Coverage Closure and Bug Hunt Project

CSEE E6863 Formal Verification

Instructor: Michael Theobald
Haoran Zhang, Jun Sha
Electrical Engineering
Columbia University
Email: hz2619@columbia.edu, js5506@columbia.edu

CONTENTS

FIFO

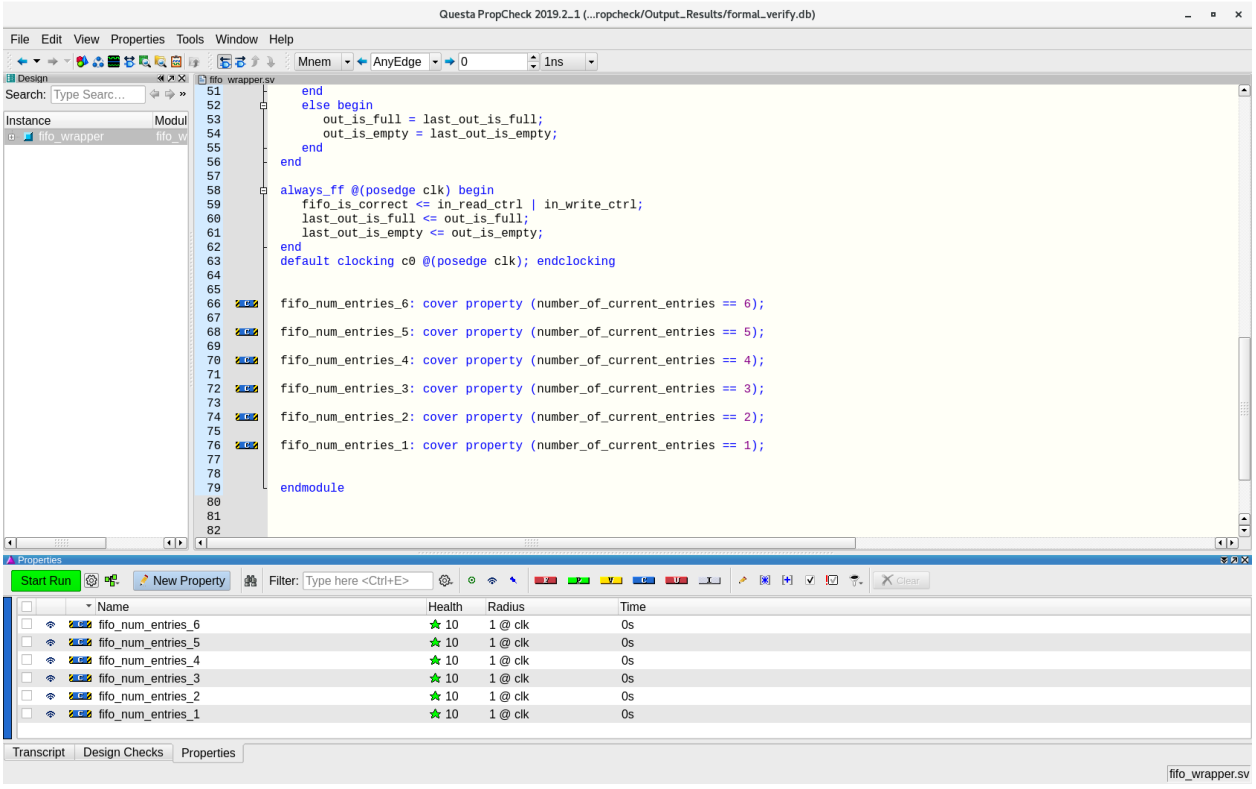
We further analyze the reason why the last if-clause should be removed to make the design work correctly. We can know that the last if-clause set both out_is_empty and out_is_full to zero when in_read_ctrl and in_write_ctrl are zero. The correct if-clause should keep **out_is_empty and out_is_full unchanged.** And in SystemVerilog, we do not need to do any assignments if want to keep these values unchanged. Hence the best way to directly debug fifo.sv is to remove these assignment lines.

However, in this scenario, we cannot do any change to fifo.sv. To make the design work, we come up with **two strategies**,which work on fifo_wrapper.sv

Strategy#1: Modify fifo_wrapper.sv to make sure **out_is_empty and out_is_full are not set to zero simultaneously** even if in_read_ctrl and in_write_ctrl are zero.

Strategy#2: Modify fifo_wrapper.sv to put extra constraints (add two assumptions) on in_read_ctrl and in_write_ctrl , making at least one of them be set 1 to avoid going to the last if-clause in fifo.sv.

a.First, test the cover property of fifo_wrapper with fifo_inst calling fifo.sv



From the picture above, we can find out that the fifo_wrapper still fails to solve the bug in the fifo.sv. The CTO's claim does not work. The CTO's fifo_wrapper cannot solve the overflow bug in fifo.sv.

Strategy#1:

Take fifo_num_entries_5 as an example:

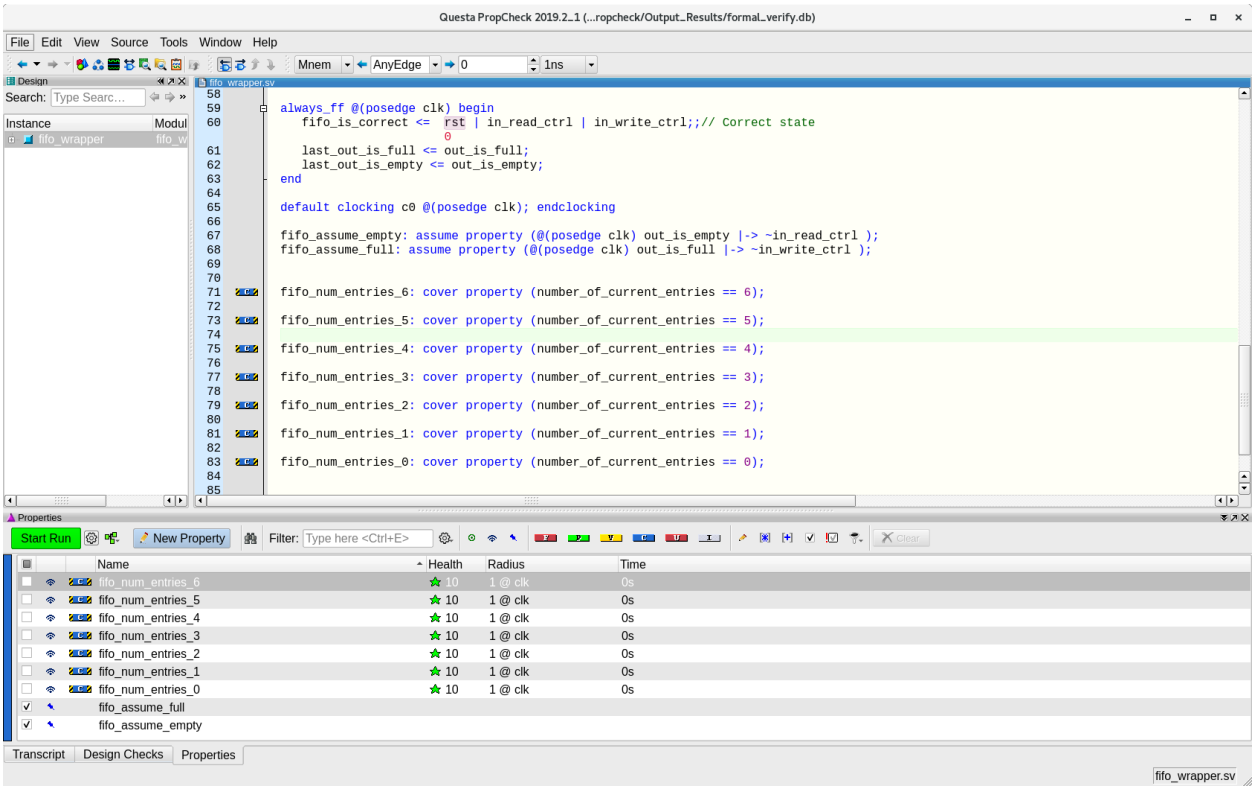
```
always_comb  begin
   if (fifo_is_correct) begin
       0
      out_is_full = out_is_full_tmp;
      0             0
      out_is_empty = out_is_empty_tmp;
      0             1->0
   end
   else begin
      out_is_full = last_out_is_full;
      0             0
      out_is_empty = last_out_is_empty;
      0             0
   end
end

always_ff @(posedge clk) begin
                    0->1
   fifo_is_correct <=  in_read_ctrl | in_write_ctrl;;// Correct state
   0                   0             0
   last_out_is_full <= out_is_full;
   0                   0
   last_out_is_empty <= out_is_empty;
   0                    0
end
```

Take the scenarios when entry_number is 5 as an example, from the picture above we cannot make sure if it is the fifo.sv that set out_is_full and out_is_empty zero or they are just kept unchanged.
So, we modify the fifo_wrapper and try to fix the bug. The modified code is as follows:

```
always_comb  begin
   if (fifo_is_correct) begin
      out_is_full = out_is_full_tmp;
      out_is_empty = out_is_empty_tmp;
   end
   else begin
      out_is_full = last_out_is_full;
      out_is_empty = last_out_is_empty;
   end
end

always_ff @(posedge clk) begin
   fifo_is_correct <= rst | in_write_ctrl | in_read_ctrl;// Correct state
   last_out_is_full <= out_is_full;
   last_out_is_empty <= out_is_empty;
end
```

We change the assignment line of fifo_is_correct. Take the scenarios when entry_number is 5 as an example.

```
always_comb  begin
   if (fifo_is_correct) begin
       1->0
      out_is_full = out_is_full_tmp;
      0              0
      out_is_empty = out_is_empty_tmp;
      1               1->0
   end
   else begin
      out_is_full = last_out_is_full;
      0              0
      out_is_empty = last_out_is_empty;
      1               0->1
   end
end

always_ff @(posedge clk) begin
                     0->1
   fifo_is_correct <=  rst | in_read_ctrl | in_write_ctrl;;// Correct state
   1->0                0     0              0
   last_out_is_full <= out_is_full;
   0                   0
   last_out_is_empty <= out_is_empty;
   0->1                 1
end
```



We can see that even if in the fifo_wrapper part we managed to make sure out_is_empty and out_is_full are kept unchanged, we still cannot solve the overflow bug.

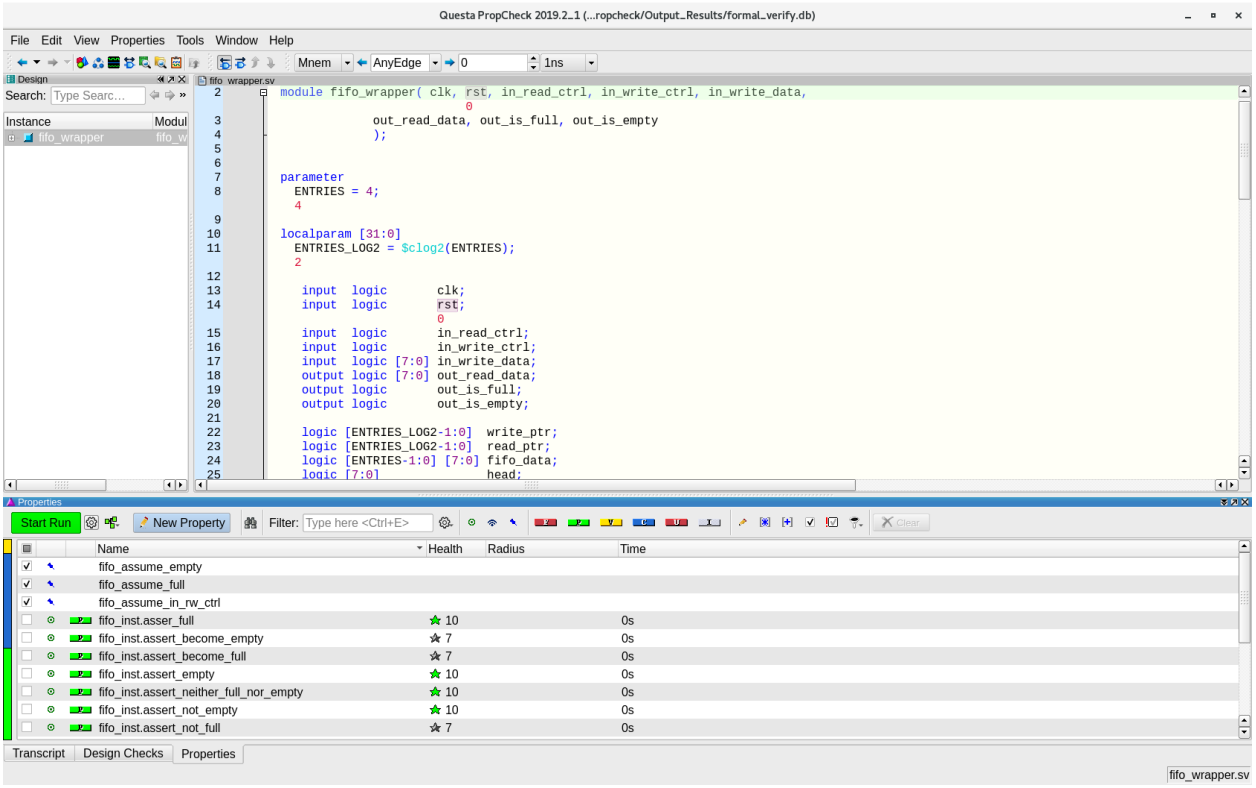
So, our strategy #1 just like what the CTO's code want to accomplish does not work in this case.

Strategy#2:

We add one assumption to constrain the in_read_ctrl and in_write_ctrl signals, making them never be set to zero simultaneously. Add a new assumption **fifo_assume_in_rw_ctrl**

```
fifo_assume_in_rw_ctrl: assume property (@(posedge clk) !(~in_write_ctrl & ~in_read_ctrl));
```

Even if we put this constraint on in_read_ctrl and in_write_ctrl signals by adding a new assumption. The result still shows that we cannot solve the overflow bug.



Strategy#2's cannot be solved easily by simply adding assumptions.

