
SymbiYosys Documentation

Release 0.1

Clifford Wolf

Jan 26, 2021

Contents

1	Installing	3
1.1	Prerequisites	3
1.2	Yosys, Yosys-SMTBMC and ABC	3
1.3	SymbiYosys	3
1.4	Yices 2	4
1.5	Z3	4
1.6	super_prove	4
1.7	Avy	4
1.8	Boolector	5
2	Getting Started	7
2.1	First step: A simple BMC example	7
2.2	Selecting the right engine	8
2.3	Beyond bounded model checks	10
3	Reference for .sby file format	13
3.1	Tasks section	13
3.2	Options section	15
3.3	Engines section	15
3.3.1	smtbmc engine	16
3.3.2	aiger engine	16
3.3.3	abc engine	17
3.4	Script section	17
3.5	Files section	17
3.6	File sections	18
3.7	Pycode blocks	18
4	Formal extensions to Verilog	21
4.1	SystemVerilog Immediate Assertions	21
4.2	SystemVerilog Functions	22
4.3	Liveness and Fairness	23
4.4	Unconstrained Variables	23
4.5	Global Clock	24
4.6	SystemVerilog Concurrent Assertions	24
5	SystemVerilog, VHDL, SVA	25
5.1	Supported SVA Property Syntax	25

5.1.1	High-Level Convenience Features	25
5.1.2	Expressions in Sequences	26
5.1.3	Sequences	26
5.1.4	Properties	27
5.1.5	Clocking and Reset	27
6	SymbiYosys License	29

SymbiYosys (sby) is a front-end driver program for Yosys-based formal hardware verification flows. SymbiYosys provides flows for the following formal tasks:

- Bounded verification of safety properties (assertions)
- Unbounded verification of safety properties
- Generation of test benches from cover statements
- Verification of liveness properties
- Formal equivalence checking [TBD]
- Reactive Synthesis [TBD]

(Items marked [TBD] are features under construction and not available at the moment.)

Follow the instructions below to install SymbiYosys and its dependencies. Yosys, SymbiYosys, and Z3 are non-optional. The other packages are only required for some engine configurations.

1.1 Prerequisites

Installing prerequisites (this command is for Ubuntu 16.04):

```
sudo apt-get install build-essential clang bison flex libreadline-dev \  
                    gawk tcl-dev libffi-dev git mercurial graphviz \  
                    xdot pkg-config python python3 libftdi-dev gperf \  
                    libboost-program-options-dev autoconf libgmp-dev \  
                    cmake
```

1.2 Yosys, Yosys-SMTBMC and ABC

<http://www.clifford.at/yosys/>

<https://people.eecs.berkeley.edu/~alanmi/abc/>

Next install Yosys, Yosys-SMTBMC and ABC (yosys-abc):

```
git clone https://github.com/YosysHQ/yosys.git yosys  
cd yosys  
make -j$(nproc)  
sudo make install
```

1.3 SymbiYosys

<https://github.com/YosysHQ/SymbiYosys>

```
git clone https://github.com/YosysHQ/SymbiYosys.git SymbiYosys
cd SymbiYosys
sudo make install
```

1.4 Yices 2

<http://yices.csl.sri.com/>

```
git clone https://github.com/SRI-CSL/yices2.git yices2
cd yices2
autoconf
./configure
make -j$(nproc)
sudo make install
```

1.5 Z3

<https://github.com/Z3Prover/z3/wiki>

```
git clone https://github.com/Z3Prover/z3.git z3
cd z3
python scripts/mk_make.py
cd build
make -j$(nproc)
sudo make install
```

1.6 super_prove

https://bitbucket.org/sterin/super_prove_build

Download the right binary .tar.gz for your system from http://downloads.bvsrc.org/super_prove/ and extract it to /usr/local/super_prove.

