Levels of Abstraction

- System Level
- High Level
- Behavioral Level
- Register-Transfer Level (RTL)
- Logical Gate Level
- Physical Gate Level
- Switch Level

Verilog模块的基本结构：

```
module <顶层模块名> (<输入输出端口列表>);
/*端口定义*/
input 输入端口列表; //输入端口声明
output 输出端口列表; //输出端口声明
/*信号类型说明*/
wire 信号名;
reg 信号名;
/*逻辑功能定义*/
//使用assign语句定义逻辑功能
assign <结果信号名>=<表达式>
//用always块描述逻辑功能
always @(<敏感信号列表>)
  begin
    //过程赋值
    //if-else,case语句;for循环语句
  end
//调用其他模块
<调用模块名>  <例化模块名> (<端口列表>);
//门元件例化
门元件关键字 <例化门元件名>(<端口列表>);
endmodule
```

- 变量类型：
  - 标量
  - 向量
  - 二维向量
- 存在并行块语句
- always块的敏感信号列表
- 循环次数必须确定
- 存在非阻塞赋值（整个过程块结束时才完成赋值操作）

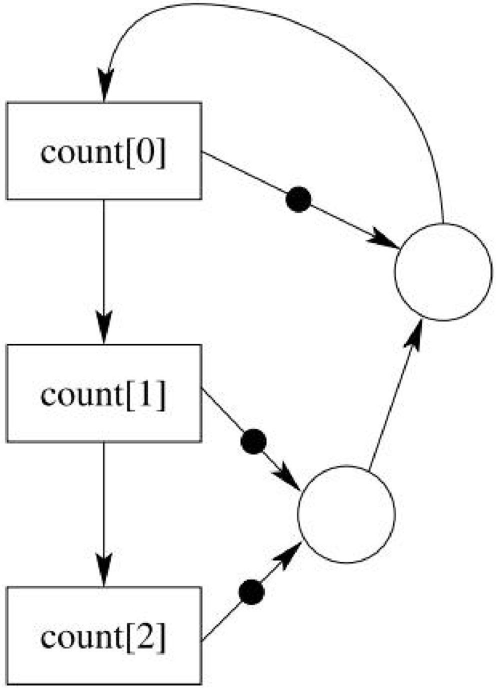
Net-Lists A net-list N is a directed graph (V_N, E_N, τ_N) where V_N is a finite set of vertices, $E_N \subseteq V_N \times V_N$ is the set of directed edges and $\tau_N : V_N \to \{\text{AND}, \text{INV}, \text{REG}, \text{INPUT}\}$ maps a node to its type, where AND is an "and" gate, INV is an inverter, REG is a register, and INPUT is a primary input. The in-degree of a vertex of type AND is at least two, of type INV and REG is exactly one and of type INPUT is zero. Any cycle in N must contain at least one REG node.

```
module counter(clk, count);
  input clk;
  output [2:0] count;
  reg [2:0] count;

  wire cin =
    ~count[0] & ~count[1] & ~count[2];

  initial count = 3'b0;

  always @ (posedge clk) begin
        count[0] <= cin;
        count[1] <= count[0];
        count[2] <= count[1];
  end
endmodule
```



And-Inverter Graph(AIG)

AIGER

bit-level techniques:

- BDD BMC
- interpolation
- IC3/PDR
- abstraction
- translate the problem into SMT formulas
- ...

extensions of the IC3/PDR:word-level IC3...

A *state* of a net-list is a mapping of its registers to the Boolean values $\mathbb{B} = \{0, 1\}$.

