SAT-based Verification Methods and Applications in Hardware Verification

Aarti Gupta
agupta@nec-labs.com
NEC Laboratories America
Princeton, U.S.A.

Acknowledgements: Pranav Ashar, Malay Ganai, Zijiang Yang, Chao Wang, Akira Mukaiyama, Kazutoshi Wakabayashi

Outline

- ❑ Background
  - – SAT Solvers
- ❑ SAT-based Verification Methods
  - – Methods for Finding Bugs
    - ➢ Bounded Model Checking (and Variations)
  - – Methods for Finding Proofs
    - ➢ Induction
    - ➢ Proof-based Abstraction
    - ➢ Unbounded Model Checking
- ❑ NEC’s VeriSol Hardware Verification Platform
  - – Interplay of Engines
  - – NEC’s High Level Synthesis Design Framework
  - – Back-end for NEC’s F-Soft Software Verification Platform

- ❑ Please see related article for further details
  - – A. Gupta, M. K. Ganai, C. Wang. SAT-based Verification Methods and Applications in Hardware Verification, in Formal Methods for Hardware Verification, SFM 2006, Lecture Notes in Computer Science, Vol. 3965, May 2006

Disclaimer: No exhaustive coverage!

What is SAT?

❑ SAT : Boolean Satisfiability Problem

– **Given a Boolean formula, find an assignment to the variables such that the formula evaluates to true, or prove that no such assignment exists**

– **Examples:**

➢ **F = ab + cd is satisfiable (c=1,d=1 is a solution)**

➢ **G = abc(b xor c) is unsatisfiable (no solution exists for a,b,c)**

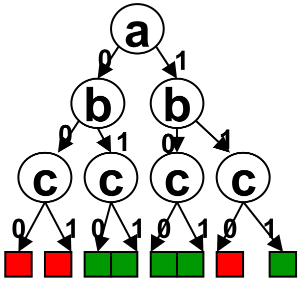
❑ Complexity of SAT Problem

– **NP-Complete Problem**

S. A. Cook, The complexity of theorem proving procedures, Proceedings, Third Annual ACM Symp. on the Theory of Computing,1971, 151-158

– **For *n* variables, examine *2^n* Boolean combinations of input variables**

H = (a + b)(a’ + b’ + c)



SAT Applications

- □ Electronic Design Automation (EDA)
  - *Verification*: Combinational equivalence checking, *Property checking*
  - Testing
  - Logic synthesis
  - FPGA routing
  - Path delay analysis
  - …
- □ AI
  - Knowledge base deduction
  - Automatic theorem proving

- □ Some classes of SAT problems are easier to solve
  - 2-SAT, Horn SAT
  - However, typical applications do not fall into these classes
  - Need a general purpose SAT solver
  - For verification, it is also useful to have a complete SAT solver

SAT Problem Representation

❑ Conjunctive Normal Form (CNF)

- Formula is a conjunction of clauses
- Clause is a disjunction of literals
- Literal is a variable or its negation

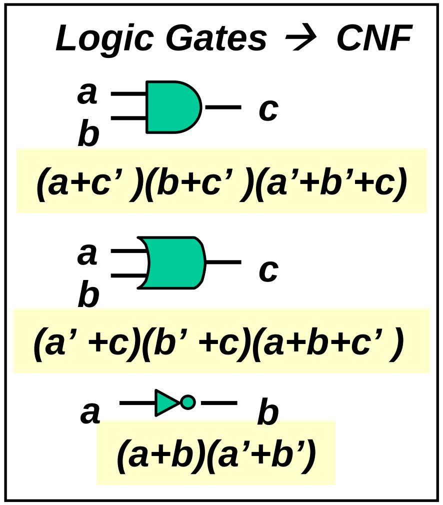
- Example: F =   (a + b) (a’ + b’ + c)

- For a formula to be satisfiable, each clause should be satisfied
- Simple representation leads to more efficient data structures

❑ Logic circuit representation

- Circuits have structural and direction information
- Circuit to CNF conversion is linear in size

❑ **Progress in both CNF-based and Circuit-based SAT Solvers**


Logic Gates → CNF

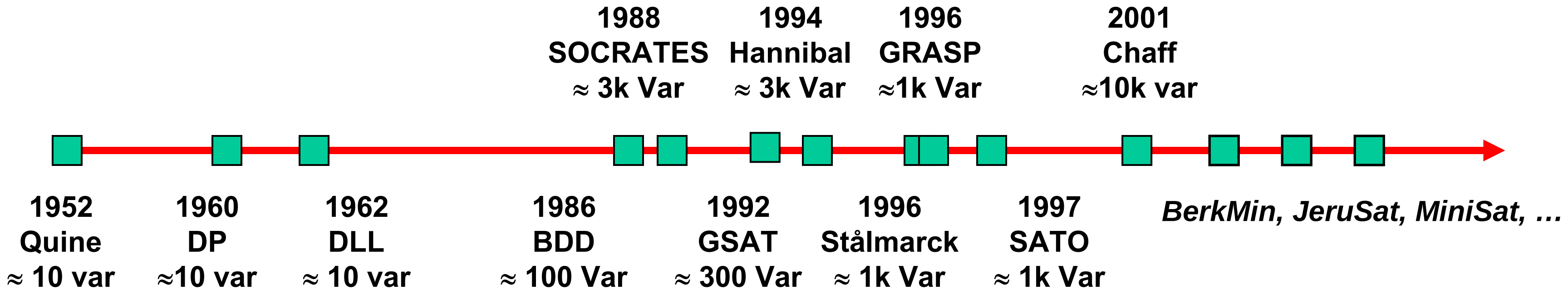
(a+c’)(b+c’)(a’+b’+c)

(a’ +c)(b’ +c)(a+b+c’)

(a+b)(a’+b’)

The Timeline

(Source: Prof. Sharad Malik, Invited Talk at CAV/CADE ’02)



1952
Quine
≈ 10 var

1960
DP
≈10 var

1962
DLL
≈ 10 var

1986
BDD
≈ 100 Var

1988
SOCRATES
≈ 3k Var

1992
GSAT
≈ 300 Var

1994
Hannibal
≈ 3k Var

1996
GRASP
≈1k Var

1996
Stålmarck
≈ 1k Var

1997
SATO
≈ 1k Var

2001
Chaff
≈10k var

BerkMin, JeruSat, MiniSat, …

SAT Solver: DLL/DPLL Algorithm

M. Davis, G. Logemann and D. Loveland, “A Machine Program for Theorem-Proving", Communications of ACM, Vol. 5, No. 7, pp. 394-397, 1962

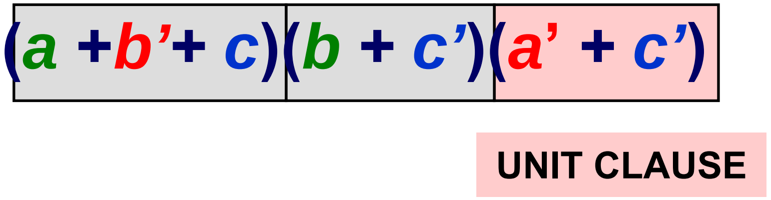
- □ **Basic framework for many modern SAT solvers**
  - **Branch and backtrack search algorithm**
  - **Prunes the search space by using a deductive procedure called BCP**
    - ➢ **Better than 2^n exhaustive search**

Boolean Constraint Propagation (BCP)

Definitions:

- *Unit clause*: An unsatisfied clause is a unit clause if it has exactly one unassigned literal
- *Implication*: A variable is forced to be assigned to be True or False based on previous assignments to other variables

a = T, b = T, c is unassigned



$(a + b' + c)(b + c')(a' + c')$

UNIT CLAUSE

Satisfied Literal

Unsatisfied Literal

Unassigned Literal

- *Unit Clause Rule*: The unassigned literal in a unit clause is implied

  c' is assigned true, i.e. c = F

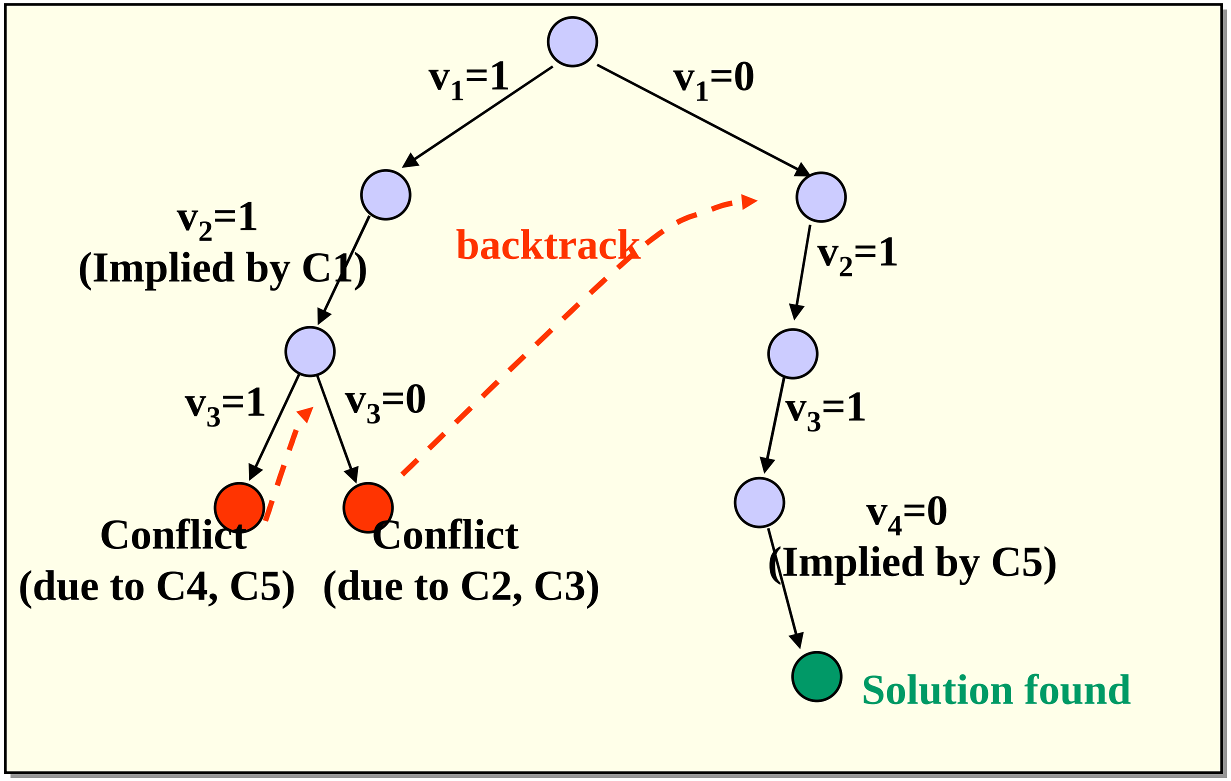
- Boolean Constraint Propagation (BCP)
  - Iteratively apply the unit clause rule until there is no unit clause available
  - *Prunes search by saving decisions on implied variables*

DPLL Algorithm Example

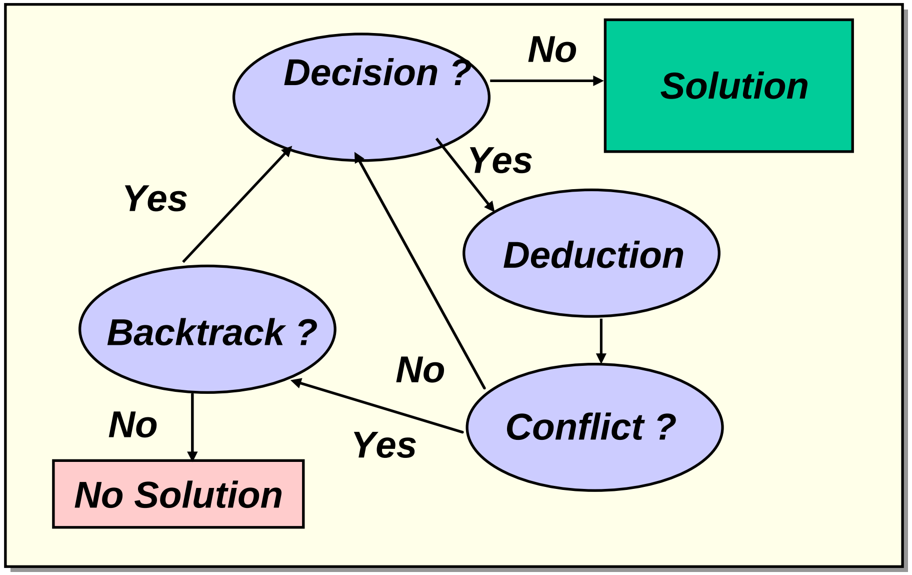
Conjunctive Normal Form (CNF)

CNF ***Clause*** ***Literal***

$$(\bar{v}_1 + v_2)(\bar{v}_1 + v_3 + v_4)\boxed{(\bar{v}_2 + v_3 + \bar{v}_4)}(\bar{v}_1 + \bar{v}_3 + v_4)(\bar{v}_2 + \boxed{\bar{v}_3} + \bar{v}_4)$$



DPLL-Based SAT Solvers



- ❑ **Main Engines**
  - **Decision: for choosing which variable/value to branch on**
  - **Deduction: for performing BCP and checking conflicts**
  - **Diagnosis: for conflict analysis and backtracking**
- ❑ **Modern SAT Solvers: Improvements in these engines**
  - **Grasp, SATO, Chaff, BerkMin, … (CNF-Based Solvers)**

Conflict Analysis Example

x1 + x4

x1 + x3’ + x8’

x1 + x8 + x12

x2 + x11

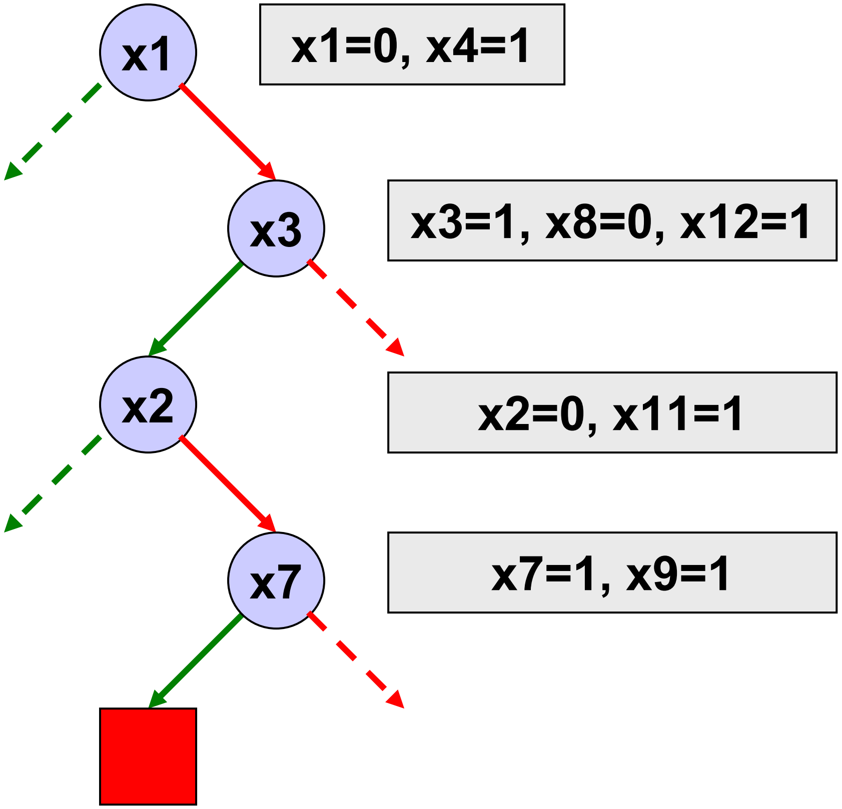
x7’ + x3’ + x9

x7’ + x8 + x9’

x7 + x8 + x10’