Then create a wrapper script /usr/local/bin/suprove with the following contents:

```
#!/bin/bash
tool=super_prove; if [ "$1" != "${1#+}" ]; then tool="${1#+}"; shift; fi
exec /usr/local/super_prove/bin/${tool}.sh "$@"
```

1.7 Avy

<https://arieg.bitbucket.io/avy/>

```
git clone https://bitbucket.org/arieg/extavy.git
cd extavy
git submodule update --init
mkdir build; cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
```

(continues on next page)

(continued from previous page)

```
make -j$(nproc)
sudo cp avy/src/{avy,avybmc} /usr/local/bin/
```

1.8 Boolector

<http://fmv.jku.at/boolector/>

```
git clone https://github.com/boolector/boolector
cd boolector
./contrib/setup-btor2tools.sh
./contrib/setup-lingeling.sh
./configure.sh
make -C build -j$(nproc)
sudo cp build/bin/{boolector,btor*} /usr/local/bin/
sudo cp deps/btor2tools/bin/btorsim /usr/local/bin/
```


The example files used in this chapter can be downloaded from [here](#).

2.1 First step: A simple BMC example

Here is a simple example design with a safety property (assertion).

```
module demo (  
    input clk,  
    output reg [5:0] counter  
);  
    initial counter = 0;  
  
    always @(posedge clk) begin  
        if (counter == 15)  
            counter <= 0;  
        else  
            counter <= counter + 1;  
        end  
  
    `ifdef FORMAL  
        always @(posedge clk) begin  
            assert (counter < 32);  
        end  
    `endif  
endmodule
```

The property in this example is true. We'd like to verify this using a bounded model check (BMC) that is 100 cycles deep.

SymbiYosys is controlled by .sby files. The following file can be used to configure SymbiYosys to run a BMC for 100 cycles on the design:

```
[options]
mode bmc
depth 100

[engines]
smtbmc

[script]
read -formal demo.sv
prep -top demo

[files]
demo.sv
```

Simply create a text file `demo.sv` with the example design and another text file `demo.sby` with the SymbiYosys configuration. Then run:

```
sby demo.sby
```

This will run a bounded model check for 100 cycles. The last few lines of the output should look something like this:

```
SBY [demo] engine_0: ##      0   0:00:00  Checking asserts in step 96..
SBY [demo] engine_0: ##      0   0:00:00  Checking asserts in step 97..
SBY [demo] engine_0: ##      0   0:00:00  Checking asserts in step 98..
SBY [demo] engine_0: ##      0   0:00:00  Checking asserts in step 99..
SBY [demo] engine_0: ##      0   0:00:00  Status: PASSED
SBY [demo] engine_0: Status returned by engine: PASS
SBY [demo] engine_0: finished (returncode=0)
SBY [demo] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY [demo] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY [demo] summary: engine_0 (smtbmc) returned PASS
SBY [demo] DONE (PASS)
```

This will also create a `demo/` directory tree with all relevant information, such as a copy of the design source, various log files, and trace data in case the proof fails.

(Use `sby -f demo.sby` to re-run the proof. Without `-f` the command will fail because the output directory `demo/` already exists.)

Time for a simple exercise: Modify the design so that the property is false and the offending state is reachable within 100 cycles. Re-run `sby` with the modified design and see if the proof now fails. Inspect the counterexample trace (`.vcd` file) produced by `sby`. ([GTKWave](#) is an open source VCD viewer that you can use.)

2.2 Selecting the right engine

The `.sby` file for a project selects one or more engines. (When multiple engines are selected, all engines are executed in parallel and the result returned by the first engine to finish is the result returned by SymbiYosys.)

Each engine has its strengths and weaknesses. Therefore it is important to select the right engine for each project. The documentation for the individual engines can provide some guidance for engine selection. (Trial and error can also be a useful method for evaluating engines.)

Let's consider the following example:

```
module testbench (
    input clk, wen,
```

(continues on next page)

(continued from previous page)

```

input [9:0] addr,
input [7:0] wdata,
output [7:0] rdata
);
memory uut (
    .clk (clk ),
    .wen (wen ),
    .addr (addr ),
    .wdata(wdata),
    .rdata(rdata)
);

(* anyconst *) reg [9:0] test_addr;
reg test_data_valid = 0;
reg [7:0] test_data;

always @(posedge clk) begin
    if (addr == test_addr) begin
        if (wen) begin
            test_data <= wdata;
            test_data_valid <= 1;
        end
        if (test_data_valid) begin
            assert(test_data == rdata);
        end
    end
end
endmodule

module memory (
    input clk, wen,
    input [9:0] addr,
    input [7:0] wdata,
    output [7:0] rdata
);
    reg [7:0] bank0 [0:255];
    reg [7:0] bank1 [0:255];
    reg [7:0] bank2 [0:255];
    reg [7:0] bank3 [0:255];

    wire [1:0] mem_sel = addr[9:8];
    wire [7:0] mem_addr = addr[7:0];

    always @(posedge clk) begin
        case (mem_sel)
            0: if (wen) bank0[mem_addr] <= wdata;
            1: if (wen) bank1[mem_addr] <= wdata;
            2: if (wen) bank1[mem_addr] <= wdata; // BUG: Should assign to bank2
            3: if (wen) bank3[mem_addr] <= wdata;
        endcase
    end

    assign rdata =
        mem_sel == 0 ? bank0[mem_addr] :
        mem_sel == 1 ? bank1[mem_addr] :
        mem_sel == 2 ? bank2[mem_addr] :
        mem_sel == 3 ? bank3[mem_addr] : 'bx;