In conclusion, if we want to fix the overflow bug in fifo.sv without modifying fifo.sv , we cannot just add new assumptions or add a new flag signal like fifo_is_correct. To make sure the fifo.sv will not reach the last if-clause, we need to put constraints on in_read_ctrl and in_write_ctrl when we write the wrapper code. The main purpose of the constraints is to make sure that both in_read_ctrl and in_write_ctrl cannot be set to zero at the sametime.

c.Now we need to do a formal verification on this 4-deep FIFO. In our comprehensive verification approach, we work on proposing several assertions so that we can find what has led to FIFO overflow/underflow more systematically.
First, when FIFO is reset, only empty flag is set high; other signals all need to be set to 0;

```
// 1. When FIFO is reset, empty flag is set high; full flag, write_ptr, read_ptr and number_of_current_entries are all set to 0.
assert_reset: assert property (@(posedge clk) rst |=> (write_ptr == 0 && read_ptr == 0 && out_is_full == 0 && out_is_empty == 1 && number_of_current_entries == 0));
```

Second, when FIFO has no entry, it is empty.;

```
// 2. When FIFO has no entry, it is empty.
assert_empty: assert property (@(posedge clk) (number_of_current_entries == 0) |-> out_is_empty);
```

Third, when FIFO is empty, it should not be full;

```
// 3. When FIFO is empty, it should not be full (full flag set to 0).
assert_not_full: assert property (@(posedge clk) out_is_empty |-> !out_is_full);
```

Fourth, when FIFO has only one entry with just one read and no write, it will become empty in the next clock cycle;

```
// 4. When FIFO has only one entry with one read and no write, it will become empty in the next clock cycle.
assert_become_empty: assert property (@(posedge clk) ((number_of_current_entries == 1) && (in_read_ctrl == 1) && (in_write_ctrl == 0)) |=> out_is_empty );
```

Fifth, when FIFO is neither full nor empty, neither full flag nor empty flag should be set high;

```
// 5. When FIFO is neither full nor empty, neither full flag nor empty flag should be set high.
assert_neither_full_nor_empty: assert property (@(posedge clk) (((number_of_current_entries > 0) && (number_of_current_entries < ENTRIES))|-> (!out_is_full && !out_is_empty)));
```

Sixth, when FIFO has N-1 entries with just one write and no read, it will become full in the next clock cycle;

```
// 6. When FIFO has N-1 entries with one write and no read, it will become full in the next clock cycle.
assert_become_full: assert property (@(posedge clk) ((number_of_current_entries == (ENTRIES - 1'b1)) && (in_read_ctrl == 0) && (in_write_ctrl == 1)) |=> out_is_full);
```

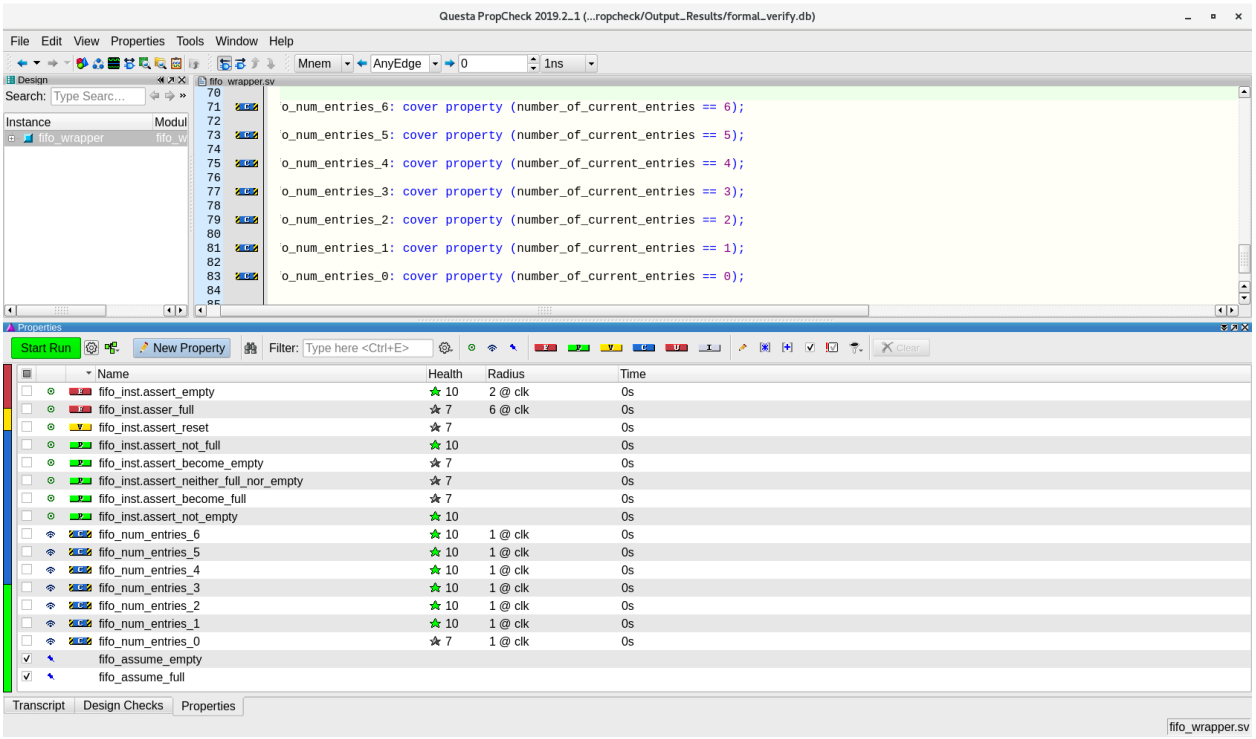
Seventh, when the number of entries has reached or exceeded the FIFO maximum entries N, it is full;

```
// 7. When the number of current entries has reached or exceeded the FIFO maximum entries N, it is full.
asser_full: assert property (@(posedge clk) (number_of_current_entries >= ENTRIES) |-> out_is_full);
```

Eighth, when FIFO is full, it should not be empty (empty flag set to 0).

```
// 8. When FIFO is full, it should not be empty (empty flag set to 0).
assert_not_empty: assert property (@(posedge clk) out_is_full |-> !out_is_empty );
```

The result is as follows:



As we can see from the picture above all comprehensive verification assertions are pass except **assertion_empty** and **assertion_full.** These two assertions are fired because in the fifo.sv the last if-clause set both out_is_full and out_is_empty to zero. If the design keep both out_is_full and out_is_empty unchanged or solved the overflow problem by redesigning the fifo_wrapper then this design can work.

Blackbox & Abstraction

a. Blackbox

i. A blackbox is a portion of a model that we choose to ignore for verification purposes when the increasing design complexity causes inconclusive analysis to happen. This technique is actually an under-constraint, which will make proofs more general. We are over-approximating the model by not considering the details in some part of the logic. So, if we can prove an assertion in a model with blackboxes, it is going to be true of the full model as well.
ii. If we obtained a counterexample, it may have two different explanations: one is that we really have a true bug in the design, the other is that we may need to add additional assumptions for the blackboxes to modify some incorrect results.
For example, for blocks with large embedded queues or memories, we need to add new assumptions when blackboxing the submodules, so it does not appear to produce incorrect results for the entire analysis.

b. Abstraction

When using abstraction, we just reduce the size of logical structures. It not only reduces the size of the logic, and hence the state space, that the tool has to search. By contrast, blackboxes is also a useful complexity reduction technique, but it reduces the complexity by ignoring submodules irrelevant to formal verification. We once used structural abstraction in our project, but never used blackboxes, because a good candidate for blackboxing is hard to find. For FIFO verification, when inconclusive analysis happened, we tried to decrease the counter size from 32 bits to 8 bits, in order for the tool to find the overflow issue as well as the bugs more easily.

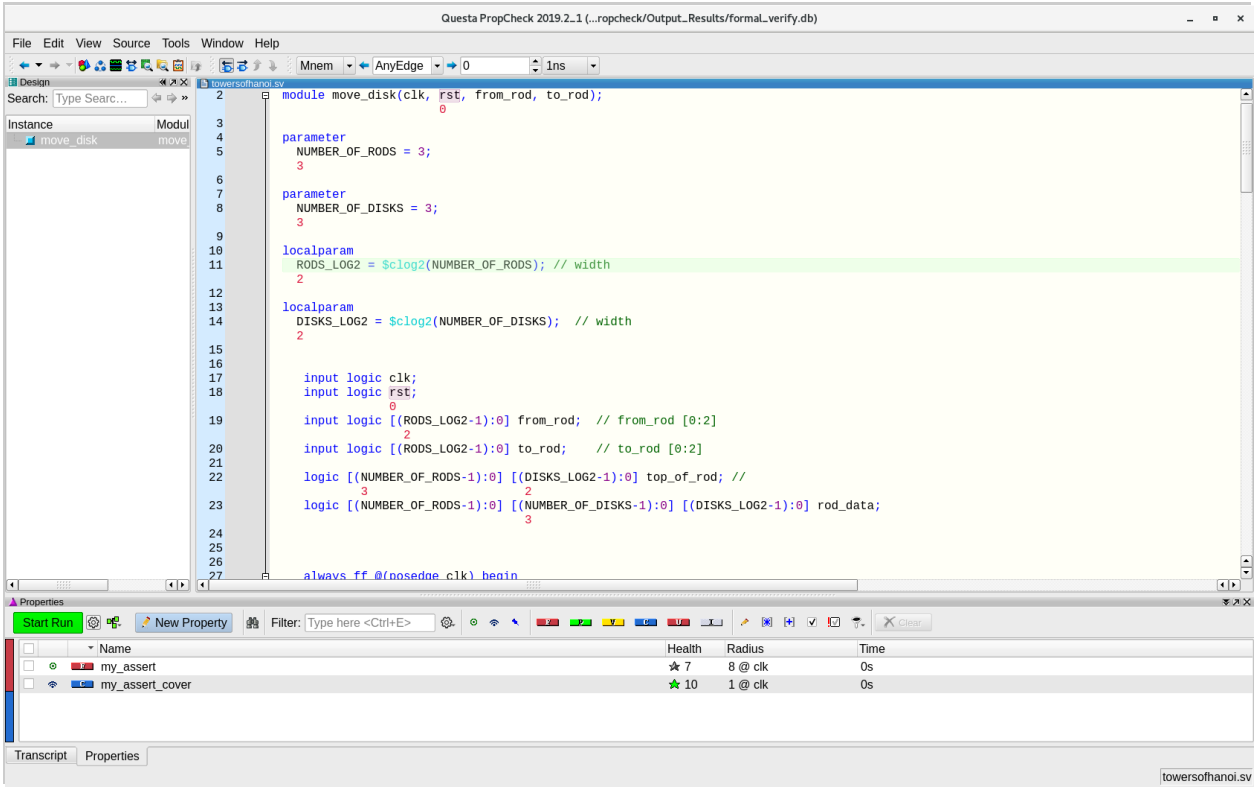
Towers of Hanoi

a. Assertion:

We call the three rods: rod[0], rod[1], rod[2], call the three disks: disk[1], disk[2], disk[3], and disk[3] is the biggest one, disk[1] is the smallest one. Divide the absolute height position into three levels: Top Level[2], Middle Level[1], Bottom Level[0].
The assertion **my_assert** means in rod [1], disk[1] cannot be put in the top level.

b. Re-write assertion as cover property

```
my_assert: assert property (@(posedge clk) rod_data[1][NUMBER_OF_DISKS-1] != 1);

my_assert_cover: cover property (@(posedge clk) rod_data[1][NUMBER_OF_DISKS-1] != 1);
```

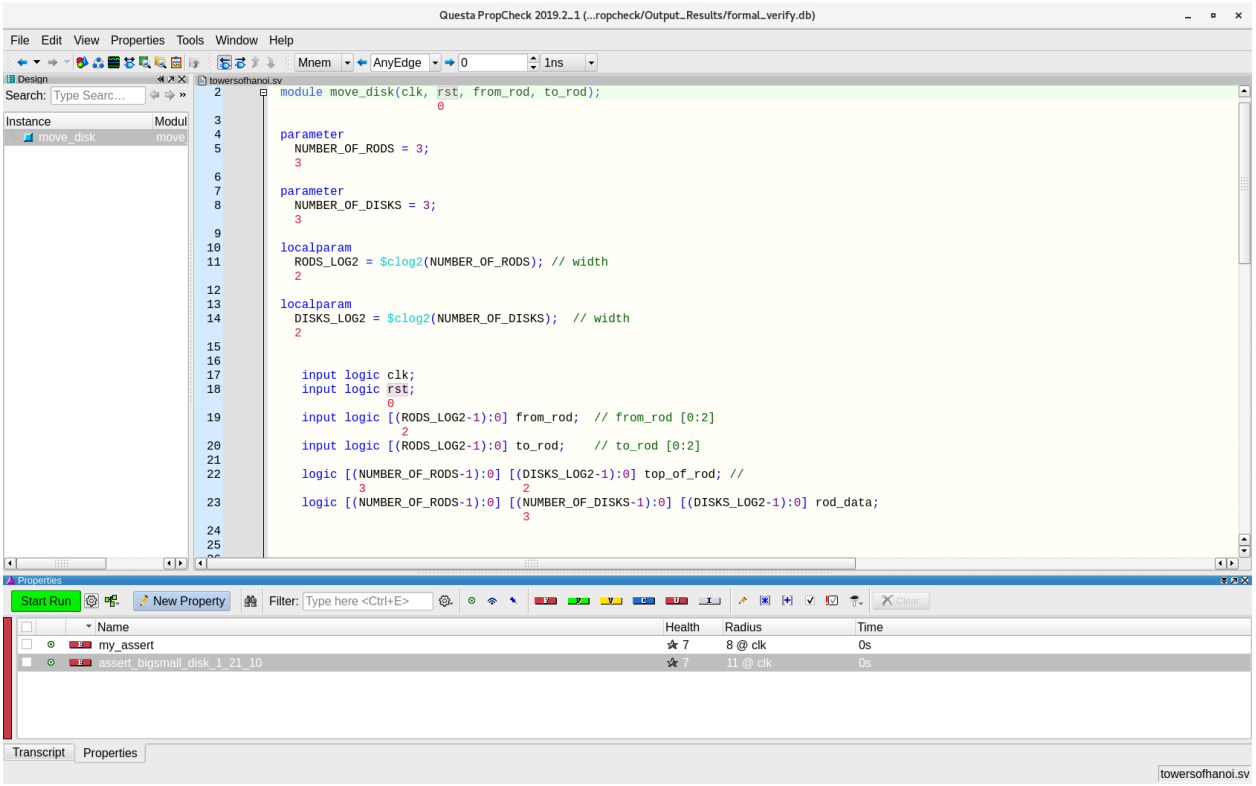


The difference between assertion and cover property is, assertion is a property that holds true all the time, and cover property is a property that ever holds.