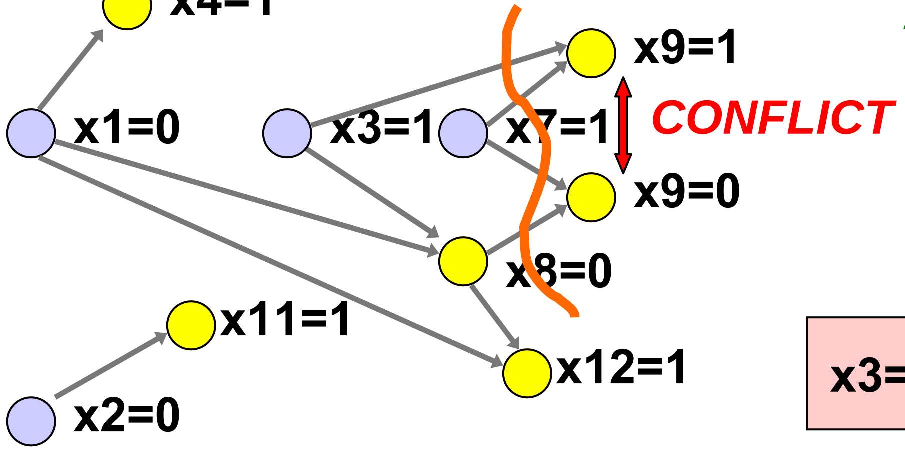
x7 + x10 + x12’

Decision Tree



$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict

Implication Graph

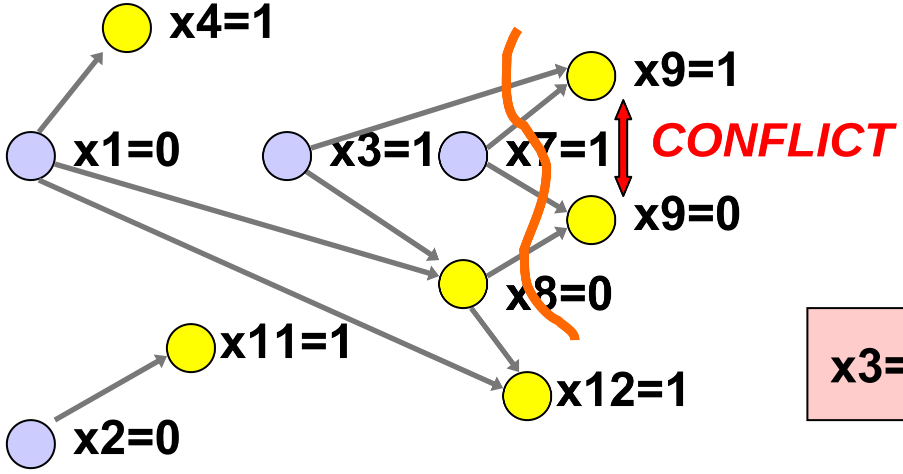


Conflict Analysis Example

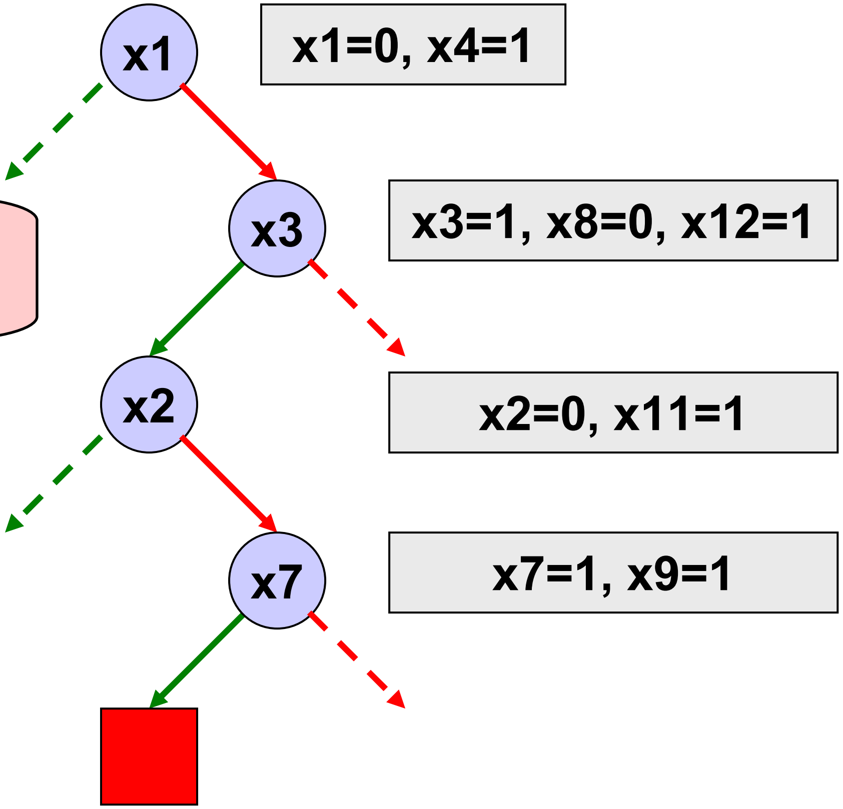
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’
x3’ + x7’ + x8

