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

Jonathan Bromley
Verilab Ltd, Edinburgh, Scotland

This presentation, and the associated paper, were prepared for the DVCon conference in March 2015.
This paper considers some issues around reuse in simulation of assertions (and related code) originally written to implement a formal model checking environment. For expert users of formal model checking, this is nothing new. However, the recent great increase in uptake of formal verification techniques, often by groups and engineers who previously have had little experience with them, means that it may be useful to codify some guidance covering how such reuse can be achieved effectively in practice.
This paper specifically reports on experience with the verification of a large bus-interconnect block. Interfaces on the boundary of this block used standard bus protocols, so that existing verification IP could be used to constrain and check those interfaces. The design was too big to be verified as a single DUT by formal model checking tools, so the overall verification strategy (which had been used successfully on earlier similar projects) called for constrained-random simulation of the entire design. Individual sub-blocks within the DUT, by contrast, were exclusively verified using formal model checking; no simulation testbenches were created for these internal blocks.

The agreed verification strategy also stipulated that formal testbenches for the internal sub-blocks should remain in place for simulation, so that all the formal assertions would also be checked in simulation. The remainder of the paper discusses why and how this was done, and indicates some of the added verification confidence that it provided, and some problems that arose in practice.
SVA enables multi-tool, multi-mode

• SVA is *the same language* for formal and simulation
• In principle, assertions are portable
  – across vendor tools
  – between verification modalities

• Some minor portability issues encountered, but...

The big problems:
inherent differences of approach
between formal and simulation

Of course, reuse of code from formal to dynamic verification would not even be possible were it not for the standardization of SystemVerilog Assertions (SVA). Not many years ago, almost every formal tool used its own notation to capture assertions and properties. Now we can write a property just once, and be confident that it will have the same meaning both in simulation and in formal.

In practice, we encountered a few minor issues of portability of language constructs. These issues were always easily worked around.

There was a much bigger set of problems to face, though, in using the same verification code in both formal and simulation. The idioms, kinds of assertion, and styles of auxiliary logic used by formal verification engineers are sharply different from those commonly encountered in simulation environments. It is those differences that caused us some difficulties in migrating from one verification modality (formal model checking) to another (simulation).
First it is useful to consider the mechanics of how we attached our formal testbenches to their device under test (DUT). We wrote each testbench as a module, and then used the SV `bind` construct to inject that module into the DUT itself as a bound instance. If the testbench's parameter and port lists exactly match those of the DUT module, then it becomes rather easy to establish the necessary connections (although it's a little tedious that .* connection doesn't work for parameters!).

Of course, the testbench module may in its turn contain instances of other modules. That internal structure does not affect the general layout described here.

This binding makes it trivially easy to include the formal testbenches into the simulation environment, because the `bind` statement does not need to specify the full hierarchical path of the DUT module instance but only its module name.
At first it seems completely unnecessary to re-run formal testbenches in simulation in this way. After all, formal model checking is inherently more thorough than simulation which only exercises individual traces. However, there were several important reasons why we obtained real value from this reuse.

For some of the more complex internal blocks, we could obtain only bounded rather than exhaustive proof of some important assertions. Although there are well-known techniques for establishing confidence in such incomplete proof, it provided considerable added confidence to see the same assertions exercised over much longer traces than were possible in formal verification.

Another interesting and often overlooked feature of assertions is that they are inherently bound to individual internal blocks of the DUT, giving a level of detailed internal probing that is difficult to replicate in a simulation environment. This gave access to excellent debug and coverage information that would have been very hard to obtain any other way.

The most important reason, though, was the ability of simulation to uncover buggy assumptions (constraints) in our formal code. We discuss this in more detail in the next few slides.
A major concern on any formal verification project is the risk that assertions and assumptions may be buggy, and fail to reflect the design's specification correctly. Broadly, we can identify faulty assertions and assumptions in four categories.

**Excessively strict assertions** will fire (indicate an error) on correct, valid DUT behavior. This is simply a debug problem, because the error is self-evident.

**Insufficiently strict (too lax) assumptions** will permit out-of-spec behavior on DUT inputs. In most cases this will give rise to faulty DUT behavior, and although the debug may be quite troublesome, this problem too is usually self-evident.

**Overconstraining (excessively limiting) assumptions** will restrict the set of possible DUT input behaviors so that some specified activity is not properly tested. This is very dangerous, because it will not usually give rise to any evident error. If the testbench is instrumented with many coverage points, such problems may come to light, but it's likely that the same misunderstandings of the spec will lead to buggy assumptions and to matching buggy cover directives. It is important to find a strategy for identifying such issues.