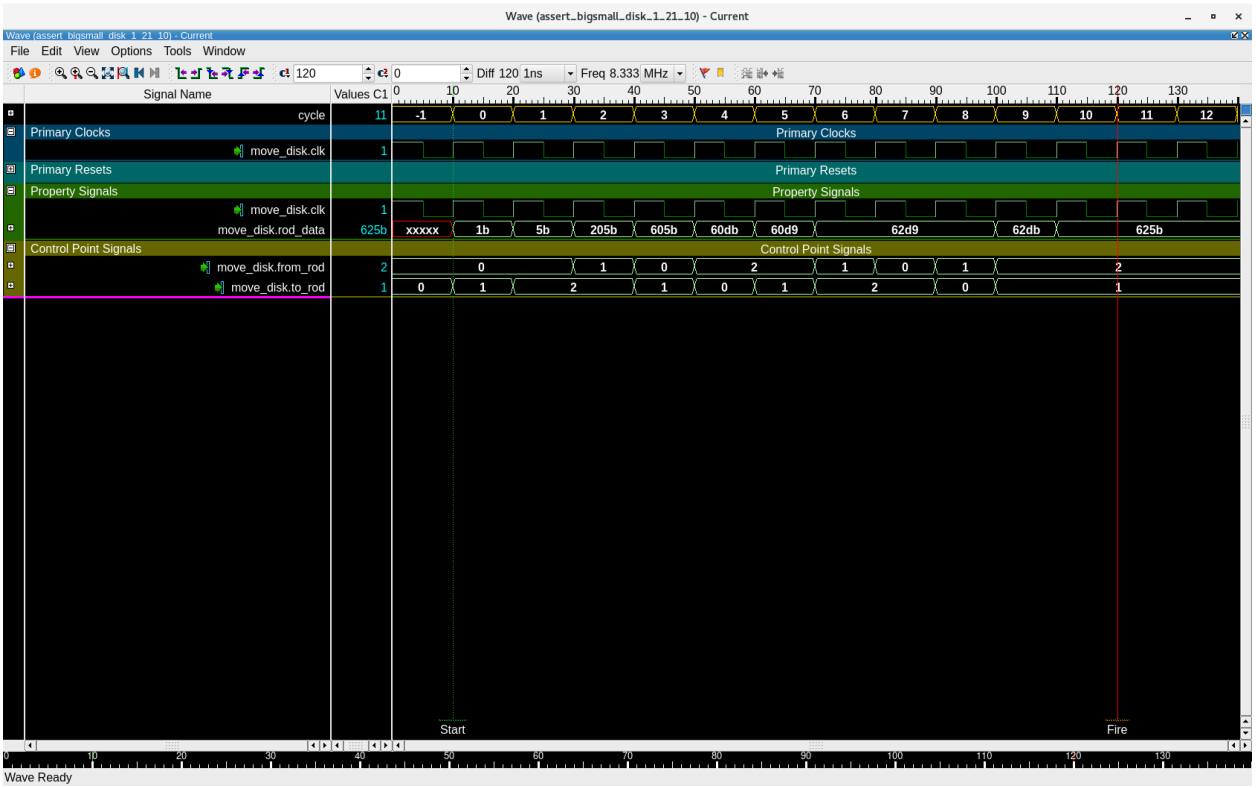
As we can see from the picture above, the assertion my_assert is fired but the cover property my_assert_cover is covered, which means that the assertion my_assert does not hold true all the time but it holds true at least once. (in the following question, we will discuss that combine assertion and cover property together, it can verify that this code can at least once satisfy this hanoi game's winning requirement.

Debug Process:

We write one assertion {assert_bigsmall_disk_1_21_10}, which means in rod [1], you can never put disk[2] and disk[1] in Middle Level[1] and Bottom Level[0].



We find that this property is fired. To find the possible reasons why it is fired, we work on its waveform.

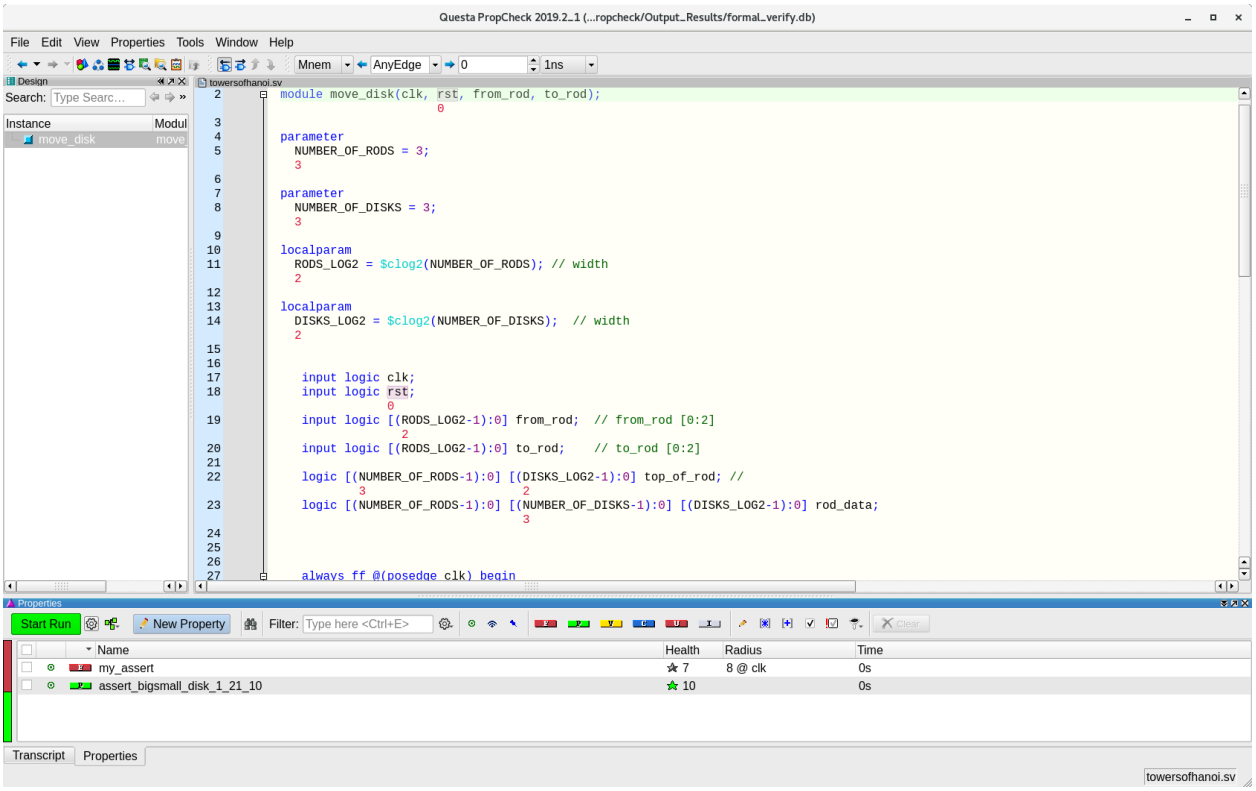


We find that the rod_data can reach 625b, which is not supposed to be like this, because in the code it was limited smaller or equal to 3.

According to the waveform, we speculate that the potential bug may be in the main body of the code. And the bug has something to do with **rod_data.** We then add the following codes to reset the value rod_data[from_rod][top_of_rod] after each time one disk is moved.

```
rod_data[from_rod][top_of_rod[from_rod]-1] <= 0; // reset the rod_data of from_rod
```

After we add this line of code, the result turns out to be **Proof** as expectation.



c. Other cover properties and assertions

From the analysis in question a and b, we believe that we need to add cover properties and assertions to verify that the code can actually satisfy the hanoi game's winning requirement.

The winning requirement is to move the entire stack to another rod obeying two rules. When we move the entire stack to another rod(rod[1] or rod[2]), the top disk of the target rod (rod[1] or rod[2]) is the smallest disk (disk[1]). Just like what we do in question a and b. We can combine assertions and cover properties together to prove that it can satisfy the winning requirements at least once.

We need to add assertions:

The event that top disk of rod[1] is disk[1] is always false.

The event that top disk of rod[2] is disk[1] is always false.

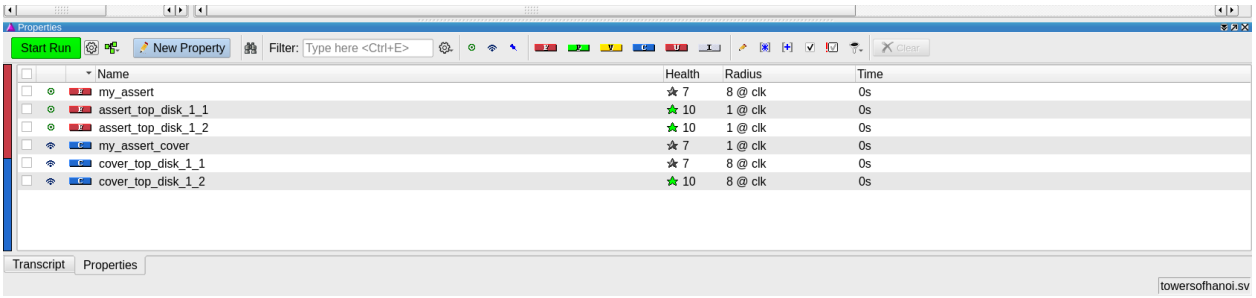
and cover property:

The event that top disk of rod[1] is disk[1] eventually happens.

The event that top disk of rod[2] is disk[1] eventually happens.

```
//////////////////////////            winning goal         //////////////////////////
assert_top_disk_1_1 :assert property (@(posedge clk) rod_data[1][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[1] is disk[1] is always false.
assert_top_disk_1_2 :assert property (@(posedge clk) rod_data[2][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[2] is disk[1] is always false.
cover_top_disk_1_1 : cover property (@(posedge clk) rod_data[1][NUMBER_OF_DISKS-1] == 1); // The event that top disk of rod[1] is disk[1] eventually happens.
cover_top_disk_1_2 : cover property (@(posedge clk) rod_data[2][NUMBER_OF_DISKS-1] == 1); // The event that top disk of rod[2] is disk[1] eventually happens.
```

And we can see the result in the following screenshot that the assertions assert_top_disk_1_1 and assert_top_disk_1_2 are fired, but cover properties are covered, which can prove that it can eventually satisfy the winning requirements (at least once).



d. Develop a formal verification strategy to verify the RTL.

To do formal verification on the RTL, we divide potential properties into three parts: Basic Functions, Game Rules and Winning Goal

Basic Functions: in this part, the properties are mainly involved with basic functions of the code like if we set 'rst' to 1, whether all the values are reset as we expected.

Game Rules

Rule #1: at any time only one disk at the top of one of the rods may be moved to the top of another rod.

Rule #2: never place a disk on top of a smaller one

Winning Goal

As for whether we should first verify "Game Rules" properties or first verify "Winning Goal" properties, we thought that it is easier to write "Winning Goal" properties. (like what we have done in question c)

But we changed our minds very soon because it is meaningless to get a right answer in a wrong way, which means if we do not first verify that it is played exactly following the rules, it is useless even if we get the right answer(result).

The assertions and cover properties of **Game Rules:**

The big disk can never be put on the top of the small disk. If we want to do a formal verification on this RTL design, we must cover all state space. In this case, the number of rods is equal to 3 and the number of disks is equal to 3. We adopt a strategy to write assertions that verify the disk size one by one. In this strategy, we want to verify, for example,

<u>The event that above disk is disk[2] and below disk is disk[1] will never happen in rod [0].</u>

```
//////////////////////////// bigsmall rod [0] disk[2] disk[1] ////////////////////////////

assert_bigsmall_disk_0_21_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-3] == 1) && (rod_data[0][NUMBER_OF_DISKS-2] == 2)));
```

Following this pattern, if we want to do a formal verification of this sort of properties, we need to cover all state space, which means we need to cover all possibilities:

$$C_3^2 * 3 * 2 \ = 18$$

C_3^2 refers to select 2 disks out of 3 disks.

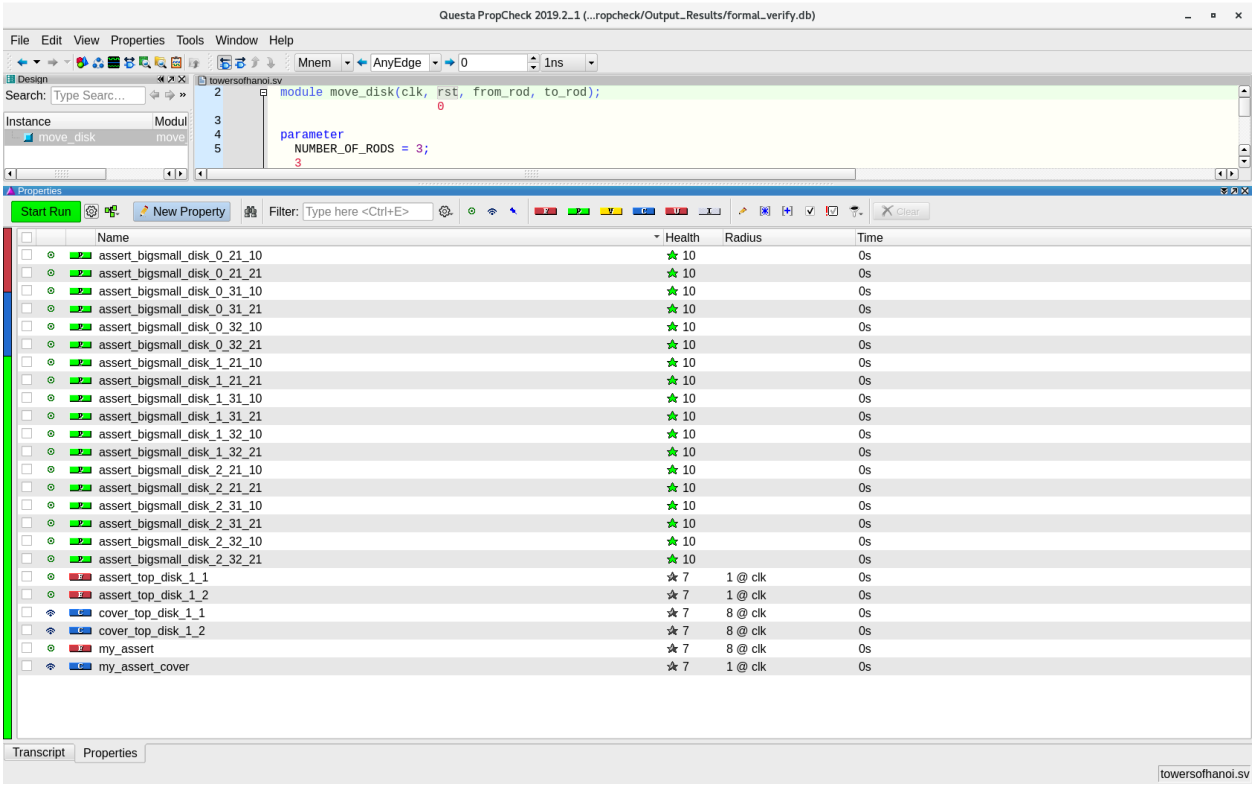
3 refers 3 rods.

