Verifying hardware designs with SymbiYosys

John Wickerson

November 29, 2023

Contents

Chapter 1

Introduction

Yosys is a collection of free and open-source (FOSS) tools for building hardware. It includes tools for synthesising designs for various FPGA architectures, and also for ASICs. Of particular relevance to this module is the part called *SymbiYosys*, which includes several tools for *verifying* hardware designs. SymbiYosys can:

- verify *safety* properties (i.e., that something *bad* doesn't happen),
- verify *liveness* properties (i.e., that something *good* eventually happens),
- verify that two designs are *equivalent* (i.e., that a specification and an implementation have the exact same behaviour),
- check *coverage* (i.e., that running a given set of inputs exercises every part of the hardware design).

In this module, we will see how to use SymbiYosys to verify hardware designs. We will also learn about the underlying technology that SymbiYosys uses to perform that verification.

https://symbiyosys.readthedocs.io/

Chapter 2

Getting started

SymbiYosys can be installed on Windows, Linux, or Mac by following the instructions at https://symbiyosys.readthedocs.io/. SymbiYosys is part of the "OSS CAD Suite" which can be downloaded from Github. Various other useful tools, such as Boolector and GTKWave, are also included in the download.

On my Mac, installation was a straightforward matter of downloading the relevant `.tgz` archive from Github, unpacking it into a directory of my choice on my machine, and updating my `PATH` environment variable to include that directory so that the various executables in the suite (`sby`, `gtkwave`, and so on) can be invoked from any working directory.

SymbiYosys is also part of the commercial "Tabby CAD Suite", which requires an additional licence (free for students taking this module). When the Tabby CAD Suite is enabled, it becomes possible to write richer assertions – assertions that don't just talk about a single state, but about a sequence of consecutive states. The Tabby CAD Suite software isn't available via GitHub – instead, a download link will be provided for students taking this module.

Chapter 3

An example

Listing 3.1 shows a very simple Verilog module. Here is a line-by-line explanation of what is going on in that file.

1. The module is called `counter`.
2. It has one input port called `clk`, which is a single wire. This is a clock signal.
3. It has one output port called `count`, which is a 6-wire bus.
5. The initial value at the output `count` is 0.
7. At every rising edge of the clock ...
8. ... if the `count` has reached 15 ...
9. ... then return it to 0 ...
10. ... and if it hasn't ...
11. ... then increment it by 1.
14. The rest of the module is only relevant to the verification process; it can be ignored when the module is synthesised.
15. At all times ...
16. ... the value of `count` should be less than 32.

In short, we have a counter that repeatedly outputs the sequence 0 to 15. Note that some of the output wires (`count[4]` and `count[5]`) stay low forever, which is a bit quirky but ok. Also note that we are checking that the output never reaches 32, which is a quite a weak property. A stronger statement could be made: that the output never actually reaches 16!

Listing 3.1: A simple Verilog module `counter.v`

```
module counter (
  input clk,
  output reg [5:0] count
);
  initial count = 0;

  always @(posedge clk) begin
    if (count == 15)
      count <= 0;
    else
      count <= count + 1;
  end

`ifdef FORMAL
  always @(*) begin
    assert (count < 32);
  end
`endif
endmodule
```

3.1 Bounded model checking

We can use SymbiYosys to check that our assertion always holds. To do so, we must provide it with a "task file". Listing 3.2 gives an example of such a file. Here is a line-by-line explanation of what is going on in that file.

2. We will be performing *bounded model checking* (BMC) to verify the correctness of our design.

3. The model checker will explore up to a depth of 100 clock cycles.

6. The bounded model checking will be performed by a tool called `smtbmc`.

9. The design should be read from the `counter.v` file.

10. The design should be prepared for verification. Among other things, this process involves flattening the design into a single module.

