C2S2 Verification Training

Aidan McNay, acm289

C2S2

This tutorial should serve as an introduction to the tools and methodologies used in formal verification. By the end, you should be familiar with SymbiYosys and the SystemVerilog functions available for formal verification, have experience describing how formal verification should be executed to SymbiYosys (both in the relevant module and an `.sby` file), know techniques to speed up formal verification, and have performed it on your own designs. Let's get started!

1 Setup

To setup your environment, first log into the C2S2 server using the SSH protocol. The server is `<NetID>@c2s2-dev.ece.cornell.edu`, and the password should be your NetID password. (If you've never logged in using SSH before, please see the Linux Development Environment Tutorial). From there, you should always begin by sourcing the setup script for C2S2:

```
% source setup-c2s2.sh
```

(Note that in this tutorial, a "%" preceeding a terminal command only indicates that it is such, and should not be included in the terminal command)

Next, we'll create the space we'll be working in, including cloning the source code from GitHub (if you haven't set up GitHub yet, please go back and complete the GitHub tutorial)

```
% mkdir -p ${HOME}/c2s2
% cd ${HOME}/c2s2
% git clone git@github.com:cornellcustomsiliconsystems/verif_tut.git
% cd verif_tut
% TUTROOT=${PWD}
```

Great! Now we're all set to start designing.

2 Formal Verification Background

Most coders are aware of the importance of test cases; they help to make sure that your code runs as expected. However, how do you know when you have enough test cases? Are you sure that you've covered *all* possible situations?

From this, a need for "solvers" was born; something that could tell you definitively whether a condition was satisfiable or not for any set of inputs. In coding, we use SAT solvers (and their extension, SMT solvers) to attempt to prove whether a condition in our code could ever be satisfied, as well as whether it could ever not be satisfied. These two checks are what our solvers are based around doing

One can think of hardware as always having a "state", which is the set of values of some or all of the registers and wires at a particular time. Using this, we can refine our statement for the two checks that solvers can check for. The first check can be expressed as "Given state A, is there a valid transition to state B?" If there is, such a transition is called a **solution**. The second check can be expressed as "Given state A, can we ensure there are no valid transitions to state B?". If there is not, then the design is said to have been **proved**, but if the solver finds a valid transition, then the design fails to prove.

Specifically, in this tutorial, our solvers are *bounded*; they will only check up to a pre-determined number of states, known as the **depth**. This isn't so much an issue for our first check; if we find a valid solution, there is no need to proceed further. However, this poses an issue for our second check; we can prove that a model won't reach a given state for N states, but how about the N+1 state? This is a natural limitation of our **Bounded Model Checker (BMC)** (a specific name for checkers who attempt to prove designs for a given number of states). The natural continuation is to use *k-induction* to show that, given N "correct" states, the model will be "correct" for all time. While that is outside the scope of this tutorial, I would encourage you to research it more.

3 Formal Verification Tools

In this tutorial, we will be using SymbiYosys (sby) as our verification tool. SymbiYosys is an open-source driver for verification flows. Specifically, we will be using its SystemVerilog extension to include verification statements in our SystemVerilog code. These statements are non-synthesizeable (meaning that they cannot be modeled as hardware like the majority of SystemVerilog code), but instead can be used by SymbiYosys to verify various attributes of your code. In particular, SymbiYosys supports three assertions:

- **`assume<expr>;`** - SymbiYosys only considers solutions where the expression inside is always 1. Use this to limit the field of solutions to only those relevant (ex. only those for where you reset at the beginning)
- **`cover<expr>;`** - SymbiYosys will attempt to find a solution to make the given expression 1. It is used to prove that a state is reachable, and is used to prove **liveliness properties**. This directly corresponds to our first verification check.
- **`assert<expr>;`** - SymbiYosys will attempt to find a way to make the given expression 0 (a.k.a. break the expression). It is used to show that a design is "safe" (it won't ever reach a "bad" state), and is used to prove **safety properties**. This directly corresponds to our second check.

Note: These assertions can only be used in an immediate context; we can only check the assertions at discrete times. Therefore, they must be used only inside `always` blocks (or less commonly, an `initial` block). For this tutorial, we will limit ourselves to only clocked blocks, as putting these checks inside `always @(*)` can lead to the solver performing excessive checks when it doesn't need to, as signals usually only change on the clock edge in RTL design, so we only need to check there.