2 refers to two possible height relations.

The total code is as follows

```
//////////////////////////// bigsmall rod [0] disk[2] disk[1] ////////////////////////////

assert_bigsmall_disk_0_21_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-3] == 1) && (rod_data[0][NUMBER_OF_DISKS-2] == 2)));

assert_bigsmall_disk_0_21_21 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-2] == 1) && (rod_data[0][NUMBER_OF_DISKS-1] == 2)));

//////////////////////////// bigsmall rod [1] disk[2] disk[1] ////////////////////////////

assert_bigsmall_disk_1_21_10 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-3] == 1) && (rod_data[1][NUMBER_OF_DISKS-2] == 2)));

assert_bigsmall_disk_1_21_21 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-2] == 1) && (rod_data[1][NUMBER_OF_DISKS-1] == 2)));

//////////////////////////// bigsmall rod [2] disk[2] disk[1] ////////////////////////////

assert_bigsmall_disk_2_21_10 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-3] == 1) && (rod_data[2][NUMBER_OF_DISKS-2] == 2)));

assert_bigsmall_disk_2_21_21 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-2] == 1) && (rod_data[2][NUMBER_OF_DISKS-1] == 2)));

//////////////////////////// bigsmall rod [0] disk[3] disk[2] ////////////////////////////

assert_bigsmall_disk_0_32_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-3] == 2) && (rod_data[0][NUMBER_OF_DISKS-2] == 3)));

assert_bigsmall_disk_0_32_21 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-2] == 2) && (rod_data[0][NUMBER_OF_DISKS-1] == 3)));

//////////////////////////// bigsmall rod [1] disk[3] disk[2] ////////////////////////////

assert_bigsmall_disk_1_32_10 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-3] == 2) && (rod_data[1][NUMBER_OF_DISKS-2] == 3)));

assert_bigsmall_disk_1_32_21 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-2] == 2) && (rod_data[1][NUMBER_OF_DISKS-1] == 3)));

//////////////////////////// bigsmall rod [2] disk[3] disk[2] ////////////////////////////

assert_bigsmall_disk_2_32_10 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-3] == 2) && (rod_data[2][NUMBER_OF_DISKS-2] == 3)));

assert_bigsmall_disk_2_32_21 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-2] == 2) && (rod_data[2][NUMBER_OF_DISKS-1] == 3)));

//////////////////////////// bigsmall rod [0] disk[3] disk[1] ////////////////////////////

assert_bigsmall_disk_0_31_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-3] == 1) && (rod_data[0][NUMBER_OF_DISKS-2] == 3)));

assert_bigsmall_disk_0_31_21 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-2] == 1) && (rod_data[0][NUMBER_OF_DISKS-1] == 3)));

//////////////////////////// bigsmall rod [1] disk[3] disk[1] ////////////////////////////

assert_bigsmall_disk_1_31_10 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-3] == 1) && (rod_data[1][NUMBER_OF_DISKS-2] == 3)));

assert_bigsmall_disk_1_31_21 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-2] == 1) && (rod_data[1][NUMBER_OF_DISKS-1] == 3)));

//////////////////////////// bigsmall rod [2] disk[3] disk[1] ////////////////////////////

assert_bigsmall_disk_2_31_10 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-3] == 1) && (rod_data[2][NUMBER_OF_DISKS-2] == 3)));

assert_bigsmall_disk_2_31_21 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-2] == 1) && (rod_data[2][NUMBER_OF_DISKS-1] == 3)));
```

We can learn from the following screenshot that all the “bigsmall” properties are passed, which means that this RTL plays the game exactly following Rule #2.



The assertions and cover properties of **Winning Goal**

Just like what we do in question b.

The winning goal is to move the entire stack to another rod obeying two rules. When we move the entire stack to another rod(rod[1] or rod[2]), the top disk of the target rod (rod[1] or rod[2]) is the smallest disk (disk[1]). Just like what we do in question a and b. We can combine assertions and cover properties together to prove that it can satisfy the winning requirements at least once.

We need to add assertions:

The event that top disk of rod[1] is disk[1] is always false.

The event that top disk of rod[2] is disk[1] is always false.

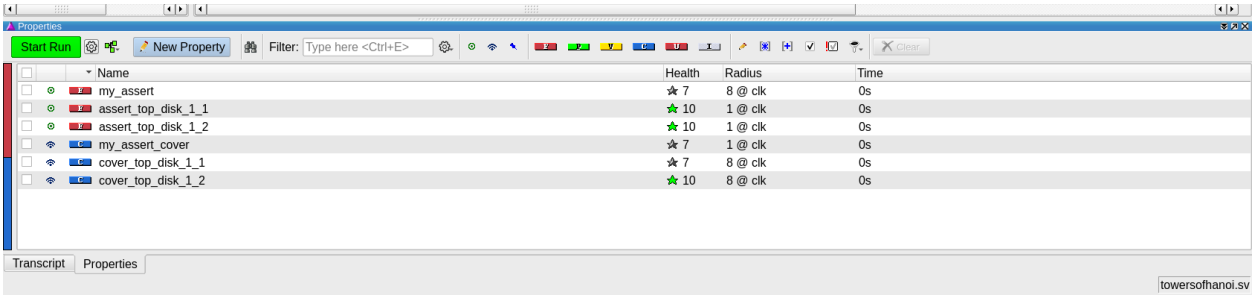
and cover property:

The event that top disk of rod[1] is disk[1] eventually happens.

The event that top disk of rod[2] is disk[1] eventually happens.

```
//////////////////////////            winning goal          //////////////////////////
assert_top_disk_1_1 :assert property (@(posedge clk) rod_data[1][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[1] is disk[1] is always false.
assert_top_disk_1_2 :assert property (@(posedge clk) rod_data[2][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[2] is disk[1] is always false.
cover_top_disk_1_1 : cover property (@(posedge clk) rod_data[1][NUMBER_OF_DISKS-1] == 1); // The event that top disk of rod[1] is disk[1] eventually happens.
cover_top_disk_1_2 : cover property (@(posedge clk) rod_data[2][NUMBER_OF_DISKS-1] == 1); // The event that top disk of rod[2] is disk[1] eventually happens.
```

And we can see the result in the following screenshot that the assertions assert_top_disk_1_1 and assert_top_disk_1_2 are fired, but cover properties are covered, which can prove that it can satisfy the winning requirements at least once.



Challenges in 3_3_hanoi game:

Challenge #1: find the bug (waveform rod_data[ ][ ] overflow - 625b)

Challenge #2: write what property? In assertions or cover properties form?

Challenge #3: same rule description may have different code forms

Challenge #4: write the adjacent property

Challenge #5: find the width bug and fix it.

Above are challenges that are **conquered**,

Following are challenges that are **not solved**

Challenge #6: basic function of hanoi RTL design, we cannot define a property that can show the if-else case when rst is set to be true.

Challenge #7: rules part of hanoi game, we cannot define a property that can show each time only one disk is moved.

e. Adjacent Rules

Can you think of an assertion and/or assumption that help find out if this puzzle can be solved?

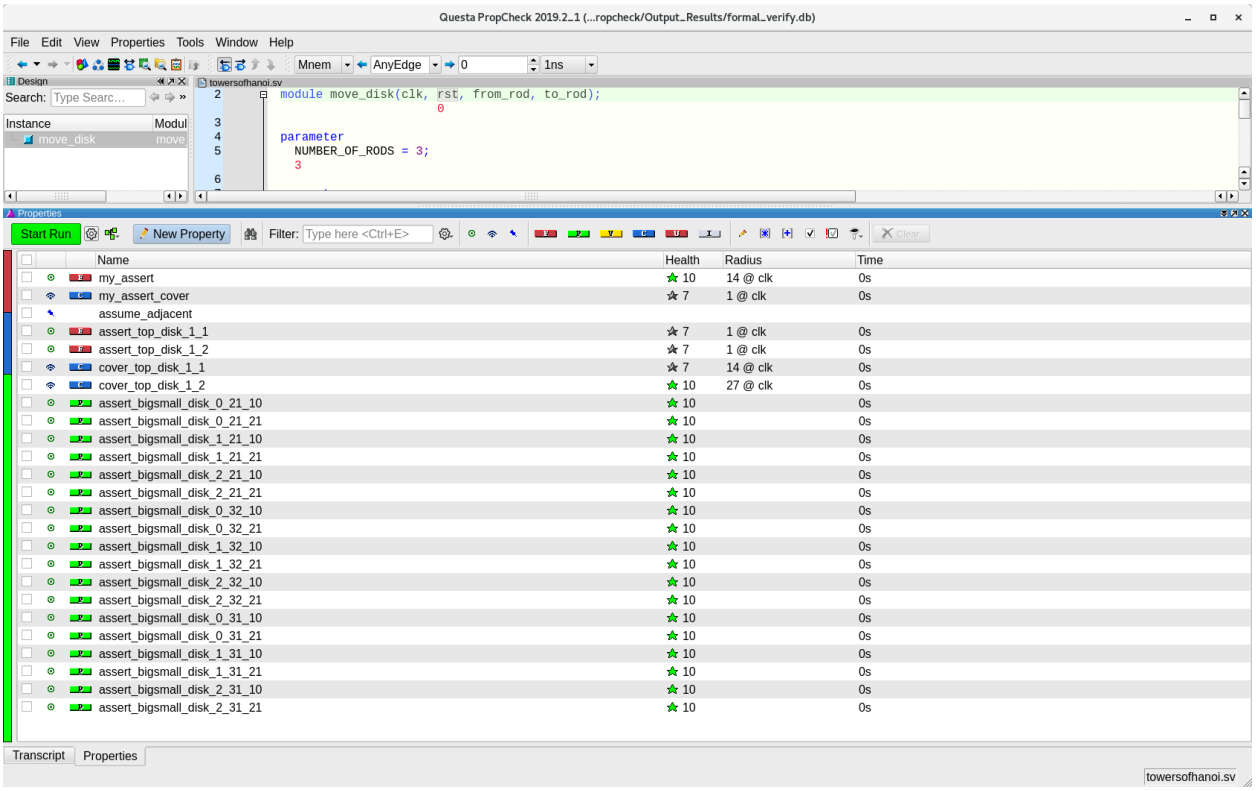
The question is to ask us when we disallow moves between rod[0] and rod[2], if this puzzle can be solved? We only need to make sure that from_rod == 0 and to_rod == 2 or from_rod ==2 and to_rod==0 won't happen at the same time.

In this scenario, we need to add an assumption that disallow from_rod == 0 and to_rod == 2 or from_rod ==2 and to_rod==0 happen at the same time.

We add an assumption **assume_adjacent**

```
//////////////////////////        adjacent                //////////////////////////

assume_adjacent : assume property (@(posedge clk)  !((from_rod == 0 && to_rod == 2) || (from_rod == 2 && to_rod == 0)));
```

After we add this assumption, we can find the result from the following screenshot, the rules and winning goal can be satisfied, which means that even if we put this constraint, this RTL can manage to solve this puzzle.



f. NUMBER_OF_DISKS = 4 and NUMBER_OF_RODS =3, redo d

To do formal verification on this RTL, we divide potential properties into three parts: Basic Functions, Game Rules and Winning Goal

Basic Functions: in this part, the properties are mainly involved with basic functions of the code like if we set rst to 1, whether all the values are reset as we expected.

Game Rules

Rule #1: at any time only one disk at the top of one of the rods may be moved to the top of another rod.

Rule #2: never place a disk on top of a smaller one

Winning Goal

As for whether we should first verify "Game Rules" properties or first verify "Winning Goal" properties, we thought that maybe it is easier to write "Winning Goal" properties. (like what we have done in question c)

But we changed our minds very soon because it is meaningless to get a right answer in a wrong way, which means if we do not first verify it is played exactly following the rules, it is useless even if we get the right answer(result).

The assertions and cover properties of **Game Rules:**

Big disk can never be put on the top of the small disk. If we want to do a formal verification on this RTL design, we must cover all state space. In this case, the number of rods is equal to 3 and the number of disks is equal to 4. We adopt a strategy to write assertions that verify the disk size one by one. In this strategy, we want to verify, for example,

<u>The event that above disk is disk[2] and below disk is disk[1] will never happen in rod [0].</u>

```
////////////////////////// bigsmall rod [0] disk[2] disk[1] ////////////////////////////
assert_bigsmall_disk_0_21_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-4] == 1) && (rod_data[0][NUMBER_OF_DISKS-3] == 2)));
```

Following this pattern, if we want to do a formal verification of this sort of properties, we need to cover all state space, which means we need to cover all possibilities:

$$C_4^2 * 3 * 3 = 54$$

C_4^2 refers to select 2 disks out of 4 disks.