Conflict-driven Learning:
Add Conflict Clause

Implication Graph



Decision Tree



x1=0, x4=1

x3=1, x8=0, x12=1

x2=0, x11=1

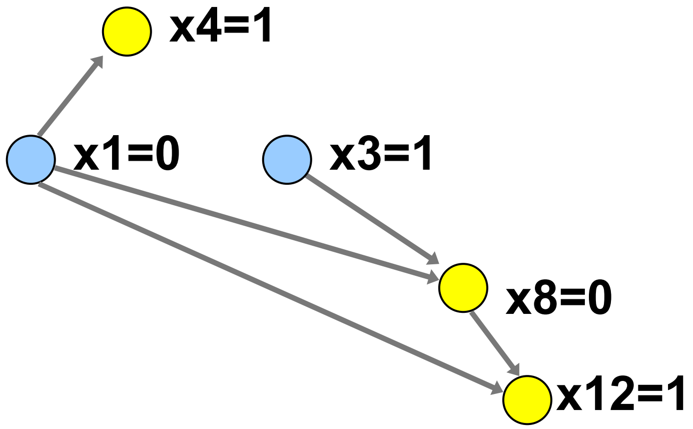
x7=1, x9=1

x3=1∧x7=1∧x8=0 → conflict

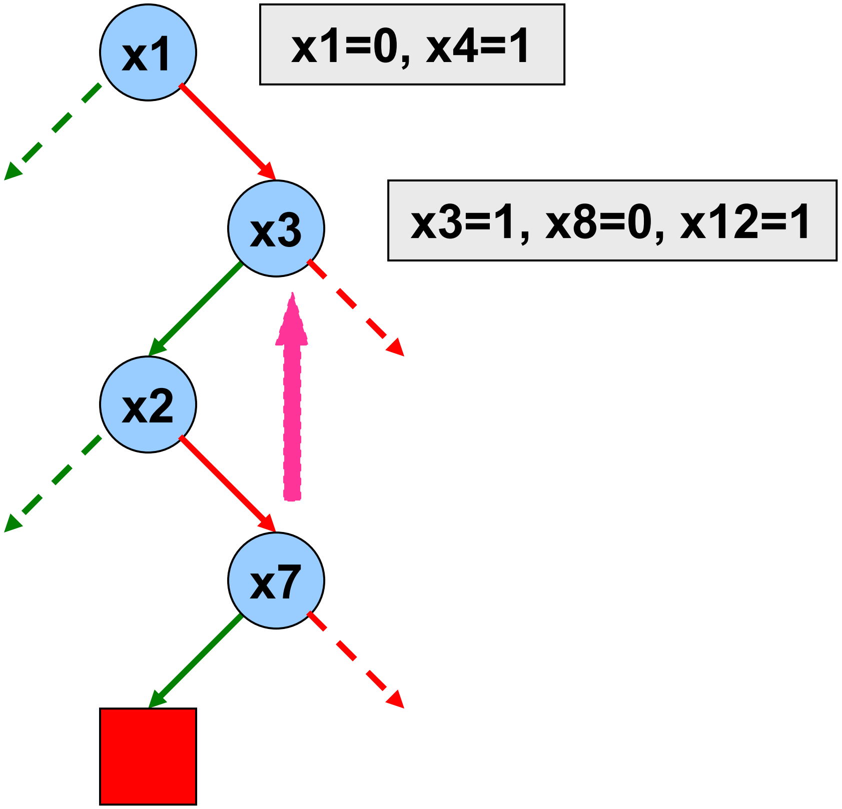
Conflict Analysis Example

x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’
x3’ + x7’ + x8

Implication Graph



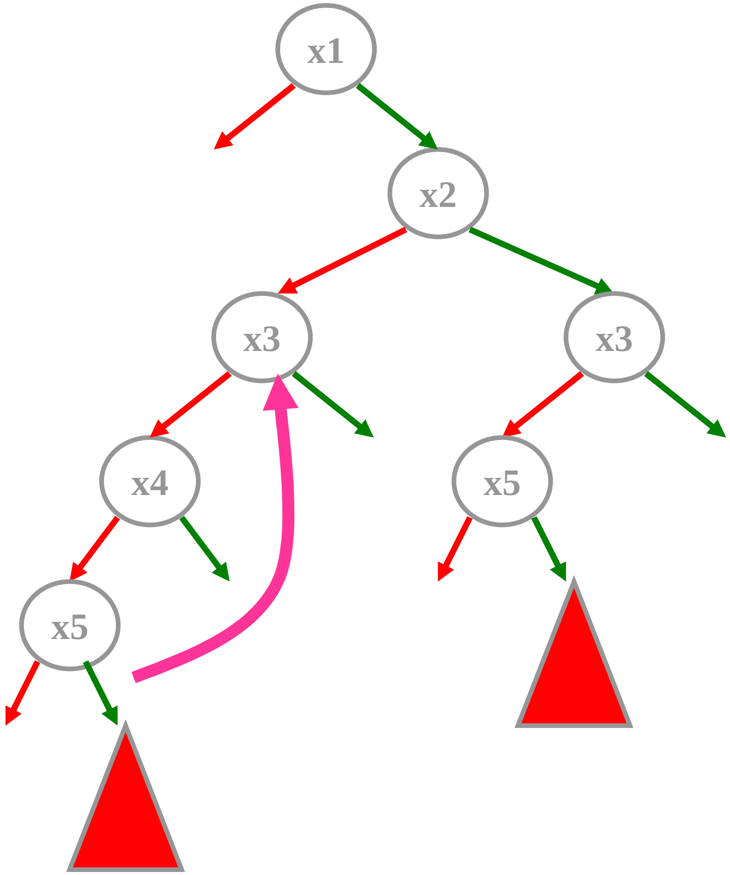
Decision Tree



Non-chronological Backtracking:
Backtrack from level 4 to level 2, not to level 3

Conflict Analysis Benefits

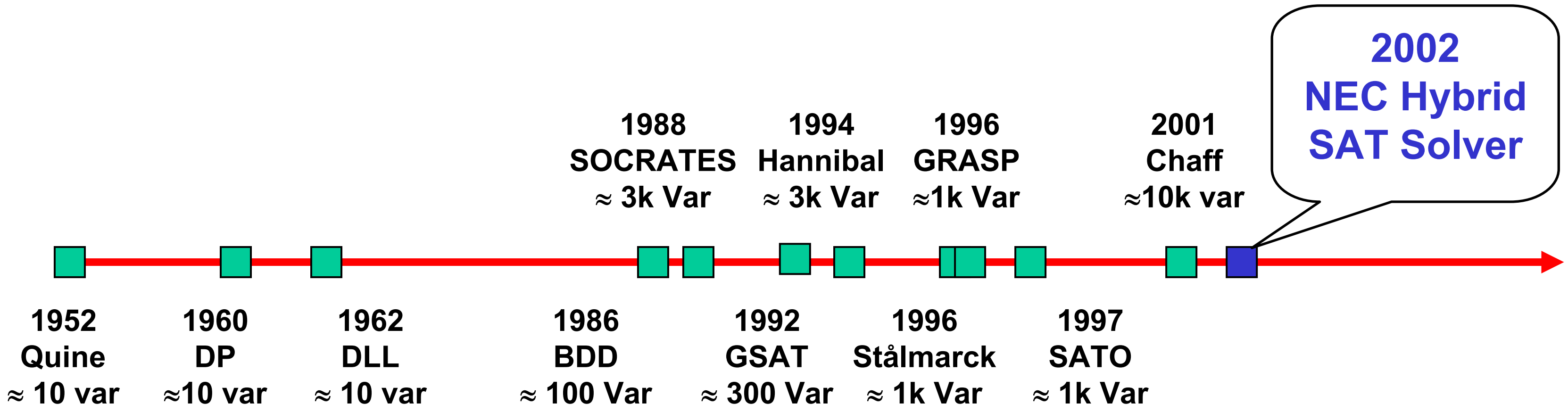
- Conflict analysis helps to *prune search space* by:
  - Avoiding same conflict using conflict-driven learning
  - Allowing non-chronological backtracking



Conflict clause: x1’+x3+x5’

The Timeline

(Source: Prof. Sharad Malik, Invited Talk at CAV/CADE ’02)



1952
Quine
≈ 10 var

1960
DP
≈10 var

1962
DLL
≈ 10 var

1986
BDD
≈ 100 Var

1988
SOCRATES
≈ 3k Var

1992
GSAT
≈ 300 Var

1994
Hannibal
≈ 3k Var

1996
GRASP
≈1k Var

1996
Stålmarck
≈ 1k Var

1997
SATO
≈ 1k Var

2001
Chaff
≈10k var

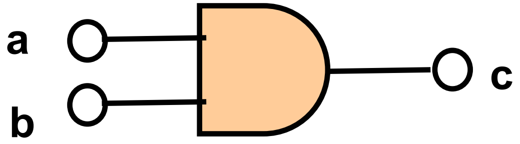
2002
NEC Hybrid
SAT Solver

NEC Hybrid (CNF+Circuit) SAT Solver

- ❑ **Circuit-domain SAT applications**
  - **ATPG, Equivalence Checking, BMC, …**

- ❑ **Combines the strengths of CNF- and Circuit-based SAT solvers**
  - **Better deduction engine**
    - ➢ **BCP: 80% of total SAT time**
    - ➢ **Handles small (circuit) and large (conflict) clauses differently**
  - **Better decision engine**
    - ➢ **Uses circuit-based information efficiently to prune search space**
      - For example, does not need to make decisions in unobservable parts
    - ➢ **Combines circuit frontier-based heuristic with Chaff's VSIDS decision heuristic**

BCP on Gate Clauses

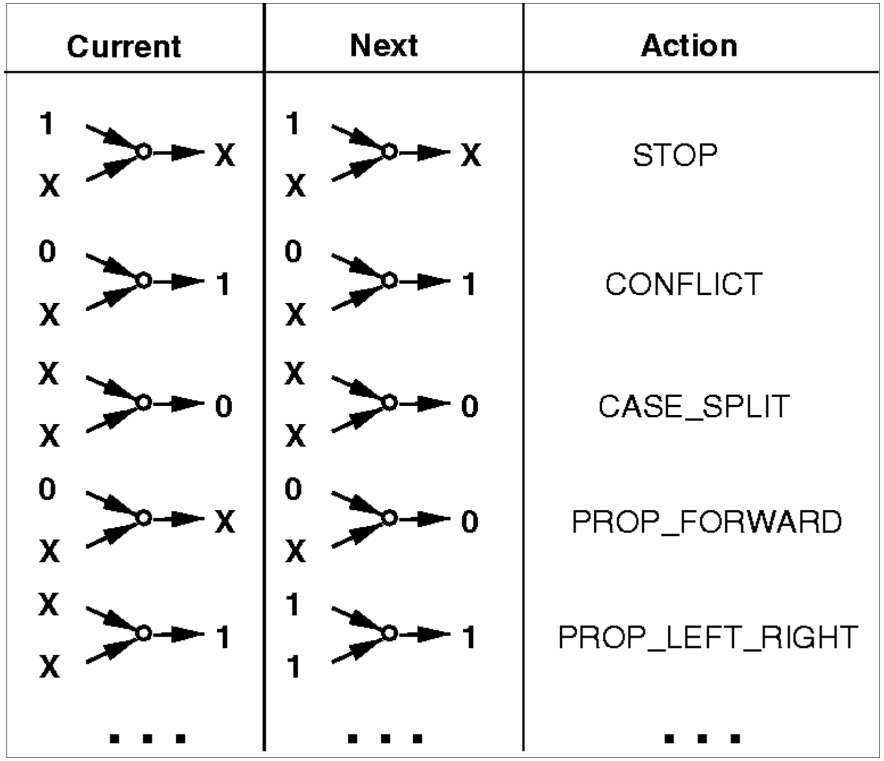


(a + c’) (b + c’) (a’ + b’ + c)

