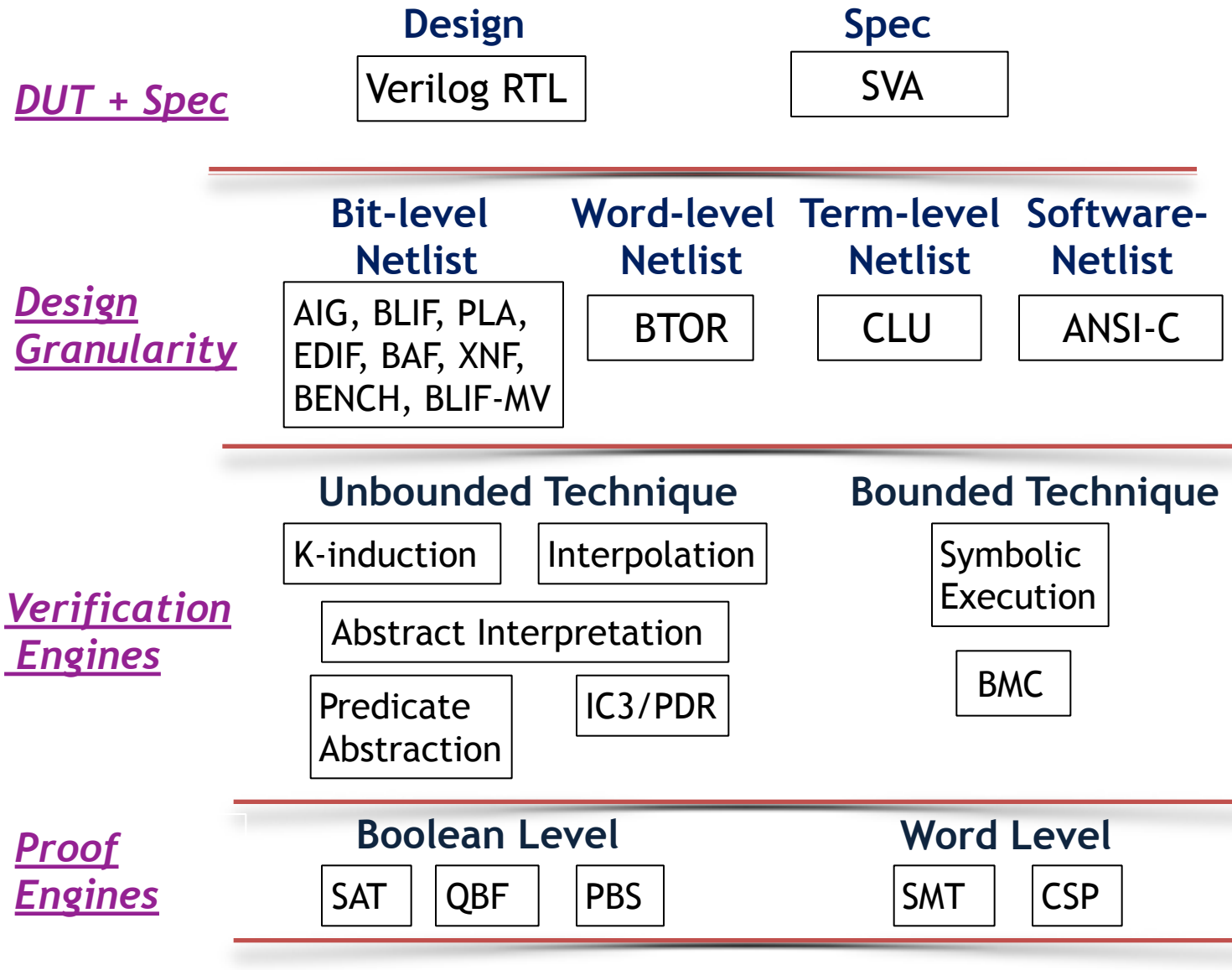
How Efficient are Software Verifiers for Hardware ?