In addition, SymbiYosys provides us with 5 functions that can assist us in our formal verification, given as follows:

Table 1: Formal Verification Functions

Function Name	Function Semantics
`$past`	`$past(<expr>)` returns the previous value of `<expr>`*
`$stable`	`$stable(<expr>)` is the same as `(<expr> == $past(<expr>))`
`$changed`	`$changed(<expr>)` is the same as `(<expr> != $past(<expr>))`
`$rose`	`$rose(<expr>)` is the same as `(<expr> && !$past(<expr>))`
`$fell`	`$fell(<expr>)` is the same as `(!<expr> && $past(<expr>))`**

***Note:** All of these functions have the notion of a "past" value of an expression. This value is defined as what the expression would've evaluated to one clock cycle ago, and therefore these functions can only be used inside of clocked `always` blocks. (While they could theoretically be valid inside `always @(*)`, you would be checking against past values from the last time the block triggered when you don't know how many times the block has triggered, so it's not very useful - I haven't checked, but Yosys might know about this and throw an error. Regardless, it is certainly best practice to put them inside clocked blocks)

Additionally, when a design starts up, `$past` doesn't have any evaluation (the simulator doesn't know what the value of any expression was before time 0), so if we try to check `$past(<expr>)`, we will get an error. To get around this, common practice is to use another variable to check if we've gone beyond the initial state, and then only check `$past` once this variable says it is valid to:

```
1 logic f_past_valid = 0;
2 always @( posedge clk ) begin
3     f_past_valid <= 1;
4     if( f_past_valid )
5         assert( $past(y)==y );
6 end
```

****Note 2:** The functions `$rose` and `$fell` are meant to be used on a single-bit signal (to see if it rose from 0 to 1, or fell from 1 to 0). If they are given a multi-bit signal, they will only consider the least-significant bit (LSB) of that signal, and will evaluate based on whether that bit rose or fell, not the entire signal.

Final Note: While Yosys is good at most of SystemVerilog, it doesn't support all of it. If you happen to find yourself using a particularly obscure SystemVerilog construct that Yosys doesn't understand, consider using Verilog instead, which Yosys fully supports (perhaps with the aid of SV2V)

4 Demonstrating Verification with an Adder

We'll begin by demonstrating the functionality of `assert`. When learning a new tool, it is always best to start simple, so we can demonstrate this using a simple 8-bit register, with the implementation below:

```
1  module eightbit_reg
2  (
3      input  logic       clk,
4      input  logic [7:0] in,
5      input  logic [7:0] out,
6  );
7
8      logic in_reg;
9      always @( posedge clk ) begin
10         in_reg <= in;
11     end
12
13     assign out = in_reg;
14
15 endmodule
```

We can modify this using our verification tools to demonstrate the functionality. In this case, I will use an `assert` statement to show that `out` always takes the past value of `in`. We can make modifications to our register like this:

```
1  module eightbit_reg
2  (
3      input  logic       clk,
4      input  logic [7:0] in,
5      input  logic [7:0] out,
6  );
7
8      logic in_reg;
9      always @( posedge clk ) begin
10         in_reg <= in;
11     end
12
13     assign out = in_reg;
14
15     `ifdef FORMAL
16
17     logic f_past_valid = 0;
18
19     always @( posedge clk ) begin
20         f_past_valid <= 1;
21         if( f_past_valid )
22             assert( out==$past(in) );
23     end
24
25     `endif // FORMAL
```

```
26
27 endmodule
```

Note that I put all of our verification logic inside an `ifdef. SymbiYosys defines FORMAL when it runs, so this is a way to ensure that our verification logic is only included when we're running verification, and not when we're doing something else (like synthesizing, where it would throw errors).

This file is given for you in $TUTROOT/reg/reg.v. Now we're all set to run our verification!