c=1 => a=1, b=1

- CNF-based
  - update of 2 clauses
- Circuit-based
  - Single table lookup

2-input AND Lookup Table



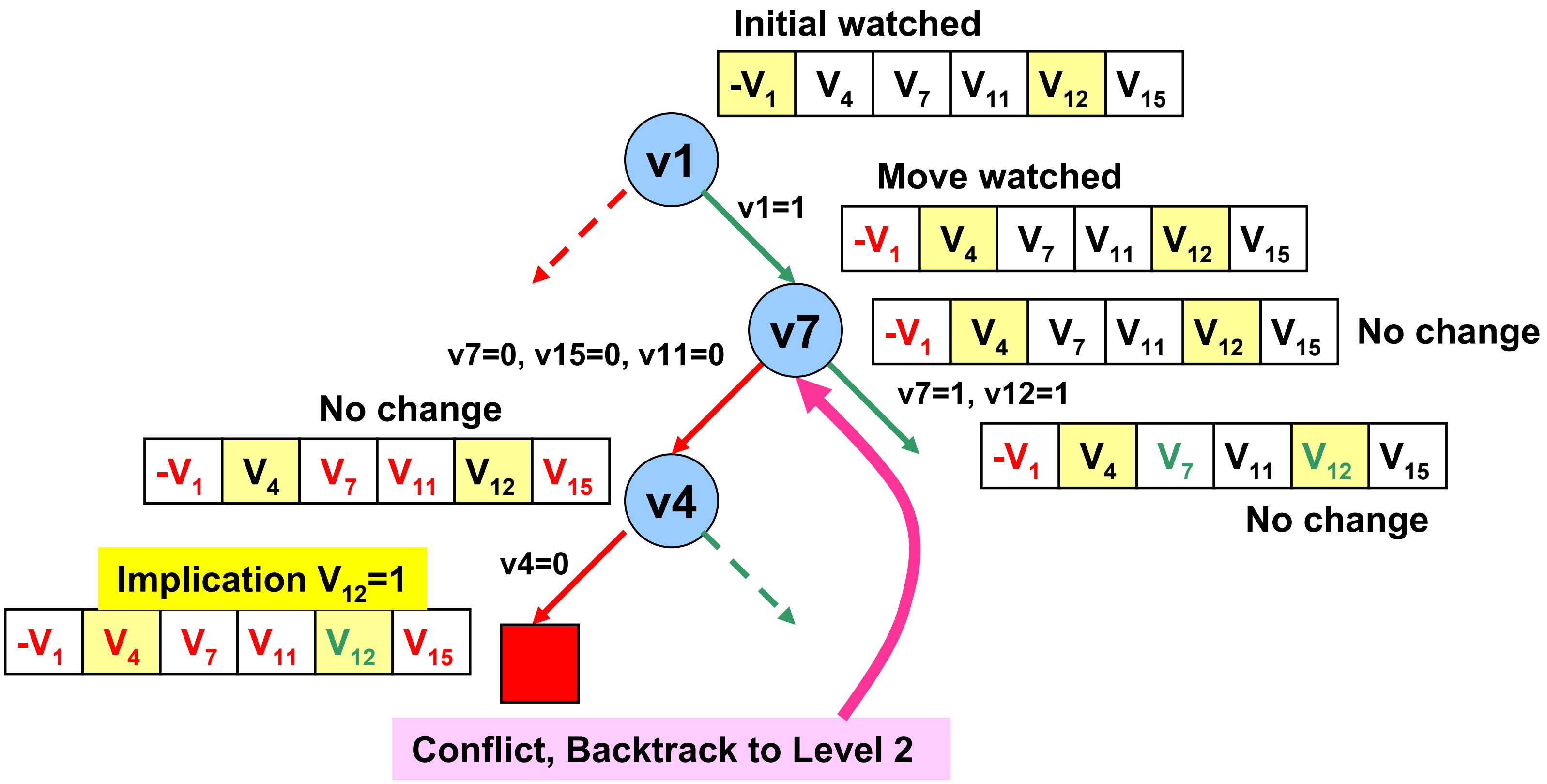
Current	Next	Action
1 X → X	1 X → X	STOP
0 X → 1	0 X → 1	CONFLICT
X X → 0	X X → 0	CASE_SPLIT
0 X → X	0 X → 0	PROP_FORWARD
X X → 1	1 1 → 1	PROP_LEFT_RIGHT
. . .	. . .	. . .

-- Kuehlmann et al. DAC ‘01

Use fast table lookup on Gate Clauses

Chaff BCP Example

- **Lazy 2-literal watching scheme** - *Moskewicz et al. DAC '01*
  - Only "two" literals (non-zero) are watched per clause
  - Clause state updated when watch pointers coincide
  - Constant time variable-unassignment during backtracking



