# Double the Return from your Property Portfolio: Reuse of Verification Assets from Formal to Simulation

Jonathan Bromley
Verilab Ltd, Edinburgh, Scotland
*jonathan.bromley@verilab.com*

***Abstract*- This paper discusses a team's experience with taking complete block-level formal testbenches and making productive use of them in a dynamic (simulation) environment. We describe the motivations for this re-use and the challenges encountered. The paper presents detailed methodology suggestions that we used successfully to overcome these challenges, and reports how both formal and dynamic verification quality was improved by re-using formal assets in this way throughout a project.**

## I. BACKGROUND

In the past few years there has been a clear and rapid shift towards adoption of formal verification in mainstream ASIC verification practice - as reported, for example, in [1]. We can identify several drivers for this remarkable and long-overdue change. The longest-established driver is without doubt the move to standardized assertion languages, notably SVA [2] and PSL [3], replacing the unhappy situation in which each formal tool used its own notation to specify properties and assertions. In addition to the obvious technical benefits, these standards made the concepts of assertion-based verification (ABV) accessible to anyone familiar with the basic ideas of cycle-by-cycle simulation of a clocked design. We can be confident that formal techniques are now better understood and more widely applied than they ever could have been without ready access to ABV in simulation.

A further remarkable feature of [1] is the strong evidence that the larger a design project, the *more* likely it is to use formal verification. Increased design complexity calls for ever greater rigor and thoroughness, and formal verification helps to answer this need.

## II. INTRODUCTION

### A. Project experience

This paper reports experience with re-use of formal verification code in simulation, as part of a recent large bus-interconnect project. The team's verification strategy, informed by experience on earlier related projects, called for all block-level verification to use only formal techniques. A constrained-random simulation testbench was used at the top level.

Early in the project, it was decided that all block-level formal verification code should be re-used in simulation wherever possible. The reasons for this decision are described more fully in section III below. We found that this re-use had many important benefits, described in detail in later sections, and our only regret was that (because of limited available debug effort) we did not complete our integration of formal testbenches into the simulation flow as early as we had hoped.

### B. A modest agenda: the limitations of this paper

Taking assertions written for formal verification, and deploying them during simulation, is not new. It is an established technique among verification groups that have a long history of using formal. However, as mentioned in section I above, the industry has recently seen a large cohort of more recent adopters of formal verification. The experiences and guidelines described in this paper are aimed at such users, in the hope of saving them some time and effort.

Furthermore, this paper for the most part addresses only one aspect of formal verification: the use of assertions to prove correctness of individual RTL blocks. Formal methods are applicable in many other parts of the design verification flow, such as connectivity checking, but our formal verification effort was predominantly concerned with traditional model checking. Even in that area there are many specialized formal techniques, such as design element abstraction, that this paper does not attempt to consider.

Despite this limited scope, the approaches described here proved useful and effective in our recent verification project, and it is hoped that they may be similarly helpful to other readers.

## III. WHY RUN ASSERTIONS IN BOTH FORMAL AND SIMULATION?

Before considering the details of *how* to re-use formal verification code in simulation, it is important to consider *why* such an approach might be valuable.

### A. *It shouldn't be necessary - formal does all you need!*

Formal verification offers the promise of exhaustive proof of correctness (or, of course, discovery of incorrectness) of a design. If a formal tool can find a proof that some property holds, then we must accept that the property holds *in all possible situations* that are permitted by the assumptions in force. The obvious conclusion is that running formal assertions in simulation is unnecessary, because it would add nothing to the verification effort. The reality is rather different, as we shall attempt to show below.

### B. *Insufficient exploration depth in formal*

For non-trivial design blocks, some of the most important assertions checking end-to-end behavior may be impossible for the formal tools to prove or disprove in a reasonable runtime. A good example of this was found on the project that led to the writing of this paper. The RTL design under test (DUT) was a complex transaction ordering block, with internal list structures sorted on various combinations of transaction arrival order, transaction ID, response order and other factors. The formal testbench included tracking structures that reflected those orderings, but were implemented independently to minimize the risk of matching design and verification errors.

After each design or specification change, formal would very quickly find counterexamples to various assertions because of comparatively minor errors and oversights in both the RTL design and the formal testbench. As both testbench and RTL design matured, though, we would reach a situation in which formal runs of many hours would fail to reach either a proof or a counterexample on some of the most important assertions. The absence of counterexamples led us to have some confidence in the design, but for some assertions it was a difficult challenge to reach acceptable exploration depth and so there was always the possibility that our verification had missed some bugs.