```

(continues on next page)

(continued from previous page)

`endmodule`

This example is expected to fail verification (see the BUG comment). The following `.sby` file can be used to show this:

```
[options]
mode bmc
depth 10
expect fail

[engines]
smtbmc boolector

[script]
read -formal memory.sv
prep -top testbench

[files]
memory.sv
```

This project uses the `smtbmc` engine, which uses SMT solvers to perform the proof. This engine uses the array-theories provided by those solvers to efficiently model memories. Since this example uses large memories, the `smtbmc` engine is a good match.

(`smtbmc boolector` selects Boolector as SMT solver, `smtbmc z3` selects Z3, and `smtbmc yices` selects Yices 2. Yices 2 is the default solver when no argument is used with `smtbmc`.)

Exercise: The engine `abc bmc3` does not provide abstract memory models. Therefore SymbiYosys has to synthesize the memories in the example to FFs and address logic. How does the performance of this project change if `abc bmc3` is used as engine instead of `smtbmc boolector`? How fast can either engine verify the design when the bug has been fixed?

2.3 Beyond bounded model checks

Bounded model checks only prove that the safety properties hold for the first N cycles (where N is the depth of the BMC). Sometimes this is insufficient and we need to prove that the safety properties hold forever, not just the first N cycles. Let us consider the following example:

```
module testbench (
    input clk,
    input reset,
    input [7:0] din,
    output reg [7:0] dout
);
    demo uut (
        .clk (clk ),
        .reset(reset),
        .din  (din  ),
        .dout (dout )
    );

    reg init = 1;
    always @(posedge clk) begin
        if (init) assume (reset);
```

(continues on next page)

(continued from previous page)

```

    if (!reset) assert (!dout[1:0]);
    init <= 0;
end
endmodule

module demo (
    input clk,
    input reset,
    input [7:0] din,
    output reg [7:0] dout
);
    reg [7:0] buffer;
    reg [1:0] state;

    always @(posedge clk) begin
        if (reset) begin
            dout <= 0;
            state <= 0;
        end else
            case (state)
                0: begin
                    buffer <= din;
                    state <= 1;
                end
                1: begin
                    if (buffer[1:0])
                        buffer <= buffer + 1;
                    else
                        state <= 2;
                end
                2: begin
                    dout <= dout + buffer;
                    state <= 0;
                end
            endcase
        end
    end
endmodule

```

Proving this design in an unbounded manner can be achieved using the following SymbiYosys configuration file:

```

[options]
mode prove

[engines]
smtbmc

[script]
read -formal prove.sv
prep -top testbench

[files]
prove.sv

```

Note that mode is now set to prove instead of bmc. The smtbmc engine in prove mode will perform a k-induction proof. Other engines can use other methods, e.g. using abc pdr will prove the design using the IC3 algorithm.

Reference for .sby file format

A .sby file consists of sections. Each section start with a single-line section header in square brackets. The order of sections in a .sby file is for the most part irrelevant, but by convention the usual order is [tasks], [options], [engines], [script], and [files].

3.1 Tasks section

The optional [tasks] section can be used to configure multiple verification tasks in a single .sby file. Each line in the [tasks] section configures one task. For example:

```
[tasks]
task1 task_1_or_2 task_1_or_3
task2 task_1_or_2
task3 task_1_or_3
```

Each task can be assigned additional group aliases, such as task_1_or_2 and task_1_or_3 in the above example.

One or more tasks can be specified as additional command line arguments when calling sby on a .sby file:

```
sby example.sby task2
```

If no task is specified then all tasks in the [tasks] section are run.