Btor2

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
static bool read_bool () {
  int ch = getc (stdin);
  if (ch == '0') return false;
  if (ch == '1') return true;
  exit (0);
}
int main () {
  bool turn;                // input
  unsigned a = 0, b = 0; // states
  for (;;) {
    turn = read_bool ();
    assert (!(a == 3 && b == 3));
    if (turn) a = a + 1;
    else      b = b + 1;
  }
}
```

⟨num⟩	::=	positive unsigned integer (greater than zero)
⟨uint⟩	::=	unsigned integer (including zero)
⟨string⟩	::=	sequence of whitespace and printable characters without '\n'
⟨symbol⟩	::=	sequence of printable characters without '\n'
⟨comment⟩	::=	';' ⟨string⟩
⟨nid⟩	::=	⟨num⟩
⟨sid⟩	::=	⟨num⟩
⟨const⟩	::=	'const' ⟨sid⟩ [0-1]+
⟨constd⟩	::=	'constd' ⟨sid⟩ ['-']⟨uint⟩
⟨consth⟩	::=	'consth' ⟨sid⟩ [0-9a-fA-F]+
⟨input⟩	::=	('input' | 'one' | 'ones' | 'zero') ⟨sid⟩ | ⟨const⟩ | ⟨constd⟩ | ⟨consth⟩
⟨state⟩	::=	'state' ⟨sid⟩
⟨bitvec⟩	::=	'bitvec' ⟨num⟩
⟨array⟩	::=	'array' ⟨sid⟩ ⟨sid⟩
⟨node⟩	::=	⟨sid⟩ 'sort' (⟨array⟩ | ⟨bitvec⟩)
		| ⟨nid⟩ (⟨input⟩ | ⟨state⟩)
		| ⟨nid⟩ ⟨opidx⟩ ⟨sid⟩ ⟨nid⟩ ⟨uint⟩ [⟨uint⟩]
		| ⟨nid⟩ ⟨op⟩ ⟨sid⟩ ⟨nid⟩ [⟨nid⟩ [⟨nid⟩]]
		| ⟨nid⟩ ('init' | 'next') ⟨sid⟩ ⟨nid⟩ ⟨nid⟩
		| ⟨nid⟩ ('bad' | 'constraint' | 'fair' | 'output') ⟨nid⟩
		| ⟨nid⟩ 'justice' ⟨num⟩ (⟨nid⟩)+
⟨line⟩	::=	⟨comment⟩ | ⟨node⟩ [⟨symbol⟩] [⟨comment⟩]
⟨btor⟩	::=	(⟨line⟩'\n')+