Authors: Rajdeep Mukherjee, Daniel Kroening, Tom Melham

FMCAD 2015

Phases of Hardware Model Checking Tool



Background

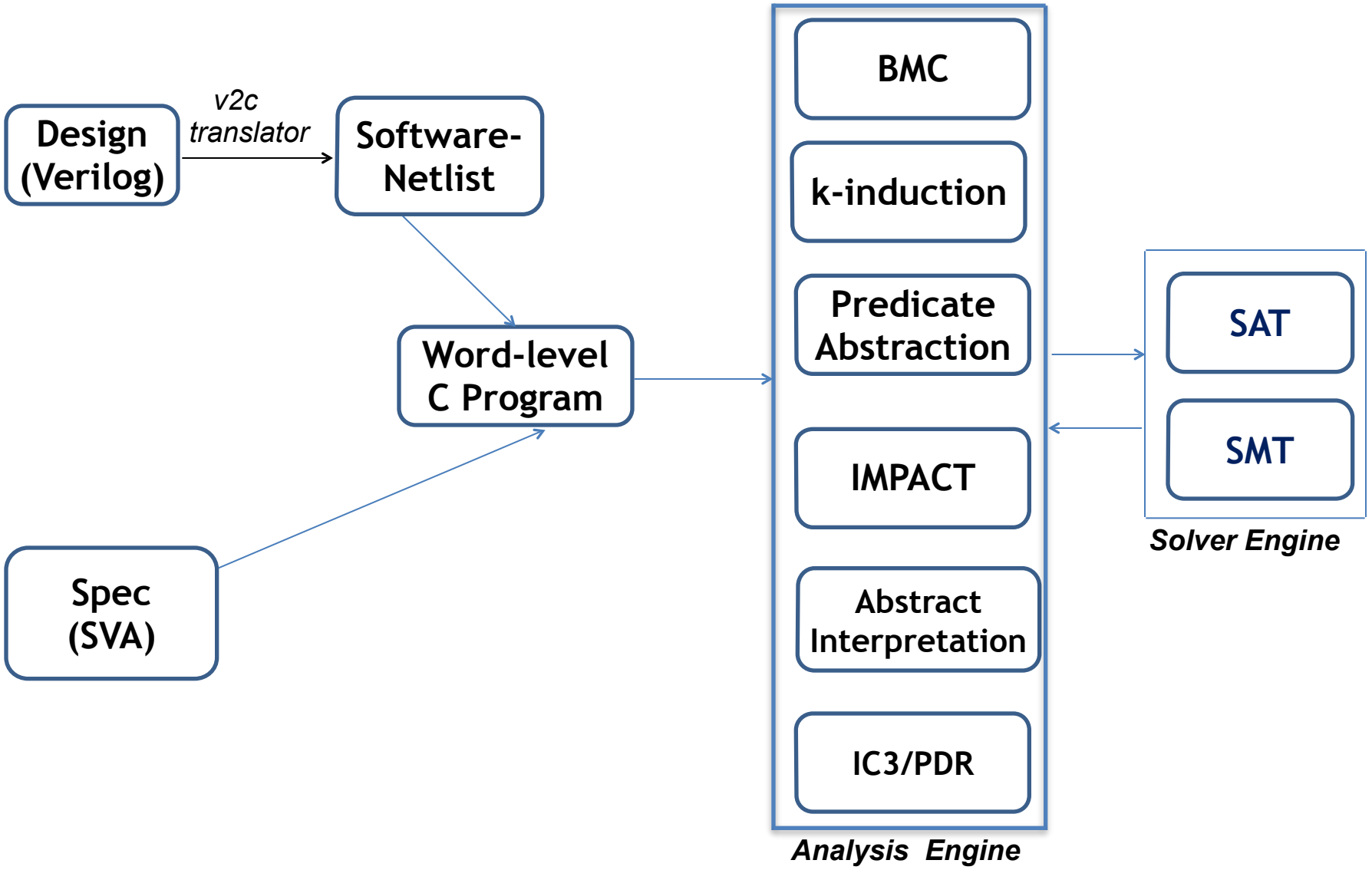
- The Simple Art of SoC Design, by Michael Keating, Synopsys Fellow
  - *High-level design contains less bug than production-ready RTL design,*
  - *Scalable Verification at higher level of abstraction*
  - *Mature synthesis technology*

- Chakraborty et. al.: Word-level symbolic trajectory evaluation, CAV'15.
- Drechsler et. al.: Wolfram- A word level framework for formal verification, RSP'09.
- Kroening et. al.: Word level predicate abstraction and refinement for verifying RTL verilog, DAC'05.
- Kroening et. al.: Lifting propositional interpolants to the word-level, FMCAD'07.
- Lahiri et. al. : The UCLID decision procedure, CAV'04.

Key Message

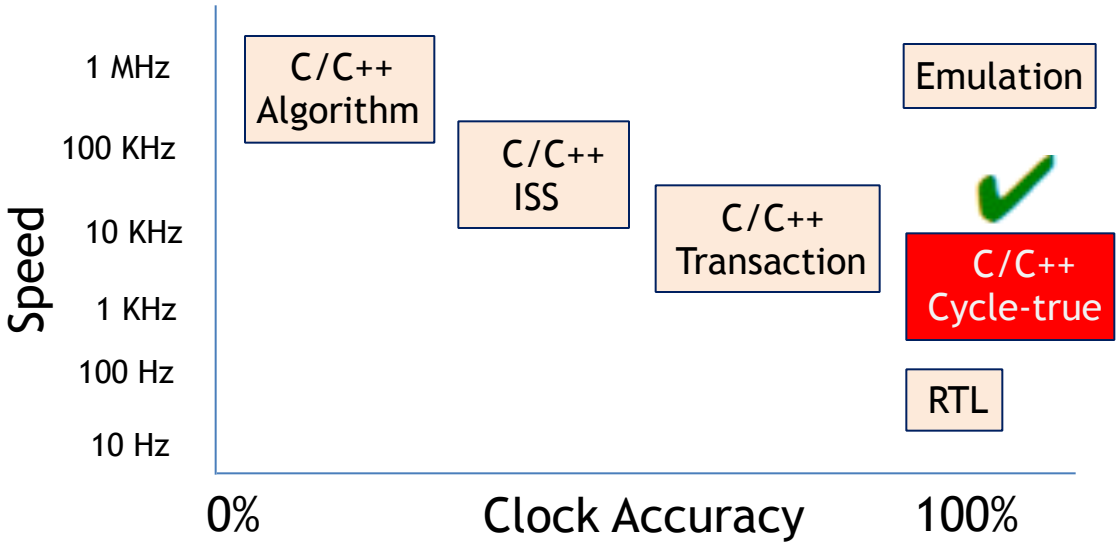
Change in frontend of Verifier enables opportunity for efficient reasoning in the backend of the tools

Hardware Property Verification Flow



Hardware Verification using Software Analyzers

Where are we working ?