It was at first very disappointing to find that some of our most important assertions remained unproven even after very long runs of the formal tool. However, as discussed in [4], this is by no means disastrous. Careful consideration of the formal exploration depths reached by assertions and by cover directives, together with the added confidence gained from formal cover directives being hit in simulation, can lead to greatly improved confidence that the formal verification effort - even though not exhaustive - has had adequate opportunity to find RTL bugs.

### C. *The canary in the RTL mine: early warning and better debugging*

Traditionally, assertion-based verification (ABV) in simulation has been useful because assertions embedded in, or applied at the boundaries of, a design block could detect bad behavior of a block long before that bad behavior's effects were visible outside the DUT. For our design, another important manifestation of this effect was improved debugging thanks to the internal structures built by our formal testbenches and also by the designers' embedded assertions. These structures were usually created as auxiliary logic to help with tracking the progress of transactions through the design so that the DUT's eventual behavior could be checked by assertions. Because this auxiliary logic was implemented as modeling code rather than design code, it often provided a more easily understood representation of the system's activity than did the RTL design itself. Engineers working on the simulation testbench soon came to rely heavily on this embedded debug information provided by code that had originally been written for formal verification.

### D. *The killer advantage: uncovering buggy assumptions (constraints)*

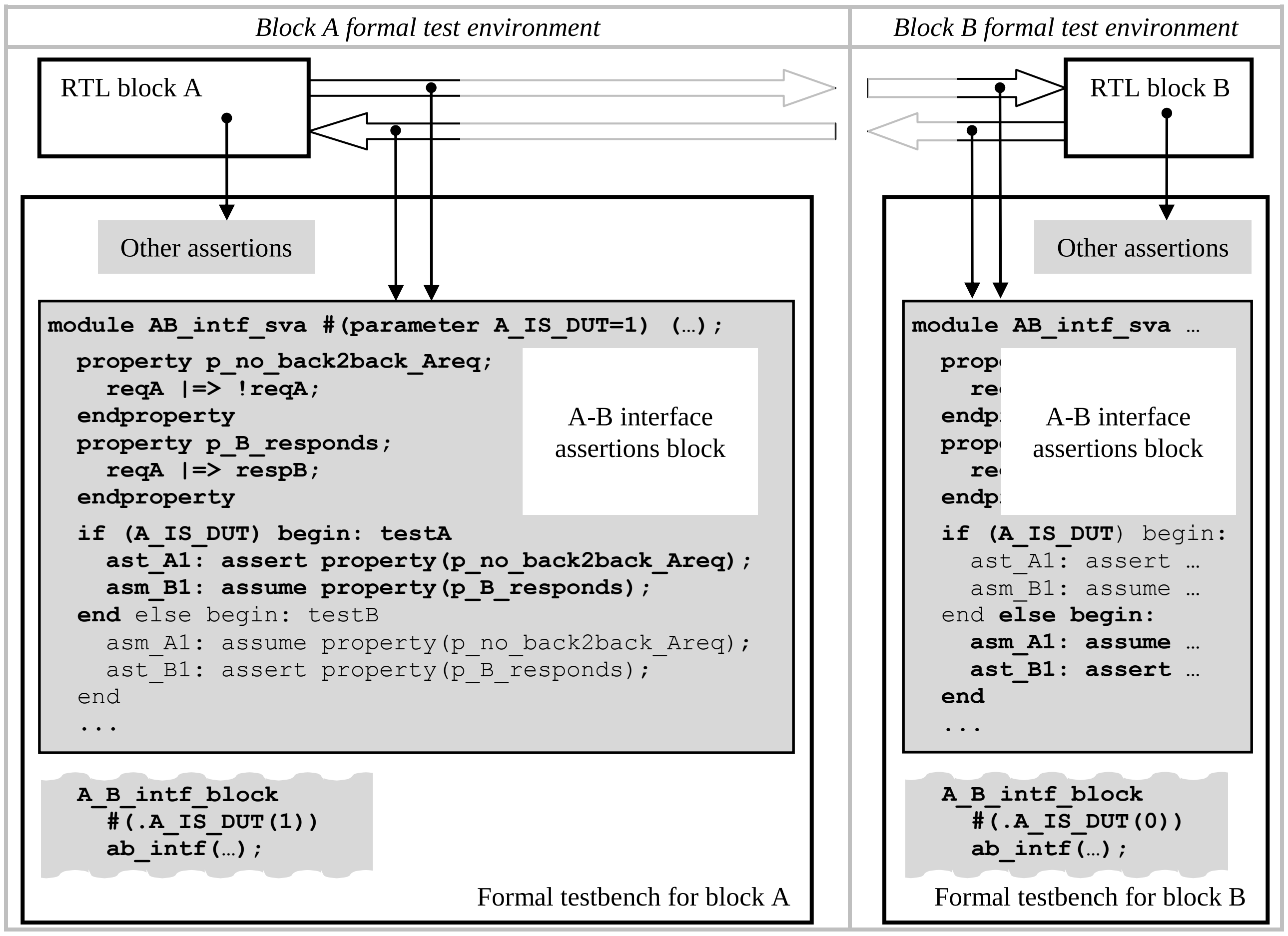
Any testbench, whether designed for formal or simulation, may be flawed. In a formal testbench, the flaws generally fall into several major categories:

1. *Faulty assertions that fire although the DUT's behavior is correct*. These flaws are usually quite easy to find and fix, because the assertion failure is self-evident and a counterexample trace is available to show a situation where within-spec DUT behavior has violated the assertion.
2. *Assumptions that insufficiently constrain the DUT's input space*, allowing illegal and impossible stimulus to be supplied to the DUT leading to bad behavior. Although these are usually much harder to debug than flawed assertions, they nevertheless are readily manifest because they will generally cause some kind of faulty DUT behavior that will be picked up by assertions elsewhere in the testbench.
3. *Excessively tolerant assertions* that detect some, but not all, potential DUT faults. These are much more troublesome, because they leave no obvious evidence. If the DUT is flawless, then the assertion will not fire

and no harm is done. But there may be DUT faults that represent genuine bugs but are not detected by an assertion that is too weak.

4. *Assumptions that overconstrain the DUT's input space*, causing some possible input scenarios to be missed from the formal verification effort.[1] As mentioned in [5] and other texts, suitably chosen cover directives can help to find these, but it is important to note that assumptions and cover directives are usually written by the same verification engineer and therefore there is a risk that they may suffer from exactly the same misunderstandings of a specification.

Of these classes of faulty formal directives, the last two are clearly the most worrying as they leave no obvious trace. Our verification methodology tried to mitigate this risk by the use of reversible assertions/assumptions at the DUT's interfaces with neighboring blocks, providing a convenient implementation of the traditional assume-guarantee methodology in formal verification as shown in Figure 1. All properties relating to the interface between blocks A and B are captured in a reversible interface block. The formal testbench for block A creates an instance of this block with assertions and assumptions appropriate for testing block A. Block B's formal testbench, conversely, flips all those assertions and assumptions, making them appropriate for testing block B. In this way, a single module - the interface block - was reused in both testbenches. In our project, reversal of assertions and assumptions was managed through parameters along with some macros to minimize code verbosity - but that is merely an implementation detail and the principle was exactly as shown in the figure below.


Block A formal test environment
Block B formal test environment
RTL block A
RTL block B
Other assertions
Other assertions
A-B interface assertions block
A-B interface assertions block
Formal testbench for block A
Formal testbench for block B

```
module AB_intf_sva #(parameter A_IS_DUT=1) (…);
  property p_no_back2back_Areq;
    reqA |=> !reqA;
  endproperty
  property p_B_responds;
    reqA |=> respB;
  endproperty
  if (A_IS_DUT) begin: testA
    ast_A1: assert property(p_no_back2back_Areq);
    asm_B1: assume property(p_B_responds);
  end else begin: testB
    asm_A1: assume property(p_no_back2back_Areq);
    ast_B1: assert property(p_B_responds);
  end
  ...
```

```
  A_B_intf_block
    #(.A_IS_DUT(1))
    ab_intf(…);
```

```
module AB_intf_sva …
  prop
    re
  endp
  prop
    re
  endp
  if (A_IS_DUT) begin:
    ast_A1: assert …
    asm_B1: assume …
  end else begin:
    asm_A1: assume …
    ast_B1: assert …
  end
  ...
```

```
  A_B_intf_block
    #(.A_IS_DUT(0))
    ab_intf(…);
```

**Figure 1: Reversible interface block implements assume-guarantee on a block-to-block interface**

An assertion on block A's output can be re-used as an assumption on block B's input. If the assertion is too lenient on block A (case 3 above), when reversed it will appear as an insufficient constraint to block B (case 2) and will

[1] Problem-specific overconstraints may be intentionally applied to restrict the space of possible behaviors and thereby simplify some proofs. They should always be clearly identified, either by using SVA's `restrict` directive (which is ignored in simulation), or by a naming convention so that they can easily be defeated for simulation.

probably be uncovered in formal verification because block B will be presented with illegal input sequences for which its behavior is not well-defined. Similarly, an overconstraint on block A's inputs (case 4), when reversed, has the effect of an over-aggressive assertion on block B (case 1). This will lead to assertion failures in the block B formal testbench even though block B's behavior is within specification, thus revealing the verification problem.

