How Efficient are Software Verifiers for Hardware ?

RAJDEEP MUKHERJEE, DANIEL KROENING, TOM MELHAM

Semiconductor Research Corporation

UNIVERSITY OF OXFORD

Hardware Model Checking

Transition system: $T = \langle I, x, \delta \rangle$, where $x = x_1, x_2, \ldots, x_n$ is the set of variables over $\mathbb{B} = \{true, false\}$, $I(x)$ is *initial state* and $\delta(x, x')$ represents the *transition relation*.

State: A state s of T is an an assignment of values to variable x.

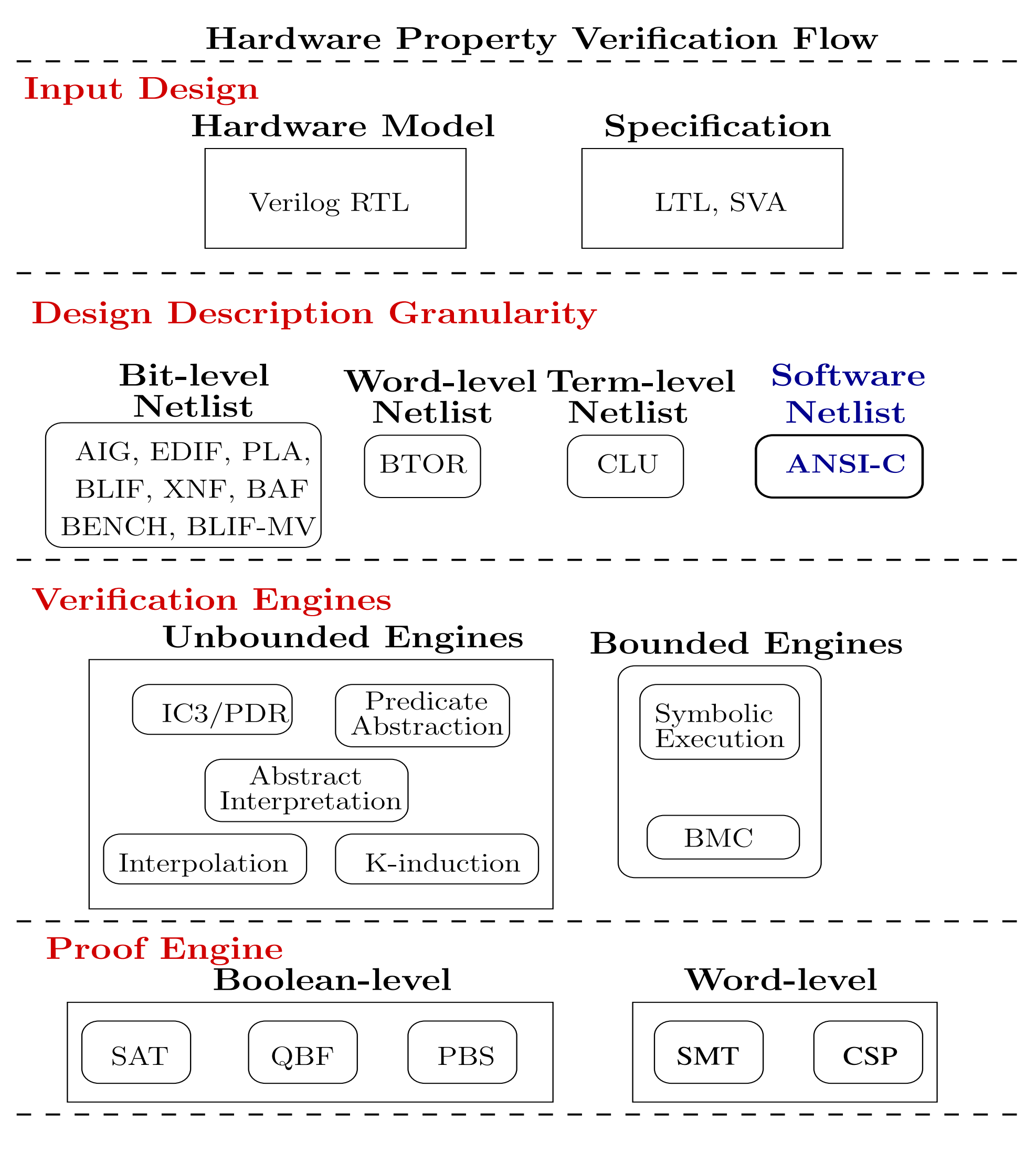
Trace: A trace $\gamma : s_0, s_1, \ldots$ is an infinite sequence of states such that $s_0 \models I$, and for each $i \geq 0$, $(s_i, s_{i+1}) \models \delta$.