Synthesis Semantics

```
module top(clk,a);
```

Signal declaration

```
input a,clk;
reg b,d,e;
wire a;
```

Continuous Assignment Statement

```
wire c = (e)? 1'b0 : d;
wire cond = a;
```

Always Clocked block

```
always @(posedge clk) begin
  b<=a;
// non-blocking
 if(cond && b)
   e<=b;
  else
   e<=0;
  d<=c;
end
endmodule
```

SEMANTICS

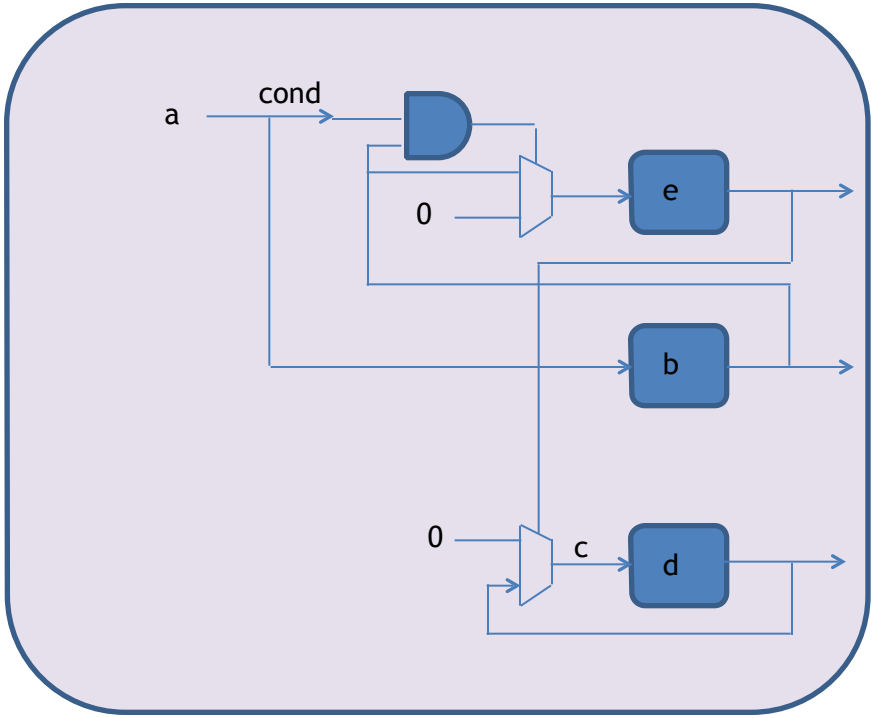
SYNTHESIZED

Combinational Logic
Continuous Assignment Update

Forall t. c(t) = if e(t) then 0 else d(t)
Forall t. cond(t) = a(t)

Sequential Logic
Always Clocked block

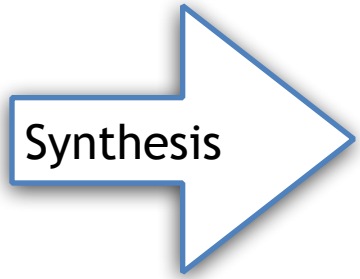
Forall t. b(t+1) = a(t)
e(t+1) = if(cond(t) && b(t)) then b(t)
d(t+1) = c(t)



From Bits to Word to Term to Software-Netlist

Verilog

```
module top(Din,En, clk,Dout);
 wire cs; reg ns;
 input clk,Din,En;
 output Dout;

// Combinational block
 assign Dout = cs;
 always @(Din or cs or En)
 begin
  if (En) ns = Din;
  else ns = cs;
 end
 ff ff(ns,clk,cs);
endmodule

// Sequential block
module ff(Din,clk,Dout);
 input clk, Din;
 output Dout; reg q;
 assign Dout = q;
 always @(posedge clk)
   q <= Din;
endmodule
```



Bit-level Netlist

Variable Map:

Inputs: top.clk=0, top.Din=1,
top.En=2, top.ff.CLK=3,
top.ff.Din=4,
input[0]=6, input[1]=7
input[2]=11, input[3]=12

Wires:

top.Dout=11, top.ff.Dout=5,
top.cs=12, top.ns=!10

Latch: top.ff.q=5

Transition constraints:

!(var(5) & !var(12)) &
!(!var(5) &var(12))
!(var(4) & !(var(2)
&var(1)) & !(!var(2)
&var(7))) & !(!var(4)

Word-level Netlist

State constraints:

top.Dout==top.cs

top.ff.Dout==top.ff.q

top.ff.Din==top.ns

top.ff.clk==top.clk

top.ff.Dout==top.cs

top.ns==top.En ?

top.Din : top.cs

Transisition constraints:

next(top.ff.q)== top.ff.Din

Software-Netlist

```
_Bool nondet_bool();
struct s_ff {_Bool q; };
struct s_en { _Bool ns; struct s_ff sff; }sen;
// Sequential Logic
_Bool ff(_Bool CLK, _Bool Din, _Bool *Dout) {
  _Bool q_old;
  q_old = sen.sff.q; sen.sff.q = Din;
  *Dout = q_old;
  return; }
// Combinational Logic
void top(_Bool clk,_Bool Din,  _Bool En, _Bool *Dout) {
_Bool cs;
 if(En) { sen.ns = Din; }
 else { sen.ns = cs; }
 ff(clk, sen.ns, &cs);
 *Dout = cs; }
int main() {
  _Bool clk,En,Din,out;
  while(1) {
   Din = nondet_bool(); En = nondet_bool();
   top(clk,Din,En,&out); }
  return; }
```