After the [tasks] section individual lines can be specified for specific tasks or task groups:

```
[options]
task_1_or_2: mode bmc
task_1_or_2: depth 100
task3: mode prove
```

If the tag <taskname>: is used on a line by itself then the conditional string extends until the next conditional block or -- on a line by itself.

```
[options]
task_1_or_2:
mode bmc
depth 100

task3:
mode prove
--
```

The tag `~<taskname>`: can be used for a line or block that should not be used when the given task is active:

```
[options]
~task3:
mode bmc
depth 100

task3:
mode prove
--
```

The following example demonstrates how to configure safety and liveness checks for all combinations of some host implementations A and B and device implementations X and Y:

```
[tasks]
prove_hAdX prove hostA deviceX
prove_hBdX prove hostB deviceX
prove_hAdY prove hostA deviceY
prove_hBdY prove hostB deviceY
live_hAdX live hostA deviceX
live_hBdX live hostB deviceX
live_hAdY live hostA deviceY
live_hBdY live hostB deviceY

[options]
prove: mode prove
live: mode live

[engines]
prove: abc pdr
live: aiger suprove

[script]
hostA: read -sv hostA.v
hostB: read -sv hostB.v
deviceX: read -sv deviceX.v
deviceY: read -sv deviceY.v
...
```

The `[tasks]` section must appear in the `.sby` file before the first `<taskname>`: or `~<taskname>`: tag.

The command `sby --dumptasks <sby_file>` prints the list of all tasks defined in a given `.sby` file.

3.2 Options section

The `[options]` section contains lines with key-value pairs. The `mode` option is mandatory. The possible values for the `mode` option are:

Mode	Description
<code>bmc</code>	Bounded model check to verify safety properties (<code>assert (...)</code> statements)
<code>prove</code>	Unbounded model check to verify safety properties (<code>assert (...)</code> statements)
<code>live</code>	Unbounded model check to verify liveness properties (<code>assert (s_eventually ...)</code> statements)
<code>cover</code>	Generate set of shortest traces required to reach all <code>cover()</code> statements
<code>equiv</code>	Formal equivalence checking (usually to verify pre- and post-synthesis equivalence)
<code>synth</code>	Reactive Synthesis (synthesis of circuit from safety properties)

All other options have default values and thus are optional. The available options are:

Option	Modes	Description
<code>expect</code>	All	Expected result as comma-separated list of the tokens <code>pass</code> , <code>fail</code> , <code>unknown</code> , <code>error</code> , and <code>timeout</code> . Unexpected results yield a nonzero return code. Default: <code>pass</code>
<code>timeout</code>	All	Timeout in seconds. Default: <code>none</code> (i.e. no timeout)
<code>multiclock</code>	All	Create a model with multiple clocks and/or asynchronous logic. Values: <code>on</code> , <code>off</code> . Default: <code>off</code>
<code>wait</code>	All	Instead of terminating when the first engine returns, wait for all engines to return and check for consistency. Values: <code>on</code> , <code>off</code> . Default: <code>off</code>
<code>aigsm2</code>	All	Which SMT2 solver to use for converting AIGER witnesses to counter example traces. Use <code>none</code> to disable conversion of AIGER witnesses. Default: <code>yices</code>
<code>tbtop</code>	All	The top module for generated Verilog test benches, as hierarchical path relative to the design top module.
<code>smtc</code>	<code>bmc</code> , <code>prove</code> , <code>cover</code>	Pass this <code>.smtc</code> file to the <code>smtbmc</code> engine. All other engines are disabled when this option is used. Default: <code>None</code>
<code>depth</code>	<code>bmc</code> , <code>cover</code>	Depth of the bounded model check. Only the specified number of cycles are considered. Default: <code>20</code>
	<code>prove</code>	Depth for the k-induction performed by the <code>smtbmc</code> engine. Other engines ignore this option in <code>prove</code> mode. Default: <code>20</code>
<code>skip</code>	<code>bmc</code> , <code>cover</code>	Skip the specified number of time steps. Only valid with <code>smtbmc</code> engine. All other engines are disabled when this option is used. Default: <code>None</code>
<code>append</code>	<code>bmc</code> , <code>prove</code> , <code>cover</code>	When generating a counter-example trace, add the specified number of cycles at the end of the trace. Default: <code>0</code>

3.3 Engines section

The `[engines]` section configures which engines should be used to solve the given problem. Each line in the `[engines]` section specifies one engine. When more than one engine is specified then the result returned by the first engine to finish is used.