```
1 sort bitvec 1
2 sort bitvec 32
3 input 1 turn
4 state 2 a
5 state 2 b
6 zero 2
7 init 2 4 6
8 init 2 5 6
9 one 2
10 add 2 4 9
11 add 2 5 9
12 ite 2 3 4 10
13 ite 2 -3 5 11
14 next 2 4 12
15 next 2 5 13
16 constd 2 3
17 eq 1 4 16
18 eq 1 5 16
19 and 1 17 18
20 bad 19
```

BTOR2

Table 1. Operators supported by BTOR2, where $\mathcal{B}^n$ represents a bit-vector sort of size n and $\mathcal{A}^{\mathcal{I} \to \mathcal{E}}$ represents an array sort with index sort $\mathcal{I}$ and element sort $\mathcal{E}$.

indexed		
[su]ext w	(un)signed extension	$\mathcal{B}^n \to \mathcal{B}^{n+w}$
slice u l	extraction, $n > u \geq l$	$\mathcal{B}^n \to \mathcal{B}^{u-l+1}$
unary		
not	bit-wise	$\mathcal{B}^n \to \mathcal{B}^n$
inc, dec, neg	arithmetic	$\mathcal{B}^n \to \mathcal{B}^n$
redand, redor, redxor	reduction	$\mathcal{B}^n \to \mathcal{B}^1$
binary		
iff, implies	Boolean	$\mathcal{B}^1 \times \mathcal{B}^1 \to \mathcal{B}^1$
eq, neq	(dis)equality	$\mathcal{S} \times \mathcal{S} \to \mathcal{B}^1$
[su]gt, [su]gte, [su]lt, [su]lte	(un)signed inequality	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^1$
and, nand, nor, or, xnor, xor	bit-wise	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
rol, ror, sll, sra, srl	rotate, shift	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
add, mul, [su]div, smod, [su]rem, sub	arithmetic	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
[su]addo, [su]divo, [su]mulo, [su]subo	overflow	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^1$
concat	concatenation	$\mathcal{B}^n \times \mathcal{B}^m \to \mathcal{B}^{n+m}$
read	array read	$\mathcal{A}^{\mathcal{I} \to \mathcal{E}} \times \mathcal{I} \to \mathcal{E}$
ternary		
ite	conditional	$\mathcal{B}^1 \times \mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
write	array write	$\mathcal{A}^{\mathcal{I} \to \mathcal{E}} \times \mathcal{I} \times \mathcal{E} \to \mathcal{A}^{\mathcal{I} \to \mathcal{E}}$

SAT-Based Model Checking Without Unrolling

Definitions

A finite-state transition system $S:(\bar{i},\bar{x},I,T)$, $\bar{i}$:input variables,$\bar{x}$:internal state variables,$I(\bar{x})$:an initial condition,$T(\bar{i},\bar{x},\bar{x}')$:a transition relation.

A safety property $P(\bar{x})$:P is invariant for the system $S(S-invariant)$ if indeed only P-states are reachable.

$\boldsymbol{inductive}$:

An inductive assertion $F(\bar{x})$ describes a set of states that:

(1)includes all initial states: $I \Rightarrow F$

(2)is closed under the transition relation:$F \wedge T \Rightarrow F'$

$\boldsymbol{inductive\ relative}$:

An assertion F is inductive relative to another assertion G if:

(1)$I \Rightarrow F$

(2)$G \wedge F \wedge T \Rightarrow F'$

$\boldsymbol{inductive\ strengthening}$:

An inductive strengthening of a safety property P is a formula F such that:

(1)$I \Rightarrow F \wedge P$

(2)$F \wedge P \wedge T \Rightarrow F \wedge P'$

SAT-Based Model Checking Without Unrolling

```
{ @post: rv iff P is S−invariant }
bool prove():
  if sat(I ∧ ¬P) or sat(I ∧ T ∧ ¬P′):
    return false
  F_0 := I, clauses(F_0) := ∅
  F_i := P, clauses(F_i) := ∅ for all i > 0
  for k := 1 to ...:
    { @rank: 2^|x̄| + 1
      @assert (A):
        (1) ∀ i ≥ 0, I ⇒ F_i
        (2) ∀ i ≥ 0, F_i ⇒ P
        (3) ∀ i > 0, clauses(F_{i+1}) ⊆ clauses(F_i)
        (4) ∀ 0 ≤ i < k, F_i ∧ T ⇒ F′_{i+1}
        (5) ∀ i > k, |clauses(F_i)| = 0 }
    if not strengthen(k):
      return false
    propagateClauses(k)
    if clauses(F_i) = clauses(F_{i+1}) for some 1 ≤ i ≤ k
      return true
```

```
bool strengthen(k : level):
  try:
    while sat(F_k ∧ T ∧ ¬P′):
      { @rank: 2^|x̄|
        @assert (B):
          (1) A.1−4
          (2) ∀ c ∈ clauses(F_{k+1}), F_k ∧ T ⇒ c′
          (3) ∀ i > k + 1, |clauses(F_i)| = 0 }
      s := the predecessor extracted from the witness
      n := inductivelyGeneralize(s, k − 2, k)
      pushGeneralization({(n + 1, s)}, k)
      { @assert (C): s ⊭ F_k }
    return true
  except Counterexample:
    return false
```

```
void propagateClauses(k : level):
  for i := 1 to k:
    { @assert: ∀ 0 ≤ j < i, ∀c ∈ clauses(F_j), if F_j ∧ T ⇒ c′ then c ∈ F_{j+1} }
    for each c ∈ clauses(F_i):
      { @assert: pre }
      if not sat(F_i ∧ T ∧ ¬c′):
        clauses(F_{i+1}) := clauses(F_{i+1}) ∪ {c}
```

SAT-Based Model Checking Without Unrolling