This use of reversible directives proved to be extremely valuable, both as a way to share verification effort between testbench authors, and for its power to detect flawed constraints. However, it works best when an interface carries interaction between two blocks that is broadly independent of the traffic on other interfaces. In our case, some blocks had coordinated interactions with numerous neighboring blocks, and assumptions typically had to capture the temporal relationships among these multiple interfaces, as indicated in **Figure 2**. Without this coordination, assumptions and assertions on each individual interface were always incomplete, because the environment in which the DUT operates was controlled not only by its own interaction with neighboring blocks, but also by interactions among those blocks that were not directly visible on interfaces to the DUT itself.

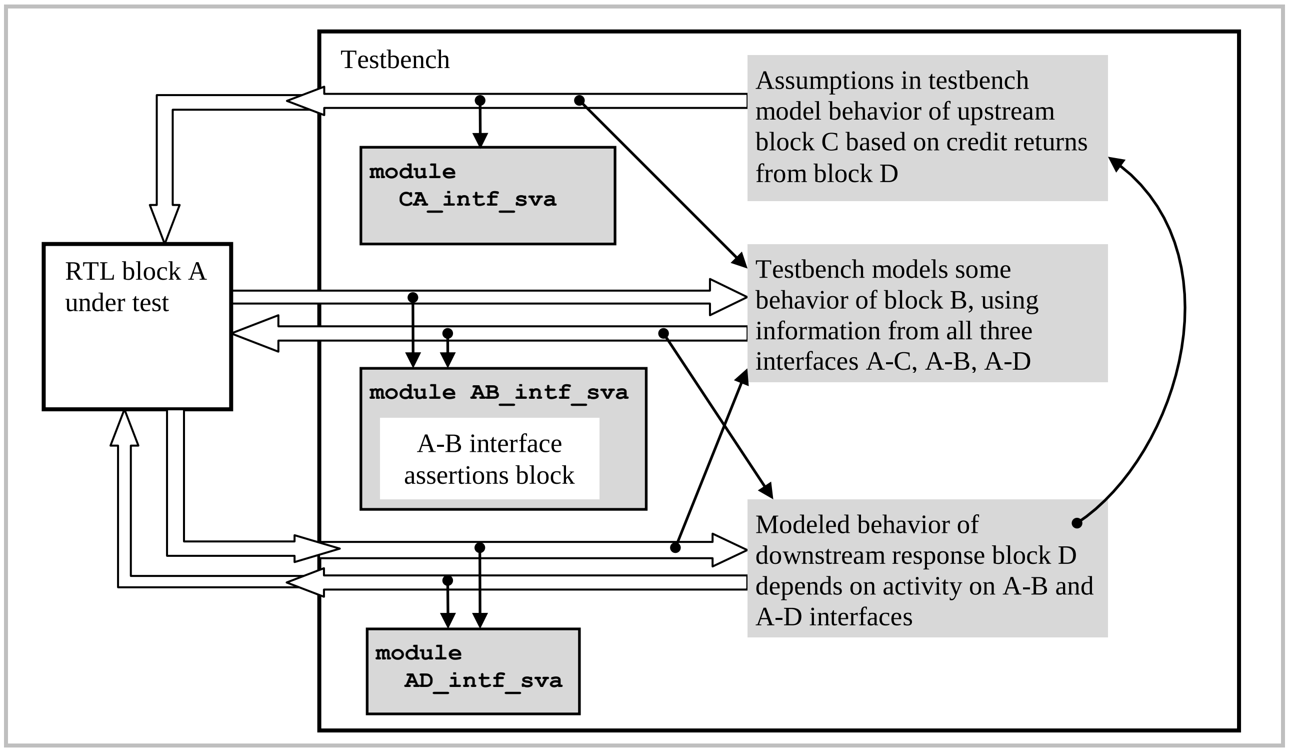

Testbench
Assumptions in testbench model behavior of upstream block C based on credit returns from block D
module CA_intf_sva
RTL block A under test
Testbench models some behavior of block B, using information from all three interfaces A-C, A-B, A-D
module AB_intf_sva
A-B interface assertions block
Modeled behavior of downstream response block D depends on activity on A-B and A-D interfaces
module AD_intf_sva

**Figure 2: Interactions between block-to-block interfaces make assume-guarantee less useful**

In that situation, the reversible assert/assume methodology was much less easy to apply, and several overconstraint bugs in our formal testbenches escaped because of the absence of such mutual self-checking. This is where our use of assertions in simulation was most valuable. With the full set of block-level and block-to-block formal assertions deployed in simulation, overconstraining assumptions were soon violated by simulation traffic. This cross-checking of formal constraints by simulation was extremely helpful to us, uncovering many subtle testbench bugs and clearly identifying our misunderstandings of the specification. It also led to a number of important refinements and corrections of the individual block-level specifications.