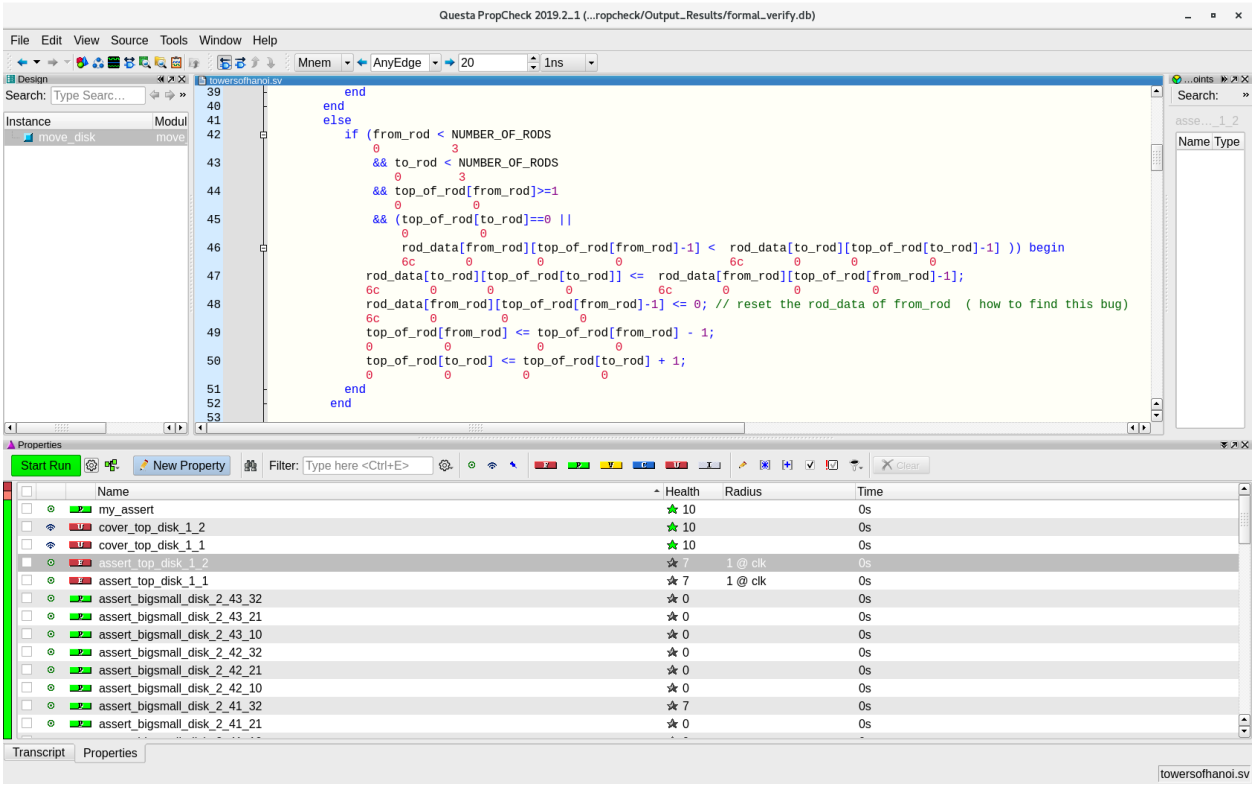
The first 3 refers 3 rods.

The second 3 refers to 3 possible height relations.

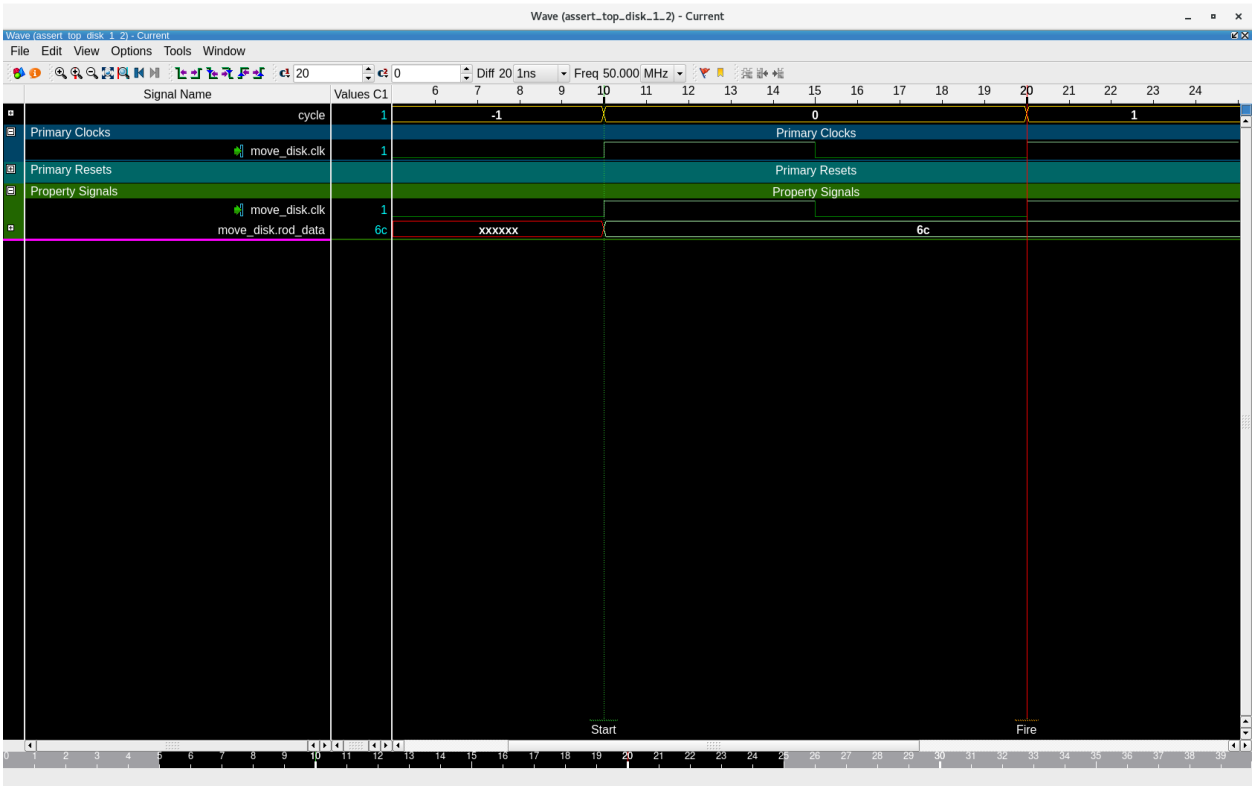
The total code is as follows

```
//////////////////////////// bigsmall rod [1] disk[2] disk[1] ////////////////////////////
assert_bigsmall_disk_1_21_10 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-4] == 1) && (rod_data[1][NUMBER_OF_DISKS-3] == 2)));
assert_bigsmall_disk_1_21_21 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-3] == 1) && (rod_data[1][NUMBER_OF_DISKS-2] == 2)));
assert_bigsmall_disk_1_21_32 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-2] == 1) && (rod_data[1][NUMBER_OF_DISKS-1] == 2)));
//////////////////////////// bigsmall rod [2] disk[2] disk[1] ////////////////////////////
assert_bigsmall_disk_2_21_10 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-4] == 1) && (rod_data[2][NUMBER_OF_DISKS-3] == 2)));
assert_bigsmall_disk_2_21_21 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-3] == 1) && (rod_data[2][NUMBER_OF_DISKS-2] == 2)));
assert_bigsmall_disk_2_21_32 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-2] == 1) && (rod_data[2][NUMBER_OF_DISKS-1] == 2)));
//////////////////////////// bigsmall rod [0] disk[3] disk[2] ////////////////////////////
assert_bigsmall_disk_0_32_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-4] == 2) && (rod_data[0][NUMBER_OF_DISKS-3] == 3)));
assert_bigsmall_disk_0_32_21 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-3] == 2) && (rod_data[0][NUMBER_OF_DISKS-2] == 3)));
assert_bigsmall_disk_0_32_32 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-2] == 2) && (rod_data[0][NUMBER_OF_DISKS-1] == 3)));
//////////////////////////// bigsmall rod [1] disk[3] disk[2] ////////////////////////////
assert_bigsmall_disk_1_32_10 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-4] == 2) && (rod_data[1][NUMBER_OF_DISKS-3] == 3)));
assert_bigsmall_disk_1_32_21 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-3] == 2) && (rod_data[1][NUMBER_OF_DISKS-2] == 3)));
assert_bigsmall_disk_1_32_32 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-2] == 2) && (rod_data[1][NUMBER_OF_DISKS-1] == 3)));
//////////////////////////// bigsmall rod [2] disk[3] disk[2] ////////////////////////////
assert_bigsmall_disk_2_32_10 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-4] == 2) && (rod_data[2][NUMBER_OF_DISKS-3] == 3)));
assert_bigsmall_disk_2_32_21 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-3] == 2) && (rod_data[2][NUMBER_OF_DISKS-2] == 3)));
assert_bigsmall_disk_2_32_32 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-2] == 2) && (rod_data[2][NUMBER_OF_DISKS-1] == 3)));
//////////////////////////// bigsmall rod [0] disk[3] disk[1] ////////////////////////////
assert_bigsmall_disk_0_31_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-4] == 1) && (rod_data[0][NUMBER_OF_DISKS-3] == 3)));
assert_bigsmall_disk_0_31_21 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-3] == 1) && (rod_data[0][NUMBER_OF_DISKS-2] == 3)));
assert_bigsmall_disk_0_31_32 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-2] == 1) && (rod_data[0][NUMBER_OF_DISKS-1] == 3)));
//////////////////////////// bigsmall rod [1] disk[3] disk[1] ////////////////////////////
assert_bigsmall_disk_1_31_10 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-4] == 1) && (rod_data[1][NUMBER_OF_DISKS-3] == 3)));
assert_bigsmall_disk_1_31_21 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-3] == 1) && (rod_data[1][NUMBER_OF_DISKS-2] == 3)));
assert_bigsmall_disk_1_31_32 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-2] == 1) && (rod_data[1][NUMBER_OF_DISKS-1] == 3)));
//////////////////////////// bigsmall rod [2] disk[3] disk[1] ////////////////////////////
assert_bigsmall_disk_2_31_10 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-4] == 1) && (rod_data[2][NUMBER_OF_DISKS-3] == 3)));
assert_bigsmall_disk_2_31_21 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-3] == 1) && (rod_data[2][NUMBER_OF_DISKS-2] == 3)));
assert_bigsmall_disk_2_31_32 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-2] == 1) && (rod_data[2][NUMBER_OF_DISKS-1] == 3)));
//////////////////////////// bigsmall rod [0] disk[4] disk[1] ////////////////////////////
assert_bigsmall_disk_0_41_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-4] == 1) && (rod_data[0][NUMBER_OF_DISKS-3] == 4)));
assert_bigsmall_disk_0_41_21 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-3] == 1) && (rod_data[0][NUMBER_OF_DISKS-2] == 4)));
assert_bigsmall_disk_0_41_32 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-2] == 1) && (rod_data[0][NUMBER_OF_DISKS-1] == 4)));
//////////////////////////// bigsmall rod [1] disk[4] disk[1] ////////////////////////////
assert_bigsmall_disk_1_41_10 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-4] == 1) && (rod_data[1][NUMBER_OF_DISKS-3] == 4)));
assert_bigsmall_disk_1_41_21 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-3] == 1) && (rod_data[1][NUMBER_OF_DISKS-2] == 4)));
assert_bigsmall_disk_1_41_32 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-2] == 1) && (rod_data[1][NUMBER_OF_DISKS-1] == 4)));
//////////////////////////// bigsmall rod [2] disk[4] disk[1] ////////////////////////////
assert_bigsmall_disk_2_41_10 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-4] == 1) && (rod_data[2][NUMBER_OF_DISKS-3] == 4)));
assert_bigsmall_disk_2_41_21 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-3] == 1) && (rod_data[2][NUMBER_OF_DISKS-2] == 4)));
assert_bigsmall_disk_2_41_32 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-2] == 1) && (rod_data[2][NUMBER_OF_DISKS-1] == 4)));
//////////////////////////// bigsmall rod [0] disk[4] disk[2] ////////////////////////////
assert_bigsmall_disk_0_42_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-4] == 2) && (rod_data[0][NUMBER_OF_DISKS-3] == 4)));
assert_bigsmall_disk_0_42_21 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-3] == 2) && (rod_data[0][NUMBER_OF_DISKS-2] == 4)));
assert_bigsmall_disk_0_42_32 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-2] == 2) && (rod_data[0][NUMBER_OF_DISKS-1] == 4)));
//////////////////////////// bigsmall rod [1] disk[4] disk[2] ////////////////////////////
assert_bigsmall_disk_1_42_10 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-4] == 2) && (rod_data[1][NUMBER_OF_DISKS-3] == 4)));
assert_bigsmall_disk_1_42_21 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-3] == 2) && (rod_data[1][NUMBER_OF_DISKS-2] == 4)));
assert_bigsmall_disk_1_42_32 : assert property (@(posedge clk)  not ((rod_data[1][NUMBER_OF_DISKS-2] == 2) && (rod_data[1][NUMBER_OF_DISKS-1] == 4)));
//////////////////////////// bigsmall rod [2] disk[4] disk[2] ////////////////////////////
assert_bigsmall_disk_2_42_10 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-4] == 2) && (rod_data[2][NUMBER_OF_DISKS-3] == 4)));
assert_bigsmall_disk_2_42_21 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-3] == 2) && (rod_data[2][NUMBER_OF_DISKS-2] == 4)));
assert_bigsmall_disk_2_42_32 : assert property (@(posedge clk)  not ((rod_data[2][NUMBER_OF_DISKS-2] == 2) && (rod_data[2][NUMBER_OF_DISKS-1] == 4)));
//////////////////////////// bigsmall rod [0] disk[4] disk[3] ////////////////////////////
assert_bigsmall_disk_0_43_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-4] == 3) && (rod_data[0][NUMBER_OF_DISKS-3] == 4)));
assert_bigsmall_disk_0_43_21 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-3] == 3) && (rod_data[0][NUMBER_OF_DISKS-2] == 4)));
```

We can find the result that all these “bigsmall” properties are passed, which means that this RTL plays the (4_3_hanoi) game exactly following Rule #2, but it cannot cover the winning goal, which means it cannot solve the (4_3_hanoi) puzzle.



To find the possible reasons that make it uncovered, we work on its waveform.



We find that the rod_data is stuck on 6C, which is pretty weird because we have exactly constrained it on the code. Then we are pretty sure that what makes the property uncovered is the value of rod_data.

In order to further check the possible reason, we write four cover properties to check if the initial value of rod[0] is covered.