```
bool strengthen(k : level):
  try:
    while sat(F_k ∧ T ∧ ¬P'):
      {@rank: 2^|x̄|
       @assert (B):
        (1) A.1−4
        (2) ∀ c ∈ clauses(F_{k+1}), F_k ∧ T ⇒ c'
        (3) ∀ i > k+1, |clauses(F_i)| = 0 }
      s := the predecessor extracted from the witness
      n := inductivelyGeneralize(s, k−2, k)
      pushGeneralization({(n+1, s)}, k)
      {@assert (C): s ⊭ F_k }
    return true
  except Counterexample:
    return false
```

```
level inductivelyGeneralize(s : state, min : level, k : level):
  if min < 0 and sat(F_0 ∧ T ∧ ¬s ∧ s'):
    raise Counterexample
  for i := max(1, min+1) to k:
    {@assert:
      (1) B
      (2) min < i ≤ k
      (3) ∀ 0 ≤ j < i, ¬s is inductive relative to F_j }
    if sat(F_i ∧ T ∧ ¬s ∧ s'):
      generateClause(s, i−1, k)
      return i−1
  generateClause(s, k, k)
  return k
```

i=k

i=k-1

or i<k-1

```
void pushGeneralization(states : (level, state) set, k : level):
  while true:
    {@rank: (k+1)2^|x̄|
     @assert (D):
      (1) pre
      (2) ∀ (i,q) ∈ states_prev, ∃j ≥ i, (j,q) ∈ states }
    (n, s) := choose from states, minimizing n
    if n > k: return
    if sat(F_n ∧ T ∧ s'):
      p := the predecessor extracted from the witness
      {@assert (E): ∀ (i,q) ∈ states, p ≠ q }
      m := inductivelyGeneralize(p, n−2, k)
      states := states ∪ {(m+1, p)}
    else:
      m := inductivelyGeneralize(s, n, k)
      {@assert (F): m+1 > n }
      states := states \ {(n, s)} ∪ {(m+1, s)}
```