Each engine configuration consists of an engine name followed by engine options, usually followed by a solver name and solver options.

Example:

```
[engines]
smtbmc --syn --nopresat z3 rewriter.cache_all=true opt.enable_sat=true
abc sim3 -W 15
```

In the first line `smtbmc` is the engine, `--syn --nopresat` are engine options, `z3` is the solver, and `rewriter.cache_all=true opt.enable_sat=true` are solver options.

In the 2nd line `abc` is the engine, there are no engine options, `sim3` is the solver, and `-W 15` are solver options.

3.3.1 smtbmc engine

The `smtbmc` engine supports the `bmc`, `prove`, and `cover` modes and supports the following options:

Option	Description
<code>--nomem</code>	Don't use the SMT theory of arrays to model memories. Instead synthesize memories to registers and address logic.
<code>--syn</code>	Synthesize the circuit to a gate-level representation instead of using word-level SMT operators. This also runs some low-level logic optimization on the circuit.
<code>--stbv</code>	Use large bit vectors (instead of uninterpreted functions) to represent the circuit state.
<code>--stdt</code>	Use SMT-LIB 2.6 datatypes to represent states.
<code>--nopresat</code>	Do not run "presat" SMT queries that make sure that assumptions are non-conflicting (and potentially warmup the SMT solver).
<code>--unroll</code> , <code>--nounroll</code>	Disable/enable unrolling of the SMT problem. The default value depends on the solver being used.
<code>--dumpsmt2</code>	Write the SMT2 trace to an additional output file. (Useful for benchmarking and troubleshooting.)
<code>--progress</code>	Enable Yosys-SMTBMC timer display.

Any SMT2 solver that is compatible with `yosys-smtbmc` can be passed as argument to the `smtbmc` engine. The solver options are passed to the solver as additional command line options.

The following solvers are currently supported by `yosys-smtbmc`:

- `yices`
- `boolector`
- `z3`
- `mathsat`
- `cvc4`

Any additional options after `--` are passed to `yosys-smtbmc` as-is.

3.3.2 aiger engine

The `aiger` engine is a generic front-end for hardware modelcheckers that are capable of processing AIGER files. The engine supports no engine options and supports the following solvers:

Solver	Modes
<code>suprove</code>	<code>prove</code> , <code>live</code>
<code>avy</code>	<code>prove</code>
<code>aigbmc</code>	<code>prove</code> , <code>live</code>

Solver options are passed to the solver as additional command line options.

3.3.3 abc engine

The `abc` engine is a front-end for the functionality in Berkeley ABC. It currently supports no engine options and supports the following solvers:

Solver	Modes	ABC Command
bmc3	bmc	bmc3 -F <depth> -v
sim3	bmc	sim3 -F <depth> -v
pdr	prove	pdr

Solver options are passed as additional arguments to the ABC command implementing the solver.

3.4 Script section

The `[script]` section contains the Yosys script that reads and elaborates the design under test. For example, for a simple project contained in a single design file `mytest.sv` with the top-module `mytest`:

```
[script]
read -sv mytest.sv
prep -top mytest
```

Or explicitly using the Verific SystemVerilog parser (default for `read -sv` when Yosys is built with Verific support):

```
[script]
verific -sv mytest.sv
verific -import mytest
prep -top mytest
```

Or explicitly using the native Yosys Verilog parser (default for `read -sv` when Yosys is not built with Verific support):

```
[script]
read_verilog -sv mytest.sv
prep -top mytest
```

Run `yosys` in a terminal window and enter `help` on the Yosys prompt for a command list. Run `help <command>` for a detailed description of the command, for example `help prep`.

3.5 Files section

The files section lists the source files for the proof, meaning all the files Yosys will need to access when reading the design, including for example data files for `$readmemh` and `$readmemb`.

`sby` copies these files to `<outdir>/src/` before running the Yosys script. When the Yosys script is executed, it will use the copies in `<outdir>/src/`. (Alternatively absolute filenames can be used in the Yosys script for files not listed in the files section.)

For example:

```
[files]
top.sv
../common/defines.vh
/data/prj42/modules/foobar.sv
```

Will copy these files as `top.v`, `defines.vh`, and `foobar.sv` to `<outdir>/src/`.