State-holding elements

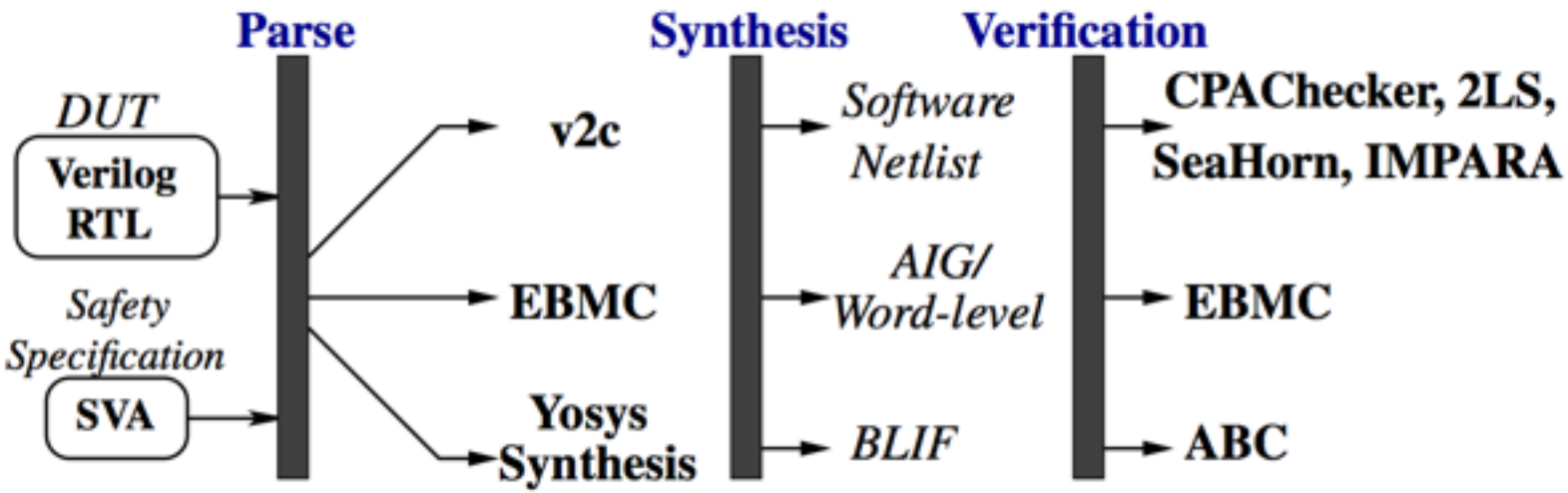
Shadow-variable Update
For Non-blocking Statement

Equivalence of Verilog and Software-netlist

- Output model is bit-precise and cycle accurate
- For unsafe benchmarks, bugs are manifested at the same clock cycle
- For safe benchmarks, properties are proven to be k-inductive, where values of k are same in Verilog and software-netlist model

Proposed Verification Tool Flow

Verification at Bit-level, Word-level, Software-Netlist level



Classical Abstract Interpretation Based Tools

- Suitable for proving absence of run-time errors, hence requires less precision
- But for functional verification, need more precision by data and control partitioning

```
typedef enum {false = 0, true = 1} _Bool;
int id(int a) {
  return a;
}
void main() {
 int x,y,z;
 _Bool c;

 if (c)
   x = -1;
 else
   x = 2;
 int k = id(x);
 → x = [-1, 2], k = [-1, 2]
 z = x * k;
 __ASTREE_assert((z!=-2));
}
```