5 Setting up SymbiYosys

We have our SystemVerilog file with all of our verification checks, so now all we need to do is to tell SymbiYosys how we want to run verification on our model. This is contained in a .sby file. This file contains 5 things (in this order, by convention only):

- **Tasks (optional):** This is where we can outline the tasks we want to run. The reason it is optional is that we can always default to only one task, but if we want to run multiple simultaneously, we can define multiple here
- **Options:** This is where we specify the type of verification we want to run (if we have multiple tasks, we would define it for each task). For this tutorial, we will only ever use `cover` and `bmc`, which are used for `cover` and `assert` statements, respectively. Additionally, we can set the depth of these checkers right below (the default depth is 20)
- **Engines:** This is where we decide which engine to give which tasks (as well as possibly specifying the specific solver that the engine should use). If we only have one task, we only need to specify the name of the engine, but if you have multiple tasks, you may need to specify which engine to assign to each task. For this tutorial, we will start off using the `smtbmc` engine, which can handle both `cover` and `bmc`
- **Script:** This is where we specify the Yosys script for the verification. For this tutorial, this will only involve reading the necessary files and synthesizing the top-level module using the `prep` command
- **Files:** This is where we tell Yosys where to find all of our source files by giving it the paths to all of them. Note that this is relative to **the .sby file**, NOT where you're running SymbiYosys from (this was the reason why I didn't use a separate build directory in this tutorial, as to avoid adding more complexity)

If you wish to learn more about how to set up this file, and the different commands you can put into it, you can read more on the SymbiYosys website, or about the scripting commands by downloading the Yosys documentation.

An example .sby file for this project is given below (assuming it's in the same directory as reg.v):

```
[options]
mode bmc
depth 100

[engines]
smtbmc

[script]
noneread -formal reg.v
prep -top reg

[files]
reg.v
```

Note that we could've been more explicit by defining a task and setting the option for that specific task, as well as setting the engine for a specific option, but since we're only using one, it isn't necessary. Nevertheless, I'll show it below for completion:

```
[tasks]
task1

[options]
task1:
mode bmc
depth 100

[engines]
task1: smtbmc

[script]
noneread -formal reg.v
prep -top reg

[files]
reg.v
```

This file is given for you in `$TUTROOT/reg/reg.sby`. From here, we can run our verification (the `-f` flag specifying that we should overwrite the previous output directory, if any):

```
% cd ${TUTROOT}/reg
% sby -f reg.sby
```

Your output should hopefully tell you that the register passed the test case! You should also get an output directory titled `reg_task1`, or simply `reg` if you didn't define it as a separate task. Going into there, we can see all of the outputs that SymbiYosys gives us.

What happens if our verification fails? Go ahead and change our assert statement in `reg.v` to:

```
assert( out==in );
```

This isn't always true; our input to a register can be different than our output. Let's try running it again:

```
% sby -f reg.sby
```

Our output should now tell us that the verification failed! In addition, in our output directory, under the engine designator for the engine we're using (in this case, `engine_0`), it will give us some output that can help us understand where our design failed to verify, including a waveform of a set of inputs that were used to violate the `assert`:

```
% cd reg_task1/engine_0
% gtkwave trace.vcd
```

(Note that you have to be in a graphical viewer to view a waveform, such as X2Go or MobaXTerm)

Great! We've demonstrated that we can verify a design is *safe* (it will never reach a "bad" state) with `assert` statements. Now, let's try to see how we can use `cover` as well!

6 Demonstrating Verification with a Logic Puzzle

`cover` statements are able to tell us whether a given state is reachable. To me, this sounds a lot like a puzzle; are there a specific set of inputs that can get us to a desired output? To demonstrate this, we can use Verilog to solve a puzzle!

I'll use the example of the sliding-puzzle. If you're not familiar with the game, it is where a person is given a grid of randomized numbered tiles with one empty slot, and by sliding the tiles into the empty slot (thus moving it around), one can re-arrange the tiles. The goal of the puzzle is to get all the tiles in order (going across rows, then down columns), with the empty slot in the bottom-right hand corner.

Figure 1: Sliding Tile Game

If we can represent this game in Verilog, we can use the formal solving engine to attempt to solve it!

