Formal Verification with Yosys-SMTBMC

Clifford Wolf

Yosys Flows

- Synthesis
  - iCE40 FPGAs
    (Project IceStorm)
  - Xilinx 7-Series FPGAs
    (Vivado for place&route)
  - ASIC Flows
    - Qflow
    - Coriolis2
    - Efabless.com Open Galaxy
  - Custom flows
    - From simple toy projects
      to PhD studies

- Formal Verification
  - Yosys-STMBMC
    - Bounded Model Checking
    - Using any SMT-LIB2 solver
      (using QF_AUFBV logic)
    - Supported solvers:
      Z3, CVC4, Yices, ...
  - Yosys built-ins
    - SAT solver
    - Equiv checking framework
  - Yosys + ABC
    - Synthesis + miter generation in Yosys,
      write BLIF, solve in ABC

Availability of various EDA tools for students, hobbyists, enthusiasts

- FPGA Synthesis
  - Free to use:
    - Xilinx Vivado WebPack, etc.
  - Free and Open Source:
    - Yosys + Project IceStorm
    - VTR (Odin II + VPR)

- HDL Simulation
  - Free to use:
    - Xilinx XSIM, etc.
  - Free and Open Source:
    - Icarus Verilog, Verilator, etc.

- Formal Verification
  - Free to use:
    - ???
  - Free and Open Source:
    - ??? *

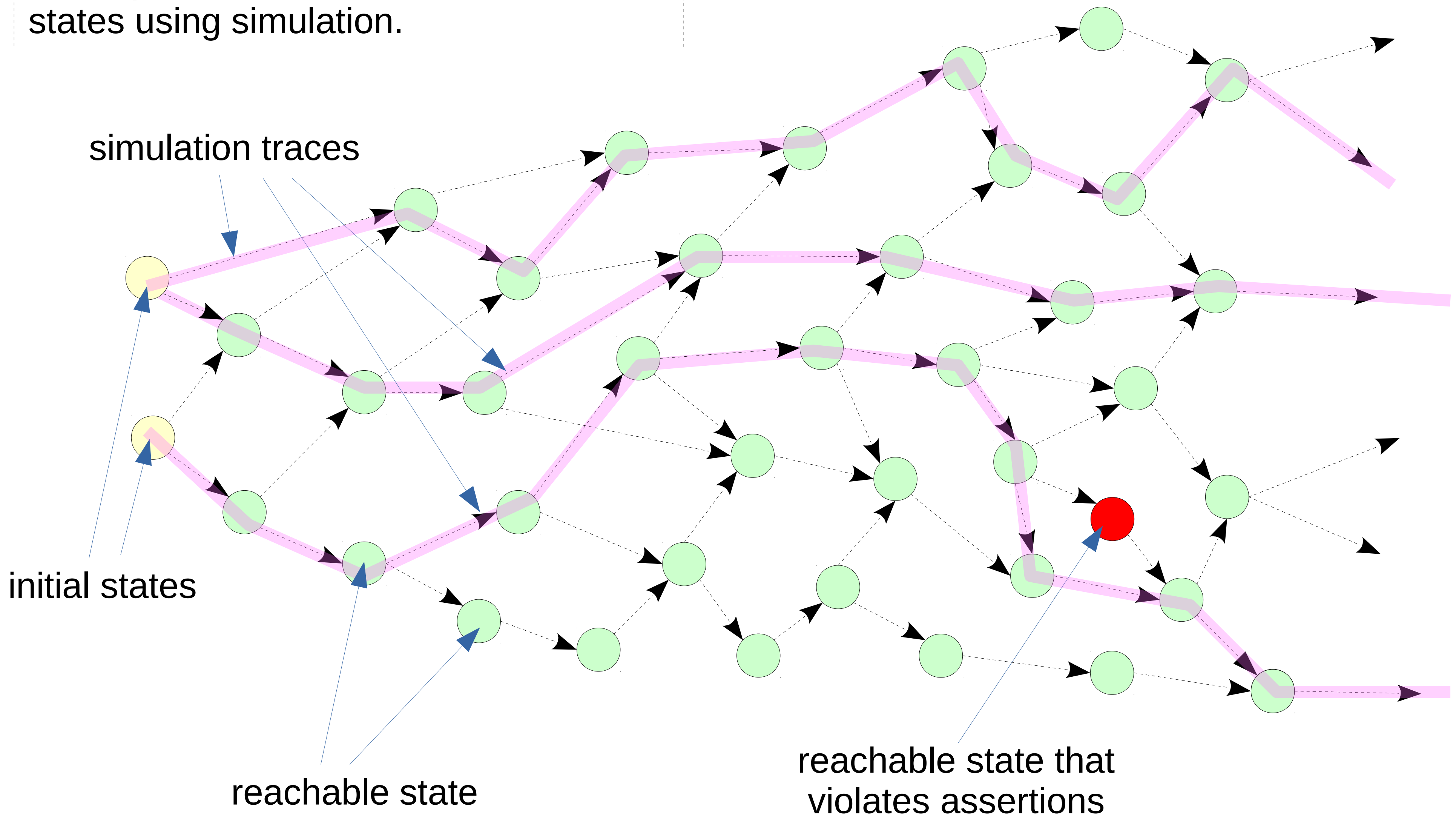
.. and people in the industry are complaining they can't find any verification experts to hire!

* There are a few contenders, but they lack complete Verilog front-ends, thus can't really be used with existing Verilog designs.

Simulation vs. Verification

Simulation checks only some of the reachable states. For non-trivial designs it is impossible to check all reachable states using simulation.

Verification uses symbolic methods to check all reachable states.



Why Formal Verification?

- Prove that a design is correct
  - Usually hard to achieve.
  - Typically only done for critical applications such as medical or aerospace.
  - Requires a full formal spec of correct behavior.

- Bughunting
  - Only requires partial specs. (The more the better of course.)
  - OK to replace one large proof with many smaller individual checks without proving that the smaller checks actually cover the original spec entirely.
  - Tends to find the most obscure bugs that would be really hard to find otherwise.
  - Finds bugs in a controlled environment.
  - Be the hunter, not the hunted!

- Optimization
  - No formal spec? Simply use the unoptimized version of the design as reference when working on optimizations.

Installing Yosys-SMTBMC (Ubuntu 16.04)

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

- Build and install Yosys (incl. Yosys-SMTBMC):
  - `git clone https://github.com/cliffordwolf/yosys`
  - `cd yosys; make`
  - `sudo make install`

- Build and install the Z3 SMT solver:
  - `git clone https://github.com/Z3Prover/z3`
  - `cd z3; python scripts/mk_make.py`
  - `cd build; make`
  - `sudo make install`

Yosys-SMTBMC Examples

- Slides and examples from this presentation:
  - http://www.clifford.at/papers/2016/yosys-smtbmc/

- Some simple examples are bundled with Yosys:
  - See `examples/smtbmc/` in Yosys source code

- See PicoRV32 for real-world example:
  - https://github.com/cliffordwolf/picorv32
  - Run “`make check`” to verify properties in `picorv32.v`
  - See `scripts/smtbmc/` for more advanced checks

Hello World

```
module hello(input clk, input rst,
output [3:0] cnt);
  reg [3:0] cnt = 0;

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

`ifdef FORMAL
  assume property (cnt != 10);
  assert property (cnt != 15);
`endif
endmodule
```

“make hello1” from examples.zip

```
yosys -ql hello1.yslog                          \
      -p 'read_verilog -formal hello1.v'        \
      -p 'prep -top hello -nordff'              \
      -p 'write_smt2 hello1.smt2'
yosys-smtbmc hello1.smt2
yosys-smtbmc -i hello1.smt2
```

Hello World

```
yosys-smtbmc hello1.smt2
##       0   0:00:00  Solver: z3
##       0   0:00:00  Checking asserts in step 0..
##       0   0:00:00  Checking asserts in step 1..
…
##       0   0:00:00  Checking asserts in step 18..
##       0   0:00:00  Checking asserts in step 19..
##       0   0:00:00  Status: PASSED
```

```
yosys-smtbmc -i hello1.smt2
##       0   0:00:00  Solver: z3
##       0   0:00:00  Trying induction in step 20..
##       0   0:00:00  Trying induction in step 19..
##       0   0:00:00  Trying induction in step 18..
##       0   0:00:00  Trying induction in step 17..
##       0   0:00:00  Trying induction in step 16..
##       0   0:00:00  Trying induction in step 15..
##       0   0:00:00  Temporal induction successful.
##       0   0:00:00  Status: PASSED
```

`assert(), assume(), restrict()`

- `assert(`*`expression`*`)`
  - Error if the expression evaluates to false
- `assume(`*`expression`*`)`
  - Simulation: Error if expression evaluates to false
  - Verification: Only consider traces where expression is true
- `restrict(`*`expression`*`)`
  - Simulation: Ignored.
  - Verification: Only consider traces where expression is true
- When to use `assume()`, when `restrict()`?
  - Use `assume()` if your asserts depend on it, use `restrict()` when it's just there to help with the proof, but the asserts would hold without it.