Reachable State: A state s is reachable in T if $\exists \gamma \in T : s \in \gamma$. We denote the reachable state space as $\mathbb{Q}$.

Safety property: A safety property P of T is a first-order formula over the varaibles X of T, which asserts that certain states s of T cannot be reached during the execution of T, often known as *bad states*, $B(x)$.

Problem Statement: Given a state-space over n boolean variables, the problem is to decide whether $T \models P$, that is, starting from initial state $I(x)$, whether a state in $B(x)$ can be reached following only transitions in $T(x, x')$.

Model Checking Phases



Software Netlist

A *Software Netlist* is defined as the six tuple, $SN = \langle In, Out, Seq, Comb, Init, Asgn \rangle$, 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 $Comb$,

- $Asgn ::= CAsgn | SAsgn$
- $CAsgn ::= (V_c = bvExpr) | (V_c = bool)$, where $V_c \in Comb \uplus Out$
- $SAsgn ::= (V_s = bvExpr) | (V_s = bool)$, where $V_s \in Seq$
- $bvExpr ::= bv_{const} | bv_{var} | ITE(cond, bv_1 \ldots bv_n) | bv_{op}(bv_1 \ldots bv_n), cond \in bool, bv_i \in \{bv_{const}, bv_{var}\}$
- $bool ::= true | false | \neg b | b_1 \wedge b_2 | b_1 \vee b_2 | bv_{rel}\{b_1 \ldots b_n\}, (n \geq 1)$

References

[1] R. Mukherjee, D. Kreoning, T. Melham. Hardware Verification Using Software Analyzers, In *ISVLSI '15*

[2] M. Brian, S. Joshi, D. Kreoning and P. Schrammel. Safety verification and refutation by k-invariants and k-induction, In *SAS '15*

[3] V.D'Silva, L. Haller and D. Kreoning. Abstract conflict driven learning, In *POPL'13*

[4] P. Bjesse. A practical approach to word level model checking of industrial netlists. In *CAV'2008*

Techniques

Bounded Model Checking:

$$I(x_0) \wedge_{i=0}^{k-1} (T(x_i, x_{i+1})) \wedge (\vee_{i=1}^{k-1} B(x_i))$$

BMC with K-induction:

$$P(x_0) \wedge_{i=0}^{k-1} (T_i \wedge P_i) \implies P_k$$

Interpolation-based Model Checking:

$$\mathbb{Q} \wedge_{i=0}^{k-1} (T_i \wedge P_i) \implies P_k$$

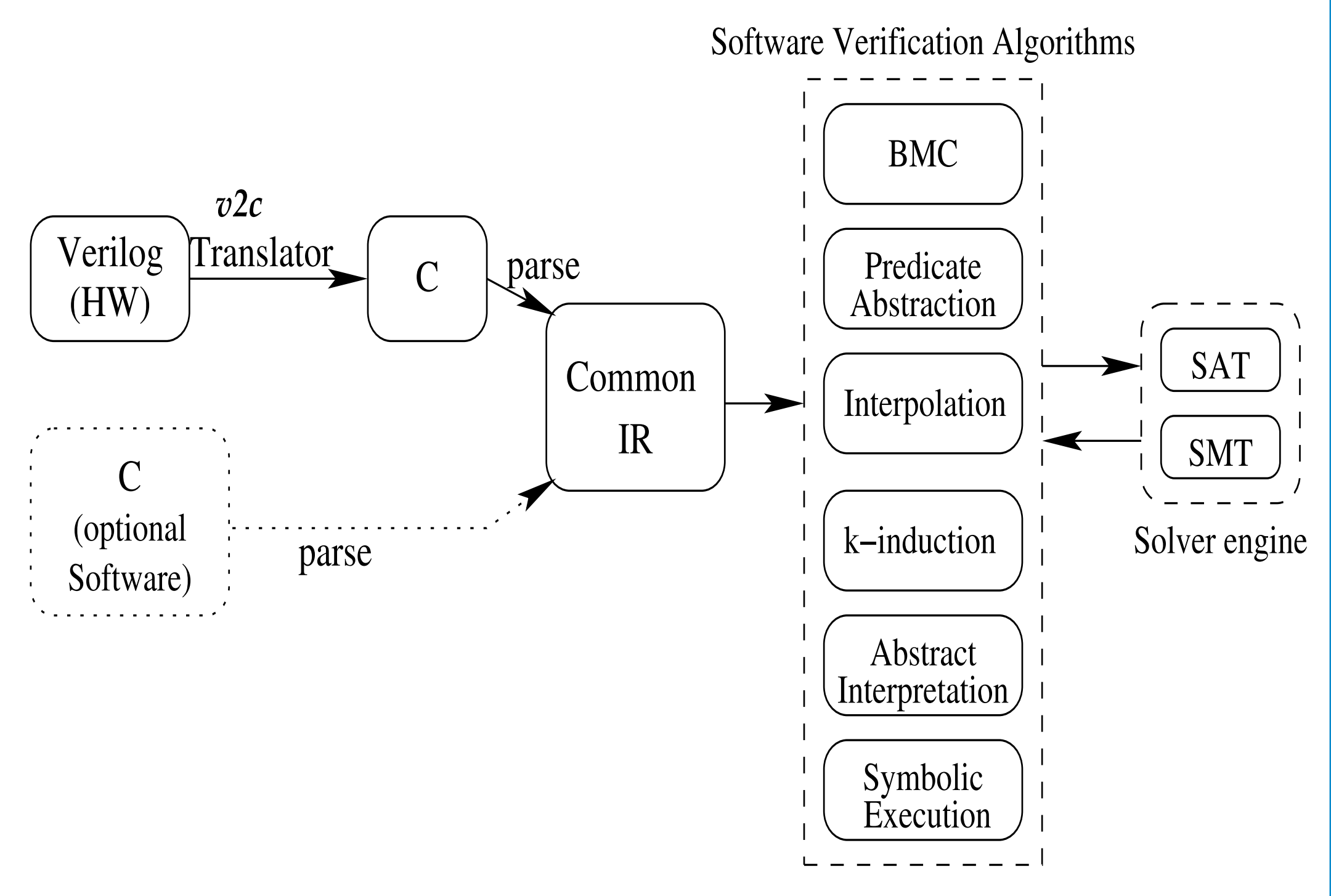
IC3/Property Directed Reachability:

1. $I(x) \implies P$
2. $\forall i, I(x) \implies \alpha_i(x)$
3. $\forall i, \wedge_{i=1}^{k} \alpha_i(x) \wedge P(x) \wedge T(x, x') \implies \alpha_k(x')$, α_k is inductive relative to $\alpha_1, \alpha_2, \ldots, \alpha_{k-1}$.
4. $\forall i, \wedge_{i=1}^{n} \alpha_i(x) \wedge P(x) \wedge T(x, x') \implies P(x')$, P is inductive related to the inductive invariants $\alpha_1, \alpha_2, \ldots, \alpha_n$.