## IV. MAKING IT WORK WELL

Although the benefits of simulating our formal testbenches were clear from the outset, there were many practical difficulties along the way. As indicated in section II above, these difficulties delayed the full integration of our formal testbench code in the top-level simulation testbench. With hindsight it would have been appropriate to put more effort into this integration activity because of the value it provided. Without exception, every firing in simulation of the formal testbench assertions led to the discovery of a flaw, either in the formal testbench or else in the RTL design or specification. Some of the more important difficulties we encountered are described below.

### A. *Behavior on assertion failure*

When an assertion proof tool (formal model checker) finds a counterexample, it can provide a wealth of relevant debug information because it understands full details of the sequence of states that led to the violation. The result is that debugging of assertion failures is very well supported by formal tools, and it is unusual to need further information beyond that which is made available automatically by the tools.

By contrast, an assertion that fails in simulation does not so easily give access to all this information. Instead, it is preferable for the assertion itself to report comprehensive diagnostics through its `$error` action, fleshing-out the rather sparse information provided by the tool on assertion failure.

Writers of formal testbenches traditionally do not bother to add `$error` actions to their assertions. If assertions are to be productively re-used in simulation, however, it is desirable to encourage the addition of these reporting actions. Formal tools generally ignore such actions, so the formal testbench is not compromised in any way by their addition.

#### 1. *Use of $sampled in assertion error messages*

The success and failure actions of a SVA directive are evaluated in the Observed region of the SystemVerilog scheduler. In all practical cases, this evaluation occurs at the time of a clock edge, but *after* nonblocking assignment updates to synchronous design variables have taken effect. On the other hand, expressions in the assertion itself have been evaluated using values that were sampled in the Preponed region, reading synchronous design variables as they were just *before* the clock edge. To avoid misleading error diagnostics, it is very important that values in the error message should be computed using the `$sampled` system function, which yields the value of its argument as it was in the scheduler's Preponed region. This issue is nicely described in section 7.2.1.1 of [5].

### B. *Issues caused by imperfect LRM compliance of tools*

The formal verification tool that we used exhibited some small but significant deviations from the SystemVerilog LRM. As a result, some flaws in the syntax of our formal testbenches were not discovered until they were run in simulation. Most of these issues were inconsequential over-generous interpretation of language syntax that had no impact on functionality of the testbench and were easily rectified after the simulator's compiler detected the error. For example, our formal tool allowed some operations to be performed on unpacked arrays that strictly should be permissible only on packed arrays. It also tolerated duplicate names for assertions, automatically providing unique names as necessary.

One issue, however, did cause some functional problems and was troublesome to track down. It concerned the sampling of variables and nets that are read in the body of a SystemVerilog function, when that function is called as part of a Boolean expression in a property. The LRM [2] specifies that actual arguments to a function called from a property should use sampled values, just like any other variable that appears in the property. However, variables that are tested within the body of a function, but are not passed to the function as arguments, should use the current value even though the function is called from a property. Our formal tool wrongly (but very conveniently!) used the sampled value of *all* variables tested within a property - even those read in the body of a function - but the simulator was strictly LRM compliant.

We reviewed all our code for instances of this problem and, in every case, it was appropriate to re-cast the functions so that all variables of interest were passed as arguments. This not only sidestepped the tool issue but also improved the quality and readability of the code.

### C. *Tensions between properties written for simulation and for formal*

A major concern with the approach described here was its impact on simulation performance. Not only were the formal testbench's assertions included in simulation, but also the RTL designers' internal assertions written to provide microarchitectural checking and coverage in the formal testbenches. Our design was highly repetitive in structure (for example, it contained 7 instances of a large RTL block, each instance containing up to 64 transaction slots; each of those slots contained a state machine, and there were about 100 small assertions checking various correct behaviors on each instance of that state machine.) This resulted in a very large number of assertions being simulated.

Reference [5] includes extensive detailed guidance on the likely performance impact of different styles of assertion in both model checking and simulation. Several specific problem cases are discussed in detail in [6].

Performance profiling soon identified some assertions that were consuming an unreasonable proportion of the simulation CPU time. Although the reasons for that poor performance were quite diverse, one major factor dominated: the unnecessary retriggering of liveness properties.

