

Yosys Application Note 012: Converting Verilog to BTOR

Ahmed Irfan and Clifford Wolf
April 2015

Abstract—Verilog-2005 is a powerful Hardware Description Language (HDL) that can be used to easily create complex designs from small HDL code. BTOR [3] is a bit-precise word-level format for model checking. It is a simple format and easy to parse. It allows to model the model checking problem over the theory of bit-vectors with one-dimensional arrays, thus enabling to model Verilog designs with registers and memories. Yosys [1] is an Open-Source Verilog synthesis tool that can be used to convert Verilog designs with simple assertions to BTOR format.

I. INSTALLATION

Yosys written in C++ (using features from C++11) and is tested on modern Linux. It should compile fine on most UNIX systems with a C++11 compiler. The README file contains useful information on building Yosys and its prerequisites.

Yosys is a large and feature-rich program with some dependencies. For this work, we may deactivate other extra features such as TCL and ABC support in the Makefile.

This Application Note is based on GIT Rev. 082550f from 2015-04-04 of Yosys [1].

II. QUICK START

We assume that the Verilog design is synthesizable and we also assume that the design does not have multi-dimensional memories. As BTOR implicitly initializes registers to zero value and memories stay uninitialized, we assume that the Verilog design does not contain initial blocks. For more details about the BTOR format, please refer to [3].

We provide a shell script `verilog2btor.sh` which can be used to convert a Verilog design to BTOR. The script can be found in the `backends/btor` directory. The following example shows its usage:

```
verilog2btor.sh fsm.v fsm.btor test
```

Listing 1. Using `verilog2btor` script

The script `verilog2btor.sh` takes three parameters. In the above example, the first parameter `fsm.v` is the input design, the second parameter `fsm.btor` is the file name of BTOR output, and the third parameter `test` is the name of top module in the design.

To specify the properties (that need to be checked), we have two options:

- We can use the Verilog `assert` statement in the procedural block or module body of the Verilog design, as shown in Listing 2. This is the preferred option.
- We can use a single-bit output wire, whose name starts with `safety`. The value of this output wire needs to be driven low when the property is met, i.e. the solver will try to find a model that makes the safety pin go high. This is demonstrated in Listing 3.

```
module test(input clk, input rst, output y);

    reg [2:0] state;

    always @(posedge clk) begin
        if (rst || state == 3) begin
            state <= 0;
        end else begin
            assert(state < 3);
            state <= state + 1;
        end
    end

    assign y = state[2];

    assert property (y != 1'b1);

endmodule
```

Listing 2. Specifying property in Verilog design with `assert`

```
module test(input clk, input rst,
            output y, output safety1);

    reg [2:0] state;

    always @(posedge clk) begin
        if (rst || state == 3)
            state <= 0;
        else
            state <= state + 1;
        end

    assign y = state[2];

    assign safety1 = !(y != 1'b1);

endmodule
```

Listing 3. Specifying property in Verilog design with output wire

We can run Boolector [2] 1.4.1¹ on the generated BTOR file:

```
$ boolector fsm.btor
unsat
```

Listing 4. Running `boolector` on BTOR file

We can also use `nuXmv` [4], but on BTOR designs it does not support memories yet. With the next release of `nuXmv`, we will be also able to verify designs with memories.

III. DETAILED FLOW

Yosys is able to synthesize Verilog designs up to the gate level. We are interested in keeping registers and memories when synthesizing the design. For this purpose, we describe a customized Yosys synthesis flow, that is also provided by the `verilog2btor.sh`

¹ Newer version of Boolector do not support sequential models. Boolector 1.4.1 can be built with `picosat-951`. Newer versions of `picosat` have an incompatible API.

script. Listing 5 shows the Yosys commands that are executed by `verilog2btor.sh`.

```

1 read_verilog -sv $1;
2 hierarchy -top $3; hierarchy -libdir $DIR;
3 hierarchy -check;
4 proc; opt;
5 opt_expr -mux_undef; opt;
6 rename -hide;;;
7 splice; opt;
8 memory_dff -wr_only; memory_collect;;
9 flatten;;
10 memory_unpack;
11 splitnets -driver;
12 setundef -zero -undriven;
13 opt;;;
14 write_btor $2;
```

Listing 5. Synthesis Flow for BTOR with memories

Here is short description of what is happening in the script line by line:

- 1) Reading the input file.
- 2) Setting the top module in the hierarchy and trying to read automatically the files which are given as include in the file read in first line.
- 3) Checking the design hierarchy.
- 4) Converting processes to multiplexers (muxs) and flip-flops.
- 5) Removing undef signals from muxs.
- 6) Hiding all signal names that are not used as module ports.
- 7) Explicit type conversion, by introducing slice and concat cells in the circuit.
- 8) Converting write memories to synchronous memories, and collecting the memories to multi-port memories.
- 9) Flattening the design to get only one module.
- 10) Separating read and write memories.
- 11) Splitting the signals that are partially assigned
- 12) Setting undef to zero value.
- 13) Final optimization pass.
- 14) Writing BTOR file.