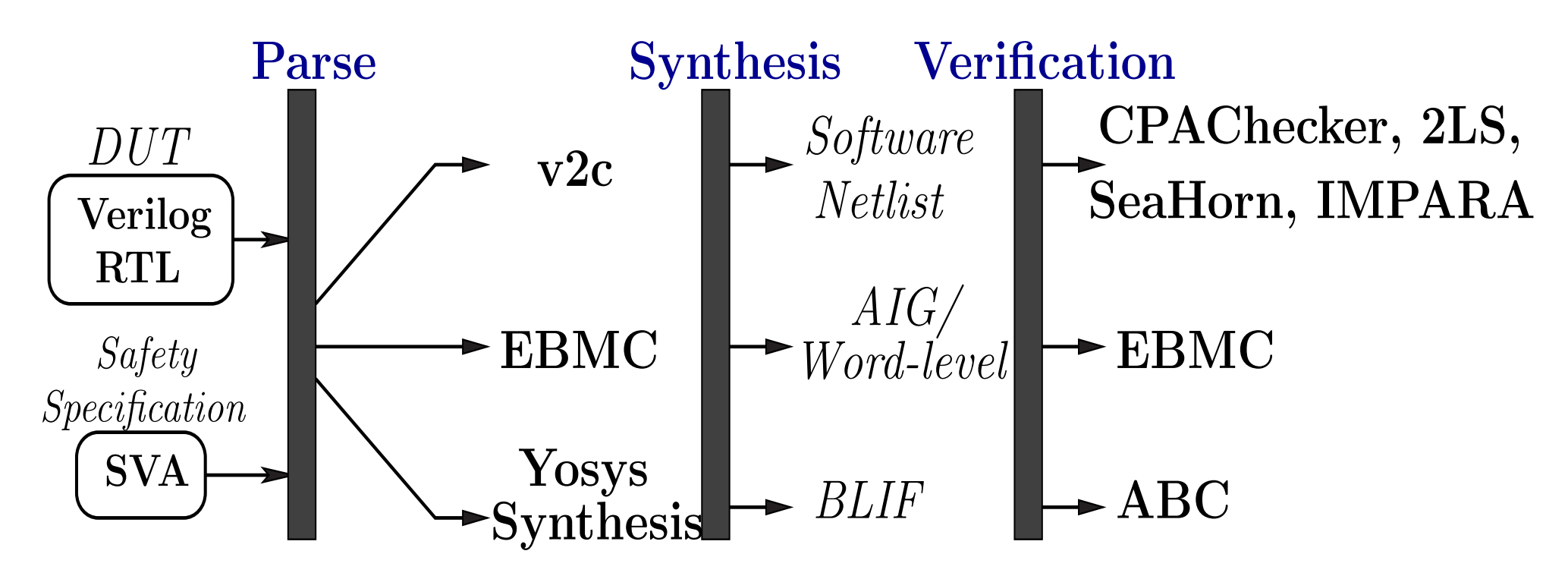
Predicate Abstraction:

1. Transition Relation: $\hat{T} := \{(b, b') | \exists x, x' \in S : T(x, x') \wedge \alpha(x) = b \wedge \alpha(x') = b'\}$, α is abstraction function, $x = \{x_1 \ldots x_n\}, b = \{b_1 \ldots b_n\}, b_i = \pi_i(x)$, π_i is the predicate on concrete variable x_i
2. Initial State: $\hat{I}(b) := \exists x \in S : (\alpha(x) = b) \wedge I(x)$
3. Safety Property: $\hat{P}(b) := \forall x \in S : (\alpha(x) = b) \implies P(x)$

Proposed Technique



Tool Flow



Results

1. BMC on software-netlist is on average >2X faster than BMC on bit-level netlist and word-level RTL model.
2. For unbounded verification, software k-induction is faster and solved more safe instances than k-induction for bit-level netlist.
3. Software PDR and bit-level PDR times are comparable for detecting deep bugs.