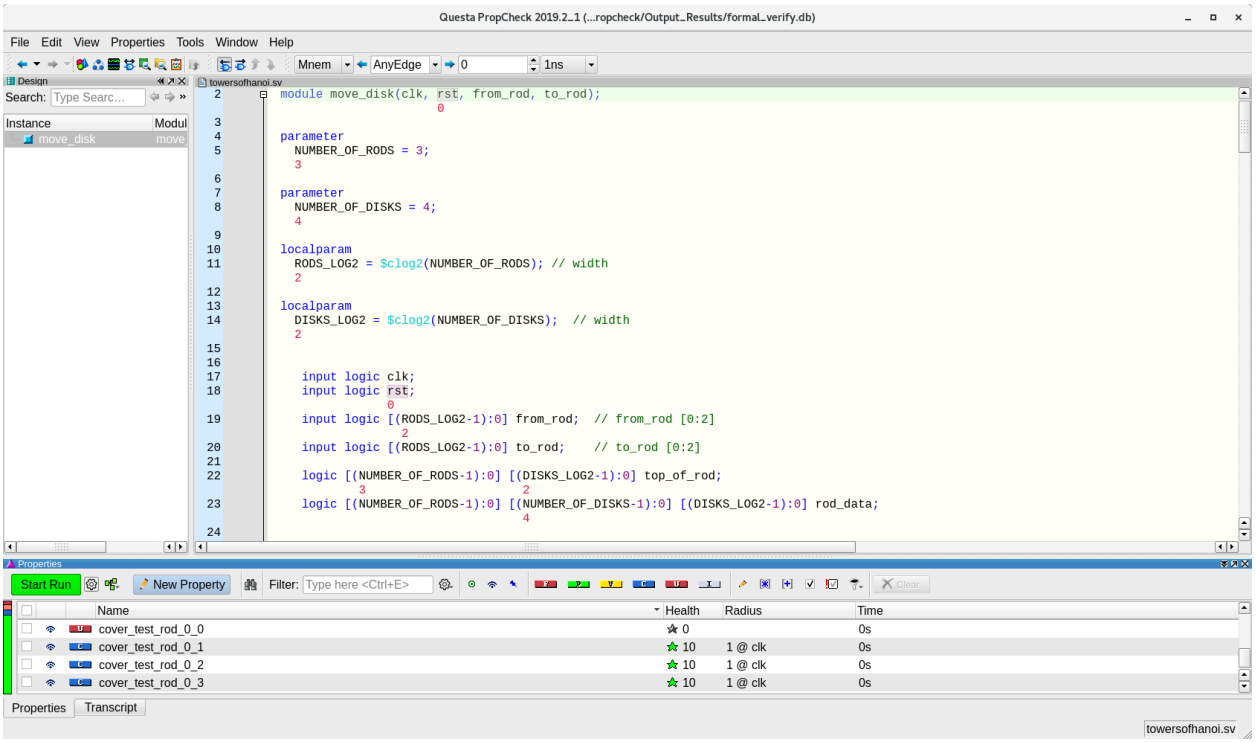
The code is as follows

```
//////////////////////////        test rod [0]         ////////////////////////////

cover_test_rod_0_0 : cover property (@(posedge clk) rod_data[0][NUMBER_OF_DISKS-4] == 4);
cover_test_rod_0_1 : cover property (@(posedge clk) rod_data[0][NUMBER_OF_DISKS-3] == 3);
cover_test_rod_0_2 : cover property (@(posedge clk) rod_data[0][NUMBER_OF_DISKS-2] == 2);
cover_test_rod_0_3 : cover property (@(posedge clk) rod_data[0][NUMBER_OF_DISKS-1] == 1);
```

From the following screenshot, we find that all rod[0] rod_data are covered except rod[0][0] which is supposed to be 4, but in fact is not covered. In this scenario, we believe that this unexpected results are caused by overflow. In other words, while we define rod_data and top_of_data, we haven't given enough bits for them, leading them to overflow in certain scenario. For example, if NUMBER_OF_RODS = 3 and NUMBER_OF_DISKS = 4, two bits are not enough to rod_data because the maximum value of rod_data is 4, and **4 is 100b in binary**.

Tricky $clog2 Function:

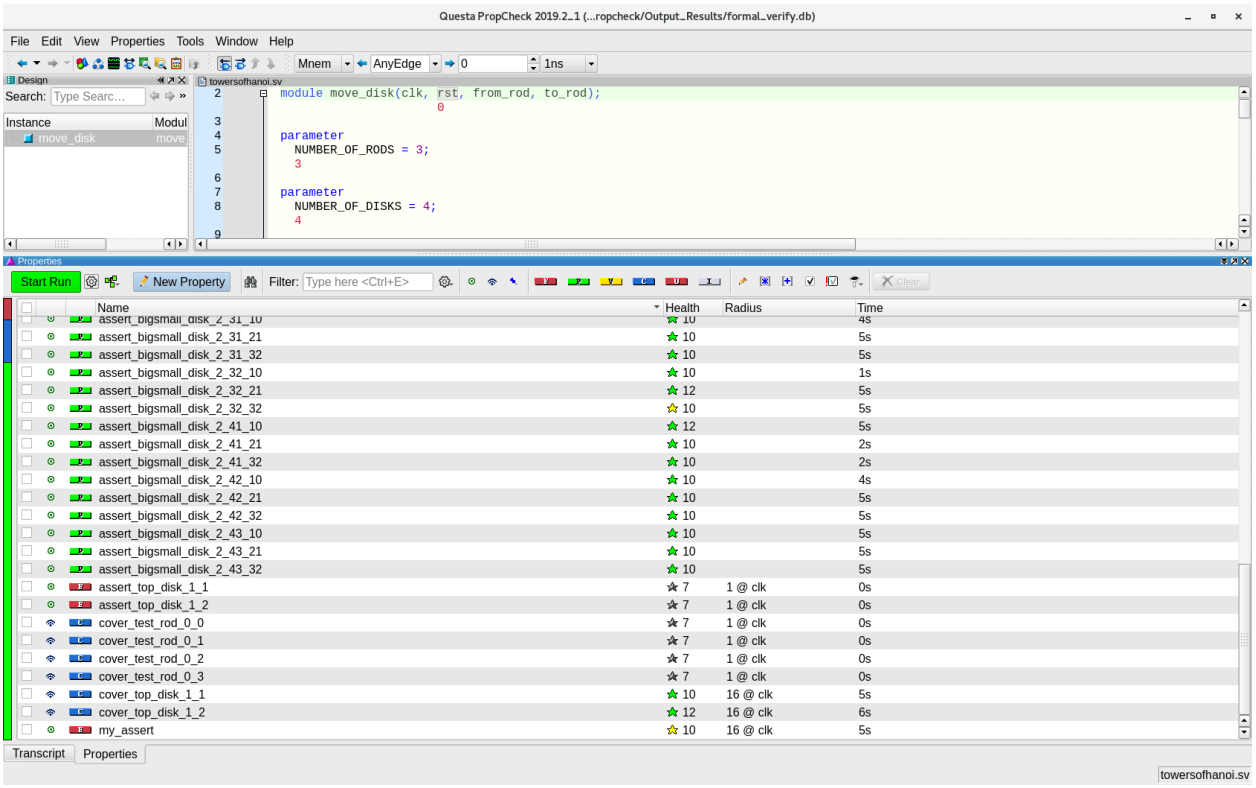
Since the function $clog2 is very tricky, when we compute **$clog2(3)** we can get 2, **when we compute $clog2(4)** we can get 2. At the beginning, the designer work on the 3_3 hanoi game, the width DISKS_LOG2-1 and RODS_LOG2-1 are enough to hold the value. To number like 4, 8, 16, 2^n the tricky $clog2 function makes width is not enough for them to solve the game. Hence, anytime when we meet number like 2^n, we need to change the width(remove the '-1') to make this RTL design work.



Hence, we modify the code by removing '-1' in defining the data width of rod_data and top_of_data.

```
//logic [(NUMBER_OF_RODS-1):0] [(DISKS_LOG2-1):0] top_of_rod;
//logic [(NUMBER_OF_RODS-1):0] [(NUMBER_OF_DISKS-1):0] [(DISKS_LOG2-1):0] rod_data;

logic [(NUMBER_OF_RODS-1):0] [(DISKS_LOG2):0] top_of_rod;
logic [(NUMBER_OF_RODS-1):0] [(NUMBER_OF_DISKS):0] [(DISKS_LOG2):0] rod_data;
```

The result goes as expected that rules assertions are passed and the winning goals are covered, which means the modified code can solve the 4_3_hanoi game.



The winning goal is to move the entire stack to another rod obeying two rules. When we move the entire stack to another rod(rod[1] or rod[2] or rod[3]), the top disk of the target rod (rod[1] or rod[2] or rod[3]) is the smallest disk (disk[1]). Just like what we do in question a and b. We can combine assertions and cover properties together to prove that it can satisfy the winning requirements at least once.

We need to add assertions:

The event that top disk of rod[1] is disk[1] is always false.

The event that top disk of rod[2] is disk[1] is always false.

The event that top disk of rod[3] is disk[1] is always false.

and cover property:

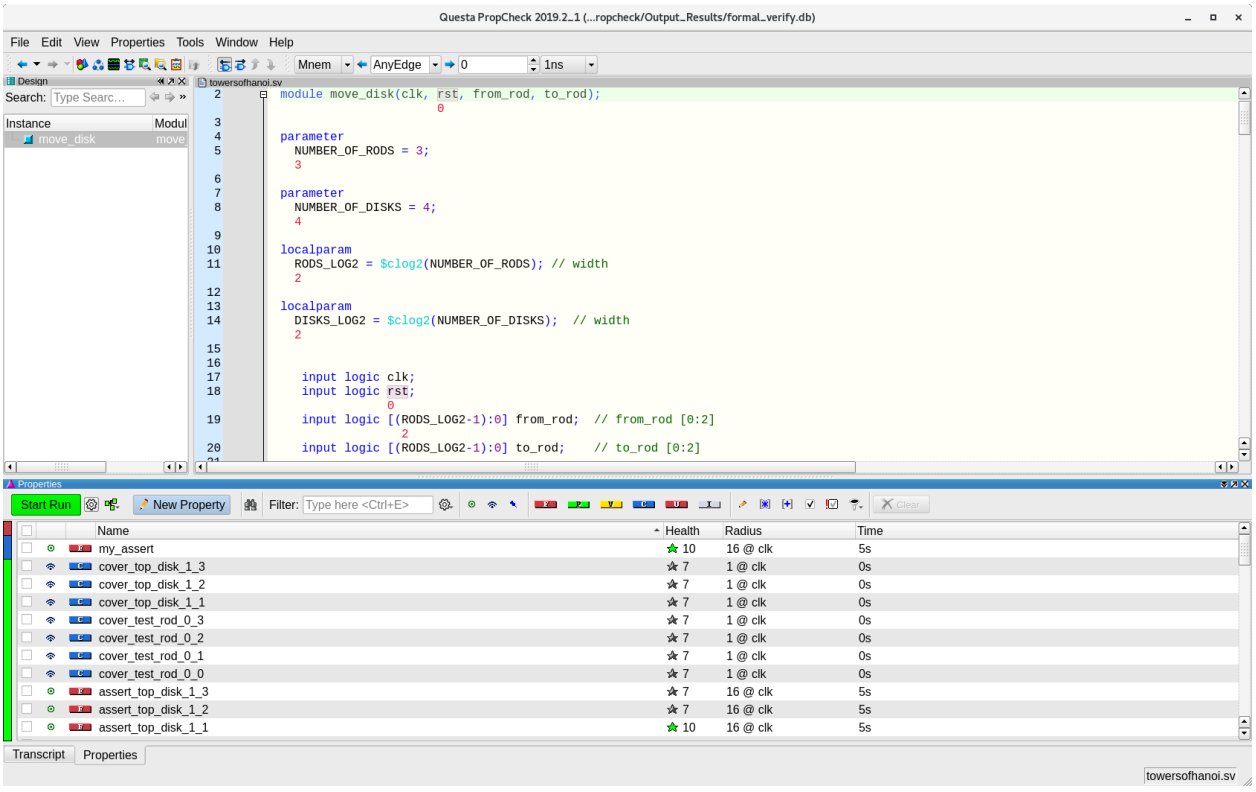
The event that top disk of rod[1] is disk[1] eventually happens.

The event that top disk of rod[2] is disk[1] eventually happens.

The event that top disk of rod[3] is disk[1] eventually happens.

```
//////////////////////////             winning goal         //////////////////////////
assert_top_disk_1_1 :assert property (@(posedge clk) rod_data[1][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[1] is disk[1] is always true.
assert_top_disk_1_2 :assert property (@(posedge clk) rod_data[2][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[2] is disk[1] is always true.
assert_top_disk_1_3 :assert property (@(posedge clk) rod_data[2][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[2] is disk[1] is always true.
cover_top_disk_1_1 : cover property (@(posedge clk) rod_data[1][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[1] is disk[1] eventually happens.
cover_top_disk_1_2 : cover property (@(posedge clk) rod_data[2][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[2] is disk[1] eventually happens.
cover_top_disk_1_3 : cover property (@(posedge clk) rod_data[2][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[2] is disk[1] eventually happens.
```

And we can see the result in the following screenshot that the assertions assert_top_disk_1_1 ,assert_top_disk_1_2 and assert_top_disk_1_3 are fired, but cover properties are covered, which can prove that it can satisfy the winning requirements at least once.



g. NUMBER_OF_DISKS = 5 and NUMBER_OF_RODS =4, redo d

To do formal verification on this RTL, we divide potential properties into three parts: Basic Functions, Game Rules and Winning Goal

Basic Functions: in this part, the properties are mainly involved with basic functions of the code like if we set rst to 1, whether all the values are reset as we expected.

Game Rules

Rule #1: at any time only one disk at the top of one of the rods may be moved to the top of another rod.

Rule #2: never place a disk on top of a smaller one

Winning Goal

As for whether we should first verify "Game Rules" properties or first verify "Winning Goal" properties, we thought that maybe it is easier to write "Winning Goal" properties. (like what we have done in question c)

But we changed our minds very soon because it is meaningless to get a right answer in a wrong way, which means if we do not first verify it is played exactly following the rules, it is useless even if we get the right answer(result).