Immediate assertions, Concurrent assertions

- Immediate assertions: `assert()`, `assume()`, or `restrict()` within an `always` or `initial` block, with an expression as argument. This is fully supported by Yosys. For example:
  - initial assume (foo < bar);
  - always @* assert (2*foo > bar);
  - always @(posedge clk) if (foo < 10) restrict(bar > 10);

- Concurrent assertions: Asserting a SystemVerilog property in module context. So far Yosys only supports simple expression properties:
  - `assert property (`*`expression`*`);`

  is identical to:
  - `always @* assert(`*`expression`*`);`

Formal Test-Benches

- Often a “test-bench” is used for formal verification, similar to simulation.
- For simple cases, i.e. verification of `assert()` statements in a regular design, the test-bench is a simple wrapper for the module under test.
- Usually the test-bench contains a few additional `assume()` or `restrict()` statements that make sure the module under test is properly reset.
- In more complex setups, the “test-bench” can be an elaborate design in itself, constructing a sophisticated proof around one or multiple modules under test.
- See `scripts/smtbmc/` in the PicoRV32 github repository for such examples.

Hello Test-Bench

```
module hello(input clk, input rst, output reg [3:0] cnt);
  always @(posedge clk) begin
    if (rst)
      cnt <= 0;
    else
      cnt <= cnt + 1;
  end
endmodule
```

`hello2.v`, `hello2_tb.v` from `examples.zip`.

```
module hello_tb(input clk, input rst);
  wire [3:0] cnt;

  hello uut (
    .clk(clk),
    .rst(rst),
    .cnt(cnt)
  );

  initial begin
    assume(rst);
  end

  always @* begin
    if (!$initstate) begin
      assume(cnt != 10);
      assert(cnt != 15);
    end
  end
endmodule
```

Reset design

Asserts

Supported SMT Solvers

- In principle, every solver that supports the SMT-LIB2 language, model generation, and incremental solving is supported. (Solver with QF_AUFBV support is recommended.)
- Yosys-SMTBMC has been tested with the following solvers:
  - Z3, CVC4, Yices2, MathSAT5, Boolector
  - Source code is available for all those solvers, but not all of them are FOSS.
  - Different solvers may perform differently for different designs. Supporting more than one solver is key!

Solver	**License**	**PicoRV32 * `make check`**
Z3	permissive	94 seconds
CVC4	copyleft	$\approx 3.4 \cdot 10^{11}$ hours
Yices2	non-commercial	10 seconds
Boolector	non-commercial	65 seconds
MathSAT5	non-commercial	345 seconds

* CPU time on AMD FX-8150 at 3600 MHz, PicoRV32 git rev 7f946d0.
Software versions: Z3 4.4.0, CVC4 1.4, Yices 2.5.1, MathSAT 5.3.13, Boolector 2.2.0

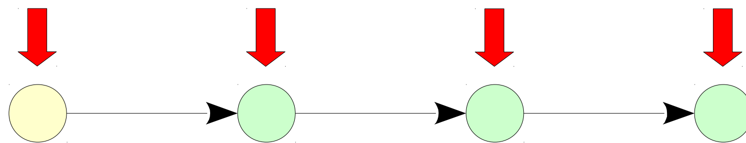
Bounded vs. Unbounded Methods

- Bounded methods only consider states reachable within *N* time steps from initial states.
- Unbounded methods consider all reachable states, regardless of the number of time steps required to reach them from the initial states.

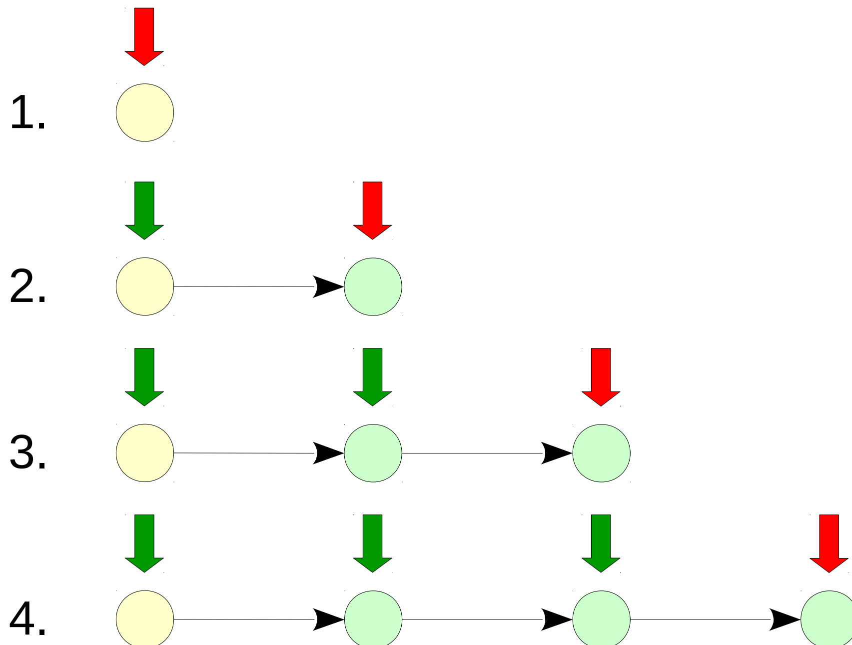
- BMC (bounded model check) is a bounded method.
- Temporal Induction can be used as a simple method for performing unbounded proofs with a bounded solver.

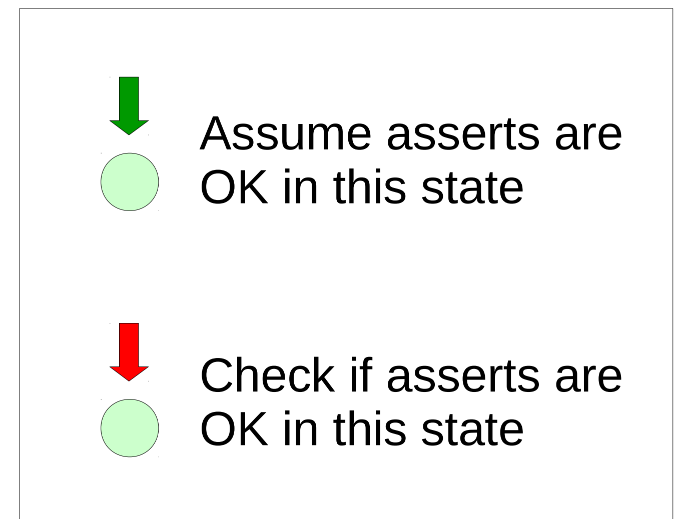
Bounded Model Check (BMC)

The naïve (one-shot) way of performing a BMC (for N = 3 time steps, i.e. 4 states):



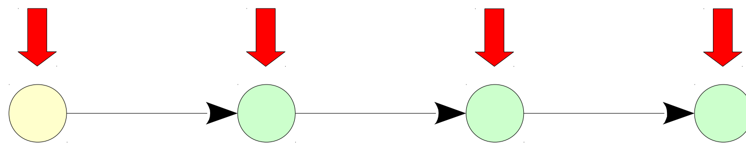
The smart (incremental) way of performing the same BMC:



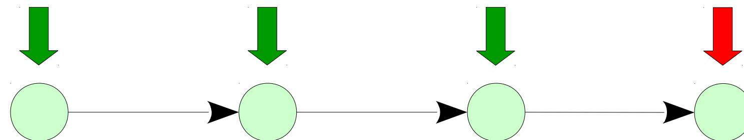


Temporal Induction

Assuming we have proven our asserts to hold for the first three non-init time steps:



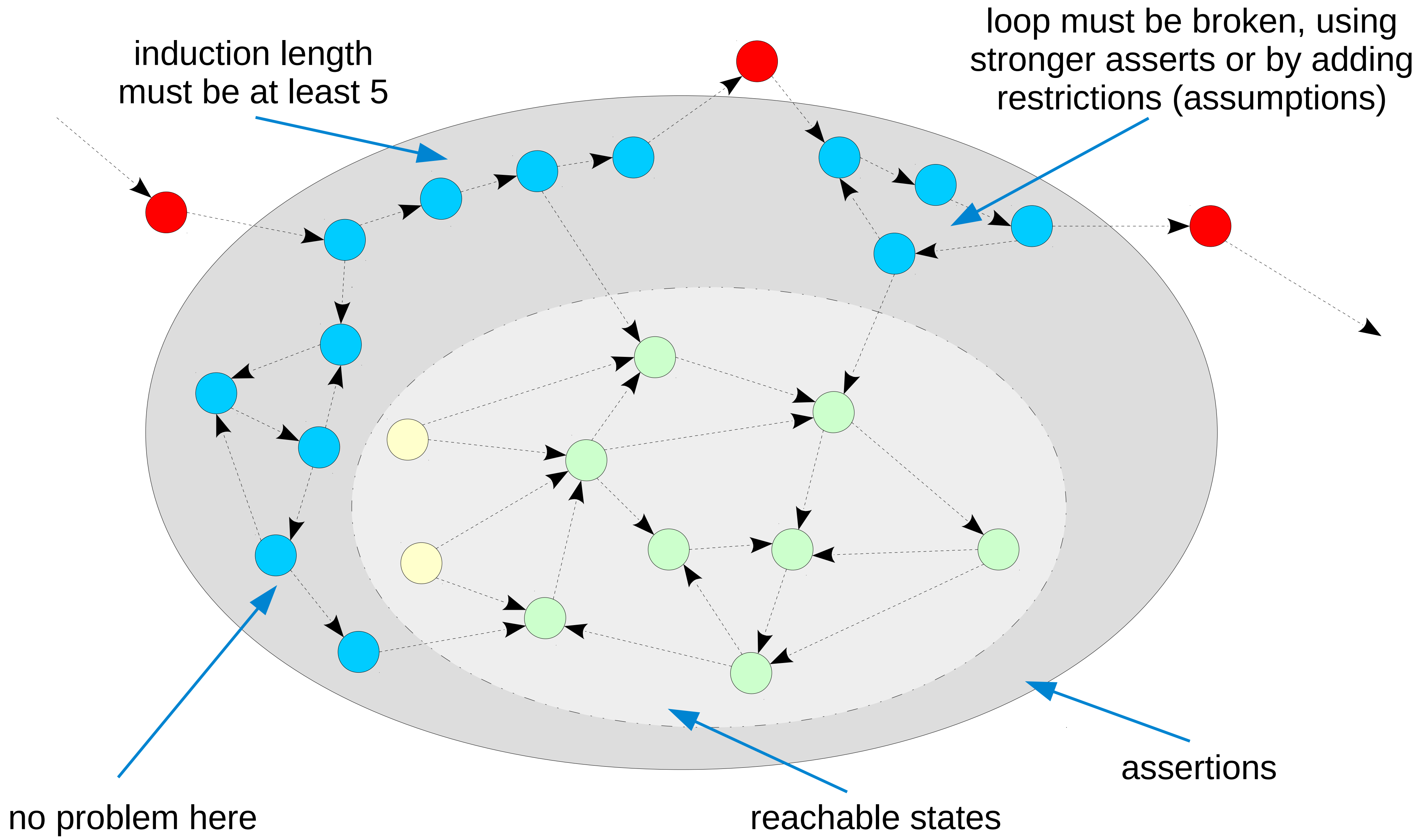
Then this will prove our asserts for all reachable time steps:



If this proof succeeds we are done. We have proven our properties to hold in all reachable states.

However, if this proof fails it does not automatically imply that there is a problem with the design. For example, the induction length could be too short, or the design might not be provable by temporal induction at all.

Temporal Induction



SAT Solving and SMT Solving

- Satisfiability (SAT) Solvers
  - Input: system of boolean formulas
  - Output: SAT/UNSAT, Witness (aka “model” for SAT)

- SAT Modulo Theory (SMT) Solvers
  - Combine SAT solving with higher-level theories, such as:
    - A: Arrays with extensionality
    - UF: Uninterpreted functions
    - BV: Bit vectors with arithmetic
    - I: Unbound integer arithmetic, …
  - Usually significantly faster than plain SAT solvers when higher-level theories are used to preserve the problem structure. (e.g. using `(bvadd)` instead of system of boolean formulas for adder circuit)

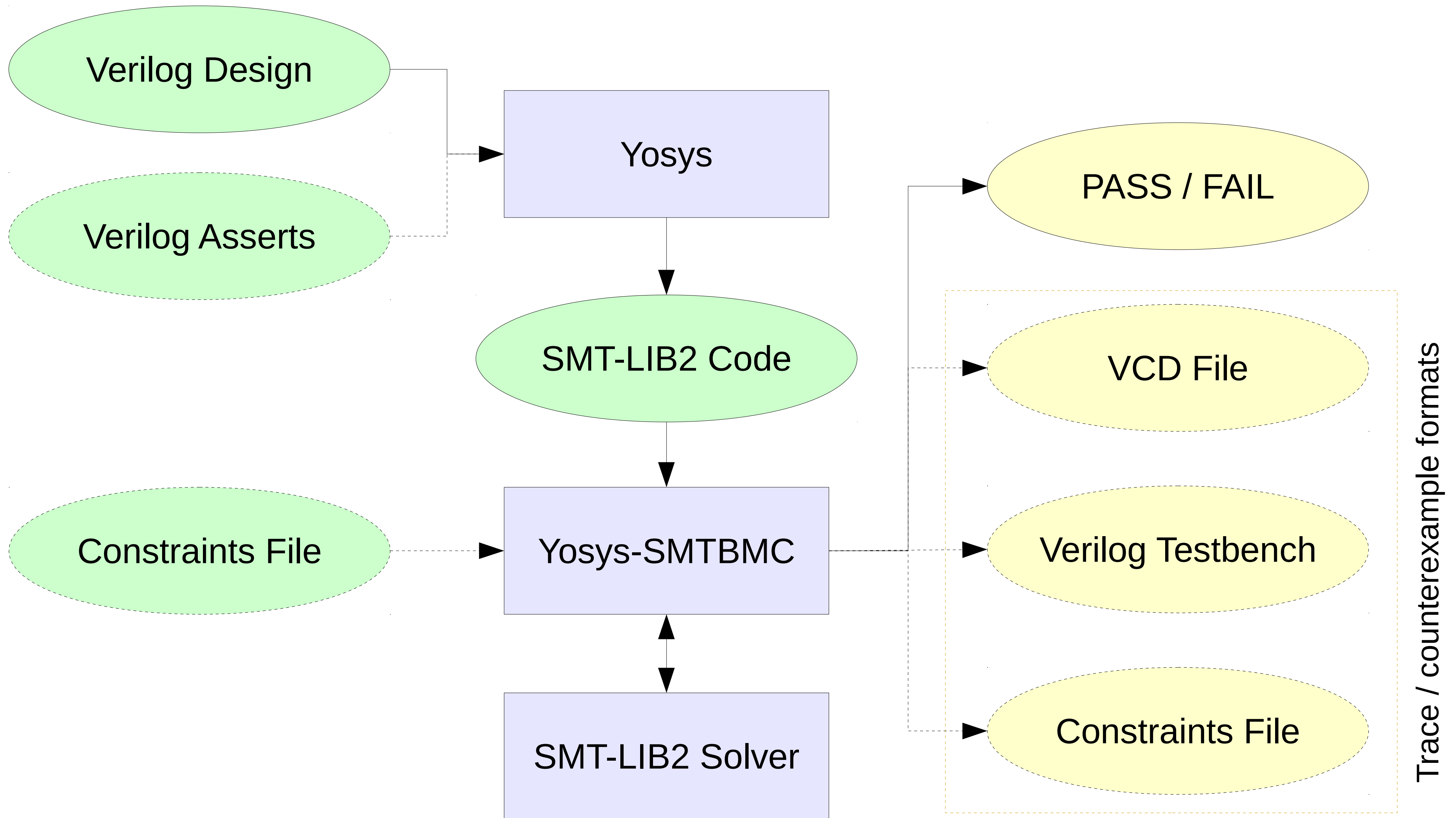
- SMT-LIB2 File Format
  - There is an annual competition for SMT solvers called SMT-COMP
  - All benchmarks used in SMT-COMP are in the SMT-LIB2 file format
  - So practically all SMT solvers support the SMT-LIB2 file format
  - Yosys-SMTBMC can use any SMT solver with support for the SMT-LIB2 QF_AUFBV logic

Preserving Problem Structure

- Preserving problem structure helps SMT solvers to excel at their task.
- Yosys-SMTBMC preserves problem structure using the following techniques (using the SMT-LIB2 QF_AUFBV theory):
  - No synthesis to gate level logic (instead use SMT-LIB2 Bit vectors)
    - Preserves arithmetic and word-wide operations
  - No synthesis of memories to circuits of FFs and MUXes (instead use Arrays)
    - Vastly reduces the model size for arrays with more elements than time steps in the BMC
  - No flattening of design hierarchy (instead use uninterpreted functions)*
    - Preserves structure when a module is instantiated many times
  - No unrolling of time steps (instead use uninterpreted functions)*
    - Preserves structure in deep bounded model checks
- Ultimately the SMT solver will have to perform this transformations in one way or another. But for many reasons it is better to let the solver do them.

* Use “yosys-smtbmc –unroll” to create SMT2 code that is already unrolled, e.g. for solvers without UF theory support. In some cases UF-capable solvers can be faster on pre-unrolled problems. See also benchmark results at the end of this slide set.