```
void generateClause(s : state, i : level, k : level):
  c := subclause of ¬s that is inductive relative to F_i
  for j := 1 to i+1:
    {@assert: B }
    clauses(F_j) := clauses(F_j) ∪ {c}
```

SAT-Based Model Checking Without Unrolling

Example

Consider the contrived transition system $S : (\bar{x}, I, T)$ with variables $\bar{x} = \{x_0, x_1, x, y_0, y_1, y, z\}$, initial condition

$$I : x_0 \wedge \neg x_1 \wedge x \wedge (y_0 = \neg y_1) \wedge y \wedge z \ , \quad \text{the safety assertion } P : z$$

and transition relation

$$T : \begin{bmatrix} (x_0' = \neg x_0) \wedge (x_1' = \neg x_1) \wedge (x' = x_0 \vee x_1) \\ \wedge (y_0' = x \wedge \neg y_0) \wedge (y_1' = x \wedge \neg y_1) \wedge (y' = y_0 \vee y_1) \\ \wedge (z' = x \wedge y) \end{bmatrix} .$$

1. F_0 is initialized to I, each of $F_1, F_2, F_3, \ldots$ to P, and k to 1.
2. $F_1 \wedge T \wedge \neg P'$ is satisfiable.

   $\neg P$- predecessor $s_1 : \neg x_0 \wedge \neg x_1 \wedge \neg x \wedge \neg y_0 \wedge \neg y_1 \wedge \neg y \wedge z$

   $F_1 \wedge \neg s_1 \wedge T$ implies $\neg s_1'$

   Inductive generalization of s_1 relative to F_1 yields the clause $c_1 : x_0 \vee x$.
   (1) $c_1 \subset \neg s_1$,
   (2) c_1 is inductive relative to F_1.

SAT-Based Model Checking Without Unrolling

Example

3.$F_1 \land T \land \neg P'$ is still satisfiable.

$\neg P - predecessor\ s_2 : x_0 \land \neg x_1 \land \neg x \land \neg y_0 \land \neg y_1 \land \neg y \land z.$

$\neg s_2$ is inductive relative to F_1.

Inductive generalization yields from $\neg s_2$ the clause $c_2 : x_1 \lor x$.

c_2 also inductive relative to F_1.

4.$F_1 \land T \land \neg P'$ is still satisfiable.

$\neg P - predecessor\ s_3 : x_0 \land x_1 \land \neg x \land y_0 \land y_1 \land \neg y \land z.$

s_3 has predecessor $s_4 : \neg x_0 \land \neg x_1 \land x \land \neg y_0 \land \neg y_1 \land y \land z.$

$\neg s_3$ is not inductive relative to F_1.

However,it is inductive relative to F_0,

inductive generalization yields from $\neg s_3$ the clause $c_3 : \neg y_0 \lor y$.

5.c_3 does not exclude $s_4 : \neg s_3$ is still not inductive relative to F_1.

s_4 has a predecessor $s_5 : x_0 \land x_1 \land x \land y_0 \land y_1 \land y \land z.$

s_4 is inductive relative to F_0,and inductive generalization yields $c_4 : x_0 \lor x_1$.

6.$\neg s_3$ is now inductive relative to F_1,

inductive generalization yields $c_5 : \neg x_0 \lor \neg x_1$.

7.$\neg s_4$ is now inductive relative to F_1,

inductive generalization yields again the clause $c_4 : x_0 \lor x_1$.

8.$F_1 \land T \land \neg P'$ is still satisfiable.

$\neg P - predecessor\ s_6 : x_0 \land \neg x_1 \land \neg x \land y_0 \land y_1 \land \neg y \land z.$

s_6 is inductive relative to F_1,inductive generalization yields the clause $c_6 : x$

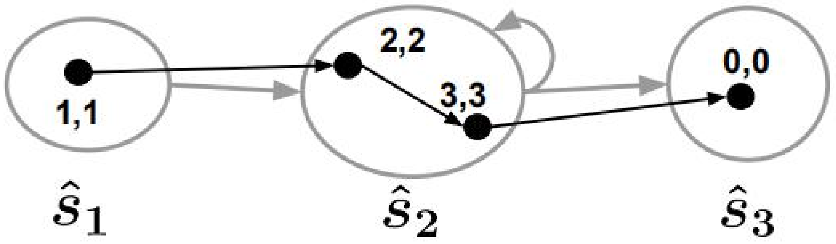
AVR using IC3 with Syntax-guided Abstraction

Syntax-guided Abstraction

Example 1: Let $\mathcal{P} = \langle \{u,v\}, (u = 1) \wedge (v = 1), (u' = ite(u < v, u+v, v+1)) \wedge (v' = v+1), ((u+v) \neq 1) \rangle$, where u, v are k-bit wide. $\mathcal{P}$ has 1 predicate $(u < v)$ and 5 words $(1, u, v, u+v, v+1)$. Consider a concrete state $s := (u,v) = (1,2)$