Many checks on forward progress and freedom from starvation were "eventually" properties such as

```
start |=> run[*1:$] ##0 done
```

This correctly expressed our intent, but was disastrously expensive to simulate in the (very common) situation where `start` was a flag that persisted possibly for many hundreds of cycles, and the delay before `done` was likewise hundreds of cycles. The resultant launching of numerous long-lived assertion threads was extremely costly of simulation runtime (though, perhaps surprisingly, not so troublesome in memory consumption: each assertion thread's memory footprint seems to be quite modest). We were able to bring this runtime cost under control simply by reworking the assertion to have only a single-cycle start condition:

```
start && !run |=> run[*1:$] ##0 done
```

or perhaps

```
$rose(start) |=> run[*1:$] ##0 done
```

These changes had no detectable impact on the usefulness or performance of the assertions in our formal verification. The original assertion, despite performing so badly in simulation, did not appear to have any special impact on the performance of formal tools.

After this experience, our team was very nervous of using *any* liveness properties because of their possible overhead. However, it soon became clear that simulation performance was not seriously impacted if such properties could be guaranteed to have only one instance running per transaction by avoiding multiple start conditions. It is not necessary to outlaw liveness (eventually) properties from simulation-based verification.

### *D. Example of a faulty assertion that was not detected in formal verification*

The simulation performance problems described in section IV.C above led one of our team to undertake extensive CPU performance profiling in simulation. Initially, suspicion fell on some assertions written by the designer to check internal behavior of RTL code. The offending assertions had the following form:

```
ast_2cyc: assert property (   // 2-cycle antecedent
  (@(posedge clock) disable iff (!reset_n)
  a ##1 b |-> c );
```

Profiling experiments suggested that we could improve performance by rewriting such assertions thus:

```
ast_past: assert property (   // 1-cycle antecedent
  (@(posedge clock) disable iff (!reset_n)
  $past(a) && b |-> c );
```

Although these are not strictly identical, the differences did not cause any difficulties in most of our examples, where both forms provided us with the desired functionality in our formal testbenches. However, we did encounter an important simulation "gotcha" with this modification that forced us to abandon it. If there is a short asynchronous reset between the first and second clocks of `ast_2cyc`, then the assertion will be aborted even if `a` was true on the first clock and `b` true on the second. By contrast, in the 1-cycle form using $past, if `a` was true before the reset, `$past(a)` is true in the cycle immediately after reset, allowing a new test of the assertion to fire *even though we wanted the assertion to be defeated by the reset.* The trace in **Figure 3** below shows an example of an inappropriate failure of `ast_past` caused by this misunderstanding. Note that `ast_2cyc` has been defeated by the asynchronous reset, and does not begin any further evaluation during the trace shown here.

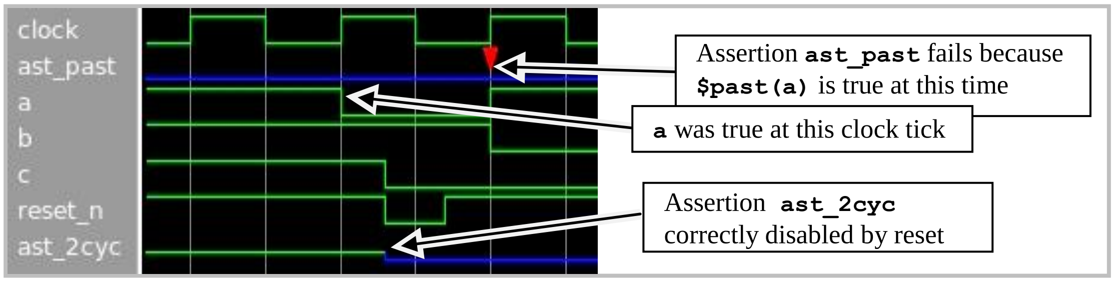

clock
ast_past
a
b
c
reset_n
ast_2cyc
Assertion **ast_past** fails because **$past(a)** is true at this time
**a** was true at this clock tick
Assertion **ast_2cyc** correctly disabled by reset

**Figure 3: $past is not affected by reset**

### *E. Freedom and responsibility: coping with free variables in simulation*

Our testbenches did not make any use of the SVA *checker* construct. The SystemVerilog LRM [2] uses the term *free variable* in a very specific way to describe `rand` variables of a checker. In the discussion below, we use "free variable" to mean any net or variable that has no driver in either the RTL code or the formal testbench. In formal verification, such signals can take any value that is admitted by the set of assumptions in force. Primary inputs of the design are almost always free variables in this sense. However, we also found it very useful to implement some internal free variables within the formal testbench.