Yosys-SMTBMC Flow



Typical Workflow

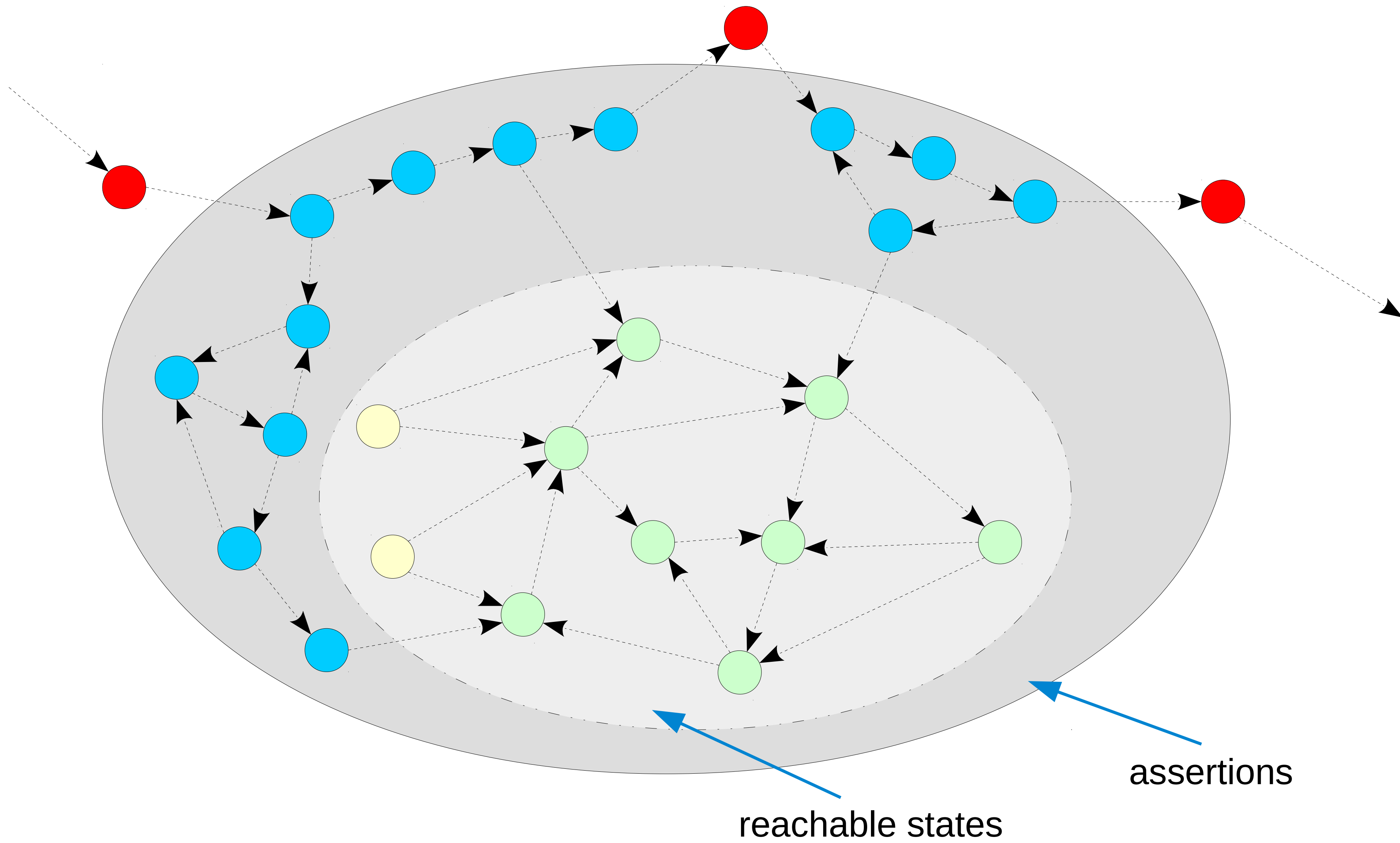
Step 1: Run Bounded Check

- FAIL → Fix design, add assumptions, or loosen asserts
- PASS → So far so good. Proceed to step 2

Step 2: Run Induction Proof

- FAIL → Investigate counterexample: Is it reachable? *
  - REACHABLE → Fix design, add assumptions, or loosen asserts
  - UNREACHABLE → Add restrictions, strengthen asserts, or increase induction length
- PASS → Do you want more asserts in your design?
  - YES → Reduce induction length or remove restrictions.
  - NO → You are done.

* Counterexample is always unreachable when induction succeeds with a larger induction length.


assertions
reachable states

demo1.v

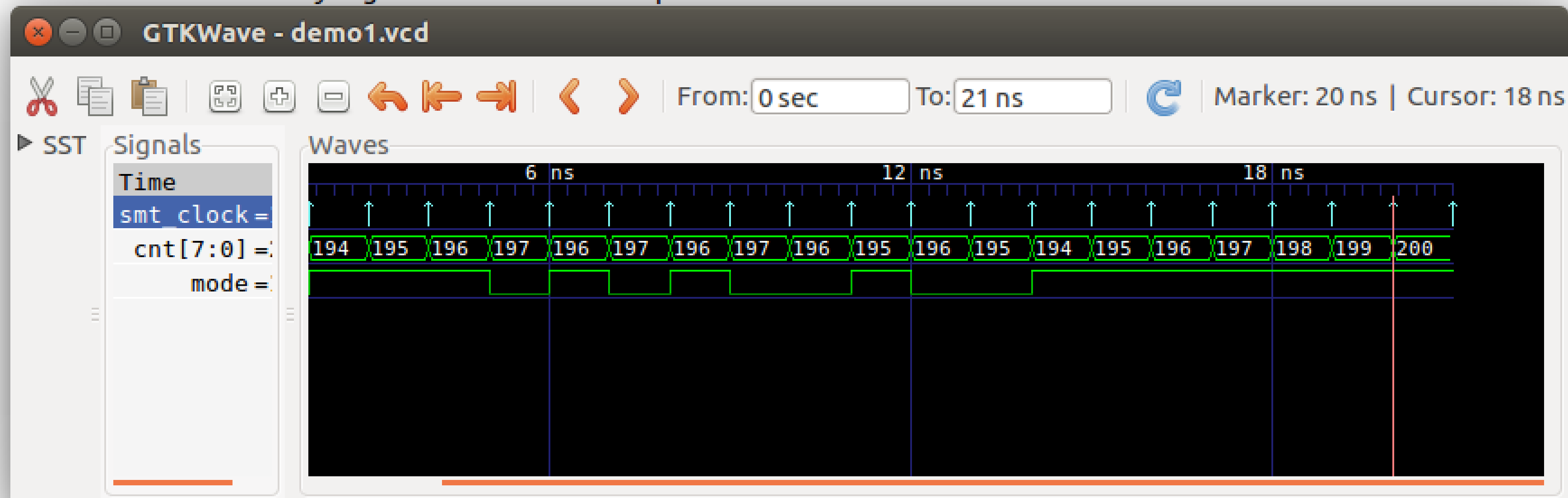
```
module demo(input clk, mode, output reg [7:0] cnt);
  always @(posedge clk) begin
    if (cnt != 0 && mode == 0)
      cnt <= cnt - 1;

    if (cnt != 99 && mode == 1)
      cnt <= cnt + 1;
  end

  initial assume (cnt == 5);
  assert property (cnt != 200);
endmodule
```

demo1.v

```
0   0:00:00  Solver: z3
0   0:00:00  Trying induction in step 20..
0   0:00:00  Trying induction in step 19..
```




```
0   0:00:00  Trying induction in step 2..
0   0:00:00  Trying induction in step 1..
0   0:00:00  Trying induction in step 0..
0   0:00:00  Temporal induction failed!
0   0:00:00  Assert failed in demo: demo1.v:11
0   0:00:00  Writing trace to VCD file: demo1.vcd
0   0:00:00  Status: FAILED (!)
```

demo2.v

```
module demo(input clk, mode, output reg [7:0] cnt);
  always @(posedge clk) begin
    if (cnt != 0 && mode == 0)
      cnt <= cnt - 1;

    if (cnt != 99 && mode == 1)
      cnt <= cnt + 1;
  end

  initial assume (cnt == 5);
  assert property (cnt != 200);
  assert property (cnt < 100);
endmodule
```

demo2.v

```
$ make demo2
yosys-smtbmc demo2.smt2
##       0   0:00:00  Solver: z3
##       0   0:00:00  Checking asserts in step 0..
##       0   0:00:00  Checking asserts in step 1..
##       0   0:00:00  Checking asserts in step 2..
…
##       0   0:00:00  Checking asserts in step 17..
##       0   0:00:00  Checking asserts in step 18..
##       0   0:00:00  Checking asserts in step 19..
##       0   0:00:00  Status: PASSED

yosys-smtbmc -i demo2.smt2
##       0   0:00:00  Solver: z3
##       0   0:00:00  Trying induction in step 20..
##       0   0:00:00  Trying induction in step 19..
##       0   0:00:00  Temporal induction successful.
##       0   0:00:00  Status: PASSED
```

parcase.v

```
module parcase (input clk, A, B, C, D, E, BUG, output reg Y);
    always @(posedge clk) begin
        Y <= 0;
        if (A != B || BUG) begin
            (* parallel_case *)
            case (C)
                A: Y <= D;
                B: Y <= E;
            endcase
        end
    end
endmodule
```

```
$ yosys -ql parcase.yslog                        \
    -p 'read_verilog -formal parcase.v' \
    -p 'prep -top parcase -nordff'         \
    -p 'assertpmux; opt -keepdc -fast'   \
    -p 'write_smt2 parcase.smt2'

$ yosys-smtbmc parcase.smt2
##   0:00:00  Solver: z3
##   0:00:00  Checking asserts in step 0..
##   0:00:00  BMC failed!
##   0:00:00  Assert failed in parcase: parcase.v:6
##   0:00:00  Status: FAILED (!)
```