**Over-lenient assertions** may fail to detect some kinds of DUT error. Again this is dangerous, because it will not usually give rise to any obvious error.
A key aspect of our formal verification strategy was the use of *reversible formal verification interface blocks* to capture the assertions and assumptions related to the interface between a pair of DUT blocks. Each block’s formal testbench contained an instance of the reversible interface block, configured by parameterization to assert properties on the DUT’s outputs and to assume properties on its inputs. This provided an implementation of the familiar assume-guarantee methodology in formal verification, because overconstraints (assumptions that excessively limit the possible behaviours) on a block’s inputs would be exposed through inappropriate failures of the same assumptions when they are used as assertions on the opposite block’s outputs. Similarly, excessively generous assertions on a DUT’s outputs that might fail to pick up DUT misbehaviour would underconstrain the opposite block’s inputs when used as assumptions, and therefore would be very likely to provoke faulty behavior and assertion failures elsewhere.

In practice we had a comprehensive system of macros and parameterization to automate this reversibility as far as possible, but the principles were exactly as illustrated here.
The reversible interface technique worked extremely well in some situations, but was much less helpful in others. It is ideal when an interface between two blocks follows some self-contained protocol, so that everything the interface block needs to know is explicit from traffic on the interface. In many parts of our internal architecture, however, there were complicated relationships among numerous interfaces linking different blocks. Although it was not difficult to capture these relationships in each formal testbench, the resulting code could not be reversed and re-used on other formal testbenches. Consequently, there were many assumptions and assertions that could not be validated by the assume-guarantee methodology in formal verification.
Limitations of reversible blocks

- Ideal for self-contained (protocol) interfaces
- Less satisfactory for multiple interdependent interfaces
- Conclusion: some important assumptions/ assertions are...
  – not checked by assume-guarantee methodology
  – because not re-used on any other formal TB

The consequence? We need to simulate the formal testbenches, to provide the desired assume-guarantee checking.
Despite having committed to including all our formal testbenches in simulation, it was some time before this goal was achieved. In practice we found numerous difficulties and, in order to make forward progress on other aspects of the project, it was necessary to disable several important formal testbenches for the simulations at first. Hindsight suggests that we should have given more attention to this integration at an early stage, as we will highlight later.

The difficulties fell into two major categories. The first was relatively straightforward: our formal tools were not as aggressive as our simulator in enforcing LRM compliance of some syntax constructs. For example, they allowed some operations on unpacked arrays that the simulator rejected as being appropriate only for packed arrays. Formal testbench authors occasionally introduced such errors because the tool didn't complain about them.

Much more challenging were the problems relating to differences of testbench and assertion architecture between formal and simulation. We examine a few of these in the next few slides.
One small but important issue we encountered was the way in which formal TB authors typically do not provide any error actions on assertion failure. Formal tools provide excellent visualization of the activity that led up to a counterexample, and therefore it is usually unnecessary to have any additional information. In any case, formal does not actually execute the code and therefore has no context in which to execute an error action and display its results. Furthermore, assumptions can never fail in formal and therefore there is no motivation for a formal TB author to add any failure action to an assume directive.

By contrast, in simulation the only information obtained on assertion failure is the point of failure and the details of the failing assertion expression. To obtain fuller debug information it is usually necessary to enable a special "assertion trace" mode in the simulator, which can be expensive especially for a long simulation. Consequently it is very useful to have assertions report detailed error diagnostics on failure. We tried to encourage our formal TB authors to do this by providing mandatory error-action arguments in the most widely used assertion/assumption macros.
There is a specific problem associated with reporting detailed information in an assertion's failure action. The action is scheduled for execution in the Reactive region of the SystemVerilog scheduler. In all realistic cases, this means that the failure action will be executed after all nonblocking assignment updates to DUT and testbench signals have taken effect - the situation after the clock tick, although at the same simulation time. However, the assertion itself invariably makes use of signal values sampled in the Preponed scheduler region, and therefore is testing signal values as they were just before the clock tick. This can give rise to bizarre and hard-to-understand error reports, as shown here.
This mismatch of reported vs. sampled signal values is very easy to fix. The failure action (and only the failure action!) should report the $sampled value of signals. This system function returns the value of a signal or expression as it was in the Preponed scheduler region, and so reports values exactly matching those used in the body of the assertion.
Recommendation

• use of $sampled well described in this book
  – along with many other things

big thank-you!

