2009-supporting tools.

B. Rules requiring formal engines

Some rules are too complex to be implemented in preprocessing, or may be more easily implemented with support from a formal verification engine.

An example is the performance hazard of liveness assertions in formal verification, which we detect in the front end of our formal property verification tool rather than during linting. From the point of view of formal verification, all assertions can be classified as either safety or liveness[2] properties [3]. A counterexample of a safety property is always finite. All other properties are liveness properties. Checking liveness properties in formal verification is much more expensive than checking safety properties [8]. It sometimes happens that people write liveness assertions without intending to. This rule flags all liveness assertions to make sure they are intentional. Consider the following assertions:

```
a31: assert property(
   $fell(rst)|->s_eventually a);

a32: assert property (
   @clk not(write ##1 read));
```

Both assertions `a31` and `a32` are liveness. Assertion `a31` states that `a` will eventually be true, and its liveness is inherent. The likely intent of assertion `a32` is to state that `read` cannot follow `write` directly. However, assertion `a32` also checks that after each `write` the clock ticks at least once. This check is usually not intended, but it makes formal verification of `a32` much more costly. The proper way to write this assertion is `not strong(write ##1 read)`.

C. Glitch sensitivity for deferred assertions

As originally defined in SVA 2005, immediate assertions in procedural code were vulnerable to glitch issues, whereby a temporary signal value during relaxation could result in an assertion failure even though the settled signal values at the end of the time step would be legal. To solve this problem, SVA 2009 introduced deferred assertions, whose evaluation would be deferred until after model activity. Unfortunately, in the 2009 definition deferred assertions could still cause glitches due to iteration with new values set by the testbench; once the testbench changed signal values, there could be further model activity. Thus we have been working to introduce a new construct in the 2012 iteration of the standard: *final assertions*. Final assertions will mature after all model and testbench activity is complete, and thus not be vulnerable to glitches.

D. Poor X/Z behavior for bit-vector functions

Bit-vector functions are not well suited for checking 4-valued data. For example, `$onehot(v)` returns 1 even if several bits of the vector `v` are X or Z, which is usually not the intended behavior. To address this, we currently use solutions such as adding a check that all bits have a known value: `$onehot(v)&& !$isunknown(v)`. To deal with this problem, the emerging standard provides a new system function, `$countbits`, that generalizes the existing bit-vector functions and takes 4-valued data into account. While we have been considering a lint rule to check that bit-vector functions are always used in conjunction with `$isunknown`, language change will enable a more robust solution.

E. System task $asserton not waking up always procedures

SVA has a mechanism for disabling/enabling assertion execution using the functions `$assertoff`/`$assertkill` and `$asserton`. However, `$asserton` does not affect the sensitivity list of the always procedures `always_comb`, `always_latch`, and `always @*`. Consider the following example:

```
always_comb begin … assert #0 (a); …; end
```

If assertions were previously disabled and then became enabled with `$asserton`, the assertion above would be checked for the first time when the value of `a` changes, and not when the assertion becomes enabled. If `a` was and remained low at the time of assertion enabling, the assertion will not fire. We have been investigating tool extensions to handle this special case, forcing such assertions to be triggered after an `$asserton`.

F. System task $asserton/kill not performing as necessary to handle multiple power planes

For the purposes of power control, assertions should be switched on and off during simulation. The language allows switching assertions on and off in a specific hierarchy, but in some cases this capability is not sufficient. Imagine that it is required to switch off all the assertions in the hierarchy `top.block1` except those in the sub-hierarchy `top.block1.subblock2`. An attempt to kill all the assertions in `top.block1` and then to reenable them in `top.block1.subblock2` results in killing all active attempts of assertions in `top.block1.subblock2`. The workaround for this situation is to enable and disable assertions individually. In the emerging standard it is planned to introduce block-level assertion locking and unlocking. This will make it possible to lock assertions in the sub-block, kill them in the outer block, and then unlock the assertions in the sub-hierarchy without affecting their execution there.

G. Performance hazard due to global clocking

Assertions governed by the global clock are usually efficient in formal verification. However, in the presence of many clock domains, the global clock should be the finest-grained event among all the clocking events. As a result, assertions governed by the global clock may significantly slow down simulation. It is therefore recommended to enable such assertions only when checking blocks belonging to one or a few clock domains. To address this issue for the full chip, the emerging standard allows different definitions of the global clock for different design hierarchies.

IV. Results and Conclusion

Overall, the use of SVA has been a very effective technique in our design and validation environments. Simulation failures are much more easily debugged when a

[2] By liveness we do not understand the pure liveness as defined in [3], but all properties that are non-safety.

nearby assertion can help point to the root cause of a problem, and our formal verification environments are built using SVA. One recent project reported that 20% of all pre-silicon RTL logic bugs were found with the aid of assertions, while only 2% of the bugs were actually due to incorrect assertions. The 20% is an underestimate; many bugs are found by assertions in early models that the designer has not yet checked into the database. In addition, some of our flows, such as emulation, have moved to assertions as their primary error-checking method.

Naturally, any instance of an incorrect assertion demands attention. We have been gradually implementing the rules presented in this paper in response to discovering the actual issues in production on our major CPU projects. As a result we have been continually improving our confidence in our SVA assertion checking. While disturbing, the corner cases presented here have been very rare, and do not negate the overall benefits of SVA.

Ultimately, the integration of SVA has been a strong win for Intel. It has dramatically improved our validation environment as well as strengthened integration between validation and design. With careful linting as described in this paper, we hope to continue strengthening our validation coverage and making our SVA usage even more successful.

ACKNOWLEDGMENTS

We would like to acknowledge the contributions of Kedar Jog, who implemented most of the rules discussed in the paper, and Wayne Clift, who helped identify many of the tricky SVA usage issues.

REFERENCES

[1] Roy Armoni, Dmitry Korchemny, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar, Deterministic Dynamic Monitors for Linear-Time Assertions. FATES/RV 2006

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

[3] Leslie Lamport, "Proving the Correctness of Multiprocess Programs," IEEE Transactions on Software Engineering, vol. SE-3, no. 2, pp. 125-143, March 1977.

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

[5] Janick Bergeron, Eduard Cerny, and Andy Nightingale, Verification Methodology Manual for SystemVerilog.: Springer, 2005.

[6] SystemVerilog Org page: http://www.systemverilog.org/

[7] Erik Seligman, Laurence Bisht, Wayne Clift, "Stumbling on SVA: Pitfalls from Real Intel Projects", Design Automation Conference, 2011.

[8] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu, "Bounded model checking," Advances in Computers, vol. 58, pp. 117-148, 2003.