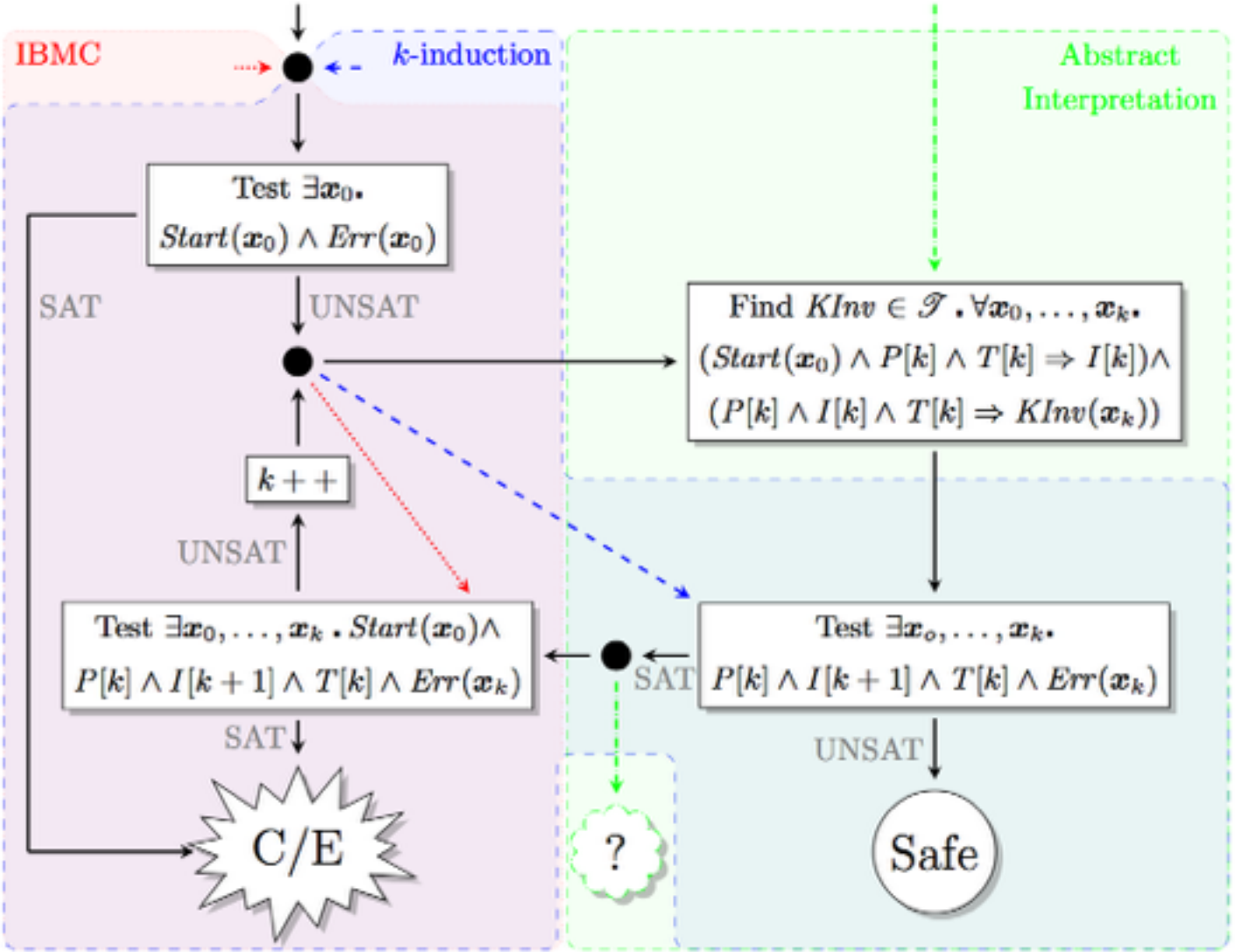
Assertion Fails !

Precision?

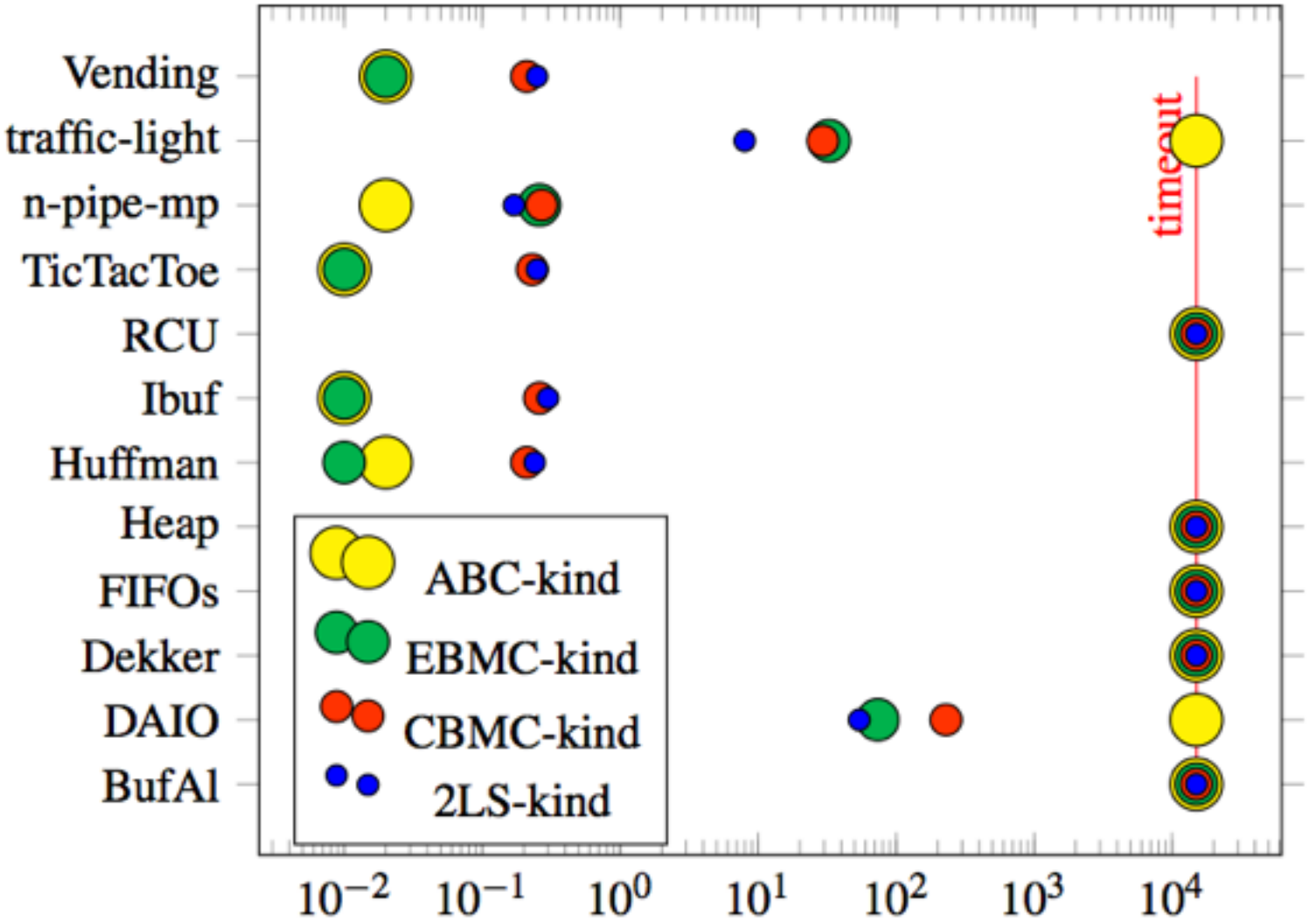
```
typedef enum {false = 0, true = 1} _Bool;
int id(int a) {
  return a;
}
void main() {
 int x,y,z;
 _Bool c;

 __ASTREE_partition_control
 if (c)
   x = -1;
 else
   x = 2;
 int k = id(x);
 z = x * k;
 __ASTREE_assert((z!=-2));
Control-flow Join after the Assertion Point
}
```