partition assignment $\hat{s} := (u < v) \wedge \{\, 1, u \mid v \mid u+v, v+1 \,\}$

Given a formula φ the solution of φ in the abstract domain is expressed as a partition assignment on terms in φ
for a partition assignment $\hat{s}$, $\hat{s} \models \varphi$ *iff* there exists a bitvector assignment s such that $s \models \varphi$ and $\hat{s} = \alpha(\varphi, s)$
α is the abstraction function that converts a bitvector assignment s to a partition assignment on terms in φ.

Example 2: Consider $\mathcal{P}$ from Example 1. Let $k = 2$. Consider the formula $\varphi = P \wedge T \wedge \neg P'$ and a satisfying concrete solution $s := (u, v, u', v') = (0, 2, 2, 3)$. Terms in φ evaluate as $(u < v, u+v, v+1, u'+v') = (\top, 2, 3, 1)$ under s, resulting in the abstract solution to be $\hat{s} := (u < v) \wedge \{\, u \mid 1, u'+v' \mid v, u+v, u' \mid v+1, v' \,\}$.



Example 3: Consider $\hat{s}$ from Example 2. $\hat{s}$ can be projected on the projection set $\sigma = \{+, 1, u', v'\}$ to get a *partial* abstract solution representing the destination states as $\hat{s}|_\sigma := \{\, 1, u'+v' \mid u' \mid v' \,\}$. The corresponding cube representation is $cube(\hat{s}|_\sigma) = ((u'+v') = 1) \wedge (u' \neq 1) \wedge (v' \neq 1) \wedge (u' \neq v')$.

$\hat{s}_1 := \neg(u < v) \wedge \{\, 1, u, v \mid u+v, v+1 \,\}$
$\hat{s}_2 := \neg(u < v) \wedge \{\, 1 \mid u, v \mid u+v \mid v+1 \,\}$
$\hat{s}_3 := \neg(u < v) \wedge \{\, 1, v+1 \mid u, v, u+v \,\}$

IC3 with Syntax-guided Abstraction (IC3+SA)

- How to **generalize a satisfiable query** from a particular solver solution?
- How to **refine** spurious counterexamples?

Algorithm 1 Syntax-guided Generalization

1. **procedure** GENERALIZE($\hat{s}$, c') ▷ $\hat{s}$ is a particular abstract solution, c' is a destination cube
2. $\sigma \leftarrow \sigma_{refine}$ ▷ initialize projection set (initially $\sigma_{refine} = \emptyset$)
3. `JustifyCOI`($\hat{s}$, c', σ) ▷ build projection set σ
4. $\sigma \leftarrow \sigma - X'$ ▷ get rid of next state symbols
5. $\hat{s}|_\sigma \leftarrow$ `Project`($\hat{s}$, σ) ▷ project $\hat{s}$ on σ
6. **return** $cube(\hat{s}|_\sigma)$ ▷ convert to a cube and return

Example 6: Let $\mathcal{P} = \langle\{u, v, w\}, (u = 1) \wedge (v = 1) \wedge (w = 1), (u' = ite((u < v) \vee (v < w), u + v, v + 1)) \wedge (v' = v + 1) \wedge (w' = w + 1), ((u + v) \neq 1))\rangle$, with u, v, w being 3-bit wide. Consider the following query and its particular solution:

$F_1 = P \qquad \varphi = F_1 \wedge T \wedge \neg P'$

$Q_1 := SAT\ ?\ [\varphi]$ gives SAT with solution s

$s = (u, v, w, u', v', w') = (0, 4, 2, 4, 5, 3) \qquad \hat{s} = \alpha(\varphi, s)$

$\hat{s} = (u < v) \wedge \neg(v < w) \wedge \{\, u \mid 1, u' + v' \mid w \mid w + 1 \mid v, u + v, u' \mid v + 1, v' \,\}$

`Generalize`($\hat{s}$, $\neg P'$) creates the generalized cube c_1 as follows:

$$\begin{aligned}\sigma &= \{\, +, u', v', 1, <, u, v \,\} - \{\, u', v', w' \,\} \\ &= \{\, +, <, u, v, 1 \,\} \\ c_1 &= cube(\hat{s}|_\sigma) = (u < v) \wedge \{\, u \mid 1 \mid v, u + v \mid v + 1 \,\}\end{aligned}$$

IC3 with Syntax-guided Abstraction (IC3+SA)

Generalization of a Satisfiable Query