We can run this task file, which is called `counter.sby`, by invoking SymbiYosys as follows:

Listing 3.2: The task file `counter.sby`

```
[options]
mode bmc
depth 100

[engines]
smtbmc

[script]
read -formal counter.v
prep -top counter

[files]
counter.v
```

```
sby -f counter.sby
```

SymbiYosys should produce several lines of output. The important part is at the end, where it reports that the design has passed verification:

```
engine_0 (smtbmc) returned pass
```

It is also instructive to see what happens when the verification process *fails*. To do this, we can try changing the correct assertion `assert (count < 32)` to the incorrect assertion `assert (count < 15)`. Now, when we invoke SymbiYosys, we get:

```
engine_0 (smtbmc) returned FAIL
counterexample trace: counter/engine_0/trace.vcd
```

The output indicates not only that SymbiYosys has failed to prove the assertion, but it has also prepared a trace showing a counterexample; that is, a timing diagram showing exactly how the hardware can execute in the lead up to the assertion failure. This timing diagram can be inspected using a waveform viewer:

```
gtkwave counter/engine_0/trace.vcd
```

If you click on `counter` in the 'SST' (signal search tree) window, and then double-click on the `clk` and `count[5:0]` signals in the window below, then they will appear in the 'Waves' window. If you then click on the `witness` in the

SST window (which may require clicking on the little '+' next to `counter`), and then double-click on the `anyinit_procdff_16` signal in the window below, then you will see a wave that shows where the assertion holds. You should be able to see that on the 16th positive edge of `clk`, the assertion stops holding: this corresponds to the value of `count` hitting 15.

3.1.1 A shortcoming of bounded model checking

Now let's see something a bit unnerving. Modify the task file by changing `depth 100` to `depth 10`. Leave the incorrect assertion `assert (count < 15)`, and rerun SymbiYosys. You should get the following result:

```
engine_0 (smtbmc) returned pass
```

What has happened here is that SymbiYosys has only been instructed to explore up to a depth of 10 clock cycles from an initial state, but the assertion only fails after 15 clock cycles. Therefore, SymbiYosys reports that there are no problems. We are seeing here the shortcoming of *bounded* model checking: it can only provide guarantees about the design up to its exploration bound. This can be fine; for instance, an engineer might know that their design will only ever be run for 42 cycles before being turned off, so a bound of 42 in these circumstances provides all the guarantees that are needed. But in most cases, we'd probably prefer to get stronger guarantees. We'd like to know that our design is correct however many cycles it runs for.

3.2 Using k-induction

Reset your design and task file to how they were in Listings 3.1 and 3.2, and then change `mode bmc` to `mode prove`. This tells SymbiYosys to try to prove the design correct (for an unbounded number of clock cycles) using a technique called k-induction. The value of k is set by the `depth` parameter, so, 100 in this example. When you rerun SymbiYosys, you should see:

```
successful proof by k-induction
```

But if you change `depth` down to 10, the k-induction fails.

```
engine_0 (smtbmc) returned FAIL for induction
```

The idea of k-induction is:

- Consider an arbitrary chain of consecutive states of length $k + 1$.

- Assume that the assertion holds in all of the first k states in that chain.
- Show that the assertion must also hold in the $(k+1)$th state in that chain.

If such a proof can be completed, then we can deduce that the assertion must hold in all states that are reachable from an initial state. To see why:

- We already know that the assertion holds in all states that are reachable in no more than k steps from an initial state, because we completed our bounded model checking.
- Now consider a state that is only reachable in $k+1$ steps. Call that state σ. State σ is preceded by a chain of k states, all of which are reachable from an initial state in no more than k steps. Therefore we have a chain of $k+1$ states where the assertion holds in the first k of them. So, by our k-induction proof above, we know that the assertion must hold in the last state in that chain, which is σ.
- We therefore now know that the assertion holds in all states that are reachable in no more than $k+1$ steps from an initial state.
- Now consider a state that is only reachable in $k+2$ steps. Call that state σ'. State σ' is preceded by a chain of k states, all of which are reachable from an initial state in no more than $k+1$ steps. Therefore we have a chain of $k+1$ states where the assertion holds in the first k of them. So, by our k-induction proof above, we know that the assertion must hold in the last state in that chain, which is σ'.
- We therefore now know that the assertion holds in all states that are reachable in no more than $k+2$ steps from an initial state.
- We can repeat this logic an unlimited number of times, moving from $k+2$ to $k+3$ to $k+4$, and so on. Ultimately, we deduce that the assertion holds in all states that are reachable (in any number of steps) from an initial state.

In our example, we saw that k-induction failed when $k = 10$. That's because it is able to form a chain of 10 steps (`count=22` $\rightarrow$ `count=23` $\rightarrow$ `count=24` $\rightarrow$ `count=25` $\rightarrow$ `count=26` $\rightarrow$ `count=27` $\rightarrow$ `count=28` $\rightarrow$ `count=29` $\rightarrow$ `count=30` $\rightarrow$ `count=31`), all of which satisfy the assertion `count < 32`, but which are followed by the state `count=32`, which violates it.