If the name of the file in `<outdir>/src/` should be different from the basename of the specified file, then the new file name can be specified before the source file name. For example:

```
[files]
top.sv
defines.vh ../common/defines_footest.vh
foo/bar.sv /data/prj42/modules/foobar.sv
```

3.6 File sections

File sections can be used to create additional files in `<outdir>/src/` from the literal content of the `[file <filename>]` section (“here document”). For example:

```
[file params.vh]
`define RESET_LEN 42
`define FAULT_CYCLE 57
```

3.7 Pycode blocks

Blocks enclosed in `--pycode-begin--` and `--pycode-end--` lines are interpreted as Python code. The function `output(line)` can be used to add configuration file lines from the python code. The variable `task` contains the current task name, if any, and `None` otherwise. The variable `tags` contains a set of all tags associated with the current task.

```
[tasks]
--pycode-begin--
for uut in "rotate reflect".split():
    for op in "SRL SRA SLL SRO SLO ROR ROL FSR FSL".split():
        output("%s_%s %s %s" % (uut, op, uut, op))
--pycode-end--

...

[script]
--pycode-begin--
for op in "SRL SRA SLL SRO SLO ROR ROL FSR FSL".split():
    if op in tags:
        output("read -define %s" % op)
--pycode-end--
rotate: read -define UUT=shifter_rotate
reflect: read -define UUT=shifter_reflect
read -sv test.v
read -sv shifter_reflect.v
read -sv shifter_rotate.v
prep -top test
```

(continues on next page)

(continued from previous page)

...

The command `sby --dumpcfg <sby_file>` can be used to print the configuration without specialization for any particular task, and `sby --dumpcfg <sby_file> <task_name>` can be used to print the configuration with specialization for a particular task.

Formal extensions to Verilog

Any Verilog file may be read using `read -formal <file>` within the SymbiYosys `script` section. Multiple files may be given on the same line, or various files may be read in subsequent lines.

`read -formal` will also define the `FORMAL` macro, which can be used to separate a section having formal properties from the rest of the logic within the core.

```
module somemodule(port1, port2, ...);
    // User logic here
    //
`ifdef FORMAL
    // Formal properties here
`endif
endmodule
```

The `bind()` operator can also be used when using the Verific front end. This will provide an option to attach formal properties to a given piece of logic, without actually modifying the module in question to do so as we did in the example above.

4.1 SystemVerilog Immediate Assertions

SymbiYosys supports three basic immediate assertion types.

1. `assume(<expr>);`

An assumption restricts the possibilities the formal tool examines, making the search space smaller. In any solver generated trace, all of the assumptions will always be true.

2. `assert(<expr>);`

An assertion is something the solver will try to make false. Any time SymbiYosys is run with mode `bmc`, the proof will fail if some set of inputs can cause the `<expr>` within the assertion to be zero (false). When SymbiYosys is run with mode `prove`, the proof may also yield an `UNKNOWN` result if an assertion can be made to fail during the induction step.

3. `cover (<expr>);`

A `cover` statement only applies when SymbiYosys is ran with option `mode cover`. In this case, the formal solver will start at the beginning of time (i.e. when all initial statements are true), and it will try to find some clock when `<expr>` can be made to be true. Such a cover run will “PASS” once all internal `cover()` statements have been fulfilled. It will “FAIL” if any `cover()` statement exists that cannot be reached in the first `N` states, where `N` is set by the `depth` option. A cover pass will also fail if an assertion needs to be broken in order to reach the covered state.

To be used, each of these statements needs to be placed into an *immediate* context. That is, it needs to be placed within an `always` block of some type. Two types of `always` block contexts are permitted:

- `always @(*)`

Formal properties within an `always @(*)` block will be checked on every time step. For synchronous proofs, the property will be checked every clock period. For asynchronous proofs, i.e. those with `multiclock on`, the property will still be checked on every time step but, depending upon how you set up your time steps, it may also be checked multiple times per clock interval.

As an example, consider the following assertion that the `error_flag` signal must remain low.

```
always @(*)
  assert(!error_flag);
```

While it is not recommended that formal properties be mixed with logic in the same `always @(*)` block, the language supports it. In such cases, the formal property will be evaluated as though it took place in the middle of the logic block.