memcmp.v

```
module memory1 (
    input clk,
    input [3:0] wstrb,
    input [15:0] waddr,
    input [15:0] raddr,
    input [31:0] wdata,
    output [31:0] rdata
);
    reg [31:0] mem [0:2**16-1];
    reg [15:0] buffered_raddr;

    // "transparent" read
    assign rdata = mem[buffered_raddr];

    always @(posedge clk) begin
        if (wstrb[3]) mem[waddr][31:24] <= wdata[31:24];
        if (wstrb[2]) mem[waddr][23:16] <= wdata[23:16];
        if (wstrb[1]) mem[waddr][15: 8] <= wdata[15: 8];
        if (wstrb[0]) mem[waddr][ 7: 0] <= wdata[ 7: 0];
        buffered_raddr <= raddr;
    end
endmodule
```

memcmp.v

```
module memory2 (
     input clk,
     input [3:0] wstrb,
     input [15:0] waddr,
     input [15:0] raddr,
     input [31:0] wdata,
     output [31:0] rdata
);
     reg [31:0] mem [0:2**16-1];
     reg [31:0] buffered_wdata;
     reg [31:0] buffered_rdata;
     reg [3:0] buffered_wstrb;
     reg waddr_is_not_raddr;

     wire [31:0] expanded_wstrb = {{8{wstrb[3]}}, {8{wstrb[2]}}, {8{wstrb[1]}}, {8{wstrb[0]}}};
     wire [31:0] expanded_buffered_wstrb = {{8{buffered_wstrb[3]}}, {8{buffered_wstrb[2]}},
                                            {8{buffered_wstrb[1]}}, {8{buffered_wstrb[0]}}};

     assign rdata = waddr_is_not_raddr ? buffered_rdata :
              (buffered_wdata & expanded_buffered_wstrb) |
              (buffered_rdata & ~expanded_buffered_wstrb);

     always @(posedge clk) begin
          mem[waddr] <= (wdata & expanded_wstrb) | (mem[waddr] & ~expanded_wstrb);
          buffered_wstrb <= wstrb;
          buffered_wdata <= wdata;
          buffered_rdata <= mem[raddr];
          waddr_is_not_raddr <= waddr != raddr;
     end
endmodule
```

memcmp.v

```
module memcmp (
    input clk,
    input [3:0] wstrb,
    input [15:0] waddr,
    input [15:0] raddr,
    input [31:0] wdata,
    output [31:0] rdata1,
    output [31:0] rdata2
);
    memory1 mem1 (
        .clk  (clk   ), .wstrb(wstrb ),
        .waddr(waddr ), .raddr(raddr ),
        .wdata(wdata ), .rdata(rdata1)
    );

    memory2 mem2 (
        .clk  (clk   ), .wstrb(wstrb ),
        .waddr(waddr ), .raddr(raddr ),
        .wdata(wdata ), .rdata(rdata2)
    );
endmodule
```

memcmp.smtc

```
initial
assume (= [mem1.mem] [mem2.mem])

always -1
assert (= [mem1.mem] [mem2.mem])
assert (= [rdata1] [rdata2])
```

```
$ yosys-smtbmc -t 5 --smtc memcmp.smtc memcmp.smt2
##       0   0:00:00  Solver: z3
##       0   0:00:00  Checking asserts in step 0..
##       0   0:00:00  Checking asserts in step 1..
##       0   0:00:00  Checking asserts in step 2..
##       0   0:00:00  Checking asserts in step 3..
##       0   0:00:00  Checking asserts in step 4..
##       0   0:00:00  Status: PASSED

$ yosys-smtbmc -t 5 --smtc memcmp.smtc -i memcmp.smt2
##       0   0:00:00  Solver: z3
##       0   0:00:00  Trying induction in step 5..
##       0   0:00:00  Trying induction in step 4..
##       0   0:00:00  Temporal induction successful.
##       0   0:00:00  Status: PASSED
```

PicoRV32 AXI Check

- The AXIS (AXI Stream) Interface:
  - `axis_ready` (1 bit, slave → master)
  - `axis_valid` (1 bit, master → slave)
  - `axis_data` (N bits, master → slave)
  - Data is transferred when `axis_valid` and `axis_ready` are both active in the same cycle.

- The example checks an AXI4-Lite interface (`picorv32_axi`), which consists of 5 individual AXI Streams.

- Properties to check:
  - Easy (`axicheck.v` *):
    - `axis_valid` must only be deasserted directly after a successful transfer
    - `axis_ready` must only be deasserted directly after a successful transfer
    - `axis_data` must be stable while `axis_valid` is asserted
  - Hard (`axicheck2.v` *):
    - `axis_valid` being asserted must not depend on `axis_ready` also being asserted

* https://github.com/cliffordwolf/picorv32/tree/master/scripts/smtbmc/

axicheck.v

```
// only deassert “valid” after successful transfer
if ($fell(mem_axi_awvalid)) begin
        assert($past(mem_axi_awready));
end

// only deassert “ready” after successful transfer
if ($fell(mem_axi_awready)) begin
        assume($past(mem_axi_awvalid));
end

// data must be stable while “valid” is active
if ($past(mem_axi_awvalid && !mem_axi_awready)) begin
        assert($stable(mem_axi_awaddr));
        assert($stable(mem_axi_awprot));
end
```

5x for the 5 AXI Streams, swapping assume() and assert() for slave interfaces

+ some additional checks to make sure the traffic on the 5 streams is consistent.
E.g. no read results are received without a read request first being sent.

axicheck2.v

Instantiate two copies of the UUT, let them have independent “ready” signals, but force them to be equal when “valid” is active:

```
if (mem_axi_awvalid_0)
    assume(mem_axi_awready_0 == mem_axi_awready_1);
```

This must not have any effect on the “valid” and “data” signals:

```
assert(mem_axi_awvalid_0 == mem_axi_awvalid_1);
assert(mem_axi_awaddr_0  == mem_axi_awaddr_1 );
assert(mem_axi_awprot_0  == mem_axi_awprot_1 );
```

Instead of resetting both UUT in the initial state, we just force them to be equal in the initial state (see `axicheck2.smtc`):

```
initial
assume (= [uut_0] [uut_1])
```

(An SMT solver with true UF support is needed in order to compare hierarchical cells like this.)

Bounded Liveness Properties

- Safety properties are asserting that certain states are unreachable from a set of initial states.
  - This are the properties we have been looking at so far.

- Liveness properties are asserting that certain state inevitable are going to be reached from a set of initial states.
  - Checking liveness is hard.
  - Usually this is used to prove that the circuit will always make progress, i.e. that the circuit cannot get stuck somehow.
  - In many cases bounded liveness properties are actually more useful: Assert that the target state is going to be reached within a given timeout:

```
integer timeout = 0;

always @(posedge clk)
    if (axis_valid && !axis_ready)
        timeout <= timeout + 1;
    else
        timeout <= 0;

assert property (timeout < 1000);
```

  - This is a regular safety property that can be proven using BMC and induction. Simply start with something like the above and then apply the regular work-flow for safety properties. (see `liveness.v` example)

Bounded Liveness Properties

```
module liveness(input clk, input axis_valid, output axis_ready);
    reg [9:0] count1 = 0, count2 = 0;

    ...

    integer timeout = 0;

    always @(posedge clk)
        if (axis_valid && !axis_ready)
            timeout <= timeout + 1;
        else
            timeout <= 0;

    assert property (timeout < 1000);

    always @* begin
        if (!state2)
            assert(count2 == 0);
        assert(3*timeout - 3*count2 == count1);
        assert(count1 <= maxcnt+6);
        assert(2*count2 <= count1+4);
    end
endmodule
```

multiclk.v

```
module multiclk(input clk, output [3:0] counter_a, counter_b);
    reg [3:0] counter_a = 0;
    reg [3:0] counter_b = 0;

    always @(posedge clk)
        counter_a <= counter_a + 1;

    always @(posedge clk)
        counter_b[0] <= !counter_b[0];

    always @(negedge counter_b[0])
        counter_b[1] <= !counter_b[1];

    always @(negedge counter_b[1])
        counter_b[2] <= !counter_b[2];

    always @(negedge counter_b[2])
        counter_b[3] <= !counter_b[3];

    assert property (counter_a == counter_b);
endmodule
```