First, let's define our interface. We should have 3 inputs; a `clk` and `reset` signal, and a 2-bit inputs that tell us which direction we want *the empty slot* to move (those being left, right, up, and down). Note that moving the empty slot left is the same as moving the block to the left of the empty slot right, so this is a valid way to describe moving the tiles. On clock edges, we will (if possible) "move" the empty slot in our Verilog representation of the board in the indicated direction:

```
module sliding_tile
(
    input logic       clk,
    input logic       reset,
    input logic [1:0] direction
);
```

Let's define which combinations of bits should indicate which direction (which I'll do with `localparam`):

```
localparam LEFT  = 2'b00;
localparam RIGHT = 2'b01;
localparam UP    = 2'b10;
localparam DOWN  = 2'b11;
```

Next, we need to find some way to represent our board. I will do this by having 9 total registers, each of which represent a space on the board, and which hold the value of the tile in that space at any given time (with 0 representing the space). I'll also have a 4-bit register named `space_loc` that details the location of the space at any given time; the first two bits represent the row (0-2), where the second two bits represent the column (0-2), where we start indexing from the top-left:

```
logic [3:0] top_left;
logic [3:0] top_middle;
logic [3:0] top_right;
logic [3:0] middle_left;
logic [3:0] middle_middle;
logic [3:0] middle_right;
logic [3:0] bottom_left;
logic [3:0] bottom_middle;
logic [3:0] bottom_right;

logic [3:0] space_loc;
```

On reset, we can reset the design to whatever we want to be our starting puzzle. Here, I've initialized it to a puzzle I know has a solution, but feel free to choose your own starting point! I've also included a register that will let us know if the design has been reset or not, so that our solver will only consider solutions that have been reset.

```
logic design_was_reset = 1'b0;

always @( posedge clk ) begin
    if( reset ) begin
```

```
        top_left      <= 4'd5;
        top_middle    <= 4'd1;
        top_right     <= 4'd6;
        middle_left   <= 4'd2;
        middle_middle <= 4'd7;
        middle_right  <= 4'd3;
        bottom_left   <= 4'd8;
        bottom_middle <= 4'd4;
        bottom_right  <= 4'd0;

        space_loc     <= 4'b1010;

        design_was_reset <= 1'b1;
    end
end
```

Alright - we've created the framework for our game, so now we have to define the mechanics! First off, we should create a signal to let us know if a move is actually valid based on where the space is (i.e. you can't move it left if it's already in the leftmost column):

```
logic move_right_valid;
logic move_left_valid;
logic move_up_valid;
logic move_down_valid;
logic move_valid;

assign move_right_valid = (direction==RIGHT) && (space_loc[1:0] < 2'd2);
assign move_left_valid  = (direction==LEFT)  && (space_loc[1:0] > 2'd0);
assign move_up_valid    = (direction==UP)    && (space_loc[3:2] > 2'd0);
assign move_down_valid  = (direction==DOWN)  && (space_loc[3:2] < 2'd2);
assign move_valid = (move_right_valid || move_left_valid || move_up_valid || move_down_valid);
```

Next, we will shift the space to the desired location by adding or subtracting from the relevant bits in `space_loc` (Note that here, I represent it as a separate clocked block, but in our code, this should be combined with the reset block, to occur if the reset doesn't)

```
logic [3:0] space_loc_left;
logic [3:0] space_loc_right;
logic [3:0] space_loc_up;
logic [3:0] space_loc_down;

assign space_loc_left  = {space_loc[3:2]    , (space_loc[1:0]-1)};
assign space_loc_right = {space_loc[3:2]    , (space_loc[1:0]+1)};
assign space_loc_up    = {(space_loc[3:2]-1), space_loc[1:0]    };
assign space_loc_down  = {(space_loc[3:2]+1), space_loc[1:0]    };

always @( posedge clk ) begin

    ... <reset logic>

    else if( move_valid ) begin
```

```
        case (direction)
            LEFT   : space_loc <= space_loc_left;
            RIGHT  : space_loc <= space_loc_right;
            UP     : space_loc <= space_loc_up;
            DOWN   : space_loc <= space_loc_down;
            default: space_loc <= space_loc;
        endcase
    end
end
```

Lastly, we need to switch the values help in the registers representing the relevant spaces. Since the code is given to you, I won't explicitly show this here for the sake of brevity. I explicitly showed everything using nested case statements to avoid using higher-up SystemVerilog constructs that SymbiYosys may not be able to handle, but you're more than welcome to try those! This logic can be found on lines 89-263 of `sliding_tile/sliding_tile.v`.

