
YosysHQ SBY

YosysHQ GmbH

Aug 04, 2026

CONTENTS

1	Installation guide	3
1.1	CAD suite(s)	3
1.2	Installing from source	3
1.2.1	Prerequisites	3
1.2.2	Required components	3
1.2.3	Recommended components	4
1.2.4	Yices 2	4
1.2.5	Optional components	4
2	Getting started	7
2.1	First In, First Out (FIFO) buffer	7
2.2	Verification properties	8
2.3	SymbiYosys	8
2.4	Exercise	11
2.5	Concurrent assertions	11
2.6	Further information	12
3	Using <i>sby</i>	13
3.1	Positional Arguments	13
3.2	Named Arguments	13
4	Reference for <i>.sby</i> file format	17
4.1	Tasks section	17
4.2	Options section	18
4.3	Cancelledby section	20
4.4	Engines section	20
4.4.1	smtbmc engine	21
4.4.2	btor engine	22
4.4.3	aiger engine	22
4.4.4	abc engine	22
4.4.5	itp engine	23
4.4.6	none engine	24
4.5	Script section	24
4.6	Files section	24
4.7	File sections	25
4.8	Pycode blocks	25
5	Autotune: Automatic Engine Selection	27
5.1	Using Autotune	27
5.1.1	Example	27

5.2	Autotune Log Output	28
5.3	Configuring Autotune	29
5.4	Autotune Options	30
6	Formal extensions to Verilog	31
6.1	SystemVerilog Immediate Assertions	31
6.2	SystemVerilog Functions	32
6.3	Liveness and Fairness	33
6.4	Unconstrained Variables	33
6.5	Global Clock	33
6.6	SystemVerilog Concurrent Assertions	34
7	SystemVerilog, VHDL, SVA	35
7.1	Supported SVA Property Syntax	35
7.1.1	High-Level Convenience Features	35
7.1.2	Expressions in Sequences	36
7.1.3	Sequences	36
7.1.4	Properties	37
7.1.5	Clocking and Reset	37
7.2	SVA properties in a VHDL design	37
8	Application Notes	39
8.1	Formal Basics and Methods	39
9	SymbiYosys license	41

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

INSTALLATION GUIDE

This document will guide you through the process of installing sby.

1.1 CAD suite(s)

Sby (SymbiYosys) is part of the [Tabby CAD Suite](#) and the [OSS CAD Suite](#)! The easiest way to use sby is to install the binary software suite, which contains all required dependencies, including all supported solvers.

- [Contact YosysHQ](#) for a [Tabby CAD Suite](#) Evaluation License and download link
- OR go to <https://github.com/YosysHQ/oss-cad-suite-build/releases> to download the free OSS CAD Suite
- Follow the [Install Instructions on GitHub](#)

Make sure to get a Tabby CAD Suite Evaluation License for extensive SystemVerilog Assertion (SVA) support, as well as industry-grade SystemVerilog and VHDL parsers!

For more information about the difference between Tabby CAD Suite and the OSS CAD Suite, please visit <https://www.yosyshq.com/tabby-cad-datasheet>.

1.2 Installing from source

Follow the instructions below to install sby and its dependencies. Yosys and sby are non-optional. Boolector is recommended to install but not required. The other packages are only required for some engine configurations.

1.2.1 Prerequisites

Installing prerequisites (this command is for Ubuntu 20.04):

```
sudo apt-get install build-essential clang bison flex \
                    libreadline-dev gawk tcl-dev libffi-dev git \
                    graphviz xdot pkg-config python3 zlib1g-dev

python3 -m pip install click
```

1.2.2 Required components

Yosys, Yosys-SMTBMC and ABC

<https://yosyshq.net/yosys/>

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

Note that this will install Yosys, Yosys-SMTBMC and ABC (as `yosys-abc`):

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

sby

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

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

1.2.3 Recommended components

Boolector

<https://boolector.github.io>

```
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/build/bin/btorsim /usr/local/bin/
```

To use the btor engine you will need to install btor2tools from [commit c35cflc](#) or newer.

1.2.4 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
```

1.2.5 Optional components

Additional solver engines can be installed as per their instructions, links are provided below.

Z3

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

super_prove

<https://github.com/sterin/super-prove-build>

Avy

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

rIC3

<https://github.com/gipsyh/rIC3/>

SBY currently supports the release version 1.5.2 of rIC3 only.

GETTING STARTED

Note

This tutorial assumes sby and boolector installation as per the *Installation guide*. For this tutorial, it is also recommended to install *GTKWave*, an open source VCD viewer. [Source files used in this tutorial](#) can be found on the sby git, under docs/examples/fifo.

2.1 First In, First Out (FIFO) buffer

From [Wikipedia](#), a FIFO is

a method for organizing the manipulation of a data structure (often, specifically a data buffer) where the oldest (first) entry, or “head” of the queue, is processed first.

Such processing is analogous to servicing people in a queue area on a first-come, first-served (FCFS) basis, i.e. in the same sequence in which they arrive at the queue’s tail.

In hardware we can create such a construct by providing two addresses into a register file. This tutorial will use an example implementation provided in *fifo.sv*.

First, the address generator module:

```
// address generator/counter
module addr_gen
#( parameter MAX_DATA=16
) ( input en, clk, rst,
    output reg [3:0] addr
);
    initial addr <= 0;

    // async reset
    // increment address when enabled
    always @(posedge clk or posedge rst)
        if (rst)
            addr <= 0;
        else if (en) begin
            if (addr == MAX_DATA-1)
                addr <= 0;
            else
                addr <= addr + 1;
        end
endmodule
```

This module is instantiated twice; once for the write address and once for the read address. In both cases, the address will start at and reset to 0, and will increment by 1 when an enable signal is received. When the address pointers increment from the maximum storage value they reset back to 0, providing a circular queue.

Next, the register file:

```
// fifo storage
// async read, sync write
wire [3:0] waddr, raddr;
reg [7:0] data [MAX_DATA-1:0];
always @(posedge clk)
    if (wen)
        data[waddr] <= wdata;
assign rdata = data[raddr];
```

Notice that this register design includes a synchronous write and asynchronous read. Each word is 8 bits, and up to 16 words can be stored in the buffer.

2.2 Verification properties

In order to verify our design we must first define properties that it must satisfy. For example, there must never be more than there is memory available. By assigning a signal to count the number of values in the buffer, we can make the following assertion in the code:

```
a_oflow: assert (count <= MAX_DATA);
```

It is also possible to use the prior value of a signal for comparison. This can be used, for example, to ensure that the count is only able to increase or decrease by 1. A case must be added to handle resetting the count directly to 0, as well as if the count does not change. This can be seen in the following code; at least one of these conditions must be true at all times if our design is to be correct.

```
a_counts: assert (count == 0
    || count == $past(count)
    || count == $past(count) + 1
    || count == $past(count) - 1);
```

As our count signal is used independently of the read and write pointers, we must verify that the count is always correct. While the write pointer will always be at the same point or *after* the read pointer, the circular buffer means that the write *address* could wrap around and appear *less than* the read address. So we must first perform some simple arithmetic to find the absolute difference in addresses, and then compare with the count signal.

```
assign addr_diff = waddr >= raddr
    ? waddr - raddr
    : waddr + MAX_DATA - raddr;
```

```
a_count_diff: assert (count == addr_diff
    || count == MAX_DATA && addr_diff == 0);
```

2.3 SymbiYosys

SymbiYosys (sby) uses a .sby file to define a set of tasks used for verification.

basic

Bounded model check of design.

nofullskip

Demonstration of failing model using an unbounded model check.

cover

Cover mode (testing cover statements).

noverific

Test fallback to default Verilog frontend.

The use of the `:default` tag indicates that by default, basic and cover should be run if no tasks are specified, such as when running the command below.

```
sby fifo.sby
```

Note

The default set of tests should all pass. If this is not the case there may be a problem with the installation of sby or one of its solvers.

To see what happens when a test fails, the below command can be used. Note the use of the `-f` flag to automatically overwrite existing task output. While this may not be necessary on the first run, it is quite useful when making adjustments to code and rerunning tests to validate.

```
sby -f fifo.sby nofullskip
```

The `nofullskip` task disables the code shown below. Because the count signal has been written such that it cannot exceed `MAX_DATA`, removing this code will lead to the `a_count_diff` assertion failing. Without this assertion, there is no guarantee that data will be read in the same order it was written should an overflow occur and the oldest data be written.

```
`ifndef NO_FULL_SKIP
    // write while full => overwrite oldest data, move read pointer
    assign rskip = wen && !ren && data_count >= MAX_DATA;
    // read while empty => read invalid data, keep write pointer in sync
    assign wskip = ren && !wen && data_count == 0;
`endif // NO_FULL_SKIP
```

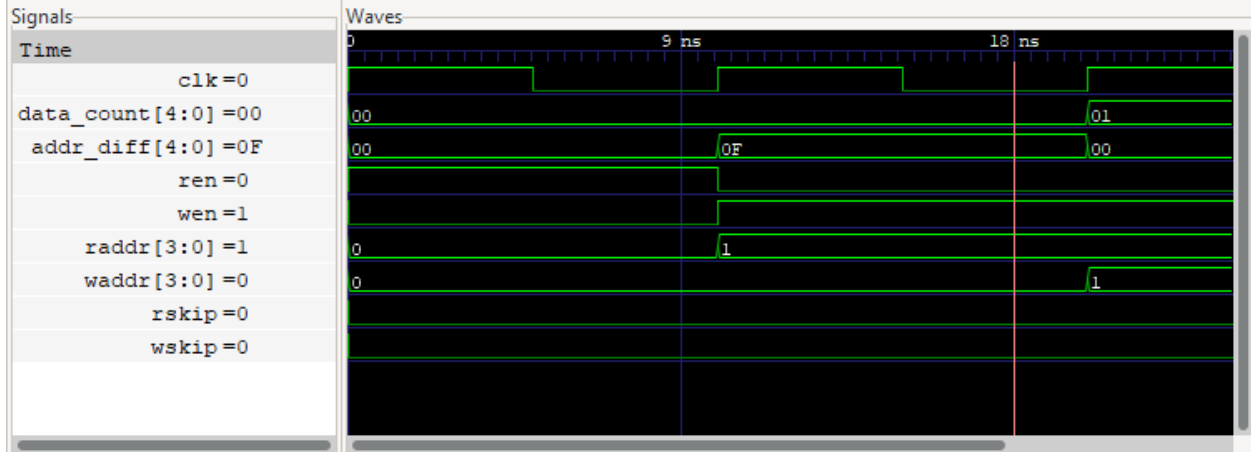
The last few lines of output for the `nofullskip` task should be similar to the following:

```
SBY [fifo_nofullskip] engine_0.basecase: ## Assert failed in fifo: a_count_diff
SBY [fifo_nofullskip] engine_0.basecase: ## Writing trace to VCD file: engine_0/trace.
↪vcd
SBY [fifo_nofullskip] engine_0.basecase: ## Writing trace to Verilog testbench: engine_
↪0/trace_tb.v
SBY [fifo_nofullskip] engine_0.basecase: ## Writing trace to constraints file: engine_0/
↪trace.smtc
SBY [fifo_nofullskip] engine_0.basecase: ## Status: failed
SBY [fifo_nofullskip] engine_0.basecase: finished (returncode=1)
SBY [fifo_nofullskip] engine_0: Status returned by engine for basecase: FAIL
SBY [fifo_nofullskip] engine_0.induction: terminating process
SBY [fifo_nofullskip] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:02 (2)
SBY [fifo_nofullskip] summary: Elapsed process time unavailable on Windows
SBY [fifo_nofullskip] summary: engine_0 (smtbmc boolector) returned FAIL for basecase
SBY [fifo_nofullskip] summary: counterexample trace: fifo_nofullskip/engine_0/trace.vcd
SBY [fifo_nofullskip] DONE (FAIL, rc=2)
SBY The following tasks failed: ['nofullskip']
```

Using the `noskip.gtkw` file provided, use the below command to examine the error trace.

```
gtkwave fifo_nofullskip/engine_0/trace.vcd noskip.gtkw
```

This should result in something similar to the below image. We can immediately see that `data_count` and `addr_diff` are different. Looking a bit deeper we can see that in order to reach this state the read enable signal was high in the first clock cycle while write enable is low. This leads to an underfill where a value is read while the buffer is empty and the read address increments to a higher value than the write address.



During correct operation, the `w_underfill` statement will cover the underflow case. Examining `fifo_cover/logfile.txt` will reveal which trace file includes the cover statement we are looking for. If this file doesn't exist, run the code below.

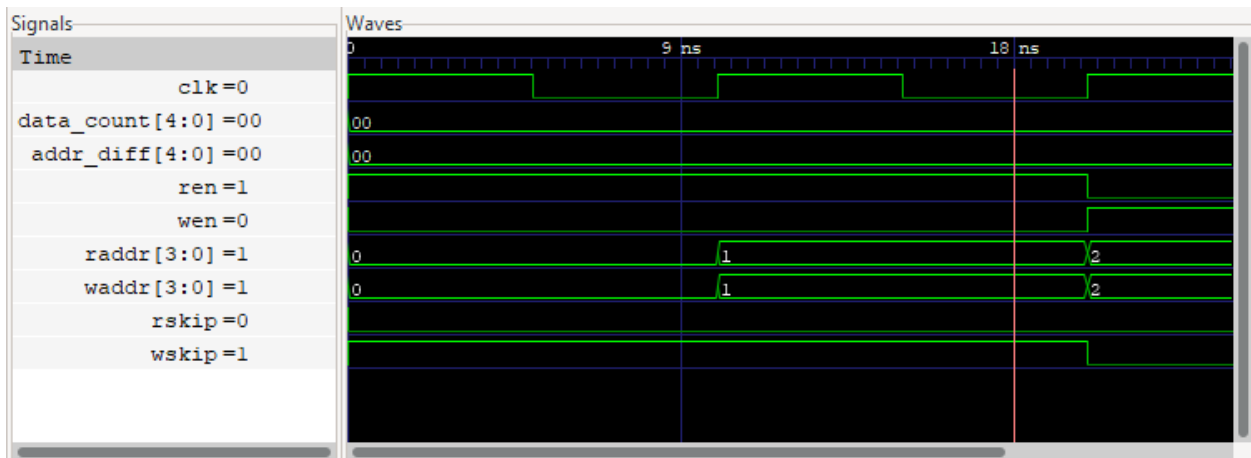
```
sby fifo.sby cover
```

Searching the file for `w_underfill` will reveal the below.