multiclk.v

```
yosys -ql multiclk.yslog \
    -p 'read_verilog -formal multiclk.v' \
    -p 'prep -top multiclk -nordff' \
    -p 'clk2fflogic; opt_clean' \
    -p 'write_smt2 multiclk.smt2'

yosys-smtbmc -s boolector multiclk.smt2
##       0   0:00:00  Solver: boolector
##       0   0:00:00  Checking asserts in step 0..
...
##       0   0:00:00  Checking asserts in step 19..
##       0   0:00:00  Status: PASSED

yosys-smtbmc -i -s boolector multiclk.smt2
##       0   0:00:00  Solver: boolector
##       0   0:00:00  Trying induction in step 20..
##       0   0:00:00  Trying induction in step 19..
##       0   0:00:00  Temporal induction successful.
##       0   0:00:00  Status: PASSED
```

setreset.v

```
module setreset(input clk, input set, rst, d, output q1, q2);
    reg q1 = 0;

    always @(posedge clk, posedge set, posedge rst)
        if (rst)      q1 <= 0;
        else if (set) q1 <= 1;
        else          q1 <= d;

    reg q2_s = 0, q2_r = 0, q2_l;
    wire q2 = q2_l ? q2_s : q2_r;

    always @(posedge clk, posedge set)
        if (set) q2_s <= 1;
        else     q2_s <= d;

    always @(posedge clk, posedge rst)
        if (rst) q2_r <= 0;
        else     q2_r <= d;

    always @* begin
        if (rst)      q2_l <= 0;
        else if (set) q2_l <= 1;

    assert property (q1 == q2);
endmodule
```

setreset.v

```
yosys -ql setreset.yslog \
    -p 'read_verilog -formal setreset.v' \
    -p 'prep -top setreset -nordff' \
    -p 'clk2fflogic; opt_clean' \
    -p 'write_smt2 setreset.smt2'
Warning: Complex async reset for dff `\q1'.

yosys-smtbmc -s boolector setreset.smt2
##       0   0:00:00  Solver: boolector
##       0   0:00:00  Checking asserts in step 0..
...
##       0   0:00:00  Checking asserts in step 19..
##       0   0:00:00  Status: PASSED

yosys-smtbmc -i -s boolector setreset.smt2
##       0   0:00:00  Solver: boolector
##       0   0:00:00  Trying induction in step 20..
##       0   0:00:00  Trying induction in step 19..
##       0   0:00:00  Temporal induction successful.
##       0   0:00:00  Status: PASSED
```

abccex.v

```
module abccex(input clk, input [7:0] din);
  reg [127:0] djb2hash = 42;
  reg [3:0] magic; // uninitialized

  always @(posedge clk) begin
    djb2hash <= ((djb2hash << 5) + djb2hash) ^ din;
    magic <= magic + 1;
  end

  function isalpha(input [7:0] ch); begin
   isalpha = 0;
   if ("0" <= din && din <= "9") isalpha = 1;
   if ("A" <= din && din <= "Z") isalpha = 1;
   if ("a" <= din && din <= "z") isalpha = 1;
  end endfunction

  assume property (isalpha(din));
  assert property (djb2hash != 0 || magic != 0);
endmodule
```

abccex.v

Solving an optimized miter circuit with ABC, then creating a counterexample VCD file from ABC .cex file with Yosys-SMTBMC:

```
yosys -ql abccex.yslog \
     -p 'read_verilog -formal abccex.v' \
     -p 'prep -top abccex -nordff' \
     -p 'write_smt2 abccex.smt2' \
     -p 'miter -assert -flatten abccex' \
     -p 'techmap; opt -fast; abc; clean' \
     -p 'write_blif abccex.blif'

yosys-abc -c 'read_blif abccex.blif; strash; logic; undc; strash; zero;
              bmc3 -F 100 -v; undc -c; write_cex -n abccex.cex'
...
Params: FramesMax = 100. Start = 0. ConfLimit = 0. TimeOut = 0. SolveAll = 0.
   0 + : Var =    1. Cla =     0. Conf =      0. Uni =  0. 0 MB  4 MB  0.02 sec
...
  26 + : Var = 9929. Cla = 40559. Conf =  61754. Uni = 23. 0 MB  9 MB  3.67 sec
Runtime:  CNF = 0.0 sec (0.3 %)  UNSAT = 0.8 sec (21.1 %)  SAT = 2.9 sec (77.9 %)  UNDEC = 0.0 sec (0.0 %)
LStart(P) = 10000  LDelta(Q) = 2000  LRatio(R) = 80  ReduceDB = 10  Vars = 9929  Used = 9929 (100.00 %)
Buffs = 306. Dups = 0.   Hash hits = 0.  Hash misses = 9604.  UniProps = 0.
Output 0 of miter "abccex" was asserted in frame 26. Time =     3.67 sec

! yosys-smtbmc -t 100 -s z3 --cex abccex.cex --dump-vcd abccex.vcd abccex.smt2
##      0   0:00:00  Solver: z3
##      0   0:00:00  Checking asserts in step 0..
...
##      0   0:00:00  Checking asserts in step 26..
##      0   0:00:00  BMC failed!
##      0   0:00:00  Assert failed in abccex: abccex.v:18
##      0   0:00:00  Writing trace to VCD file: abccex.vcd
##      0   0:00:00  Status: FAILED (!)
```

Benchmark

- See `benchmark.zip` and `benchmark_log.zip`

- `benchmark_picorv32_check.v`
  - Based on PicoRV32 “make check”
  - BMC for 20 cycles, expected to pass

- `benchmark_picorv32_comp.v`
  - Compares two PicoRV32 cores in different configurations (based on PicoRV32 “tracecmp2” test)
  - Added bug in the core with the two-stage shifter implementation, expected to fail after 16 cycles
  - Proof limited to ADDI and SLLI instructions

- `benchmark_picorv32_hard.v`
  - Same test as `benchmark_picorv32_comp.v`, but with all instructions allowed

Both benchmarks use a slightly modified version of PicoRV32 and only use simple immediate assertions. Every formal Verilog verification tool should be able to process this benchmark files as they are.

Modifications to PicoRV32: Removed parallel_case, full_case, removed explicit don’t cares, removed optional MUL/DIV co-processor support.