While it's true that this sequence of states does lead to an assertion failure, it doesn't actually matter, because the first state, `count=22`, isn't reachable from an initial state. So how can we stop SymbiYosys coming up with this spurious counterexample? We can do one of two things. We can either increase the value

of k so that SymbiYosys is unable to find k consecutive good states that precede a bad one ($k = 17$ should do the trick). Alternatively, we can define more of the unreachable states to be bad. It doesn't affect the hardware if we define unreachable states as 'good' or 'bad', because they're unreachable either way, so we might as well define as many of them as possible as 'bad'. This amounts to adding an assertion to rule out all of the (unreachable) states where `count` is above 15; or, to put it another way, changing `assert (count < 32)` to `assert (count < 16)`. If we do this, k-induction with $k = 1$ will suffice.

3.3 Writing simple temporal assertions

The assertions we have written so far have only talked about a single state. It is also possible to write assertions that need to be evaluated over multiple states. This allows us to express properties like "whenever the reset signal goes high, the data-out signal goes low within three clock cycles". Properties like this are called 'temporal' properties, because unlike for properties of a single state, there is the added dimension of the advancement of time.

SymbiYosys allows assertions to use the `$past(...)` operator to refer to the value that a given expression had in the previous clock cycle. For instance, we might extend our previous example with the following extra assertion:

```
always @(posedge clk) begin
  assert (count == 0 || count == $past(count) + 1);
end
```

This assertion says that the value of `count` must either be zero or be 1 more than the value of `count` in the previous clock cycle. This assertion is evaluated at every rising edge of the clock (`posedge clk`), and indeed, assertions that use `$past(...)` *must* be evaluated in this way.

3.4 Writing SVA assertions

If SymbiYosys is extended with SystemVerilog assertions (SVA) (which requires an additional licence), then these assertions can become more expressive. For instance, the assertion can be rewritten as an implication, like so:

```
always @(posedge clk) begin
  assert property (count > 0 |->
                   count == $past(count) + 1);
end
```

It is also possible to give the desired property a name, and then assert it using that name, like so:

```
property going_up;
  count > 0 |-> count == $past(count) + 1;
endproperty

always @(posedge clk) begin
  assert property (going_up);
end
```

It is also possible to move the trigger *inside* the assertion, like so:

```
property going_up;
  @(posedge clk)
  count > 0 |-> count == $past(count) + 1;
endproperty

assert property (going_up);
```

SVA provides a 'disable iff' construction, which is a convenient way to express that certain properties shouldn't be checked under certain conditions. We could use it instead of our implication, like so:

```
property going_up;
  @(posedge clk)
  disable iff (count == 0)
  count == $past(count) + 1;
endproperty

assert property (going_up);
```

One final SVA feature that is worth mentioning is the two kinds of implication. We write `p |-> q` for an *overlapping implication*, which means that if `p` holds in the current state, then `q` must also hold in that state. The other kind is the *non-overlapping implication*, which is written `p |=> q`, and means that if `p` holds in the current state, then `q` must hold in the *next* state.

3.5 Checking coverage

We might like to know not only that certain *undesirable* states are *not* reachable, but also that certain *desirable* states *are* reachable. To do this, we can write `cover` statements. As a simple example, we might like to reassure ourselves that

not only does `count` never go as high as 32, but also that at some point it *does* reach 13. We add the following statement to our code:

```
always @(*) begin
  cover (count == 13);
end
```

We can ask SymbiYosys to perform coverage checking at a depth of 100, like so.

```
[options]
mode cover
depth 100
```

SymbiYosys reports that it was able to find a trace that reaches the state `count=13` in just 13 steps.

```
        engine_0 (smtbmc) returned pass
cover trace: counter/engine_0/trace0.vcd
reached cover statement [...] in step 13
```

(Note that when SymbiYosys is in `bmc` or `prove` mode, finding a trace indicates a *problem* with the design, but when it is in `cover` mode, the situation is reversed, and finding a trace is a *good* thing!)

We can also ask SymbiYosys to perform coverage checking at a depth of just 10, like so.

```
[options]
mode cover
depth 10
```

Of course, it cannot reach the state `count=13` in just 10 steps, so the check fails.

```
      engine_0 (smtbmc) returned FAIL
engine_0 did not produce any traces
```