```
$ grep "w_underfill" fifo_cover/logfile.txt -A 1
SBY [fifo_cover] engine_0: ## Reached cover statement at w_underfill in step 2.
SBY [fifo_cover] engine_0: ## Writing trace to VCD file: engine_0/trace4.vcd
```

We can then run `gtkwave` with the trace file indicated to see the correct operation as in the image below. When the buffer is empty, a read with no write will result in the `wkskip` signal going high, incrementing *both* read and write addresses and avoiding underflow.

```
gtkwave fifo_cover/engine_0/trace4.vcd noskip.gtkw
```



Note

Implementation of the `w_underfill` cover statement depends on whether Verific is used or not. See the [Concurrent assertions](#) section for more detail.

2.4 Exercise

Adjust the `[script]` section of `fifo.sby` so that it looks like the below.

```
[script]
nofullskip: read -define NO_FULL_SKIP=1
noverific: read -noverific
read -formal fifo.sv
hierarchy -check -top fifo -chparam MAX_DATA 17
prep -top fifo
```

The `hierarchy` command we added changes the `MAX_DATA` parameter of the top module to be 17. Now run the basic task and see what happens. It should fail and give an error like `Assert failed in fifo: a_count_diff`. Can you modify the verilog code so that it works with larger values of `MAX_DATA` while still passing all of the tests?

Note

If you need a **hint**, try increasing the width of the address wires. 4 bits supports up to $2^4=16$ addresses. Are there other signals that need to be wider? Can you make the width parameterisable to support arbitrarily large buffers?

Once the tests are passing with `MAX_DATA=17`, try something bigger, like 64, or 100. Does the basic task still pass? What about cover? By default, `bmc` & `cover` modes will run to a depth of 20 cycles. If a maximum of one value can be loaded in each cycle, how many cycles will it take to load 100 values? Using the [.sby reference page](#), try to increase the cover mode depth to be at least a few cycles larger than the `MAX_DATA`.

Note

Reference files are provided in the `fifo/golden` directory, showing how the verilog could have been modified and how a `bigtest` task could be added.

2.5 Concurrent assertions

Until this point, all of the properties described have been *immediate* assertions. As the name suggests, immediate assertions are evaluated immediately whereas concurrent assertions allow for the capture of sequences of events which occur across time. The use of concurrent assertions requires a more advanced series of checks.

Compare the difference in implementation of `w_underfill` depending on the presence of Verific. `w_underfill` looks for a sequence of events where the write enable is low but the write address changes in the following cycle. This is the expected behaviour for reading while empty and implies that the `w_skip` signal went high. Verific enables elaboration of SystemVerilog Assertions (SVA) properties. Here we use such a property, `write_skip`.

```
property write_skip;
  @(posedge clk) disable iff (rst)
    !wen ==> $changed(waddr);
```

(continues on next page)

(continued from previous page)

```
endproperty
w_underfill: cover property (write_skip);
```

This property describes a *sequence* of events which occurs on the `clk` signal and are disabled/restarted when the `rst` signal is high. The property first waits for a low `wen` signal, and then a change in `waddr` in the following cycle. `w_underfill` is then a cover of this property to verify that it is possible. Now look at the implementation without Verific.

```
reg past_nwen;
initial past_nwen <= 0;
always @(posedge clk) begin
    if (rst) past_nwen <= 0;
    if (!rst) begin
        w_underfill: cover (past_nwen && $changed(waddr));
        past_nwen <= !wen;
    end
end
```

In this case we do not have access to SVA properties and are more limited in the tools available to us. Ideally we would use `$past` to read the value of `wen` in the previous cycle and then check for a change in `waddr`. However, in the first cycle of simulation, reading `$past` will return a value of `X`. This results in false triggers of the property so we instead implement the `past_nwen` register which we can initialise to `0` and ensure it does not trigger in the first cycle.

As verification properties become more complex and check longer sequences, the additional effort of hand-coding without SVA properties becomes much more difficult. Using a parser such as Verific supports these checks *without* having to write out potentially complicated state machines. Verific is included for use in the *Tabby CAD Suite*.

Note

The Verific frontend for Yosys requires the commercial [Tabby CAD Suite](#). This is not the same as simply having a Verific license when using Yosys.

2.6 Further information

For more information on the uses of assertions and the difference between immediate and concurrent assertions, refer to appnote 109: [Property Checking with SystemVerilog Assertions](#).

USING SBY

Once SBY is installed and available on the command line as *sby*, either built from source or using one of the available CAD suites, it can be called as follows. Note that this information is also available via *sby -help*. For more information on installation, see [Installation guide](#).