Hybrid tools for Software Verification

Combines BMC+ K-induction + Abstract Interpretation



K-induction at Bit-level,Word-level, Software-Netlist level

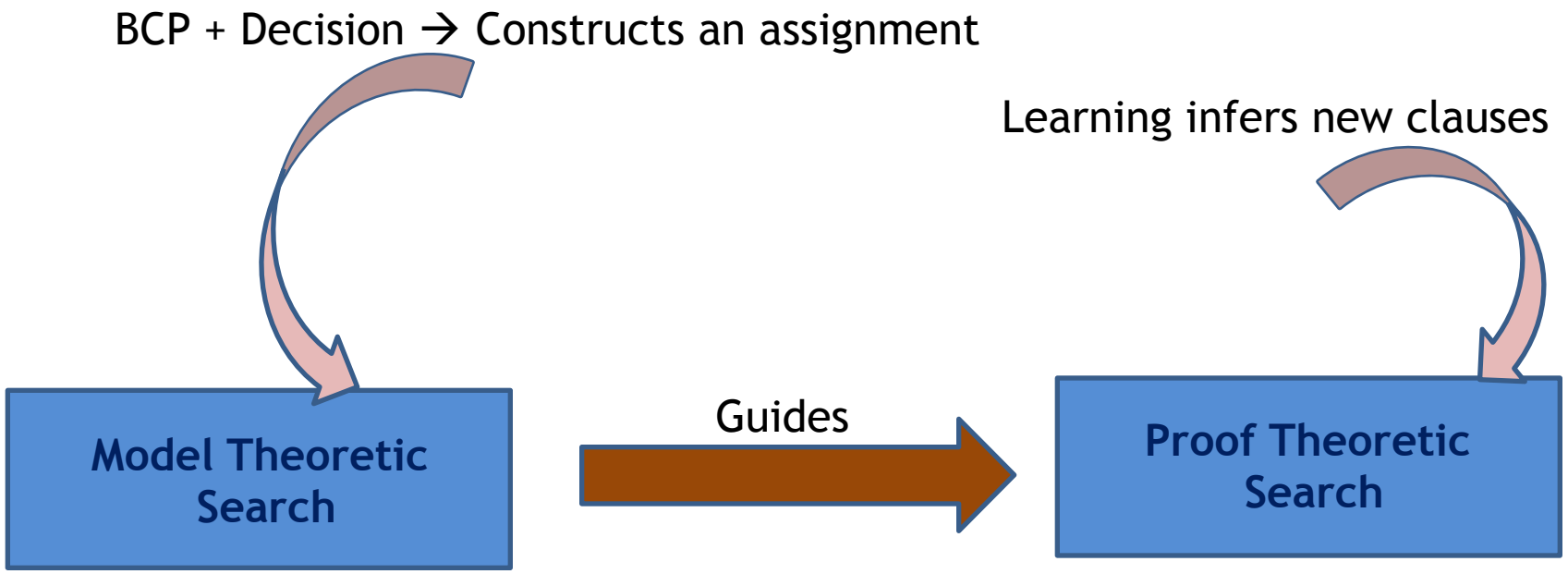


Results

- BMC on Software-netlist is > 2X faster than BMC on bit-level netlist
- Software Interpolation (IMPACT) and hardware interpolation-based techniques have comparable times.
- Bit-level PDR did not terminate for RCU and took > 4 hrs for FIFO controller and Buffer Allocation

Ongoing Work

Conflict Driven Clause Learning Solvers



SAT Solvers: **Precise but inefficient**
Abstract Interpreter: **Efficient but Imprecise**

How to make SAT solvers more efficient?

→ Choose a domain that's better suited to your problem than the Boolean constants domain!

How to make Abstract Interpretation more precise?

→ Wrap them in the SAT architecture!

Abstract Conflict Driven Clause Learning

New program analysis that embeds an abstract domain inside Conflict Driven Clause Learning algorithm of modern day SAT solvers.

What is ACDCL?

From AI Point of View → ACDCL is an abstract interpreter that uses Decision and Learning to increase transformer precision

From a decision procedure perspective → ACDCL is a SAT solver for program analysis constraints. It is a strict generalisation of propositional CDCL.