For detailed description of the commands mentioned above, please refer to the Yosys documentation, or run `yosys -h command_name`.

The script presented earlier can be easily modified to have a BTOR file that does not contain memories. This is done by removing the line number 8 and 10, and introduces a new command `memory` at line number 8. Listing 6 shows the modified Yosys script file:

```

read_verilog -sv $1;
hierarchy -top $3; hierarchy -libdir $DIR;
hierarchy -check;
proc; opt;
opt_expr -mux_undef; opt;
rename -hide;;;
splice; opt;
memory;;
flatten;;
splitnets -driver;
setundef -zero -undriven;
opt;;;
write_btor $2;
```

Listing 6. Synthesis Flow for BTOR without memories

IV. EXAMPLE

Here is an example Verilog design that we want to convert to BTOR:

```

module array(input clk);

    reg [7:0] counter;
    reg [7:0] mem [7:0];

    always @(posedge clk) begin
        counter <= counter + 8'd1;
        mem[counter] <= counter;
    end

    assert property (!(counter > 8'd0) ||
        mem[counter - 8'd1] == counter - 8'd1);

endmodule
```

Listing 7. Example - Verilog Design

The generated BTOR file that contain memories, using the script shown in Listing 5:

```

1 var 1 clk
2 array 8 3
3 var 8 $auto$rename.cc:150:execute$20
4 const 8 00000001
5 sub 8 3 4
6 slice 3 5 2 0
7 read 8 2 6
8 slice 3 3 2 0
9 add 8 3 4
10 const 8 00000000
11 ugt 1 3 10
12 not 1 11
13 const 8 11111111
14 slice 1 13 0 0
15 one 1
16 eq 1 1 15
17 and 1 16 14
18 write 8 3 2 8 3
19 acond 8 3 17 18 2
20 anext 8 3 2 19
21 eq 1 7 5
22 or 1 12 21
23 const 1 1
24 one 1
25 eq 1 23 24
26 cond 1 25 22 24
27 root 1 -26
28 cond 8 1 9 3
29 next 8 3 28
```

Listing 8. Example - Converted BTOR with memory

And the BTOR file obtained by the script shown in Listing 6, which expands the memory into individual elements:

```

1 var 1 clk
2 var 8 mem[0]
3 var 8 $auto$rename.cc:150:execute$20
4 slice 3 3 2 0
5 slice 1 4 0 0
6 not 1 5
7 slice 1 4 1 1
8 not 1 7
9 slice 1 4 2 2
10 not 1 9
11 and 1 8 10
12 and 1 6 11
13 cond 8 12 3 2
14 cond 8 1 13 2
15 next 8 2 14
16 const 8 00000001
17 add 8 3 16
18 const 8 00000000
19 ugt 1 3 18
20 not 1 19
21 var 8 mem[2]
22 and 1 7 10
23 and 1 6 22
24 cond 8 23 3 21
25 cond 8 1 24 21
26 next 8 21 25
27 sub 8 3 16
:
:
54 cond 1 53 50 52
55 root 1 -54
:
:
77 cond 8 76 3 44
78 cond 8 1 77 44
79 next 8 44 78

```

Listing 9. Example - Converted BTOR without memory

V. LIMITATIONS

BTOR does not support initialization of memories and registers, i.e. they are implicitly initialized to value zero, so the initial block for memories need to be removed when converting to BTOR. It should also be kept in consideration that BTOR does not support the x or z values of Verilog.

Another thing to bear in mind is that Yosys will convert multi-dimensional memories to one-dimensional memories and address decoders. Therefore out-of-bounds memory accesses can yield unexpected results.

VI. CONCLUSION

Using the described flow, we can use Yosys to generate word-level verification benchmarks with or without memories from Verilog designs.

REFERENCES

- [1] Clifford Wolf. The Yosys Open SYnthesis Suite.
<http://www.clifford.at/yosys/>
- [2] Robert Brummayer and Armin Biere, Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays
<http://fmv.jku.at/boolector/>
- [3] Robert Brummayer and Armin Biere and Florian Lonsing, BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking
<http://fmv.jku.at/papers/BrummayerBiereLonsing-BPR08.pdf>
- [4] Roberto Cavada and Alessandro Cimatti and Michele Dorigatti and Alberto Griggio and Alessandro Mariotti and Andrea Micheli and Sergio Mover and Marco Roveri and Stefano Tonetta, The nuXmv Symbolic Model Checker
<https://es-static.fbk.eu/tools/nuxmv/index.php>