```
usage: sby [options] [<jobname>.sby [tasknames] | <dirname>]
```

3.1 Positional Arguments

<jobname>.sby | <dirname> .sby file OR directory containing config.sby file

tasknames tasks to run (only valid when <jobname>.sby is used)

3.2 Named Arguments

- | | |
|---------------------|--|
| -d | set workdir name. default: <jobname> or <jobname>_<taskname>. When there is more than one task, use -prefix instead |
| --prefix | set the workdir name prefix. _<taskname> will be appended to the path for each task |
| -f | remove workdir if it already exists
Default: False |
| -b | backup workdir if it already exists
Default: False |
| -t | run in a temporary workdir (remove when finished)
Default: False |
| -T | add taskname (useful when sby file is read from stdin)
Default: [] |
| -E | throw an exception (incl stack trace) for most errors
Default: False |
| -j | maximum number of processes to run in parallel |
| --sequential | run tasks in sequence, not in parallel
Default: False |

--live	Possible choices: csv, jsonl print live updates of property statuses during task execution, may be specified multiple times
--autotune	automatically find a well performing engine and engine configuration for each task Default: <code>False</code>
--autotune-config	read an autotune configuration file (overrides the sby file's autotune options)
--yosys	Default: <code>{}</code>
--abc	Default: <code>{}</code>
--smtbmc	Default: <code>{}</code>
--witness	Default: <code>{}</code>
--suprove	Default: <code>{}</code>
--aigbmc	Default: <code>{}</code>
--avy	Default: <code>{}</code>
--rIC3	Default: <code>{}</code>
--btormc	Default: <code>{}</code>
--itp-bmc	Default: <code>{}</code>
--pono	configure which executable to use for the respective tool Default: <code>{}</code>
--dumpcfg	print the pre-processed configuration file Default: <code>False</code>
--dumptags	print the list of task tags Default: <code>False</code>
--dumptasks	print the list of tasks Default: <code>False</code>
--dumpdefaults	print the list of default tasks Default: <code>False</code>
--dumptaskinfo	output a summary of tasks as JSON Default: <code>False</code>
--dumpfiles	print the list of source files Default: <code>False</code>
--setup	set up the working directory and exit Default: <code>False</code>
--link	make symbolic links to source files instead of copying them Default: <code>False</code>
--status	summarize the contents of the status database Default: <code>False</code>

--statusfmt	Possible choices: csv, jsonl print the most recent status for each property in specified format Default: ''
--latest	only check statuses from the most recent run of a task Default: False
--statusreset	reset the contents of the status database Default: False
--statuscancels	intertask cancellations can be triggered by the status database Default: False
--taskstatus	display the status of tasks in the status database Default: False
--init-config-file	create a default .sby config file
--version	show program's version number and exit

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].

4.1 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.

Note that there is currently no way to specify dependencies on other tasks. For complex flows where such dependencies are needed, consider using separate `.sby` files, or a single file with external scripting. For an advanced example which uses tasks and external scripting to implement a multi-stage verification flow, see [AppNote 130: Multi-Stage Verification](#).

4.2 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
bmc	Bounded model check to verify safety properties (<code>assert(...)</code> statements)
prove	Unbounded model check to verify safety properties (<code>assert(...)</code> statements)
live	Unbounded model check to verify liveness properties (<code>assert(s_eventually ...)</code> statements)
cover	Generate set of shortest traces required to reach all <code>cover()</code> statements
prep	Prepare the design and write the preprocessed model without running solver engines

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

Option	Modes	Description
expect	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>
timeout	All	Timeout in seconds. Default: <code>none</code> (i.e. no timeout)
multic	All	Create a model with multiple clocks and/or asynchronous logic. Values: <code>on</code> , <code>off</code> . Default: <code>off</code>
wait	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>
vcd	All	Write VCD traces for counter-example or cover traces. Values: <code>on</code> , <code>off</code> . Default: <code>on</code>
vcd_sim	All	When generating VCD traces, use Yosys's <code>sim</code> command. Replaces the engine native VCD output. Values: <code>on</code> , <code>off</code> . Default: <code>off</code>
fst	All	Generate FST traces using Yosys's <code>sim</code> command. Values: <code>on</code> , <code>off</code> . Default: <code>off</code>
cycle_w	All	Cycle width used by Yosys's <code>sim</code> command. Values: even numbers ≥ 2 . Default: <code>10</code>
aigsm	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>
tbtop	All	The top module for generated Verilog test benches, as hierarchical path relative to the design top module.
make_model	All	Force generation of the named formal models. Takes a comma-separated list of model names. For a model <code><name></code> this will generate the <code>model/design_<name>.*</code> files within the working directory, even when not required to run the task.
skip_prep	All	Skip SBY's internal preparation step. Values: <code>on</code> , <code>off</code> . Default: <code>off</code> . For an example of how it can be useful to disable the preparation step in complex flows, see AppNote 130: Multi-Stage Verification .
smtc	bmc, prove, cover	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>
depth	bmc, cover	Depth of the bounded model check. Only the specified number of cycles are considered. Default: <code>20</code>
	prove	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>
skip	bmc, cover	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>
append	bmc, prove, cover	When generating a counter-example trace, add the specified number of cycles at the end of the trace. Default: <code>0</code>
append	bmc, prove, cover	Uphold assumptions when appending cycles at the end of the trace. Depending on the engine and options used this may be implicitly on or not supported (as indicated in SBY's log output). Values: <code>on</code> , <code>off</code> . Default: <code>on</code>
cover_	cover	Check for assertion properties during <code>cover</code> mode. Values: <code>on</code> , <code>off</code> . Default: <code>off</code>

4.3 Cancelledby section

At times it may be desirable for one task to be able to stop another task early, such as when a `prove` task and a `bmc` task are testing the same properties and the `prove` task finishes first, making the `bmc` task redundant. This is where the optional `[cancelledby]` section comes in. Each line corresponds to the name of a task which, if finished, will cancel the current task. No distinction is made for whether the task finished successfully, only that it is finished. Remember that tags can be used to control which tasks contain a given configuration line.

Example:

```
[cancelledby]
task1:
task2
task3

task2:
task3
task4
```

In this example, `task1` can be cancelled by `task2` or `task3`, while `task2` can be cancelled by `task3` or `task4`. If `task4` is the first to finish, then `task2` will be cancelled, which in turn leads to `task1` being cancelled. A task can only be cancelled if it has not yet finished, and will check the finished tasks before starting and periodically while the attached engines are running.

Normally, only the current taskloop is checked for finished tasks. This means that intertask cancellations are not possible across separate SBY invocations or when using the `--sequential` flag. By providing SBY with the `--statuscancels` flag however the status database will be used, allowing tasks to be cancelled independently of taskloop. Note that this also means that a task may be cancelled by a previous invocation unless `--statusreset` is called first.

Example:

```
sby $sbyfile --statusreset
sby $sbyfile a b &
sby $sbyfile c d --statuscancels
```

In this example, the `a` and `b` tasks can only be cancelled by each other, while the `c` and `d` tasks can be cancelled by any task.

4.4 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.

The following mode/engine/solver combinations are currently supported:

Mode	Engine
bmc	smtbmc [all solvers] btor btormc btor pono btor rIC3 abc bmc3 abc sim3 aiger aigbmc aiger rIC3
prove	smtbmc [all solvers] abc pdr aiger avy aiger suprove aiger rIC3 btor rIC3
cover	smtbmc [all solvers] btor btormc
live	aiger suprove

4.4.1 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>--keep-going</code>	In BMC mode, continue after the first failed assertion and report further failed assertions. In prove mode, <code>--keep-going</code> is currently only supported with <code>abc pdr</code> (see the <code>abc</code> engine section below).
<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`
- `bitwuzla`

- z3
- mathsat
- cvc4
- cvc5

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

4.4.2 btor engine

The `btor` engine supports hardware modelcheckers that accept `btor2` files. The engine supports no engine options and supports the following solvers:

Solver	Modes
btormc	bmc, cover
pono	bmc
rIC3	bmc, prove

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

4.4.3 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
suprove	prove, live
avy	prove
rIC3	bmc, prove
aigbmc	bmc

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

4.4.4 abc engine

The `abc` engine is a front-end for the functionality in Berkeley ABC. It supports the following solvers:

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

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

When using the `pdr` solver, the `abc` engine supports the following engine option (currently the only engine option available for `abc`):

Option	Description
<code>--keep-go</code>	Continue after the first proven or failed assertion and report individual per-property results, rather than a single global result.

Example:

```
[engines]
abc --keep-going pdr
```

4.4.5 itp engine

The `itp` engine implements interpolation-based model checking using Craig interpolants extracted from SAT resolution proofs. It uses the external `itp-bmc` tool.

Supported modes: `prove`, `bmc`

Installation:

Install the `itp-bmc` binary to `PATH`:

```
git clone https://github.com/inquisitour/itp-bmc.git
cd itp-bmc
make
sudo cp bmc /usr/local/bin/itp-bmc
```

Or set the `ITP_BMC` environment variable, or use the `--itp-bmc` command-line flag to specify the path directly.

Engine arguments:

```
[engines]
itp <bound> <skip>
```

Argument	Description
<code>bound</code>	Maximum unrolling depth. Default: value of <code>depth</code> option
<code>skip</code>	Number of initial timeframes to skip before checking bad states. Useful for designs requiring reset cycles. Default: value of <code>skip</code> option or 0

Example:

```
[options]
mode prove

[engines]
itp 20 0
```

For designs requiring reset cycles (e.g. `riscv-formal`):

```
[engines]
itp 15 10
```

Note

The `itp` engine does not currently produce counterexample witness traces. When a property violation is found, only `FAIL` status is reported.

4.4.6 none engine

The none engine does not run any solver. It can be used together with the `make_model` option to manually generate any model supported by one of the other engines. This makes it easier to use the same models outside of sby.

4.5 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`.

4.6 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
```

4.7 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
```

4.8 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

...
```

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.