Lastly, we need to have some sense of whether we're done or not. I did this by defining a 9-bit wire called `game_state`, where each bit reflects whether the corresponding space on the board has the correct tile:

```
logic [8:0] game_state; // This tells us how many tiles are currently in their correct spot

assign game_state = {
    (top_left      == 4'b1),
    (top_middle    == 4'b2),
    (top_right     == 4'b3),
    (middle_left   == 4'b4),
    (middle_middle == 4'b5),
    (middle_right  == 4'b6),
    (bottom_left   == 4'b7),
    (bottom_middle == 4'b8),
    (bottom_right  == 4'b0)
};
```

Alright - we're finally done with setting up the mechanics of our sliding tile game! From here, it's relatively simple to set up the machine - we only need two verification statements!

- We want to make sure that the machine starts out with reset high, so that on the first clock edge, we will reset to our starting game position. We can do this with an `assume` statement inside an `initial` statement
- We want to see if we can get to a point where all the bits of `game_state` are 1. We can do this with a `cover` statement

Implementing this, we get:

```
`ifdef FORMAL

logic f_past_valid = 0;

initial assume( reset );
```

```
always @( posedge clk ) begin
    f_past_valid <= 1

    cover(game_state==9'b111111111);
end

`endif // FORMAL
```

Great - our module is complete! (This code is given for you in `$TUTROOT/sliding_tile/sliding_tile.v`) Now, we just need to tell SymbiYosys how to verify it. This is very similar to the register, the main difference being that we are verifying cover statements, so our mode should be `cover` (as well as changing our files and modules that we're referencing)

```
1  [tasks]
2  task1
3
4  [options]
5  task1:
6  mode cover
7  depth 100
8
9  [engines]
10 task1:
11 smtbmc
12
13 [script]
14 read -formal sliding_tile.v
15 prep -top sliding_tile
16
17 [files]
18 sliding_tile.v
```

Now, we *could* run it to verify our design with the commands below:

```
% cd ${TUTROOT}/sliding_tile
% sby -f sliding_tile.sby
```

...***BUT***...

7 Speeding up Verification

Computation time can be *very* long for verification! When creating this tutorial, I ran the puzzle above for ~40 minutes before it finished. This is because what the solver is doing is creating a model of the module for every set of inputs (every possible state), and applying it. Then, if that doesn't satisfy the cover statement, it will apply every possible set of inputs *on each of the created models,* ***creating a new model for each possible next state***, creating many more models. This isn't an issue for a small number of steps or for a small number of inputs (as there aren't that many possible sets of inputs - we saw this with the register). However, as the number of steps required increases or

the number of inputs increases, this can become an issue quickly. If you run the game as it is above, you will notice that each progressive step takes longer than the previous, as the simulator now has to keep track of more models as the number of steps increases.

Alright - we know that computation time is an issue, so how do we fix it? There are a couple options:

7.a Choose a better engine

Not all engines were created equally! Some will naturally be better at certain tasks than others. SymbiYosys can help us with this; if you run it with the `--autotune`, it will find the optimal engine/solver configuration (including flags to pass) for your task by running them all in parallel. Doing this for our design, I found that `btor btormc` was the optimal engine/solver configuration, so we can modify our `.sby` file to use it instead. I would encourage you to look here to explore the options for engines - all of them should be installed, so you're more than welcome to play around and experiment with them!

7.b Restrict number of inputs:

The number of possible states grows factorially with the number of inputs, so if we can limit the number of inputs, we will limit the number of states we need to check. Often times, this isn't possible for overall designs, but it clues us into the fact that our testing should include *unit testing*. It doesn't make sense to test the entire chip when you're just worried about the functionality of a multiplier. Verifying your design in smaller elements (and then perhaps using conventional testing to test the interconnects) can be a good way to speed up the process. (This isn't, however, to say that our software is bad at handling large numbers of states, just that at some point, anything can be overwhelmed. The puzzle above was solved at step 26 - given the number of inputs, how many models existed at that time?)

7.c Restrict the number of valid states:

Often times, we know some states won't lead to the desired outcome (especially when using cover statements). While resetting a design over and over again is a valid thing to do (and the solver will try it, as it doesn't know the semantics of the reset), it is unlikely to achieve your goal, but will only waste computation time and power.