A typical example of such usage was the choice of slot in a transaction tracking structure in the testbench. Such tracking structures represented and modeled, but did not exactly replicate, structures in the DUT itself. Some of these testbench structures were simple FIFOs in which the allocation of slots was straightforward and deterministic. Some, however, depended on pointers or links to maintain ordering, and therefore a new transaction can be stored in an arbitrarily chosen free slot in the structure. Given the knowledge (assured by other mechanisms) that there is at least one slot available, we can choose a slot by providing a free variable for the slot index and writing an assumption to constrain it, as indicated in the following simplified example:

```
bit [SLOT_INDEX_BITS-1:0] slot_index;  // undriven (free) variable
bit slot_in_use[0:(1<<SLOT_INDEX_BITS)-1];

// Ensure that slot_index always points at a free slot
asm_freeslot: assume property (@(posedge clock) !slot_in_use[slot_index]);

always @(posedge clock) if (new_transaction) begin
  slot_in_use[slot_index] <= 1;
  … fill slot structure at [slot_index] with transaction info …
end
```

However, this idiom is clearly useless in simulation, where there is no driver on variable `slot_index` and therefore it will always have the value zero. Assumptions are treaded exactly as assertions in simulation, and assumption `asm_free_slot` will therefore fail whenever slot `[0]` is occupied.

Rather than removing this code for simulation and replacing it with a different implementation, we chose to leave all the existing code in place and add a simulation-only driver to give the offending variable a suitable value in a deterministic manner:

```
`ifdef USING_SIMULATION_NOT_FORMAL
always_comb begin
  slot_index = 0;   // to satisfy always_comb "no latch" rules
  foreach (slot_in_use[i])
    if (!slot_in_use[i]) begin
      slot_index = i;
      break;
    end
end
`endif
```

In this way, all our formal code continues to be exercised in simulation, albeit with a deterministic choice of `slot_index`. In particular, the assumption `asm_freeslot` now operates as an assertion, confirming that the simulation-only slot chooser code is doing its job correctly.

If there is a significant risk that deterministic simulation-only code may repeatedly choose only a subset of possible DUT behaviors, the variable can also be randomized. In the simple slot-chooser case described above, this would not be useful because the choice of slot in a testbench structure has no impact on the stimulus seen by our DUT. For situations where a free variable affects DUT stimulus, however, the complexity and performance cost of randomization is clearly justified. Once again, our formal assumptions on the free variables become assertions in simulation, checking that all stimulus (whether derived from testbench randomization, or from some upstream RTL component) meets the specified requirements.

## V. SURPRISE BENEFITS

In addition to the planned value already outlined, we found that including our formal testbench code in simulation gave rise to several benefits that we did not anticipate.

### *A. Formal and dynamic: two viewpoints on one engineer's work*

Several members of the team were involved in development of both formal and simulation test environments. All these engineers reported that they found themselves applying a distinctly different mindset when working on the two different kinds of verification. This led to excellent cross-checking of each engineer's formal testbench development effort against insights they gained from work on the simulation environment, and *vice versa*.

In particular, our formal testbenches were applied to individual RTL blocks whose microarchitectural specification did not always relate in an obvious way to the overall behavior of the full design. In simulation, though, we were able to visualize and understand[2] the relationship between top-level activity and the resultant (and sometimes surprising) behaviors of each RTL block.

Engineers working full-time on the simulation testbench were often able to point out errors or misunderstandings in the formal testbenches by careful examination of assertions that fired only in simulation. In this way, our decision to apply all formal testbench assertions in simulation gave us yet another kind of cross-checking.

### *B. Reaping the rewards of microarchitecture coverage*

Not only our formal testbenches, but also the RTL designers' modules of low-level assertions were attached to each DUT using SystemVerilog `bind` directives. This made it very straightforward to include any of these assertions in top-level simulation. As a result, our top-level simulation environment was in effect instrumented with low-level probes and checks in each RTL block. These low-level assertions, and especially any cover directives, gave extremely useful internal coverage within design blocks. The level of detail they provided came close to the fine granularity provided by automatic code coverage, but the resulting information was much better focused on design intent and, thanks to descriptive names of each assertion and cover, much easier to interpret. The project made extensive use of this to achieve its microarchitecture coverage goals.

## VI. SUMMARY OF RECOMMENDATIONS AND SUGGESTIONS

Although we encountered many problems of detail when migrating our formal testbenches into simulation, the most important aspects of our experience can be summarized in just a few key points:

*1. Ensure that everyone on the team understands how to avoid inappropriate coding styles and idioms*

Although SVA provides a precise and usually clear language for describing temporal properties, there are numerous coding pitfalls that need to be considered, as described in [7].