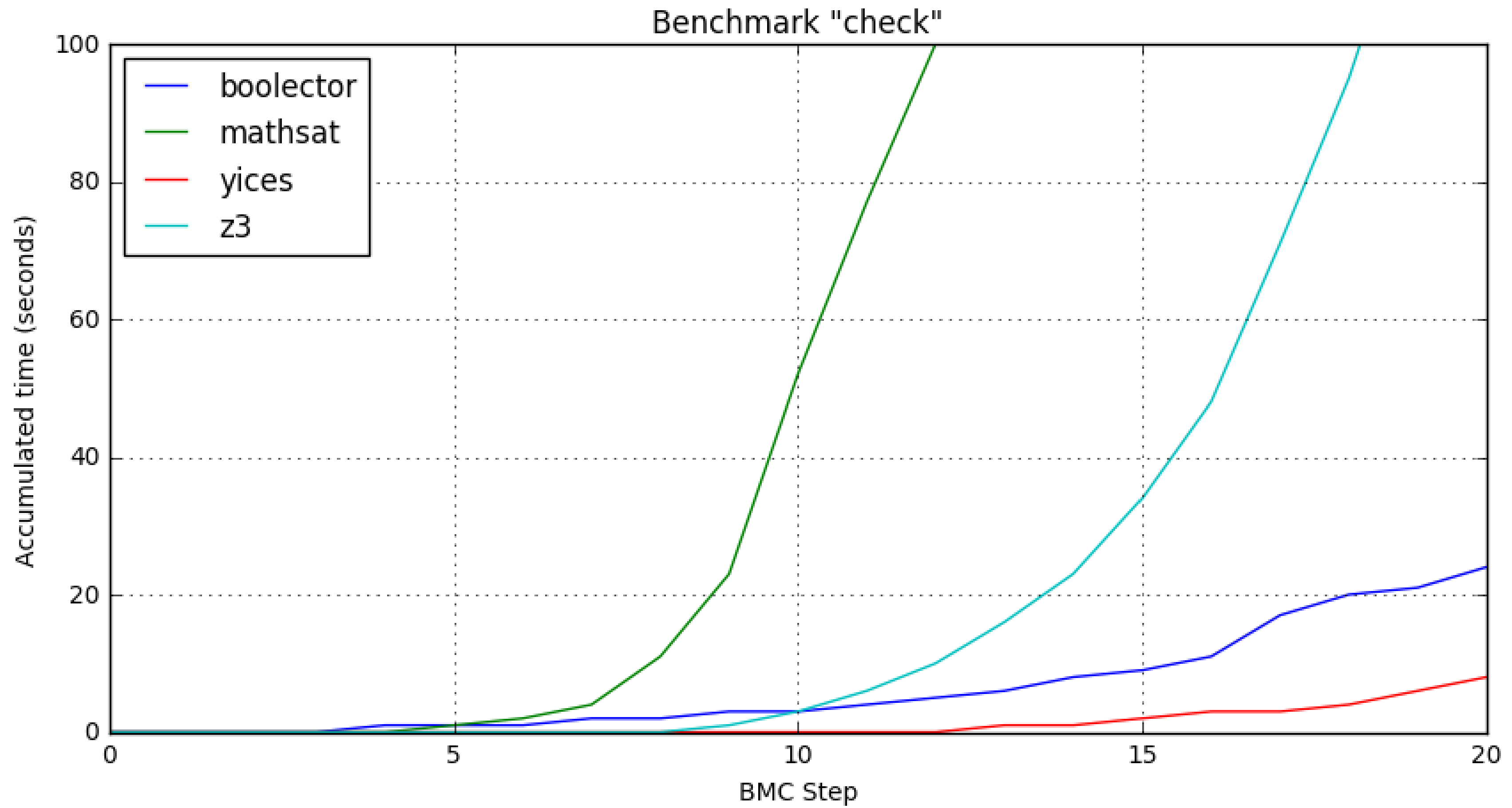
Benchmark

	Benchmark "PicoRV32 check"	Benchmark "PicoRV32 comp"	Benchmark "PicoRV32 hard"
Yosys Synthesis *	+ 2 seconds	+ 3 seconds	+ 3 seconds
Yosys SAT	44 seconds	143 seconds	> 2 hours
z3	157 seconds	654 seconds	> 2 hours
Yices2	8 seconds	282 seconds	5451 seconds
Boolector	25 seconds	16 seconds	629 seconds
MathSAT	730 seconds	> 2 hours	> 2 hours
Yosys Synthesis **	+ 15 seconds	+ 25 seconds	+ 25 seconds
Yosys SAT	61 seconds	250 seconds	> 2 hours
ABC BMC	3 seconds	20 seconds	> 2 hours

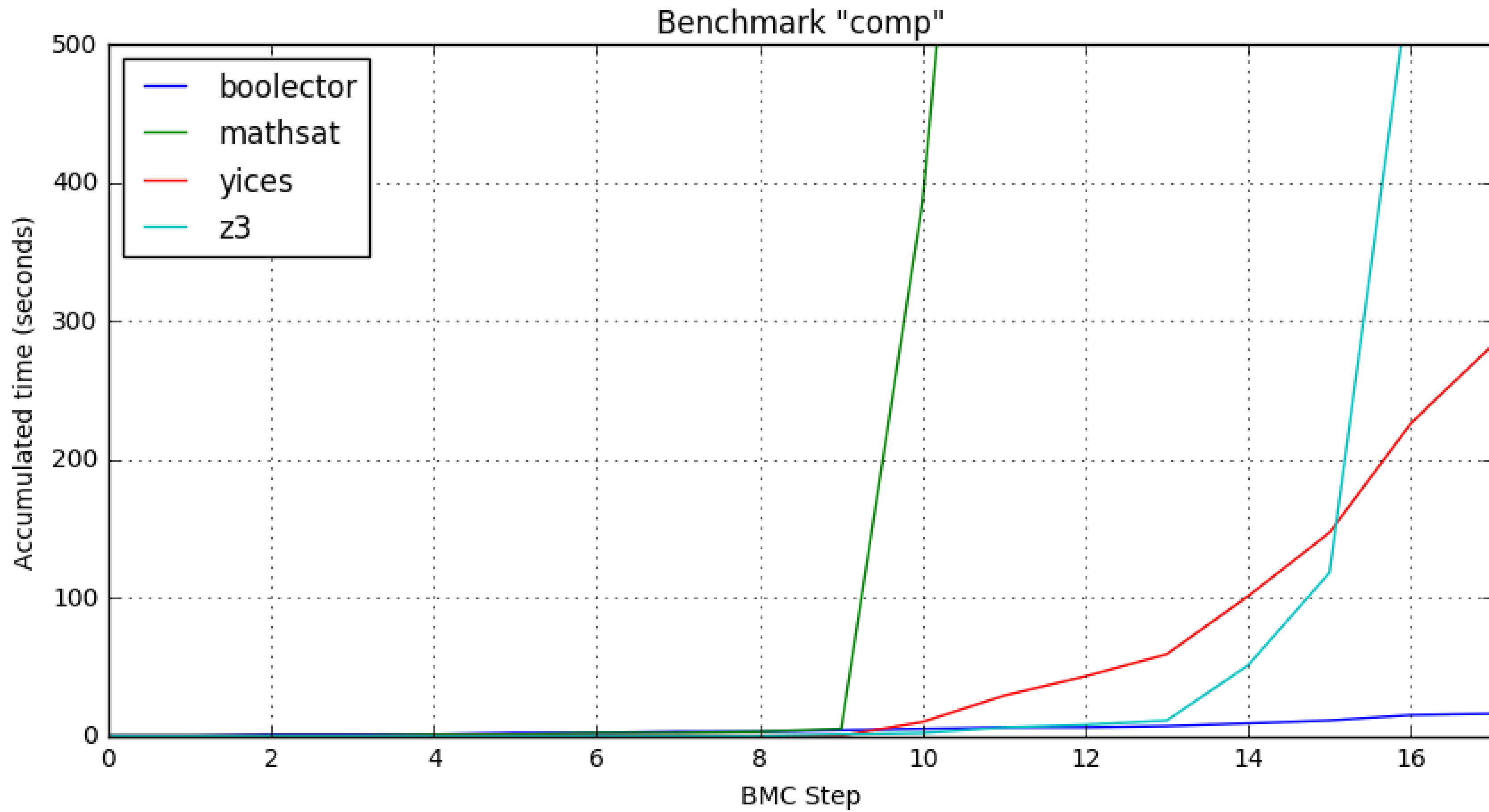
* Verilog to smt2 file with 'prep' and 'write_smt2'