```
procedure JustifyCOI(ŝ, φ, σ)                              ▷ φ is a FOL expression, σ is passed by reference
    if φ is a conditional operation then                   ▷ if φ is an if-then-else expression
        ⟨cond, v_⊤, v_⊥⟩ ← BreakCondition(φ)               ▷ get condition and arguments
        JustifyCOI(ŝ, cond, σ)
        val ← Evaluate(cond, ŝ)                            ▷ evaluate cond under ŝ
        JustifyCOI(ŝ, (val = ⊤) ? v_⊤ : v_⊥, σ)            ▷ recurse only on the relevant branch
    else if φ is a logical operation then
        val ← Evaluate(φ, ŝ)                               ▷ evaluate φ under ŝ
        if IsControlling(val, φ) then                      ▷ if assigned a controlling value (⊥ for ∧, ⊤ for ∨)
            JustifyCOI(ŝ, GetControlling(φ, ŝ), σ)         ▷ recurse only on controlling arg.
        else
            for each a ∈ Argument(φ) do
                JustifyCOI(ŝ, a, σ)
    else
        for each a ∈ Argument(φ) do
            JustifyCOI(ŝ, a, σ)
        if φ is a next state variable then
            JustifyCOI(ŝ, GetRelation(φ), σ)               ▷ get the next state relation for φ from T
        Add symbol(φ) to σ                                 ▷ add symbol of φ to the projection set