Decision Procedure		Abstract Interpretation
Partial Assignments	→	Abstract Domain
Decision	→	Restrict range of variables
Unit Rule	→	Best Abstract Clause Transformer
BCP	→	GFP Iteration
Learning	→	Generate program analysis constraint (Implicit form of Trace Partitioning)

Conclusion

- SoC designs are increasingly written at higher level than RTL
- Change in front-end of verifiers allows opportunity for scalable reasoning in the backend
- Verifiers must employ software-like representations of circuits
- Abstract domain needed : Bit-field domain, Interval, Octagon, Equality, Polyhedra
- Need bit-precise reasoning
- Need new solver based on ACDL

UNIVERSITY OF OXFORD

Thank You for Your Attention !

Systems Verification Group

www.cprover.org/hardware/v2c-isvlsi/

Dependency Analysis

```
module top(clk,a);
input a,clk;
reg b,d,e;
wire a;

Continuous Assignment Statement
wire c = (e)? 1'b0 : b;
wire cond = b;

Always Clocked block
always @(posedge clk) begin
  b<=a;

Non-blocking Assignments
 if(cond)
   e<=b;
 else
   e<=0;
 d<=c;
end
endmodule
```

Dependency Analysis

```
State-holding elements
struct state_elements_top {
 unsigned int b,  e;
};
struct state_elements_top u1;

unsigned int c;
_Bool cond;
void top_ns(unsigned int clk, unsigned int a)
{
    _Bool a_old, b_old, d_old, e_old;

    b_old = u1.b;
    d_old = u1.d;
    e_old = u1.e;
                    Shadow-variable Update
                    For Non-blocking Statement
  Sequential Logic
  State Update Phase
  u1.b= a;
    if(cond)
       u1.e = b_old;
     else
       u1.e = 0;
    u1.d = c;

    Combinational Logic
    Continuous Assignment Statements
   cond = u1.b;
   c = (u1.e) ? 0 : u1.b;
}
```

Circuits with Loopback

```
module foo(c,a,z,y);
 input c;
 input [31:0] a;
 output [31:0] z;
 reg [31:0] y;
 output [31:0] y;
```

Continuous Assignment Statement

```
 assign z[0] = c;
 assign z[31:1] = a[30:0];
```

Always Clocked block

```
always@(posedge clk)
   y <= z;
assert property1: ((a == z));
endmodule

module top();
 wire [31:0] loopback;
 wire [31:0] y;
 wire data;
 foo f(.c(data),.a(loopback),.z(loopback),.y());
endmodule
```

```
// type declarations
typedef unsigned int _u32;
// Module Verilog::foo
struct module_foo {
  _Bool c;
  _u32 a;
  _u32 z;
  _u32 y;
  _Bool clk;
};
// Module Verilog::top
struct module_top {
  _u32 loopback;
  _u32 y;
  _Bool data;
  struct module_foo f;
};
// top module
extern struct module_top top;

void main() {
  __CPROVER_assume(top.f.z ==  ((top.f.z & 0xfffffffe)
                                | (top.f.c & 0x1)));
  __CPROVER_assume(top.f.z ==  ((top.f.z & 0x00000001)
                                | ((top.f.a & 0x7fffffff) << 1)));
  __CPROVER_assume(top.f.c == top.data);
  __CPROVER_assume(top.f.a == top.loopback);
  __CPROVER_assume(top.f.z == top.loopback);
  // Next state function
  __CPROVER_assume(top.f.y == top.f.z);
}
```

Do we need fix-point ?

```
module foo(c,a,z,y);
 input c;
 input [31:0] a;
 output [31:0] z;
 reg [31:0] y;
 output [31:0] y;
```

Continuous Assignment Statement

```
 assign z[0] = c;
 assign z[31:1] = a[30:0];
```

Always Clocked block

```
always@(posedge clk)
   y <= z;
assert property1: ((a == z));
endmodule

module top();
 wire [31:0] loopback;
 wire [31:0] y;
 wire data;
 foo f(.c(data),.a(loopback),.z(loopback),.y());
endmodule
```

```
struct state_elements_foo{
  unsigned int y;
};
struct state_elements_foo  sfoo;
```

Sequential Logic
State Update Phase

```
void foo_nextstate(_Bool c, unsigned int a, unsigned int *z, unsigned int *y) {
  unsigned int tmp0;
  tmp0 = *z;
  sfoo.y = tmp0;
  *y = tmp0;
}
```

Combinational Logic
Continuous Assignment Statements

```
void foo_output(_Bool c, unsigned int a, unsigned int *z, unsigned int *y) {
  *z = (*z & 0xfffffffe) | (c & 0x1);      // z[0] = c;
  *z = (*z & 0x00000001) | ((a & 0x7fffffff) << 1);    // z[31:1] = a[30:0];
}

void main() {
  unsigned int y, loopback;
```

Compute fixpoint

```
for(int i=1;i<33;i++)
    foo_output(1, loopback, &loopback, &y);
  foo_nextstate(1, loopback, &loopback, &y);
}
```

Need Richer Domains like ..