AUTOTUNE: AUTOMATIC ENGINE SELECTION

Selecting the best performing engine for a given verification task often requires some amount of trial and error. To reduce the manual work required for this, `sby` offers the `--autotune` option. This takes an `.sby` file and runs it using engines and engine configurations. At the end it produces a report listing the fastest engines among these candidates.

5.1 Using Autotune

To run `autotune`, you can add the `--autotune` option to your usual `sby` invocation. For example, if you usually run `sby demo.sby` you would run `sby --autotune demo.sby` instead. When the `.sby` file contains multiple tasks, `autotune` is run for each task independently. As without `--autotune`, it is possible to specify which tasks to run on the command line.

`Autotune` runs without requiring further interaction, and will eventually print a list of engine configurations and their respective solving times. To permanently use an engine configuration you can copy it from the `sby --autotune` output into the `[engines]` section of your `.sby` file.

5.1.1 Example

The Sby repository contains a [small example](#) in the `docs/examples/autotune` directory.

The `divider.sby` file contains the following `[engines]` section:

```
[engines]
smtbmc
```

We notice that running `sby -f divider.sby medium` takes a long time and want to see if another engine would speed things up, so we run `sby --autotune -f divider.sby medium`. After a few minutes this prints:

```
SBY [divider_medium] finished engines:
SBY [divider_medium]   #4: engine_7: smtbmc --nopresat bitwuzla -- --noincr (32 seconds,
↳ status=PASS)
SBY [divider_medium]   #3: engine_2: smtbmc boolector -- --noincr (32 seconds,
↳ status=PASS)
SBY [divider_medium]   #2: engine_3: smtbmc --nopresat boolector -- --noincr (32 seconds,
↳ status=PASS)
SBY [divider_medium]   #1: engine_6: smtbmc bitwuzla -- --noincr (31 seconds,
↳ status=PASS)
SBY [divider_medium] DONE (AUTOTUNED, rc=0)
```

This tells us that for the `medium` task, the best engine choice (#1) is `smtbmc bitwuzla -- --noincr`. To use this engine by default we can change the `[engines]` section of `divider.sby` to:

```
[engines]
smtbmc bitwuzla -- --noincr
```

5.2 Autotune Log Output

The log output in `--autotune` mode differs from the usual `sby` log output.

It also starts with preparing the design (this includes running the user provided `[script]`) so it can be passed to the solvers. This is only done once and will be reused to run every candidate engine.

```
SBY [demo] model 'base': preparing now...
SBY [demo] base: starting process "cd demo/src; yosys -ql ../model/design.log ../model/
↳ design.ys"
SBY [demo] base: finished (returncode=0)
SBY [demo] prepared model 'base'
```

This is followed by selecting the engine candidates to run. The design and `sby` configuration are analyzed to skip engines that are not compatible or unlikely to work well. When an engine is skipped due to a recommendation, a corresponding log message is displayed as well as the total number of candidates to try:

```
SBY [demo] checking more than 20 timesteps (100), disabling nonincremental smtbmc
SBY [demo] testing 16 engine configurations...
```

After this, the candidate engine configurations are started. Depending on the configuration, engines can run in parallel. The engine output itself is not logged to stdout when running autotune, so you will only see messages about starting an engine:

```
SBY [demo] engine_1 (smtbmc --nopresat boolector): starting... (14 configurations_
↳ pending)
SBY [demo] engine_2 (smtbmc bitwuzla): starting... (13 configurations pending)
SBY [demo] engine_3 (smtbmc --nopresat bitwuzla): starting... (12 configurations pending)
...
```

The engine log that would normally be printed is instead redirected to files named `engine_*_autotune.txt` within `sby`'s working directory.

To run an engine, further preparation steps may be necessary. These are cached and will be reused for every engine requiring them, while properly accounting for the required preparation time. Below is an example of the log output produced by such a preparation step. Note that this runs in parallel, so it may be interspersed with other log output.

```
SBY [demo] model 'smt2': preparing now...
SBY [demo] smt2: starting process "cd demo/model; yosys -ql design_smt2.log design_smt2.
↳ ys"
...
SBY [demo] smt2: finished (returncode=0)
SBY [demo] prepared model 'smt2'
```

Whenever an engine finishes, a log message is printed:

```
SBY [demo] engine_4 (smtbmc --unroll yices): succeeded (status=PASS)
SBY [demo] engine_4 (smtbmc --unroll yices): took 30 seconds (first engine to finish)
```

When an engine takes longer than the current hard timeout, it is stopped:


```
SBY [demo] engine_2 (smtbmc bitwuzla): timeout (150 seconds)
```

Depending on the configuration, autotune will also stop an engine earlier when reaching a soft timeout. If no other engine finishes in less time, the engine will be retried later with a longer soft timeout:

```
SBY [demo] engine_0 (smtbmc boolector): timeout (60 seconds, will be retried if ↵
↵necessary)
```

Finally, a summary of all finished engines is printed, sorted by their solving time:

```
SBY [demo] finished engines:
SBY [demo] #3: engine_1: smtbmc --nopresat boolector (52 seconds, status=PASS)
SBY [demo] #2: engine_5: smtbmc --nopresat --unroll yices (41 seconds, status=PASS)
SBY [demo] #1: engine_4: smtbmc --unroll yices (30 seconds, status=PASS)
SBY [demo] DONE (AUTOTUNED, rc=0)
```

If any tried engine encounters an error or produces an unexpected result, autotune will also output a list of failed engines. Note that when the sby file does not contain the `expect` option, autotune defaults to `expect pass,fail` to simplify running autotune on a verification task with a currently unknown outcome.