This is where the `assume` statements come in! While we use `cover` and `assert` to define the goals of our verification, we can also use `assume` statements to limit the number of states that the solver will consider. If you've heard of neural network pruning, this is the same concept; if we know that a state won't be part of the solution, we can just eliminate it.

Let's revisit our sliding tile puzzle solver with this in mind. Here are some facts about our best solution that, while they may seem obvious to you and me, the solver wouldn't explicitly know about:

- The design should begin in reset (we might do this with an `initial` statement)

- Once it's been reset, the design shouldn't be reset again
- We should only move the space in a valid direction (previously, we simply didn't do anything if this was an input, but we can also directly tell the solver to not consider this as an option)
- We shouldn't move the space in the direction it just came from

Knowing these (and anything else you can come up with!), see if you can revise the verification logic in the `sliding_tile` module to speed up our solver. Once you're done, you can check my solution in `$TUTROOT/sliding_tile_fast`.

Now we have a verification solver that can solve our sliding tile puzzle in a much more reasonable amount of time! While this previously took ∼40 minutes, with the proper configurations, our new model (for me) was able to solve it in **4 seconds**

```
% cd ${TUTROOT}/sliding_tile_fast
% sby -f sliding_tile_fast.sby
```

Lastly, this tells us that a solution exists, but we can look at the waveform to determine the exact solution it found (note again that this will only work in a graphical viewer):

```
% cd ${TUTROOT}/sliding_tile_fast/sliding_tile_fast_task1/engine_0
% gtkwave trace.vcd
```

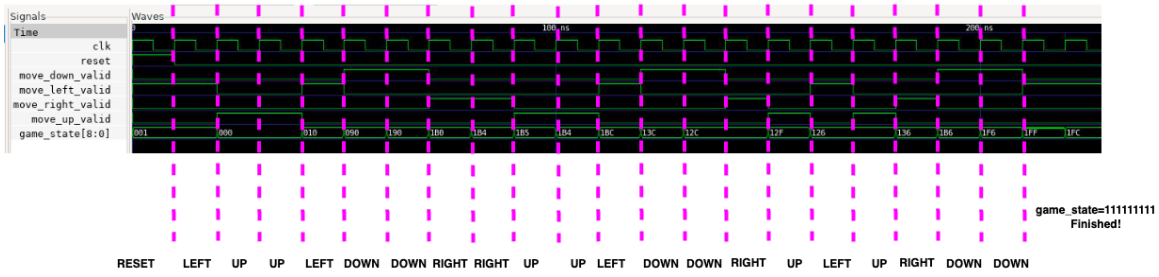

Figure 2: Sliding Tile solution from waveform

8 Your own example!

Now that you've seen how one can use formal verification solvers to solve puzzles, try it yourself! Create a logic puzzle framework within Verilog, and see if you can use formal verification to solve it. Some suggestions might be:

- The wolf-goat-cabbage problem
- A logic-grid problem
- See below:

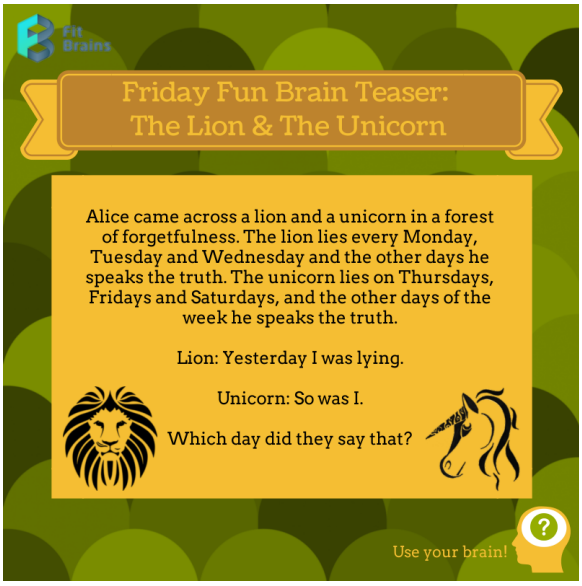

Figure 3: Sample puzzle

- Whatever you can think of!!

9 Conclusion

Hopefully by now you understand and are comfortable using SymbiYosys and the associated commands. If you ever want more practice, feel free to reach out for ideas, or practice on any Verilog design you have! Other than that, this wraps it up for this tutorial - hope you enjoyed!