From BITS to WORD 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 ff1(ns,CLK,cs);
endmodule
// ~Sequential Block~
module ff(Din, clk, Dout);
input Din, clk;
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,
convert::input[0]=6
convert::input[1]=7
convert::input[2]=11
convert::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)
& !(!(var(2) & var(1))
& !(!var(2) & var(7))))
!(var(3) & !var(0)) &
!(!var(3) & var(0))
Next state functions:
NEXT(top.ff.q)=var(4)
```

Word-level Netlist

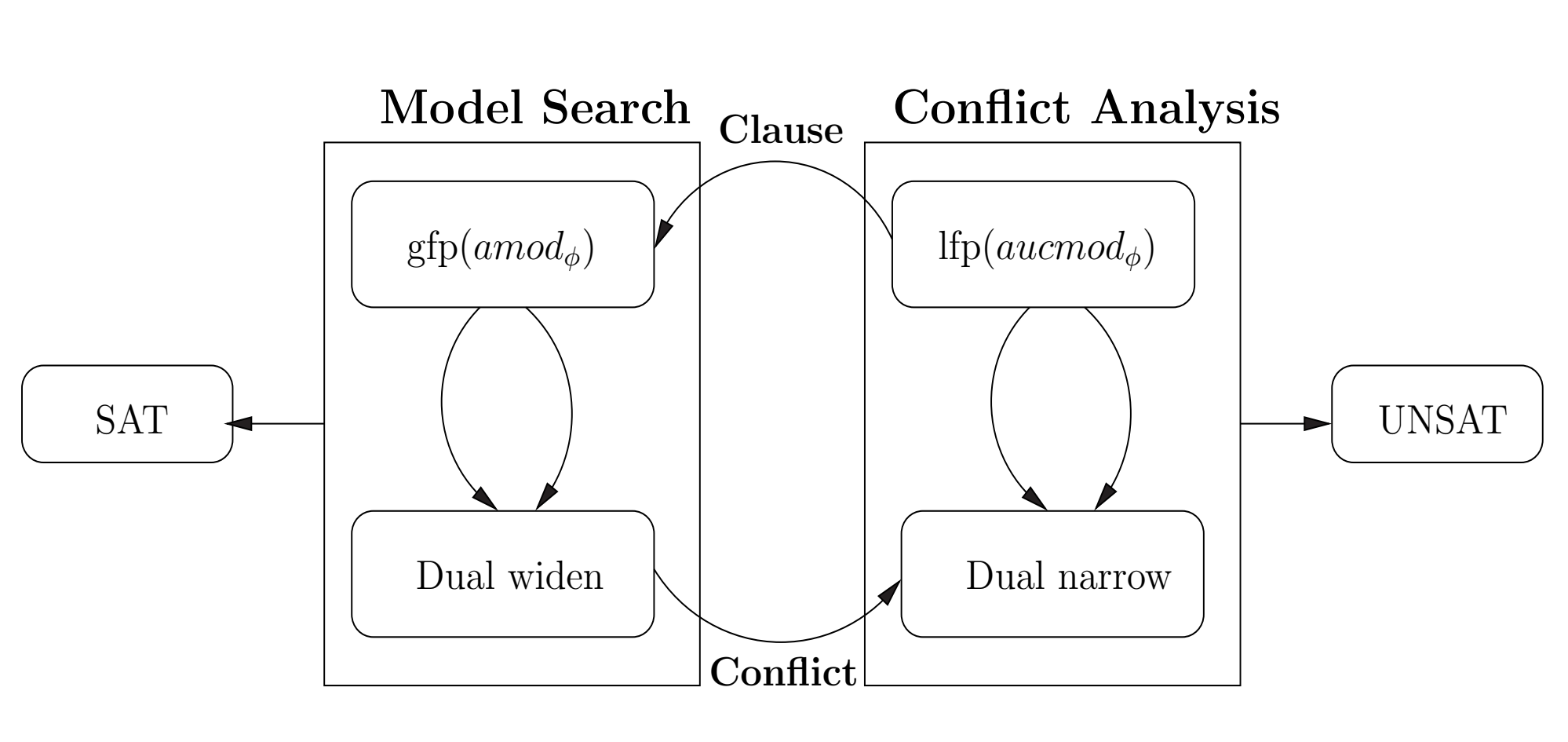
```
State constraints:
top.Dout==top.cs
top.ff.Dout==top.ff.q
top.ff.Din==top.ns
top.ff.clk==top.clk
top.ff.Dout==top.cs
top.ns==top.En ?
top.Din : top.cs