```

IC3 with Syntax-guided Abstraction (IC3+SA)

Refinement

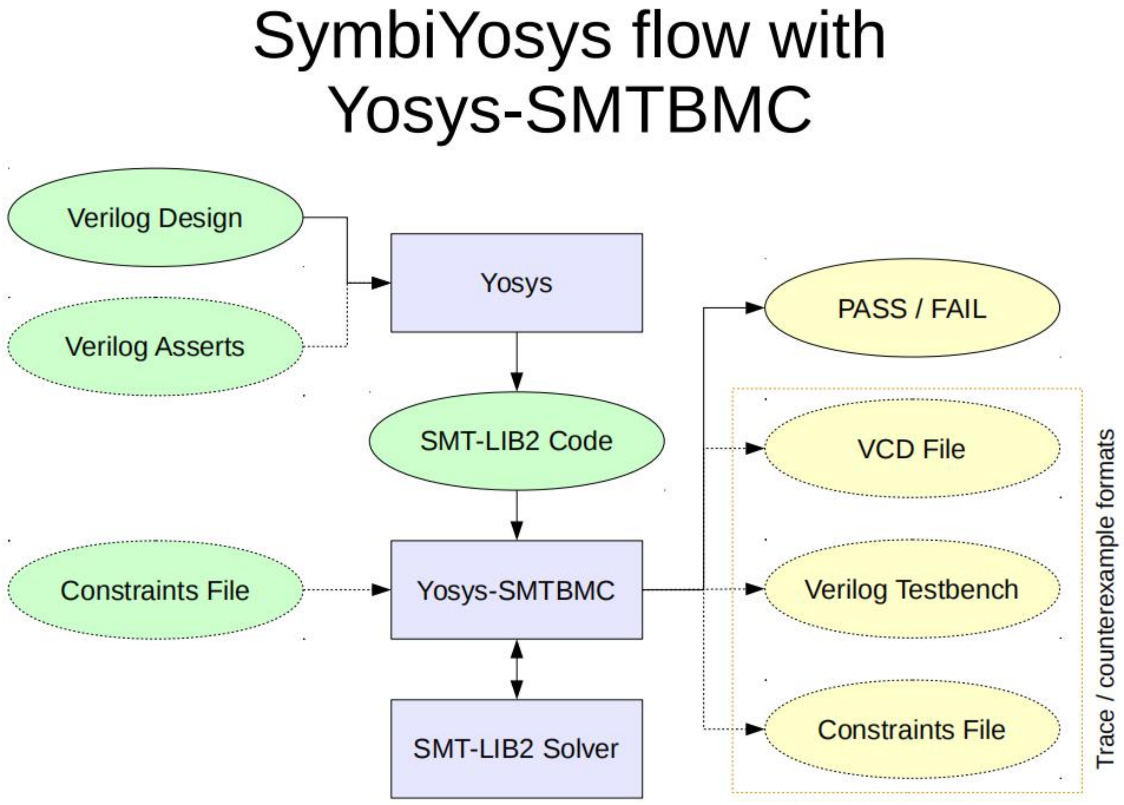
Algorithm 2 Refinement of SA

procedure $\textsc{Refine}(\hat{\mathcal{C}})$
 $p_0 \leftarrow I$
 for $i = 1$ to n **do**
  $\psi_i \leftarrow p_{i-1} \land c_{i-1} \land T \land c'_i$
  if `SAT ?` $[\psi_i]$`: solution` s **then**
   $p_i \leftarrow$ `PostImage`$(p_{i-1} \land c_{i-1},\ s)$ ▷ compute image$(p_{i-1} \land c_{i-1})$ under s
  else   ▷ i.e. $\mathcal{C}$ is spurious
   $m \leftarrow$ `MUS`(ψ_i) ▷ find MUS for the UNSAT query
   $m \leftarrow$ `Substitute`(m) ▷ eliminate symbolic constants
   $\Phi \leftarrow \neg m$
   $T \leftarrow T \land \Phi$ ▷ conjoin axiom to $\hat{T}$
   $\sigma_{new} \leftarrow$ `symbols(NewTerms(`Φ`))`   ▷ find symbols in new terms
   $\sigma_{refine} \leftarrow \sigma_{refine} \cup \sigma_{new}$ ▷ add permanent symbols
   return $\emptyset$
 return $\mathcal{C}$ ▷ i.e. $\mathcal{C}$ is a true counterexample

some tools

SymbiYosys:

- smtbmc engine
- aiger engine
- abc engine



Verilog2SMV :

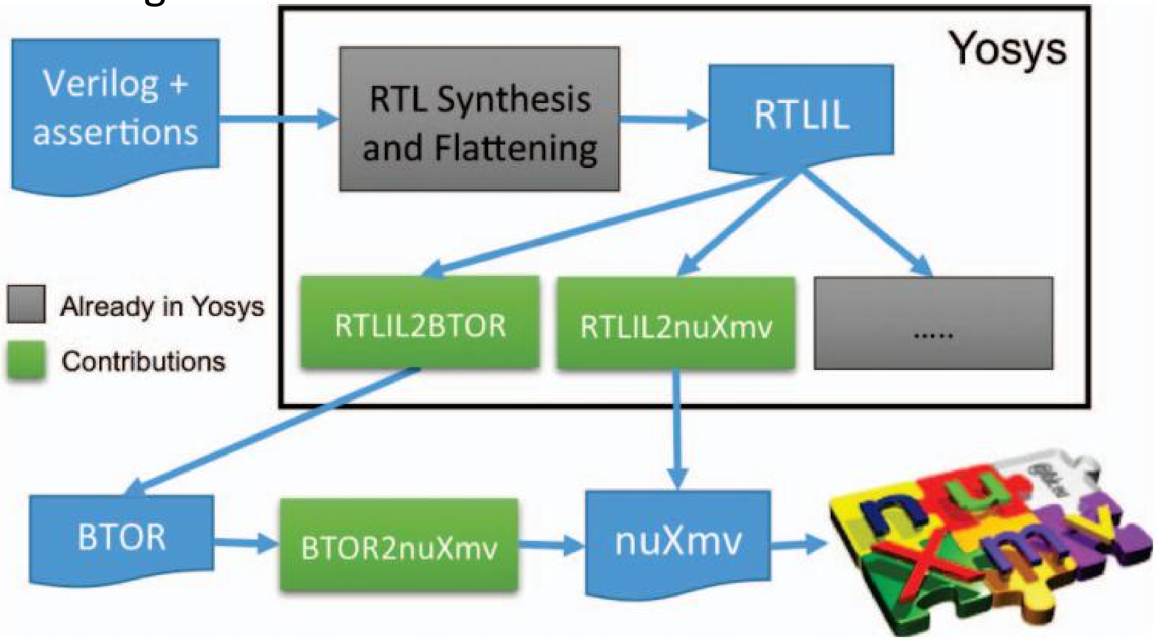


Fig. 1. *verilog2smv* architecture and verification tool-chain

BtorMC
CoSA2: CoreIR Symbolic Analyzer 2
CoNPS-btormc-THP
...