The assertions and cover properties of **Game Rules:**

Big disk can never be put on the top of the small disk. If we want to do a formal verification on this RTL design. We must cover all state space. In this case, the number of rods is equal to 4 and the number of disks is equal to 5. We adopt a strategy to write assertions that verify the disk size one by one. In this strategy, we want to verify, for example,

<u>The event that above disk is disk[2] and below disk is disk[1] will never happen in rod [0].</u>

```
////////////////////////// bigsmall rod [0] disk[2] disk[1] ////////////////////////////
assert_bigsmall_disk_0_21_10 : assert property (@(posedge clk)  not ((rod_data[0][NUMBER_OF_DISKS-4] == 1) && (rod_data[0][NUMBER_OF_DISKS-3] == 2)));
```

Following this pattern, if we want to do a formal verification of this sort of properties, we need to cover all state space, which means we need to cover all possibilities:

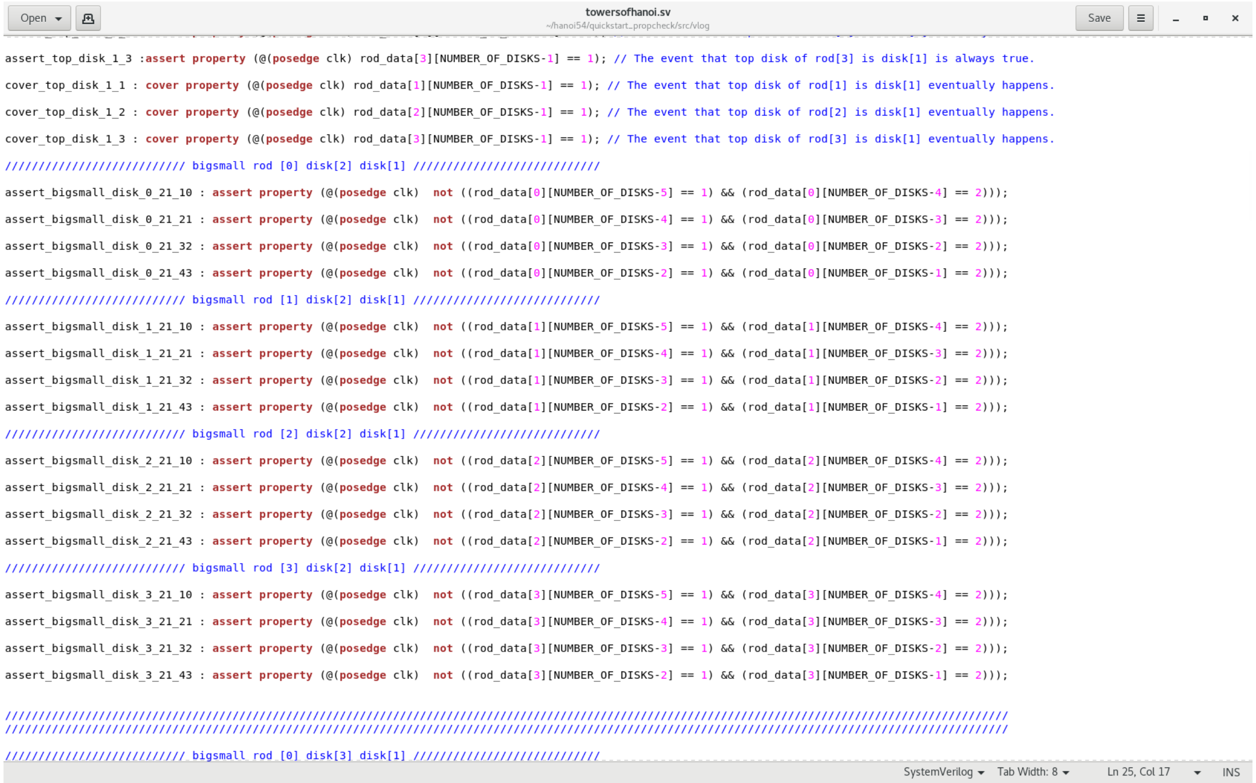
$$C_5^2 * 4 * 4 = 160$$

C_5^2 refers to select 2 disks out of 5 disks.

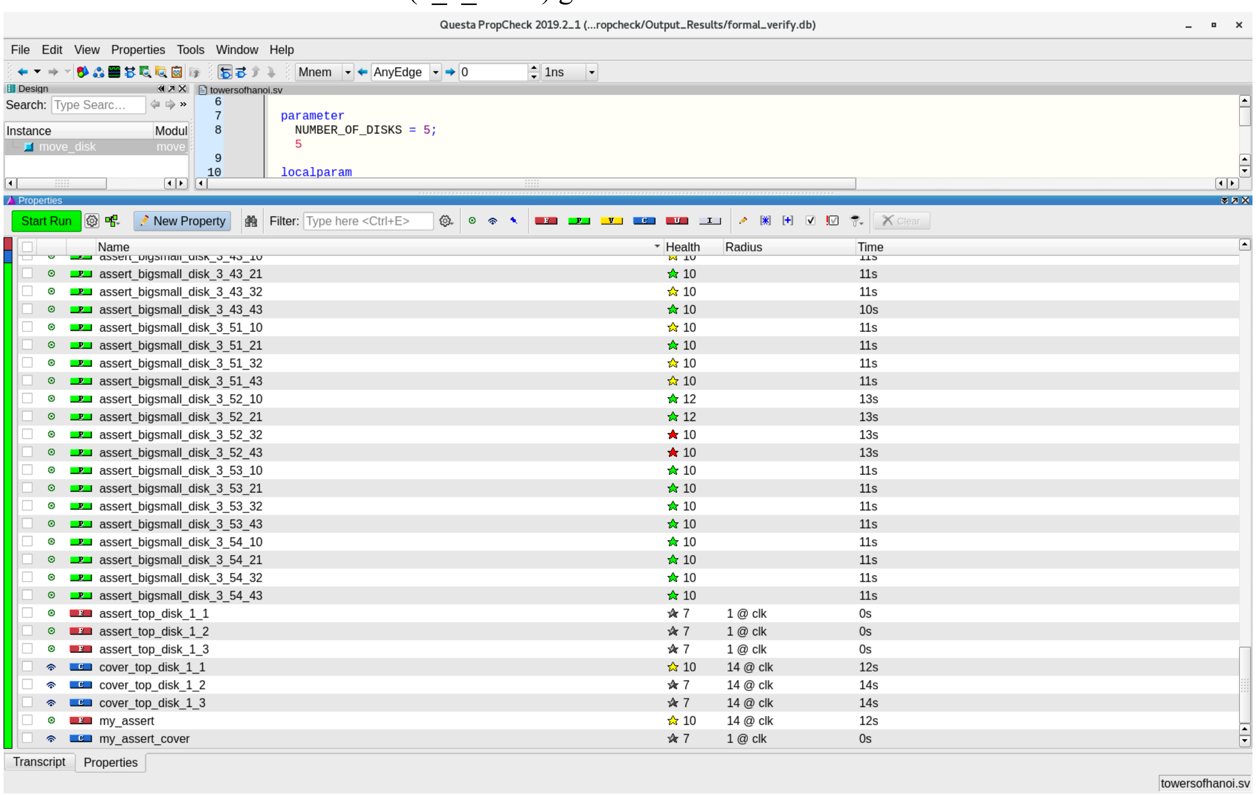
The first 4 refers 4 rods.

The second 4 refers to 4 possible height relations.

Part of rules properties code is as follows



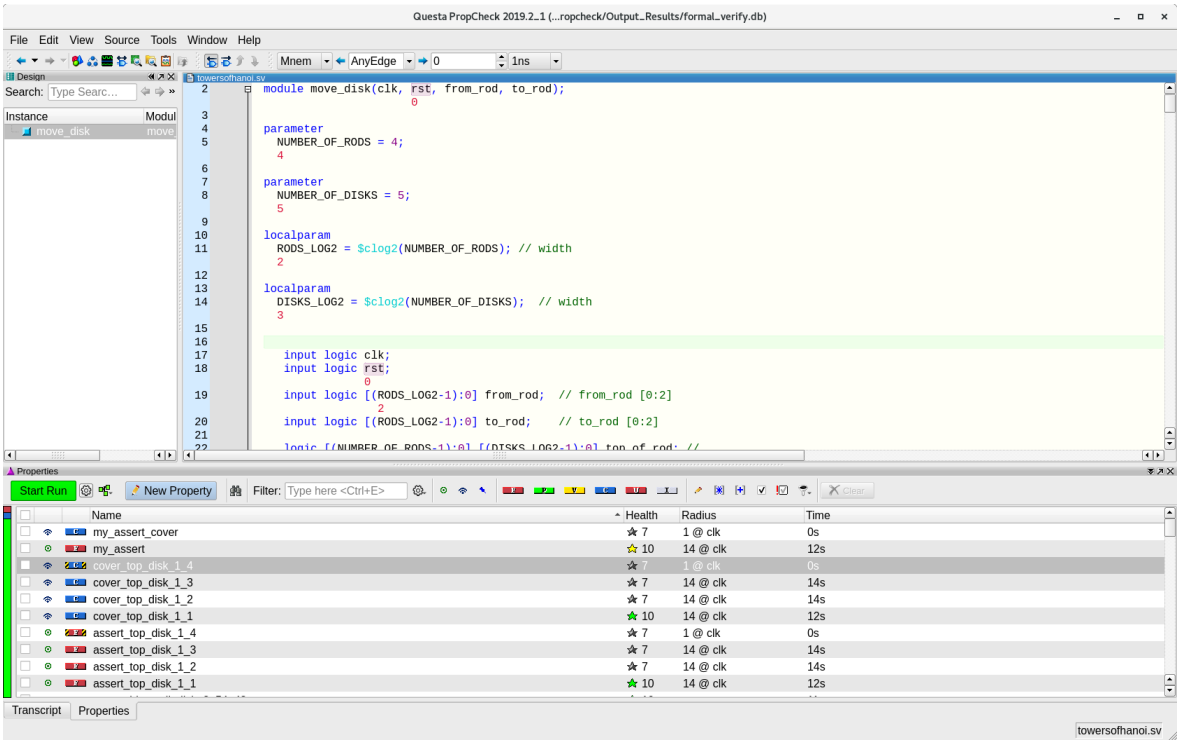
We can find the result that all these "bigsmall" properties are passed, which means that this RTL plays the (5_4_hanoi) game exactly following the Rule #2, and all winning goal cover properties are covered which means that this code can solve the (5_4_hanoi) game.



The winning goal is to move the entire stack to another rod obeying two rules. When we move the entire stack to another rod(rod[1] or rod[2] or rod[3] or rod[4]), the top disk of the target rod (rod[1] or rod[2] or rod[3] or rod[4]) is the smallest disk (disk[1]). Just like what we do in question a and b. We can combine assertions and cover properties together to prove that it can satisfy the winning requirements at least once.
We need to add assertions:
The event that top disk of rod[1] is disk[1] is always false.
The event that top disk of rod[2] is disk[1] is always false.
The event that top disk of rod[3] is disk[1] is always false.
The event that top disk of rod[4] is disk[1] is always false.
and cover property:
The event that top disk of rod[1] is disk[1] eventually happens.
The event that top disk of rod[2] is disk[1] eventually happens.
The event that top disk of rod[3] is disk[1] eventually happens.
The event that top disk of rod[4] is disk[1] eventually happens.

```
//////////////////////////            winning goal         ////////////////////////////

assert_top_disk_1_1 :assert property (@(posedge clk) rod_data[1][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[1] is disk[1] is always false.

assert_top_disk_1_2 :assert property (@(posedge clk) rod_data[2][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[2] is disk[1] is always false.

assert_top_disk_1_3 :assert property (@(posedge clk) rod_data[3][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[3] is disk[1] is always false.

assert_top_disk_1_4 :assert property (@(posedge clk) rod_data[4][NUMBER_OF_DISKS-1] != 1); // The event that top disk of rod[4] is disk[1] is always false.

cover_top_disk_1_1 : cover property (@(posedge clk) rod_data[1][NUMBER_OF_DISKS-1] == 1); // The event that top disk of rod[1] is disk[1] eventually happens.

cover_top_disk_1_2 : cover property (@(posedge clk) rod_data[2][NUMBER_OF_DISKS-1] == 1); // The event that top disk of rod[2] is disk[1] eventually happens.

cover_top_disk_1_3 : cover property (@(posedge clk) rod_data[3][NUMBER_OF_DISKS-1] == 1); // The event that top disk of rod[3] is disk[1] eventually happens.

cover_top_disk_1_4 : cover property (@(posedge clk) rod_data[4][NUMBER_OF_DISKS-1] == 1); // The event that top disk of rod[4] is disk[1] eventually happens.
```

And we can see the result in the following screenshot that the assertions assert_top_disk_1_1 ,assert_top_disk_1_2 and assert_top_disk_1_3 assert_top_disk_1_4 are fired, but cover properties are covered, which can prove that it can satisfy the winning requirements at least once.



Redo Project Hanoi Tower using Cadence Jasper

```
# ----------------------------------------
#  Copyright (c) 2017 Cadence Design Systems, Inc. All Rights
#  Reserved.  Unpublished -- rights reserved under the copyright
#  laws of the United States.
# ----------------------------------------

# Analyze design under verification files
set ROOT_PATH ../designs/reference_design/verilog_sva
set RTL_PATH ${ROOT_PATH}/source/design
set PROP_PATH ${ROOT_PATH}/source/properties

analyze -sv \
  ${RTL_PATH}/towersofhanoi33.sv \
 #${RTL_PATH}/towersofhanoi43.sv \
 #${RTL_PATH}/towersofhanoi53.sv \

# Elaborate design and properties
elaborate -top move_disk

# Set up Clocks and Resets
clock clk
reset rst

# Get design information to check general complexity
get_design_info

# Prove properties
# 1st pass: Quick validation of properties with default engines
set_max_trace_length 10
prove -all
#
# 2nd pass: Validation of remaining properties with different engine
set_max_trace_length 50
set_prove_per_property_time_limit 30s
set_engine_mode {K I N}
prove -all

# Report proof results
report
```

Hanoi_3_3

```
============================================================
RESULTS
============================================================

-------------------------------------------------------------------------------------------------
      Name                                        |    Result    | Engine |  Bound  |  Time
-------------------------------------------------------------------------------------------------

---[ <embedded> ]------------------------------------------------------------------------------
[1]   move_disk.my_assert                         cex            K            14    0.388 s
[2]   move_disk.my_assert_cover                   covered        Hp            1    0.060 s
[3]   move_disk.assert_top_disk_1_1               cex            K            14    0.388 s
[4]   move_disk.assert_top_disk_1_2               cex            K            27    9.801 s
[5]   move_disk.cover_top_disk_1_1                covered        K            14    0.388 s
[6]   move_disk.cover_top_disk_1_2                covered        K            27    9.801 s
[7]   move_disk.assert_bigsmall_disk_0_21_10      proven         N      Infinite    0.319 s
[8]   move_disk.assert_bigsmall_disk_0_21_21      proven         N      Infinite    0.166 s
[9]   move_disk.assert_bigsmall_disk_1_21_10      proven         N      Infinite    0.415 s
[10]  move_disk.assert_bigsmall_disk_1_21_21      proven         N      Infinite    0.138 s
[11]  move_disk.assert_bigsmall_disk_2_21_10      proven         N      Infinite    0.298 s
[12]  move_disk.assert_bigsmall_disk_2_21_21      proven         N      Infinite    0.136 s
[13]  move_disk.assert_bigsmall_disk_0_32_10      proven         I      Infinite    0.069 s
[14]  move_disk.assert_bigsmall_disk_0_32_21      proven         N      Infinite    0.070 s
[15]  move_disk.assert_bigsmall_disk_1_32_10      proven         I      Infinite    0.057 s
[16]  move_disk.assert_bigsmall_disk_1_32_21      proven         N      Infinite    0.107 s
[17]  move_disk.assert_bigsmall_disk_2_32_10      proven         N      Infinite    0.029 s
[18]  move_disk.assert_bigsmall_disk_2_32_21      proven         N      Infinite    0.023 s
[19]  move_disk.assert_bigsmall_disk_0_31_10      proven         N      Infinite    0.027 s
[20]  move_disk.assert_bigsmall_disk_0_31_21      proven         N      Infinite    0.036 s
[21]  move_disk.assert_bigsmall_disk_1_31_10      proven         N      Infinite    0.038 s
[22]  move_disk.assert_bigsmall_disk_1_31_21      proven         N      Infinite    0.067 s
[23]  move_disk.assert_bigsmall_disk_2_31_10      proven         N      Infinite    0.017 s
[24]  move_disk.assert_bigsmall_disk_2_31_21      undetermined   K      51 - 16777216  0.163 s
[<embedded>] %
```