Transition constraints:
next(top.ff.q)==top.ff.Din
```

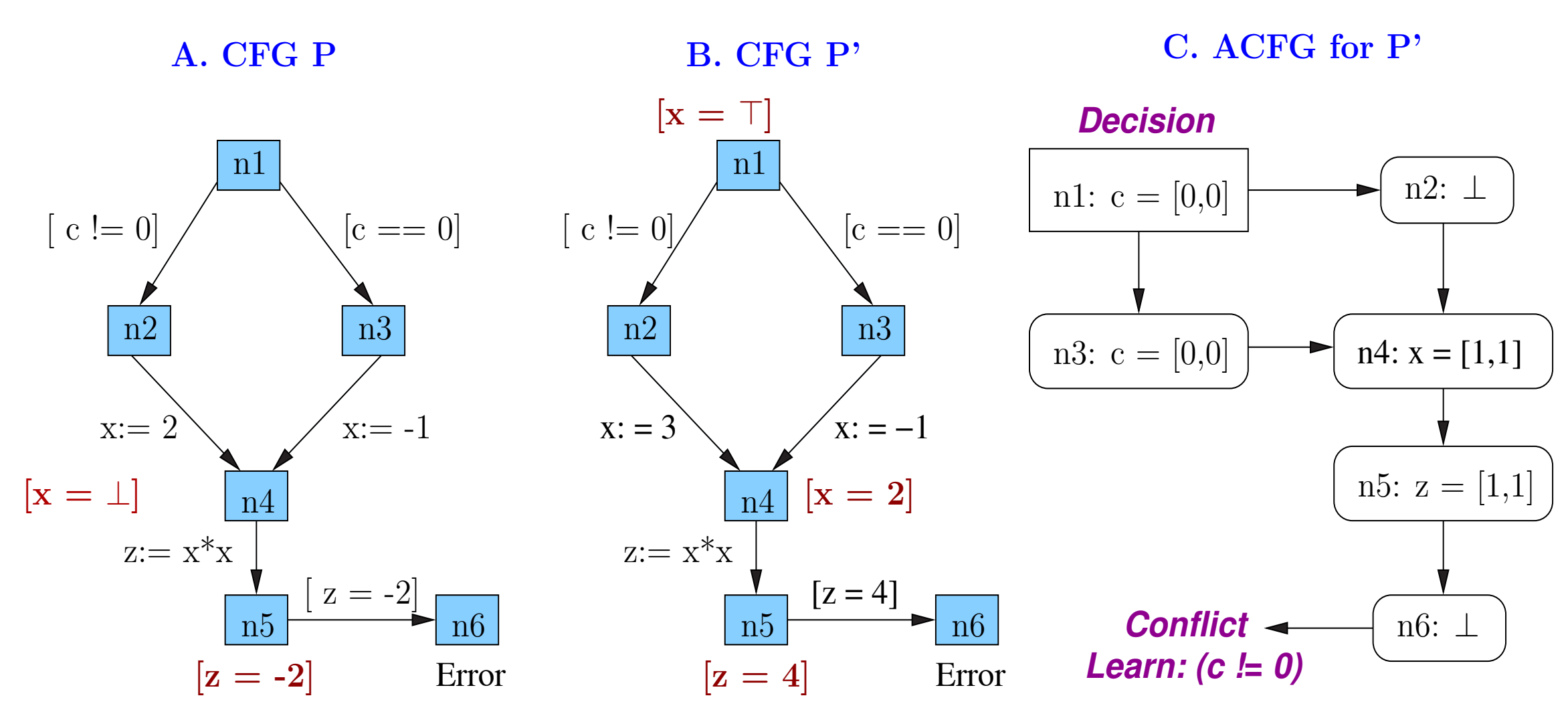
Software Netlist

```
_Bool nondet_bool();
struct s_ff{_Bool q;};
struct s_en{ _Bool ns;
struct s_ff sff; }sen;
_Bool ff(_Bool CLK,_Bool Din,
_Bool *Dout){
_Bool q_old;
q_old = sen.sff.q;
sen.sff.q = Din;
*Dout = q_old;
return; }
_Bool cs;
void top(_Bool clk,_Bool Din,_Bool En,
_Bool *Dout) {
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;
}
```

Abstract Interpretation view of CDCL



Abstract CDCL



Automatic Program and Property Driven Partitioning

Partial assignments ⇔ Abstract domain
Unit Rule ⇔ Abstract transformer
BCP ⇔ Greatest fix point iteration

Decision ⇔ Meet irreducible
Conflict Analysis ⇔ Graph cuts
Learning ⇔ Trace partitioning