5.3 Configuring Autotune

Autotune can be configured by adding an `[autotune]` section to the `.sby` file. Each line in that section has the form `option_name value`, the possible options and their supported values are described below. In addition, the `--autotune-config` command line option can be used to specify a file containing further autotune options, using the same syntax. When both are used, the command line option takes precedence. This makes it easy to run autotune with existing `.sby` files without having to modify them.

5.4 Autotune Options

Auto-tune Option	Description
<code>timeout</code>	Set a different timeout value (in seconds) used only for autotune. The value <code>none</code> can be used to disable the timeout. Default: uses the non-autotune timeout option.
<code>soft_timeout</code>	Initial timeout value (in seconds) used to interrupt a candidate engine when other candidates are pending. Increased every time a candidate is retried to ensure progress. Default: 60
<code>wait</code>	Additional time to wait past the time taken by the fastest finished engine candidate so far. Can be an absolute time in seconds, a percentage of the fastest candidate or a sum of both. Default: 50%+10
<code>parallel</code>	Maximal number of engine candidates to try in parallel. When set to <code>auto</code> , the number of available CPUs is used. Default: <code>auto</code>
<code>presat</code>	Filter candidates by whether they perform a presat check. Values <code>on</code> , <code>off</code> , <code>any</code> . Default: <code>any</code>
<code>incr</code>	Filter candidates by whether they use incremental solving (when applicable). Values <code>on</code> , <code>off</code> , <code>any</code> , <code>auto</code> (see next option). Default: <code>auto</code>
<code>incr_threshold</code>	Number of timesteps required to only consider incremental solving when <code>incr</code> is set to <code>auto</code> . Default: 20
<code>mem</code>	Filter candidates by whether they have native support for memory. Values <code>on</code> , <code>any</code> , <code>auto</code> (see next option). Default: <code>auto</code>
<code>mem_threshold</code>	Number of memory bits required to only consider candidates with native memory support when <code>mem</code> is set to <code>auto</code> . Default: 10240
<code>forall</code>	Filter candidates by whether they support <code>\$allconst/\$allseq</code> . Values <code>on</code> , <code>any</code> , <code>auto</code> (on when <code>\$allconst/allseq</code> are found in the design). Default: <code>auto</code>

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. Refer to *SystemVerilog*, *VHDL*, *SVA* for more on the Verific front end.

6.1 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.

6.2 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>)`.

6.3 Liveness and Fairness

TBD

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

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

6.4 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.

6.5 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));
```

6.6 SystemVerilog Concurrent Assertions

TBD, see *Supported SVA Property Syntax*.

SYSTEMVERILOG, VHDL, SVA

Note

This document describes features only available with the commercial [Tabby CAD Suite](#) and the included Verific frontend. This is not the same as simply having a Verific license when using Yosys.

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.

7.1 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.

7.1.1 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.

7.1.2 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.

7.1.3 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:\$]*

7.1.4 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*

7.1.5 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)*

7.2 SVA properties in a VHDL design

The below code snippet, taken from an example SBY configuration included in [docs/examples/vhd](#), shows a VHDL design `updowncount.vhd` being loaded, followed by a SystemVerilog file `formal_bind.sv`.

Listing 1: `formal_bind.sby` script section

```
verific -vhdl updowncount.vhd
verific -sv formal_bind.sv
prep -top updowncount
```

Listing 2: `formal_bind.sv`

```
module formal_bind(input clk, rst, up, down, [3:0] count);

    initial assume(rst);

    assert property(@(posedge clk) count != 4'd15);
    cover property(@(posedge clk) count == 4'd10);
```

(continues on next page)

(continued from previous page)

```
endmodule  
  
bind updowncount formal_bind fb_inst(.*);
```

As you can see, the `formal_bind.sv` file includes a `formal_bind` module and makes use of the `bind` keyword in SystemVerilog to create an instance of this module connecting the inputs to the signals of the same name in the VHDL design. SVA properties can then be applied to those signals as if the whole design was in SystemVerilog.

APPLICATION NOTES

8.1 Formal Basics and Methods

- 109 Property Checking with SVA
- 120 Weak precondition cover and witness for SVA properties
- 130 Multi-Stage Verification

SYMBIYOSYS LICENSE

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

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

```
Copyright (C) 2016  Claire Xenia Wolf <claire@yosyshq.com>
```

```
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.