Property Table

No filter | Filter on name

Type	Name	Engine	Bound	Time	Task	Traces	Source
Assert	move_disk.my_assert	K	14	0.4	<embedded>	1	Analysis Session
Cover	move_disk.my_assert_cover	Hp	1	0.1	<embedded>	1	Analysis Session
Assume	move_disk.assume_adjacent	?		0.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_top_disk_1_1	K	14	0.4	<embedded>	1	Analysis Session
Assert	move_disk.assert_top_disk_1_2	K	27	9.8	<embedded>	1	Analysis Session
Cover	move_disk.cover_top_disk_1_1	K	14	0.4	<embedded>	1	Analysis Session
Cover	move_disk.cover_top_disk_1_2	K	27	9.8	<embedded>	1	Analysis Session
Assert	move_disk.assert_bigsmall_disk_0_21_10	N (14)	Infinite	0.3	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_0_21_21	N (17)	Infinite	0.2	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_1_21_10	N (13)	Infinite	0.4	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_1_21_21	N (16)	Infinite	0.1	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_21_10	N (15)	Infinite	0.3	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_21_21	N (17)	Infinite	0.1	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_0_32_10	I (14)	Infinite	0.1	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_0_32_21	N (15)	Infinite	0.1	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_1_32_10	I (14)	Infinite	0.1	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_1_32_21	N (14)	Infinite	0.1	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_32_10	N (14)	Infinite	0.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_32_21	N (14)	Infinite	0.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_0_31_10	N (14)	Infinite	0.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_0_31_21	N (15)	Infinite	0.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_1_31_10	N (13)	Infinite	0.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_1_31_21	N (16)	Infinite	0.1	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_31_10	N (14)	Infinite	0.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_31_21	K	51 - 1677...	0.2	<embedded>	0	Analysis Session

Hanoi_4_3

```
---[ <embedded> ]-------------------------------------------------------------------------------
[1]   move_disk.my_assert                          cex          K           16    13.093 s
[2]   move_disk.assert_top_disk_1_1                cex          K           16    13.093 s
[3]   move_disk.assert_top_disk_1_2                cex          K           16    12.861 s
[4]   move_disk.assert_top_disk_1_3                cex          K           16    12.861 s
[5]   move_disk.cover_top_disk_1_1                 covered      Hp           1    0.055 s
[6]   move_disk.cover_top_disk_1_2                 covered      Hp           1    0.055 s
[7]   move_disk.cover_top_disk_1_3                 covered      Hp           1    0.055 s
[8]   move_disk.cover_test_rod_0_0                 covered      Hp           1    0.055 s
[9]   move_disk.cover_test_rod_0_1                 covered      Hp           1    0.055 s
[10]  move_disk.cover_test_rod_0_2                 covered      Hp           1    0.055 s
[11]  move_disk.cover_test_rod_0_3                 covered      Hp           1    0.055 s
[12]  move_disk.assert_bigsmall_disk_0_21_10       proven       N      Infinite    0.912 s
[13]  move_disk.assert_bigsmall_disk_0_21_21       proven       N      Infinite    0.654 s
[14]  move_disk.assert_bigsmall_disk_0_21_32       proven       N      Infinite    1.392 s
[15]  move_disk.assert_bigsmall_disk_1_21_10       proven       N      Infinite    0.631 s
[16]  move_disk.assert_bigsmall_disk_1_21_21       proven       N      Infinite    0.786 s
[17]  move_disk.assert_bigsmall_disk_1_21_32       proven       N      Infinite    0.693 s
[18]  move_disk.assert_bigsmall_disk_2_21_10       proven       N      Infinite    0.587 s
[19]  move_disk.assert_bigsmall_disk_2_21_21       proven       N      Infinite    0.699 s
[20]  move_disk.assert_bigsmall_disk_2_21_32       proven       N      Infinite    0.506 s
[21]  move_disk.assert_bigsmall_disk_0_32_10       proven       N      Infinite    1.489 s
[22]  move_disk.assert_bigsmall_disk_0_32_21       proven       N      Infinite    1.663 s
[23]  move_disk.assert_bigsmall_disk_0_32_32       proven       N      Infinite    1.544 s
[24]  move_disk.assert_bigsmall_disk_1_32_10       proven       N      Infinite    1.169 s
[25]  move_disk.assert_bigsmall_disk_1_32_21       proven       N      Infinite    4.712 s
[26]  move_disk.assert_bigsmall_disk_1_32_32       proven       N      Infinite    1.721 s
[27]  move_disk.assert_bigsmall_disk_2_32_10       proven       N      Infinite    0.782 s
[28]  move_disk.assert_bigsmall_disk_2_32_21       proven       N      Infinite    2.397 s
[29]  move_disk.assert_bigsmall_disk_2_32_32       proven       N      Infinite    2.207 s
[30]  move_disk.assert_bigsmall_disk_0_31_10       proven       N      Infinite    0.636 s
[31]  move_disk.assert_bigsmall_disk_0_31_21       proven       N      Infinite    17.755 s
[32]  move_disk.assert_bigsmall_disk_0_31_32       proven       N      Infinite    14.916 s
[33]  move_disk.assert_bigsmall_disk_1_31_10       proven       N      Infinite    2.434 s
[34]  move_disk.assert_bigsmall_disk_1_31_21       proven       N      Infinite    0.867 s
[35]  move_disk.assert_bigsmall_disk_1_31_32       proven       N      Infinite    1.038 s
[36]  move_disk.assert_bigsmall_disk_2_31_10       proven       N      Infinite    0.502 s
[37]  move_disk.assert_bigsmall_disk_2_31_21       proven       N      Infinite    0.743 s
[38]  move_disk.assert_bigsmall_disk_2_31_32       proven       N      Infinite    1.004 s
[39]  move_disk.assert_bigsmall_disk_0_41_10       proven       N      Infinite    1.670 s
[40]  move_disk.assert_bigsmall_disk_0_41_21       proven       N      Infinite    3.110 s
```

Property Table

No filter | Filter on name

Type	Name	Engine	Bound	Time	Task	Traces	Source
Assert	move_disk.my_assert	K	16	13.1	<embedded>	1	Analysis Session
Cover	move_disk.cover_top_disk_1_3	Hp	1	0.1	<embedded>	1	Analysis Session
Cover	move_disk.cover_top_disk_1_2	Hp	1	0.1	<embedded>	1	Analysis Session
Cover	move_disk.cover_top_disk_1_1	Hp	1	0.1	<embedded>	1	Analysis Session
Cover	move_disk.cover_test_rod_0_3	Hp	1	0.1	<embedded>	1	Analysis Session
Cover	move_disk.cover_test_rod_0_2	Hp	1	0.1	<embedded>	1	Analysis Session
Cover	move_disk.cover_test_rod_0_1	Hp	1	0.1	<embedded>	1	Analysis Session
Cover	move_disk.cover_test_rod_0_0	Hp	1	0.1	<embedded>	1	Analysis Session
Assert	move_disk.assert_top_disk_1_3	K	16	12.9	<embedded>	1	Analysis Session
Assert	move_disk.assert_top_disk_1_2	K	16	12.9	<embedded>	1	Analysis Session
Assert	move_disk.assert_top_disk_1_1	K	16	13.1	<embedded>	1	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_43_32	N (12)	Infinite	1.2	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_43_21	N (11)	Infinite	0.8	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_43_10	N (11)	Infinite	0.7	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_42_32	N (10)	Infinite	1.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_42_21	N (11)	Infinite	1.1	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_42_10	N (10)	Infinite	0.8	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_41_32	N (12)	Infinite	0.7	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_41_21	N (11)	Infinite	0.9	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_41_10	N (11)	Infinite	0.6	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_32_32	N (19)	Infinite	2.2	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_32_21	N (17)	Infinite	2.4	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_32_10	N (11)	Infinite	0.8	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_31_32	N (14)	Infinite	1.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_31_21	N (10)	Infinite	0.7	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_31_10	N (10)	Infinite	0.5	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_21_32	N (11)	Infinite	0.5	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_21_21	N (10)	Infinite	0.7	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_2_21_10	N (12)	Infinite	0.6	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_1_43_32	N (12)	Infinite	1.8	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_1_43_21	N (18)	Infinite	4.7	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_1_43_10	N (12)	Infinite	0.9	<embedded>	0	Analysis Session

Hanoi_5_4

Name	Result	Engine	Bound	Time
<embedded>				
[1] move_disk.my_assert	cex	N	14	48.046 s
[2] move_disk.my_assert_cover	covered	Hp	1	0.059 s
[3] move_disk.assert_top_disk_1_1	cex	N	14	48.046 s
[4] move_disk.assert_top_disk_1_2	cex	K	14	43.110 s
[5] move_disk.assert_top_disk_1_3	cex	K	14	37.057 s
[6] move_disk.cover_top_disk_1_1	covered	N	14	48.046 s
[7] move_disk.cover_top_disk_1_2	covered	K	14	43.110 s
[8] move_disk.cover_top_disk_1_3	covered	K	14	37.057 s
[9] move_disk.assert_bigsmall_disk_0_21_10	proven	N	Infinite	1.256 s
[10] move_disk.assert_bigsmall_disk_0_21_21	proven	N	Infinite	2.557 s
[11] move_disk.assert_bigsmall_disk_0_21_32	proven	N	Infinite	3.992 s
[12] move_disk.assert_bigsmall_disk_0_21_43	proven	N	Infinite	3.635 s
[13] move_disk.assert_bigsmall_disk_1_21_10	proven	N	Infinite	1.189 s
[14] move_disk.assert_bigsmall_disk_1_21_21	proven	N	Infinite	2.763 s
[15] move_disk.assert_bigsmall_disk_1_21_32	proven	N	Infinite	1.917 s
[16] move_disk.assert_bigsmall_disk_1_21_43	proven	N	Infinite	1.592 s
[17] move_disk.assert_bigsmall_disk_2_21_10	proven	N	Infinite	3.135 s
[18] move_disk.assert_bigsmall_disk_2_21_21	proven	N	Infinite	4.293 s
[19] move_disk.assert_bigsmall_disk_2_21_32	proven	N	Infinite	3.837 s
[20] move_disk.assert_bigsmall_disk_2_21_43	proven	N	Infinite	5.444 s
[21] move_disk.assert_bigsmall_disk_3_21_10	proven	N	Infinite	2.056 s
[22] move_disk.assert_bigsmall_disk_3_21_21	proven	N	Infinite	8.784 s
[23] move_disk.assert_bigsmall_disk_3_21_32	proven	N	Infinite	1.856 s
[24] move_disk.assert_bigsmall_disk_3_21_43	proven	N	Infinite	4.182 s
[25] move_disk.assert_bigsmall_disk_0_31_10	proven	N	Infinite	0.999 s
[26] move_disk.assert_bigsmall_disk_0_31_21	proven	N	Infinite	4.746 s
[27] move_disk.assert_bigsmall_disk_0_31_32	proven	N	Infinite	1.689 s
[28] move_disk.assert_bigsmall_disk_0_31_43	proven	N	Infinite	5.952 s
[29] move_disk.assert_bigsmall_disk_1_31_10	proven	N	Infinite	1.845 s
[30] move_disk.assert_bigsmall_disk_1_31_21	proven	N	Infinite	21.901 s
[31] move_disk.assert_bigsmall_disk_1_31_32	proven	N	Infinite	1.386 s
[32] move_disk.assert_bigsmall_disk_1_31_43	proven	N	Infinite	8.228 s
[33] move_disk.assert_bigsmall_disk_2_31_10	proven	N	Infinite	6.777 s

Property Table

No filter | Filter on name

Type	Name	Engine	Bound	Time	Task	Traces	Source
Cover	move_disk.my_assert_cover	Hp	1	0.1	<embedded>	1	Analysis Session
Assert	move_disk.my_assert	N	14	48.0	<embedded>	1	Analysis Session
Cover	move_disk.cover_top_disk_1_3	K	14	37.1	<embedded>	1	Analysis Session
Cover	move_disk.cover_top_disk_1_2	K	14	43.1	<embedded>	1	Analysis Session
Cover	move_disk.cover_top_disk_1_1	N	14	48.0	<embedded>	1	Analysis Session
Assert	move_disk.assert_top_disk_1_3	K	14	37.1	<embedded>	1	Analysis Session
Assert	move_disk.assert_top_disk_1_2	K	14	43.1	<embedded>	1	Analysis Session
Assert	move_disk.assert_top_disk_1_1	N	14	48.0	<embedded>	1	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_54_43	N (13)	Infinite	7.8	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_54_32	N (17)	Infinite	48.9	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_54_21	N (15)	Infinite	11.1	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_54_10	N (11)	Infinite	2.2	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_53_43	N (10)	Infinite	2.8	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_53_32	N (12)	Infinite	1.7	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_53_21	N (15)	Infinite	22.2	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_53_10	N (12)	Infinite	1.6	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_52_43	N (11)	Infinite	2.9	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_52_32	N (10)	Infinite	2.3	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_52_21	N (10)	Infinite	1.8	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_52_10	N (10)	Infinite	2.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_51_43	N (11)	Infinite	3.5	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_51_32	N (13)	Infinite	3.8	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_51_21	N (12)	Infinite	1.4	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_51_10	N (13)	Infinite	5.1	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_43_43	N (11)	Infinite	2.9	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_43_32	N (10)	Infinite	2.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_43_21	N (15)	Infinite	6.0	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_43_10	N (14)	Infinite	2.8	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_42_43	N (19)	Infinite	10.7	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_42_32	N (11)	Infinite	2.7	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_42_21	N (11)	Infinite	1.5	<embedded>	0	Analysis Session
Assert	move_disk.assert_bigsmall_disk_3_42_10	N (15)	Infinite	4.5	<embedded>	0	Analysis Session

Total: 168 | Filtered: 168 | Selected: 0 | Validity: 164:4:0:0 | Run: 0:0:0:168

The difference between JasperGold and Questa is that JasperGold spends much more time on producing the verification report but it can produce a more well-rounded report and can detect more errors, for example, JasperGold can find the rod_data and top_of_rod width error that Questa cannot detect(just give it a fired or uncovered).