** Verilog to optimized gate-level miter netlist

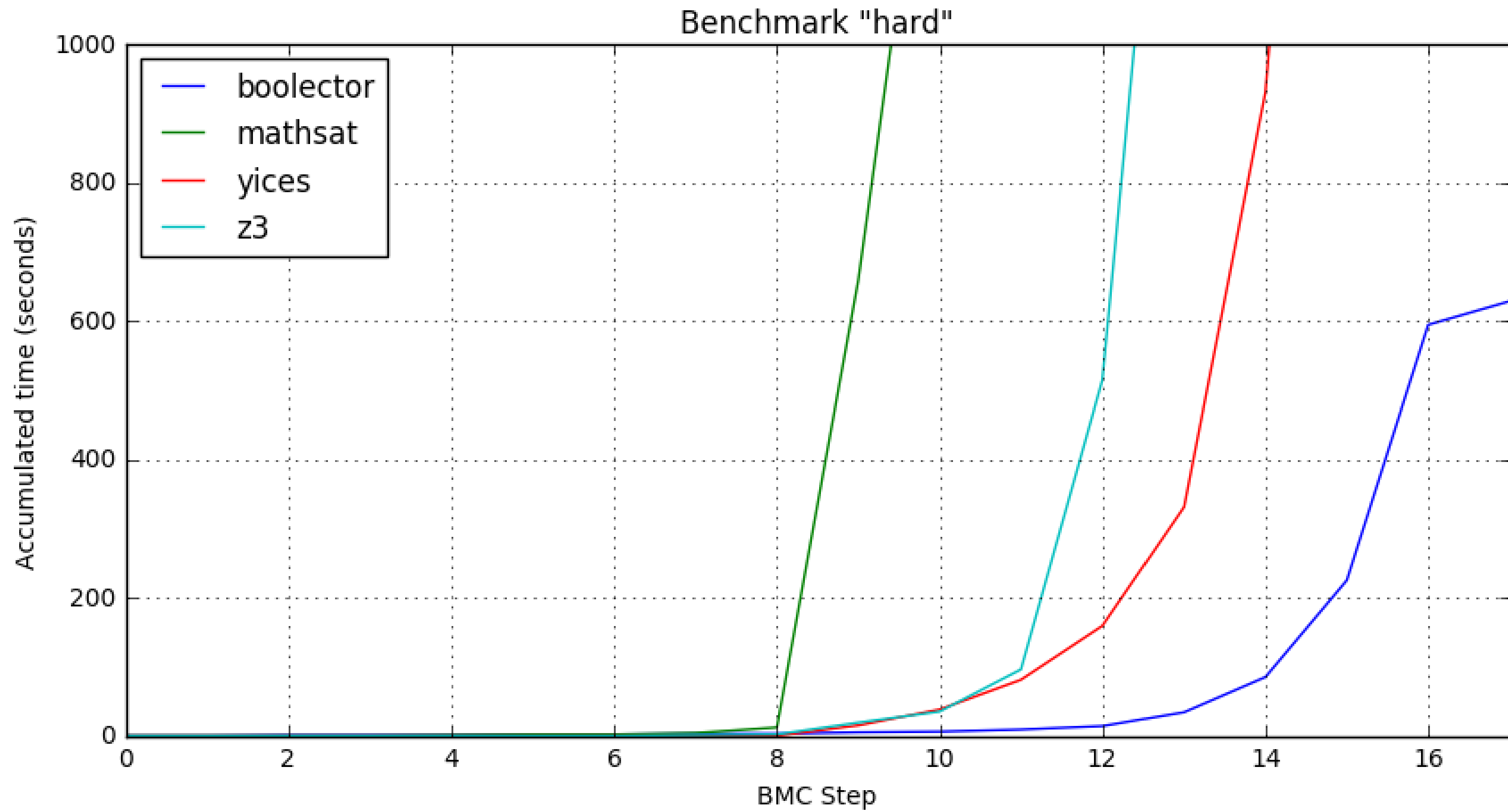
Benchmark



Benchmark



Benchmark



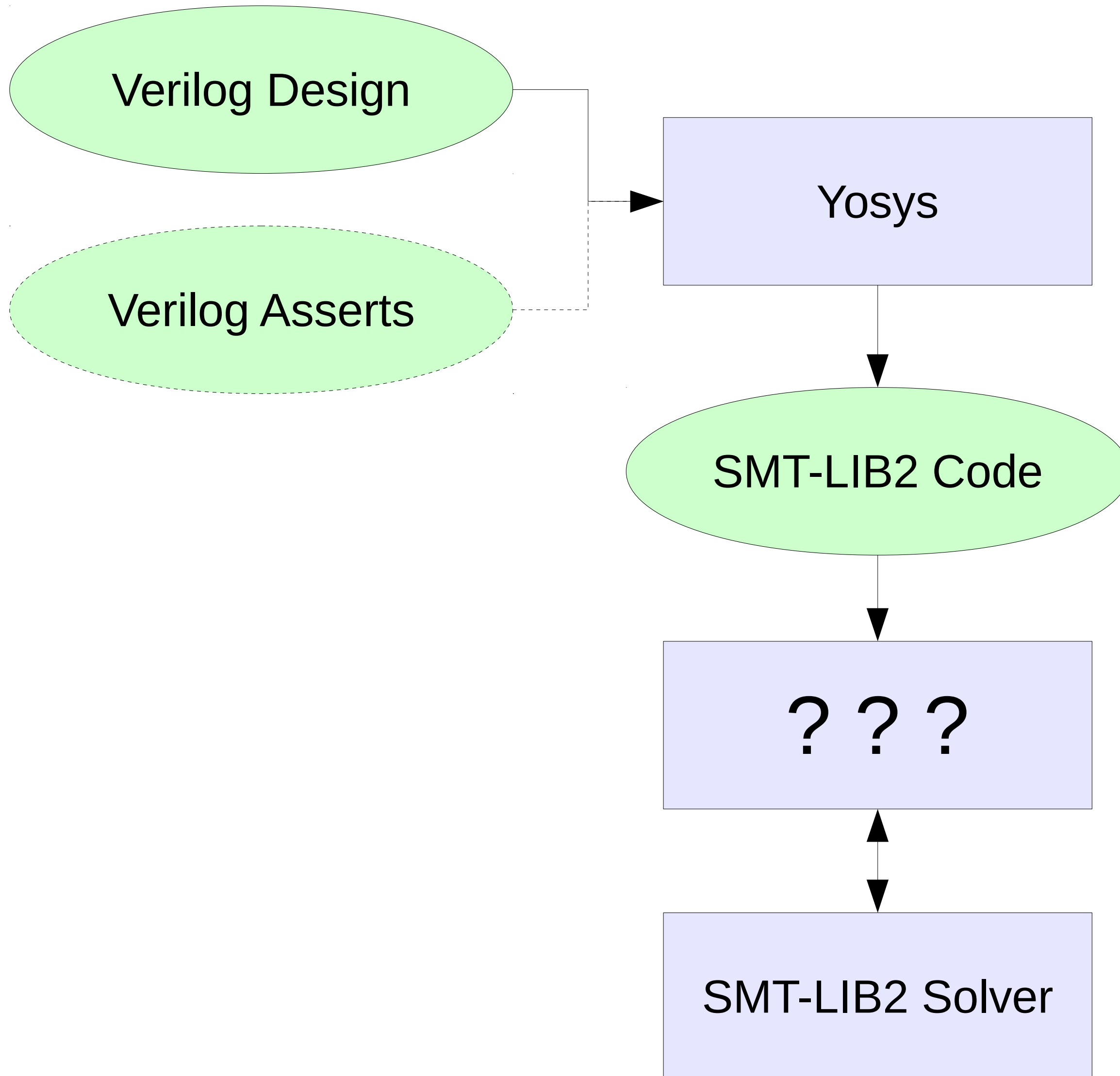
Benchmark

Solver performance for stand-alone SMT2 files generated from yosys-smtbmc for the different supported logics. The SMT2 benchmark files have been submitted for SMTCOMP.

	Benchmark "PicoRV32 check"			Benchmark "PicoRV32 comp"			Benchmark "PicoRV32 hard"		
	z3	yices	bool-ector	z3	yices	bool-ector	z3	yices	bool-ector
QF_AUFBV	158	8	-----	668	279	-----	14/17	5326	-----
QF_UFBV	164	10	-----	493	64	-----	13/17	14/17	-----
QF_UF	13	7	-----	103	29	-----	15/17	15/17	-----
QF_ABV	169	6	21	744	152	9	14/17	5214	623
QF_BV	173	7	37	547	48	11	13/17	14/17	7168

1234 … Benchmark completed after 1234 seconds
12/17 … Benchmark reached timeout, completed 12 of 17 steps
Timeout was 8000 seconds (about 2 hours 15 minutes) for all benchmarks

Custom Flows



Options for writing custom proofs:

- Hand-written SMT2 code

- Custom python script using `smtio.py` (the python lib implementing most of yosys-smtbmc)

- Any other app using any SMT-LIB2 solver (e.g. using C/C++ API for proofs that involve many `(check-sat)` calls.

cycle3.v

```
module cycle3 (input clk, a, b, c, output reg [3:0] state);
    always @(posedge clk) begin
        state <= 0;
        case (state)
            0: state <= 1;
            1: state <= 0;
            2: if (a && !b && !c) state <= 3;
            3: if (!a && b && !c) state <= 4;
            4: if (!a && !b && c) state <= 2;
        endcase
    end
endmodule
```

Verification Problem: Show that this design contains a three states cycle.

cycle3.tpl

```
(set-logic QF_UFBV)

; insert the auto-generated code here
%%

; declate three states
(declare-fun s1 () cycle3_s)
(declare-fun s2 () cycle3_s)
(declare-fun s3 () cycle3_s)

; setup three states
(assert (and (cycle3_h s1) (not (cycle3_is s1))))
(assert (and (cycle3_h s2) (not (cycle3_is s2))))
(assert (and (cycle3_h s3) (not (cycle3_is s3))))

...
```

cycle3.tpl

```
...

; create cyclic transition relation
(assert (cycle3_t s1 s2))
(assert (cycle3_t s2 s3))
(assert (cycle3_t s3 s1))

; find model
(check-sat)

; print states
(get-value ((|cycle3_n a| s1) (|cycle3_n b| s1) (|cycle3_n c| s1)))
(get-value ((|cycle3_n a| s2) (|cycle3_n b| s2) (|cycle3_n c| s2)))
(get-value ((|cycle3_n a| s3) (|cycle3_n b| s3) (|cycle3_n c| s3)))
```

cycle3.tpl

```
$ yosys -ql cycle3.yslog \
    -p 'read_verilog cycle3.v' \
    -p 'prep -top cycle3 -nordff' \
    -p 'write_smt2 -tpl cycle3.tpl cycle3_with_tpl.smt2'

$ z3 cycle3_with_tpl.smt2
sat
(((|cycle3_n a| s1) true)
 ((|cycle3_n b| s1) false)
 ((|cycle3_n c| s1) false))
(((|cycle3_n a| s2) false)
 ((|cycle3_n b| s2) true)
 ((|cycle3_n c| s2) false))
(((|cycle3_n a| s3) false)
 ((|cycle3_n b| s3) false)
 ((|cycle3_n c| s3) true))
```

End-to-end Formal Verification of RISC-V Cores with `riscv-formal`

- `riscv-formal` is a framework for formal verification of RISC-V Processor Cores, using Yosys-SMTBMC and ither Yosys-based verification flows.
- A separate verification task for each instruction
- And a few additional verification tasks to verify consistent state between instructions and correct implementation of memory I/O
- Project is in its early stages and under construction! Only RV32I support at the moment.

https://github.com/cliffordwolf/riscv-formal

Future Work

- Limited support for SVA properties
  - Using AST transformations to clocked always blocks with immediate assertions.
  - However: I highly recommend sticking to immediate assertions in new code. It does not look like FOSS simulators are going to support SVA properties anytime soon.

- Improved support for Verilog x-propagation
  - Currently only available with Yosys “sat” flow
  - Using a Yosys pass that adds transforms into a circuit problem without x-progatation (using explicit `*__x` nets)

- Yosys C Back-End
  - There exist a few FOSS formal verification tools for C (e.g. ESBMC)
  - A C back-end would enable verification flows that check against formal specs written in C.

- SymbiYosys
  - A front-end driver for Yosys-SMTBMC and other Yosys-based verification flows
  - “Push-button solution” for a wide range of HDL verification tasks.

http://www.clifford.at/papers/2016/yosys-smtbmc/

Questions ?