This, and many other important and subtle aspects of SVA behavior, is very thoroughly and clearly described in this book.
We spent considerable effort on debugging a problem that eventually was found to be the result of our chosen formal tool behaving in a manner that does not comply with the SystemVerilog LRM, but appears entirely sensible at first glance. As a result, we had written some code in our formal TBs that was incorrect, but behaved exactly as we intended in our formal tool. When the same code was run in simulation, our simulator of course fully honored the LRM semantics and gave unintended results.

When a function is called as part of an expression in an assertion, the function's input arguments are evaluated using the same $sampled (Preponed) semantics as any other variable in the assertion. However, if the function reads any non-argument (global) variable, such as the mode bit invertedMode in our example, that variable is evaluated in the Observed scheduler region - effectively, its value immediately after the clock tick. In most realistic cases, this behavior is surprising and does not match the desired intent. Our formal tool erroneously gave the results we hoped for, by using $sampled semantics for all such variables.

It's not feasible to use $sampled within the function, because it is not necessarily in a properly clocked context. Consequently the best fix was to ensure that all signals of interest were passed as arguments to the function. Although this was quite tedious to implement, it probably led to better code style.
When we finally incorporated our formal testbenches into simulation, there was some evidence of a reduction in simulation performance. Considerable effort was invested into examining profiler results to seek out the reasons for this performance hit.

By far the most important performance problem was caused by liveness properties that retrigger multiple times for any given transaction, as indicated on this slide. The property shown at the bottom of the slide causes repeated restarts for the indicated trace, leading to very large numbers of concurrent assertion threads that are very costly of simulation performance (although there is essentially no impact on formal tool performance).

By simply recoding such assertions so that they fire only once per transaction, as indicated in the upper code box, performance was dramatically improved. There is certainly no need to avoid all liveness properties, but great care must be taken to avoid liveness properties (which, of course, are usually long-lived) that retrigger repeatedly.
A more troublesome issue was caused because several of our formal testbenches used a technique that we colloquially called "free variables". (Note, this is not exactly the same as the strict definition of "free variable" that you will find in the SystemVerilog LRM's description of the checker construct.)

As you can see from the code example, our formal testbench used an undriven variable `txn` to capture the "type" of a transaction, simplifying later coding. The various DUT-facing strobe and attribute signals, such as `busWnR`, can be derived from this transaction type by assumptions. Later, the type variable can be used in checks and other logic, rather than having to inspect various strobe signals individually. This approach can greatly contribute to code clarity.

However, it simply doesn't work in simulation. The `txn` variable is undriven, and will conflict with the values of strobe signals that are driven by the testbench or other parts of the overall DUT.

We worked around this, without disturbing the formal testbench code, by writing functions that could derive the transaction type from the actual values of strobe and other signals. With this simulation-only driver in place, the assumptions now act as a check on the validity of the conversion function, assuring consistency between formal and simulation testbenches.
In addition to the important and anticipated benefit of robust checking of assertions and assumptions using assume-guarantee, we obtained a number of useful but unexpected benefits from inclusion of our formal testbenches into the simulation environment.

First, and most important, was the cross-checking that arose because multiple engineers were now taking an interest in the details of each formal testbench. There is no doubt that this peer review had a strong positive effect on code quality.

Additionally, we found that the deep embedding of formal code into the DUT, combined with the auxiliary logic that is a major part of any formal testbench, made it much easier for us to add coverage points that observed microscopic RTL activity in a useful way. For example, fill levels of internal data structures in key DUT blocks were readily accessible to coverage constructs. Thanks to the bind methodology described earlier, this advantage came without any special effort.
Four key technical guidelines that emerged from our efforts to integrate formal testbenches with the simulation environment. Although all these guidelines are fairly straightforward and obvious, we would have saved ourselves a lot of time if we had taken careful note of them from the outset!

- Provide informative error messages on assertion fail
- Be aware of possible semantic inconsistencies
- Avoid liveness assertions that create numerous long-lived threads
- Plan for handling of undriven TB variables in simulation
The benefits of integrating formal testbenches into the simulation were sufficiently telling that it soon became clear that we should have applied much more early debug effort to it than we did. Reviewing the project led us to suggest the methodology guidance given here. We certainly intend to follow it ourselves on future projects.
Verilab always welcomes feedback on its published materials. If you have any comments or questions, please feel free to contact the author.

Questions?
jonathan.bromley@verilab.com