YosysHQ AppNote-011

YosysHQ GmbH

Jan 08, 2024

CONTENTS

CHAPTER

ONE

FAQS RELATING TO SBY

1.1 Documentation

Q: Where are the docs?

A: SBY docs

1.2 SystemVerilog Assertions (SVA)

Q: What subset of SVA is supported?

A: Refer to the SBY docs: Supported SVA property syntax

1.3 Choosing an engine and solver

Q: How do I know which formal engine and/or which solver to use for verifying my design?

A: Different combinations of engine and solver will perform differently and may support different features or functionalities. The SBY engines reference lists all currently supported engines, their options, available solvers, and which modes they operate in. At this time, a comprehensive comparison of when to use different engines/solvers is not yet available.

With the introduction of the `--autotune` option it is now possible to automatically compare available engines for a given SBY task. Autotune will run each engine, providing a report of time taken and the status returned. This can be used to quickly determine the fastest configuration while continuing to iterate on a design. Check out the autotune docs for more information.

1.4 Tool runtime

Q: I tried one of the examples included with the tool, and formal run looked stuck and didn't get completed in half an hour. Not sure if engine selection has anything to do with it. How much to wait in such case?

A: How long to wait is impossible to know, it depends entirely on the problem you are working on. Half an hour is not unusual though.

1.5 Proof complexity

Q: Does SBY, or any of the supported solvers, provide functionality to evaluate the complexity of a proof? Or is there some hint on how to restructure the RTL Code to simplify the problem?

A: We don't have anything like that currently. The question has generated a bit of discussion on our slack for how we could estimate it - it's not easy to correlate the activity of variables that end up in the SAT solver with anything in the design, given the amount of rewriting that happens in the intermediate layers. There were two suggestions that can give some proxy values that should correlate with the complexity:

- Systematically simplifying the problem by adding assumptions that fix different parts of the inputs (so basically moving a slider between full proof and specific testcase) until things are solved quickly is probably the most approachable way to get some insight into solver performance for a specific problem.
- Using the amount of logic that is in the COI of the proof as a crude estimate for the complexity. We don't have a user-friendly interface for this but it can be done with the yosys CLI. If you have already run SBY, you can use the command `yosys -p 'stat t:$assert t:$assume %ci*' <sby task folder>/model/design.il` to print some statistics of the logic feeding into the assume and assert statements in your code (use `'stat t:$cover t:$assume %ci*'` for cover tasks). This won't take into account all of the factors that affect performance - e.g. deeply pipelined code, where the variable state depends on many previous cycles, often has particularly poor performance beyond what the amount of registers would suggest. (If you want to see this data before running SBY, open a yosys shell and run the commands from the `[script]` section before running `'stat t:$assert t:$assume %ci*'`.)

1.6 Where do assertions fail

Q: How do I see what is causing my assertion to fail?

A: If an assertion is failing, SBY will provide a counterexample trace. Provided you are not using the `append` option, the final cycle in this trace is the cycle in which the assertion does not hold. Check out our quickstart guide for a worked example of examining and fixing a failing assertion.

1.7 Design initialisation

Q: Why does my design not get reset properly at the start?

A: SBY does not consider reset signals special. If you want to restrict your proof to only certain behaviors of the reset signal, add an `assume()` statement enforcing the reset sequence, e.g. `initial assume(reset);` (Yosys also adds the non-standard `$initstate` for use in conditionals, e.g. assume property (`@(posedge clk) $initstate |-> reset [*3]);`).

Where possible we encourage writing your properties in such a way as to be able to leave the reset signal unconstrained after the initial cycles, so as to check for bugs that might occur after a soft reset.

1.8 Clock signals

Q: How does SBY detect and handle clock signals?

A: How SBY treats the clock signals differs depending on if you are using multi-clock mode (`multiclock on` in the `[options]` section) or not. In single-clock mode, the clock signal's actual value is disregarded, we assume all registered signals update simultaneously, and the solver has one variable per signal per clock cycle to determine. So internally, the solver will actually never see the clock change, and we artificially add the toggling of the clock signal when generating the trace - but that can lead to some discrepancies if there are any signals that are assigned the value of the clock signal (such as with submodules).

In multiclock mode, the clock signal is instead treated as a regular input, and the solver can freely choose whether to toggle it, unless you add assumptions. This means that the clock signal will not obey the implicit rules of clock signals like having a consistent period or duty cycle, but while surprising at first, this is actually not a disadvantage most of the time - when the clocks are not related, it's almost always possible to eventually reach a specific interleaving of clock edges if you let the system run long enough. Not having those constraints in place means that the solver can find the worst case in only a few steps, giving you a short trace. With the constraints, getting to that point might take so long that the problem becomes computationally intractable. If your clocks are actually related, do add an assumption about that.

Q: When do I need to enable multi-clock mode?

A: You need to set `multiclock on` in the `[options]` section whenver the design contains entities that are sensitive to different events. This includes:

- multiple clock signals
- multiple edges of the same clock signal
- any asynchronous logic (with the exception of asynchronous resets that should be treated as synchronous)

1.9 Semantics of "disable iff"

Q: I would have expected the following to pass. Why does it not pass?

```
assume property (@(posedge clock) A |-> B disable iff (reset));
assert property (@(posedge clock) A && !reset |-> B );
```

A: Both of those properties are two simulation cycles long, because the clock edge between those two cycles is part of the property. The `disable iff` statement behaves similar to an *asynchronous* reset that is not sampled by the clock, thus the sequence `A && !B && !reset ##1 reset` will disable the assumption, but will not disable the assertion in the above example.

1.10 Witness cover traces

Q: How do I produce witness cover traces for a passing assertion?

A: Check out the witness cover section of our whitepaper, Weak precondition cover and witness for SVA properties.

1.11 Can liveness properties fail

Q: Is it possible to have liveness property to fail? Or will it just get stuck in formal run

A: We don't recommend using liveness properties - it's almost always better to replace with an assertion of something happening within a certain timeframe.

The example our CTO gives is of a design that is stuck in a deadlock, but it has a 64 bit counter and when that overflows, things start up again. Liveness will tell you "yup, this design will do things eventually" but it really doesn't help you because that 64 bit counter is so large that your design will basically never start again.