*2. Run formal testbench assertions in simulation as early as possible*

Early in our project we attempted to add our formal testbenches to simulation. Partly because of the formal code's immaturity at that stage, but primarily because of limited available debug effort, we found this was impractical at first. Later in the project, as we integrated more and more of the formal code into simulation, the benefits we obtained (improved simulation debug, correction of many flaws in the formal code, general code improvements) were more than sufficient to justify the integration effort.

*3. Encourage all team members to get involved*

In our verification team of about 9 engineers, three were primarily concerned with formal block-level verification and the remainder worked mainly on the top-level simulation testbench. However, some of the most useful debug activity took place when engineers with complementary skills worked together, or sanity-checked each others' work.

*4. Use assume-guarantee methodology, leaving assumptions in place for use as assertions by simulation*

By reversing the roles of assertions and assumptions in re-usable formal verification modules applied to block-to-block interfaces, we were able to implement the traditional assume-guarantee methodology in a straightforward way that also worked successfully in simulation. This approach allowed us to find numerous flaws in the formal verification code. On several occasions, flaws of this type were discovered in simulation *before* they were discovered in formal verification, because the complicated interactions among interfaces hid the symptoms in a single-block testbench, whereas simulation tested all the assumptions and assertions in the full system context.

[2] In a perfect world all levels of the design specification would be complete and there would be no need for such cross-checking. In reality, though, the ability to "grok", or understand viscerally, the overall behavior was valuable to avoid minor misunderstandings of the specification, and to support refinement of each block-level specification in discussion with its author.

*5. Provide informative error messages on assertion failure, even when writing formal code*

Formal testbench writers found it easy, and useful, to add failure error messages to their assertions that would help to debug any assertion failure in simulation. When doing so, it is important to be aware of the correct use of `$sampled` in such messages, as discussed in section IV.A.1 above.

*6. Be aware of possible language inconsistencies between tools*

As mentioned in section IV.B, subtle differences in language support between tools can cause difficulties in simulation. In particular, avoid using any function that reads any variables that are not arguments of the function. Within a property, such variables are sampled in an unexpected and unhelpful manner by simulation - but some formal tools do not honor the language standard in this respect, hiding the coding error and giving convenient but non-portable behavior.

*7. Ensure that liveness properties start only one assertion thread for any given transaction*

This well-known performance problem is discussed in section IV.C.

*8. Be aware that $past, $fell and $rose are not cleared or cancelled by an asynchronous reset*

Section IV.D notes an important pitfall that must be considered: the value returned by `$past` is not in any way affected by an intervening reset. This is rarely a concern in formal verification where reset is used only to establish a starting state, but it can give misleading assertion failures in simulation.

*9. When using free (undriven) variables in formal, consider how they will be given meaningful values in simulation*

Undriven variables, which can take arbitrary values in formal verification, must be given useful values in a deterministic manner for simulation. Auxiliary simulation-only code can accomplish this, and its correctness can usefully be checked by the same assumptions that constrained the variable in formal - see section IV.E.

*10. Take advantage of the simulation coverage that can be obtained from formal code*

As noted in section V.B, formal code can provide useful coverage points on internal block behavior.

## VII. ACKNOWLEDGEMENTS

The author wishes to thank Erik Seligman and Harry Foster for their detailed and expert reviews, and (as always) his colleagues in Verilab for their unfailing support and encouragement.

## VIII. BIBLIOGRAPHY

[1] Mentor Graphics, Inc., "The 2012 Wilson Research Group Functional Verification Study (Part 11)," 26 August 2013. [Online]. Available: http://blogs.mentor.com/verificationhorizons/blog/2013/08/26/9951/. [Accessed 27 January 2015].

[2] IEEE, Standard 1800-2012 for SystemVerilog Hardware Design and Verification Language, New York, NJ: IEEE, 2012.

[3] IEEE, Standard 1850-2005: Property Specification Language (PSL), Piscataway, NJ: IEEE Standards Association, 2005.

[4] V. Singhal, H. Singh, J. Park and N. Kim, "Sign-off with Bounded Formal Verification Proofs," in *DVCon*, San Jose, 2014.

[5] E. Cerny, S. Dudani, J. Havlicek and D. Korchemny, SVA: The Power of Assertions in SystemVerilog (2nd ed), Springer, 2014.

[6] J. Long, A. Seawright and H. Foster, "SVA Local Variable Coding Guidelines for Efficient Use," in *DVCon*, San Jose, 2007.

[7] L. Bisht, D. Korchemny and E. Seligman, "SystemVerilog Assertion Linting: Closing Potentially Critical Verification Holes," in *DVCon*, San Jose, 2012.