Octagon Domain Analysis:

lo ≤ X ± Y ≤ hi

```
int get_random_1to10()
{
        int result;
        __ASTREE_known_range((result, [1; 10]));
        return result;
}

int main()
{
  int  x = get_random_1to10();
  int  y = 10 - x;
 Intervals:  x = [1, 10], y = [0, 9]
 Octagons:   -8 ≤  (x - y) ≤  10 ,  (x+y)  = 10
  __ASTREE_assert((x+y <= 10));
}
```

Equality Domain Analysis :

X == Y, X != Y

```
typedef enum {false = 0, true = 1} _Bool;
int id(int a) {
  return a;
}
void main() {
  int x,y,z;
  _Bool c;
  if (c)
    x = -1;
  else
    x = 2;
  int k = id(x);
  Intervals:  x = [-1, 2], k = [-1, 2]
  Equality:  (x == k)
  z = x * k;
  __ASTREE_assert((z!=-2));
}
```

Semantic Loop Unrolling

Program P:

```
int main() {
  int init = 0;
  int i=0;
  float x=0.0, div=0.0;
  while (i<10) {
    if(init) {
       x+=x/div;
    }
    else {
      init = 1;
      x = 1.0;
      div = 2.0;
    }
    i++;
}
```

Unroll= 0:

- One invariant for all loop iterations:
- i ε [0,9]; init ε [0,1]; div ε [0.0,2.0]

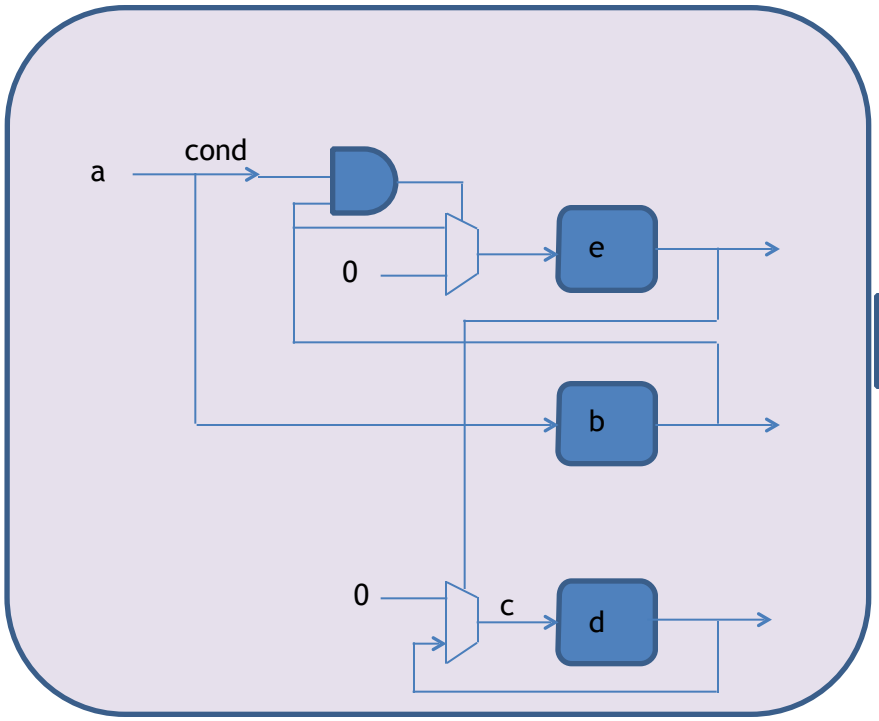
➢ False alarm: potential runtime error

Unroll = 1:

- One invariant for first loop iterations:
- At entry: i ε {0}; init ε {0}
- One invariant for all other loop iterations: i ε [1,9]; init = 1; div = 2.0

➢ No alarms reported

Translation to C (Clock Accurate)



Synthesis Semantics

State-holding elements

```
struct state_elements_top {
  unsigned int b, d, e, c;
};
struct state_elements_top  u1;

void top (unsigned int clk, unsigned int a)
{
  _Bool  b_old, d_old, e_old;
```

Internal Wires

```
  _Bool cond;

  b_old = u1.b;
  d_old = u1.d;
  e_old = u1.e;
```

Shadow-variable Update For Non-blocking Statement

Combinational Logic
Continuous Assignment Statements

```
  cond = a;
  u1.c = (u1.e) ? 0 : u1.d;
```

Sequential Logic
State Update Phase

```
 u1.b = a;
 if(cond && b_old)
    u1.e = b_old;
 else
    u1.e = 0;
  u1.d = u1.c;
}
```

Actual State update

Software-Netlist Model

- A software-netlist model is defined as six tuple, SN=⟨In,Out,Seq,Comb,Init,Asgn⟩, where

In, *Out*, *Seq*, *Comb*, *Init* are *input*, *output*, *sequential/state-holding*, *combinational / stateless* signals and *initial* states respectively. *Asgn* is a finite set of assignments to *Out*, *Seq* and *Comb* where,

- $Asgn ::= CAsgn | SAsgn$
- $CAsgn ::= (V_C = bvExpr) | (V_C = bool), V_C \in Comb \uplus Out$
- $SAsgn ::= (V_S = bvExpr) | (V_S = bool), V_S \in Seq$
- $bvExpr ::= bv_{const} \mid bv_{var} \mid ITE(cond, bv1 \ldots bv_n) \mid bv_{op}(bv1 \ldots bv_n), cond \in bool, bv_i \in \{bv_{const}, bv_{var}\}$
- $bool ::= true \mid false \mid \neg b \mid b1 \wedge b2 \mid b1 \vee b2 \mid bv_{rel}\{b1 \ldots b_n\}, (n \geq 1)$

- **Cycle accurate and Bit-precise Model**
- **Generated following synthesis semantics**