- `always @(posedge <clock>)`

The second form of immediate assertion is one within a clocked `always` block. This form of assertion is required when attempting to use the `$past`, `$stable`, `$changed`, `$rose`, or `$fell` SystemVerilog functions discussed in the next section.

Unlike the `@(*)` assertion, this one will only be checked on the clock edge. Depending upon how the clock is set up, that may mean that there are several formal time steps between when this assertion is checked.

The two types of immediate assertions, both with and without a clock reference, are very similar. There is one critical difference between them, however. The clocked assertion will not be checked until the positive edge of the clock following the time period in question. Within a synchronous design, this means that the fault will not lie on the last time step, but rather the time step prior. New users often find this non-intuitive.

One subtlety to be aware of is that any `always @(*)` assertion that depends upon an `always @(posedge <clock>)` assumption might fail before the assumption is applied. One solution is to use all clocked or all combinatorial blocks. Another solution is to move the assertion into an `always @(posedge <clock>)` block.

4.2 SystemVerilog Functions

Yosys supports five formal related functions: `$past`, `$stable`, `$changed`, `$rose`, and `$fell`. Internally, these are all implemented in terms of the implementation of the `$past` operator.

The `$past (<expr>)` function returns the value of `<expr>` from one clock ago. It can only be used within a clocked `always` block, since the clock is used to define “one clock ago.” It is roughly equivalent to,

```
reg past_value;
always @(posedge clock)
  past_value <= expression;
```

There are two keys to the use of `$past`. The first is that `$past (<expr>)` can only be used within a clocked always block. The second is that there is no initial value given to any `$past (<expr>)`. That means that on the first clock period of any design, `$past (<expr>)` will be undefined.

Yosys supports both one and two arguments to `$past`. In the two argument form, `$past (<expr>, N)`, the expression returns the value of `<expr>` from `N` clocks ago. `N` must be a synthesis time constant.

`$stable (<expr>)` is short hand for `<expr> == $past (<expr>)`.

`$changed (<expr>)` is short hand for `<expr> != $past (<expr>)`.

While the next two functions, `$rose` and `$fell`, can be applied to multi-bit expressions, only the least significant bits will be examined. If we allow that `<expr>` has only a single bit within it, perhaps selected from the least significant bit of a larger expression, then we can express the following equivalencies.

`$rose (<expr>)` is short hand for `<expr> && !$past (<expr>)`.

`$fell (<expr>)` is short hand for `!<expr> && $past (<expr>)`.

4.3 Liveness and Fairness

TBD

```
assert property (eventually <expr>);
```

```
assume property (eventually <expr>);
```

4.4 Unconstrained Variables

Yosys supports four attributes which can be used to create unconstrained variables. These attributes can be applied to the variable at declaration time, as in

```
(* anyconst *) reg some_value;
```

The `(* anyconst *)` attribute will create a solver chosen constant. It is often used when verifying memories: the proof allows the solver to pick a constant address, and then proves that the value at that address matches however the designer desires.

`(* anyseq *)` differs from `(* anyconst *)` in that the solver chosen value can change from one time step to the next. In many ways, it is similar to how the solver will treat an input to the design, with the difference that an `(* anyseq *)` variable can originate internal to the design.

Both `(* anyseq *)` and `(* anyconst *)` marked values can be constrained with assumptions.

Yosys supports two other attributes useful to formal processing, `(* allconst *)` and `(* allseq *)`. These are very similar in their functionality to the `(* anyseq *)` and `(* anyconst *)` attributes we just discussed for creating unconstrained values. Indeed, for both assertions and cover statements, the two sets are identical. Where they differ is with respect to assumptions. Assumed properties of an `(* allseq *)` or `(* allconst *)` value will be applied to all possible values of that variable may take on. This gets around the annoying reality associated with defining a property using `(* anyconst *)` or `(* anyseq *)` only to have the solver pick a value which wasn't the one that was constrained.

4.5 Global Clock

Accessing the formal timestep becomes important when verifying code in any asynchronous context. In such asynchronous contexts, there may be multiple independent clocks within the design. Each of the clocks may be defined by an assumption allowing the designer to carefully select the relationships between them.

All of this requires the `multiclock` on line in the SBY options section.

It also requires the `(* gclk *)` attribute.

To use `(* gclk *)`, define a register with that attribute, as in:

```
(* gclk *) reg formal_timestep;
```

You can then reference this `formal_timestep` in the clocking section of an `always` block, as in,

```
always @(posedge formal_timestep)
    assume(incoming_clock == !$past(incoming_clock));
```

4.6 SystemVerilog Concurrent Assertions

TBD, see *Supported SVA Property Syntax*.

SystemVerilog, VHDL, SVA

Run `verific -sv <files>` in the `[script]` section of your `.sby` file to read a SystemVerilog source file, and `verific -vhdl <files>` to read a VHDL source file.

After all source files have been read, run `verific -import <topmodule>` to import the design elaborated at the specified top module. This step is optional (will be performed automatically) if the top-level module of your design has been read using Verific.

Use `read -sv` to automatically use Verific to read a source file if Yosys has been built with Verific.

Run `yosys -h verific` in a terminal window and enter for more information on the `verific` script command.

5.1 Supported SVA Property Syntax

SVA support in Yosys' Verific bindings is currently in development. At the time of writing, the following subset of SVA property syntax is supported in concurrent assertions, assumptions, and cover statements when using the `verific` command in Yosys to read the design.

5.1.1 High-Level Convenience Features

Most of the high-level convenience features of the SVA language are supported, such as

- `default clocking... endclocking`
- `default disable iff... ;`
- `property... endproperty`
- `sequence... endsequence`
- `checker... endchecker`
- Arguments to sequences, properties, and checkers
- Storing sequences, properties, and checkers in packages

In addition the SVA-specific features, the SystemVerilog `bind` statement and deep hierarchical references are supported, simplifying the integration of formal properties with the design under test.

The `verific` command also allows parsing of VHDL designs and supports binding SystemVerilog modules to VHDL entities and deep hierarchical references from a SystemVerilog formal test-bench into a VHDL design under test.

5.1.2 Expressions in Sequences

Any standard Verilog boolean expression is supported, as well as the SystemVerilog functions `$past`, `$stable`, `$changed`, `$rose`, and `$fell`. These functions can also be used outside of SVA sequences.

Additionally the `<sequence>.triggered` syntax for checking if the end of any given sequence matches the current cycle is supported in expressions.

Finally the usual SystemVerilog functions such as `$countones`, `$onehot`, and `$onehot0` are also supported.

5.1.3 Sequences

Most importantly, expressions and variable-length concatenation are supported:

- *expression*
- *sequence* ##N *sequence*
- *sequence* ## [*] *sequence*
- *sequence* ## [+] *sequence*
- *sequence* ## [N:M] *sequence*
- *sequence* ## [N:\$] *sequence*

Also variable-length repetition:

- *sequence* [*]
- *sequence* [+]
- *sequence* [*N]
- *sequence* [*N:M]
- *sequence* [*N:\$]

And the following more complex operators:

- *sequence* or *sequence*
- *sequence* and *sequence*
- *expression* throughout *sequence*
- *sequence* intersect *sequence*
- *sequence* within *sequence*
- `first_match(sequence)`
- *expression* [=N]
- *expression* [=N:M]
- *expression* [=N:\$]

- *expression* [->N]
- *expression* [->N:M]
- *expression* [->N:\$]

5.1.4 Properties

Currently only a certain set of patterns are supported for SVA properties:

- [*antecedent_condition*] *sequence*
- [*antecedent_condition*] *not sequence*
- *antecedent_condition sequence until_condition*
- *antecedent_condition not sequence until_condition*

Where *antecedent_condition* is one of:

- *sequence* |->
- *sequence* |=>

And *until_condition* is one of:

- *until expression*
- *s_until expression*
- *until_with expression*
- *s_until_with expression*

5.1.5 Clocking and Reset

The following constructs are supported for clocking and reset in most of the places the SystemVerilog standard permits them. However, properties spanning multiple different clock domains are currently unsupported.

- @ (posedge *clock*)
- @ (negedge *clock*)
- @ (posedge *clock* iff *enable*)
- @ (negedge *clock* iff *enable*)
- disable iff (*expression*)

CHAPTER 6

SymbiYosys License

SymbiYosys (sby) itself is licensed under the ISC license:

```
SymbiYosys (sby) -- Front-end for Yosys-based formal verification flows

Copyright (C) 2016 Clifford Wolf <clifford@clifford.at>

Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

Note that the solvers and other components used by SymbiYosys come with their own license terms.