Initial watched

$-V_1$	V_4	V_7	V_{11}	V_{12}	V_{15}

v1

v1=1

Move watched

$-V_1$	V_4	V_7	V_{11}	V_{12}	V_{15}

v7

$-V_1$	V_4	V_7	V_{11}	V_{12}	V_{15}

No change

v7=1, v12=1

$-V_1$	V_4	V_7	V_{11}	V_{12}	V_{15}

No change

v7=0, v15=0, v11=0

No change

$-V_1$	V_4	V_7	V_{11}	V_{12}	V_{15}

v4

v4=0

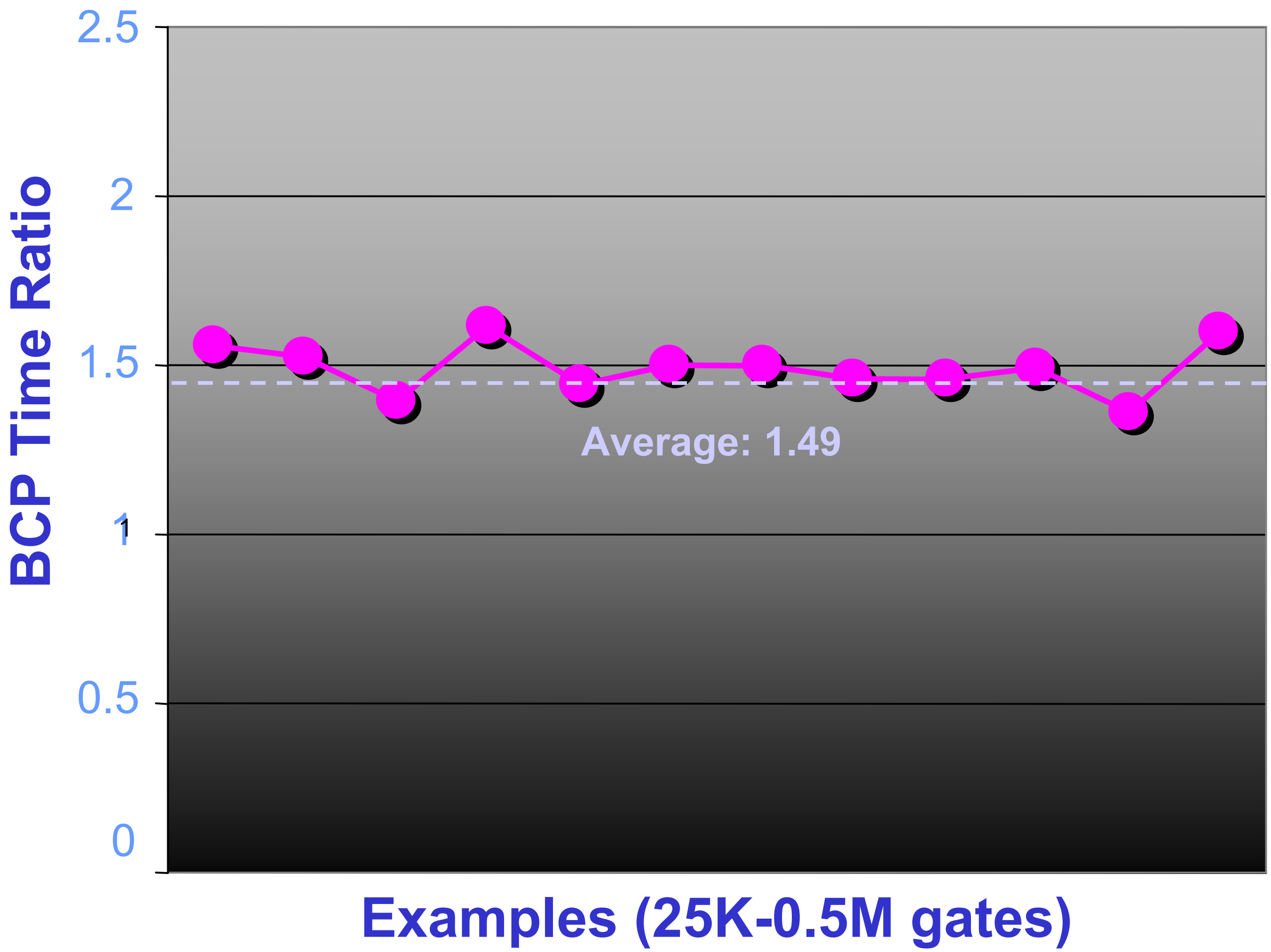
Implication V_{12}=1

$-V_1$	V_4	V_7	V_{11}	V_{12}	V_{15}

Conflict, Backtrack to Level 2

BCP Results (gate clauses only)

BCP Time Comparison (*per million implications*)



CNF: **BCP on Gate Clauses as CNF**

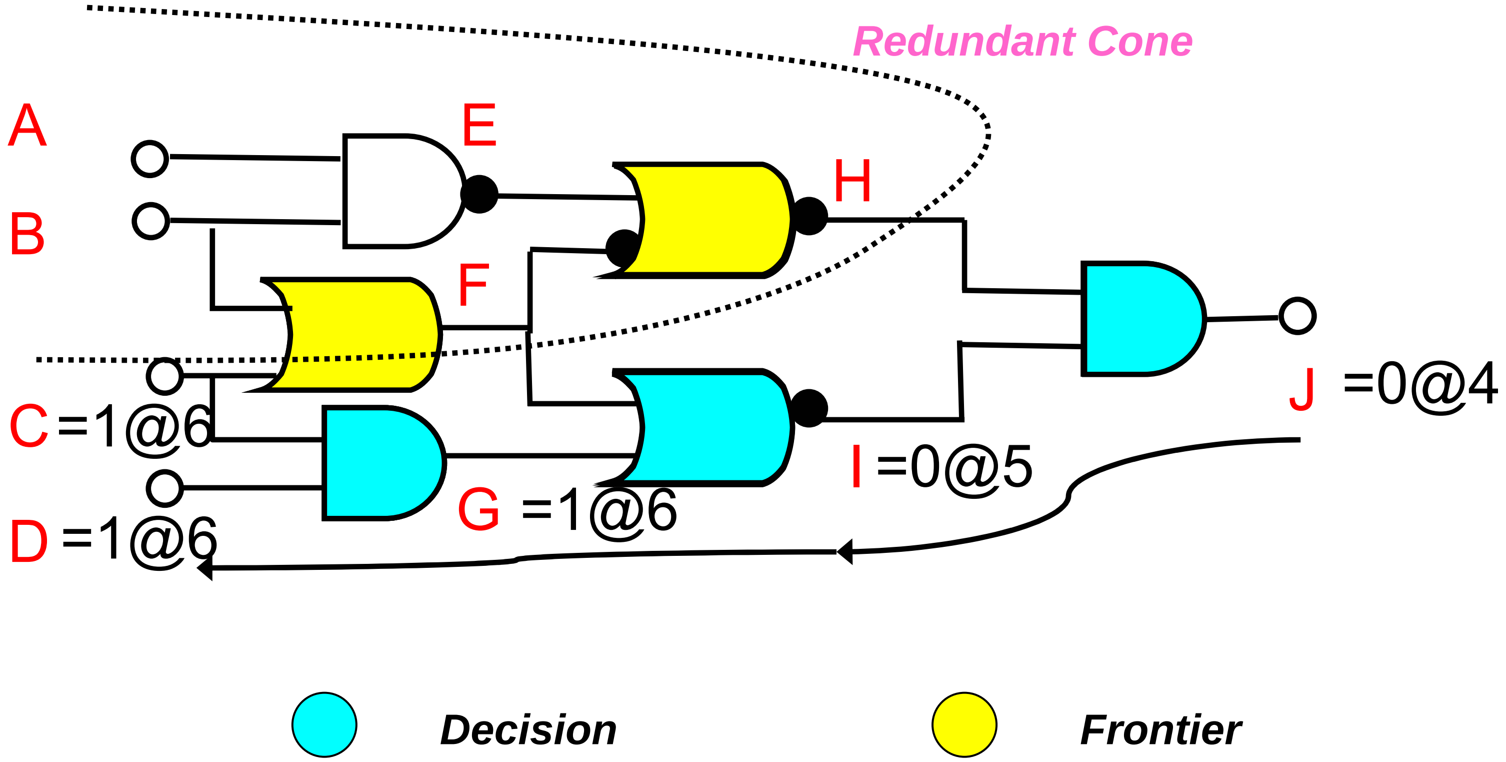
Ckt: **BCP on Gate Clauses as Circuit**

Platform:

RH Linux 7.1, PIII 750Mhz 256 Mb

Decision Heuristic: Justification Frontiers

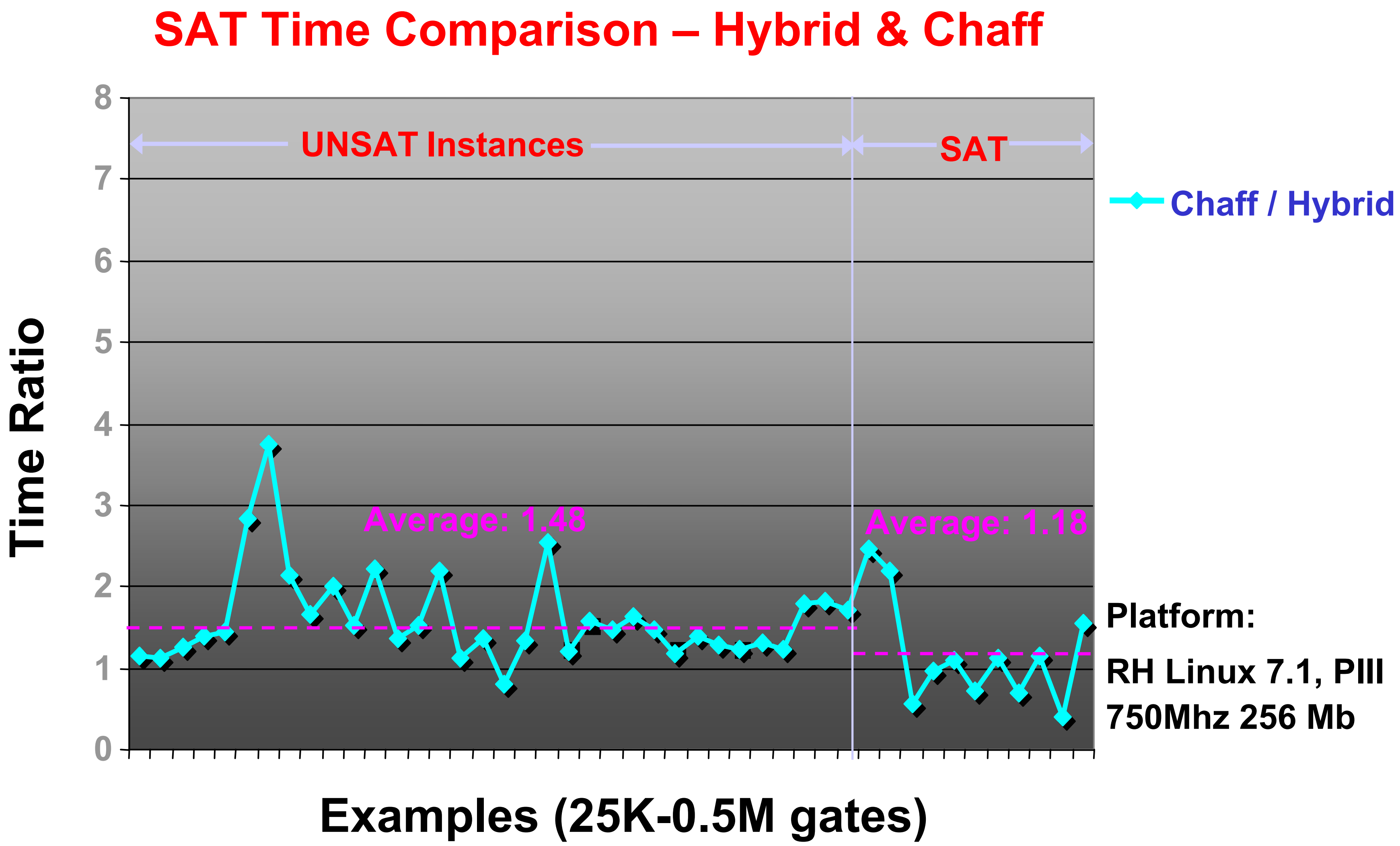
- Decision is restricted to variables required to justify the fanout
- Helps in pruning the search space for the SAT solver



NEC Hybrid (Circuit+CNF) SAT Solver

- **Deduction Engine – Hybrid BCP**
  - **Circuit-based BCP on gate clauses using *fast table lookup***
  - **CNF-based BCP on learned clauses using *lazy update***
- **Decision Engine**
  - **Use of circuit-based heuristics**
- **Diagnostic Engine**
  - **Record both clauses and gate nodes as reasons for conflict**

SAT Results (same decision heuristics)

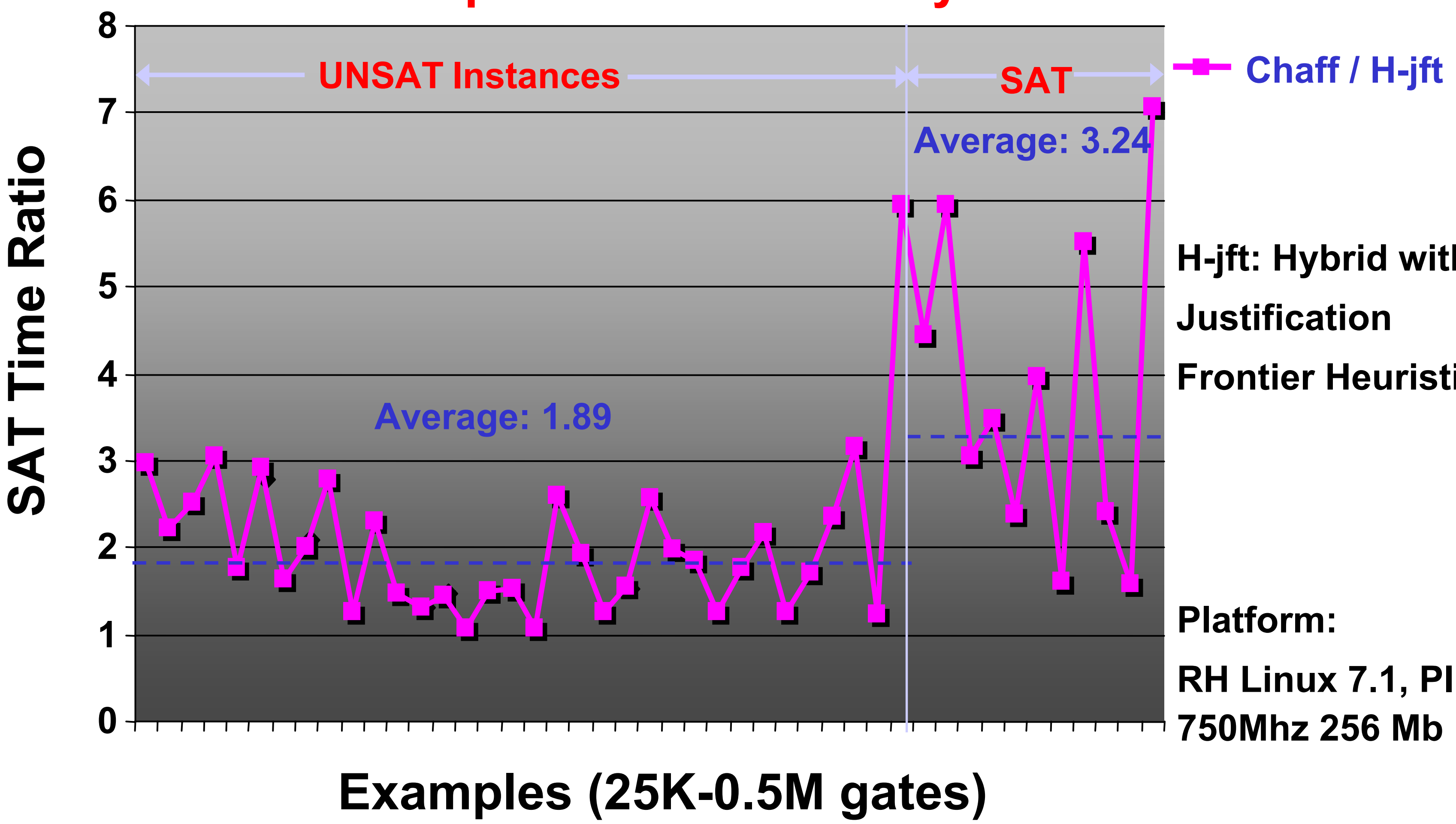


Platform:

RH Linux 7.1, PIII
750Mhz 256 Mb

SAT Results (circuit decision heuristic)

SAT Time Comparison – Chaff & Hybrid w/ JFT



H-jft: Hybrid with Justification Frontier Heuristic

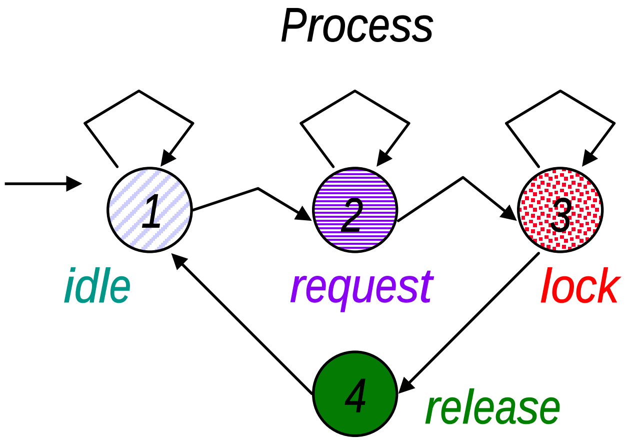
Platform:

RH Linux 7.1, PIII 750Mhz 256 Mb

SAT-based Verification Methods

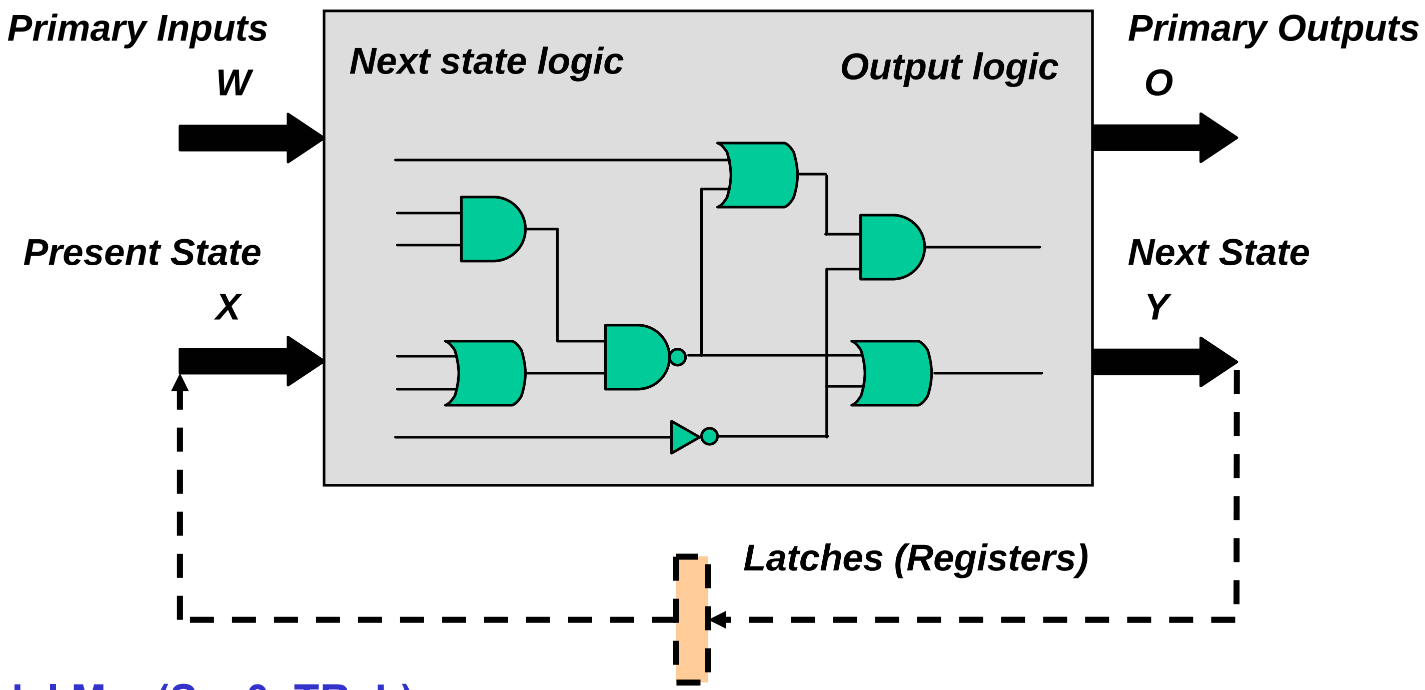
Implementation Model

- ❑ Labeled Transition System (LTS)
- ❑ Model M = (S, s0, TR, L)
  - – S: Set of states (usually finite)
  - – s0: Initial state
  - – TR: Transition Relation between states
  - – L: Labeling of propositions (signals) that are true in a state
- ❑ Example: mutual exclusion for critical section



S = { 1, 2, 3, 4}
s0 = { 1 }
TR = { (1,1), (1, 2), (2, 2), (2, 3),
(3, 3), (3, 4), (4, 1) }
L: L(1) = { idle }
L(2) = { request }
L(3) = { lock }
L(4) = { release }

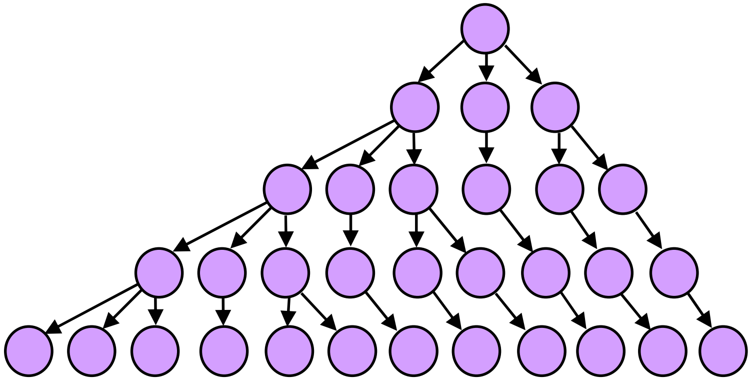
Hardware Circuit Model (Symbolic LTS)



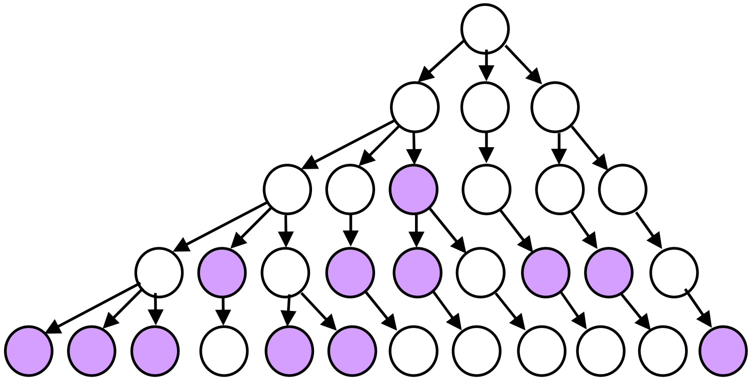
- ❑ Model M = (S, s0, TR, L)
- ❑ Set of States S is encoded by a vector of binary variables *X*
  - – Implemented as the outputs of latches (registers)
  - – *NOTE: Size of state space: |S| = 2* $^{|X|}$
- ❑ Initial state s0 comprises initial values of the latches
- ❑ Transition relation TR is implemented as next state logic (Boolean gates)
  - – Y = TR(X, W), where TR is a Boolean function of present state X and inputs W
- ❑ Labeling L is implemented as output logic (Boolean gates)
  - – O = f(X) or O = g(X,W)

Temporal Logic Specifications

A G p “on all (A) paths,
p holds globally (G) in every state”



A F p “on all (A) paths,
p holds in the future (F) eventually”

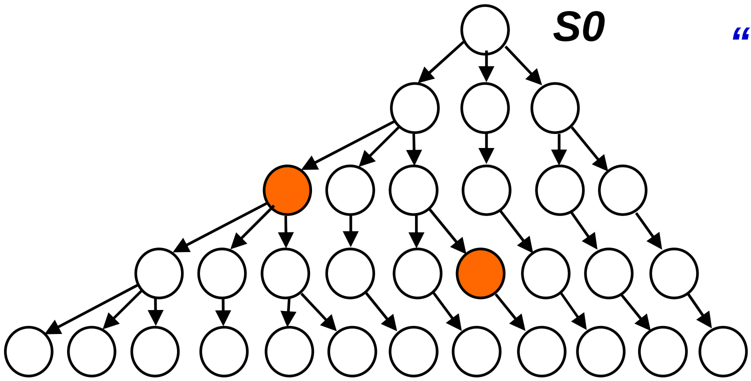


- ❑ **Safety property**
  - Nothing bad will happen
  - Example: Mutual exclusion
  - Formula
    - ➢ AG ! (p1_lock && p2_lock)
    - ➢ p1 and p2 cannot be in the lock state simultaneously

- ❑ **Liveness property**
  - Something good will happen
  - Example: resource allocation
  - Formula
    - ➢ AF bus_grant
    - ➢ The bus is granted eventually

Property Verification

- ❑ **Two Main Approaches**
  - **Proof Approach**
    - ➢ **Exhaustive state space exploration, i.e. all states in the LTS are covered to check for property satisfaction**
    - ➢ **Usually maintains some representation of visited states**
    - ➢ **Very expensive for medium to large-size LTS**
  - **Falsification Approach**
    - ➢ **State space search for bugs (counter-examples) only**
    - ➢ **Typically does not maintain representation of visited states**
    - ➢ **Less expensive, but needs good search heuristics**

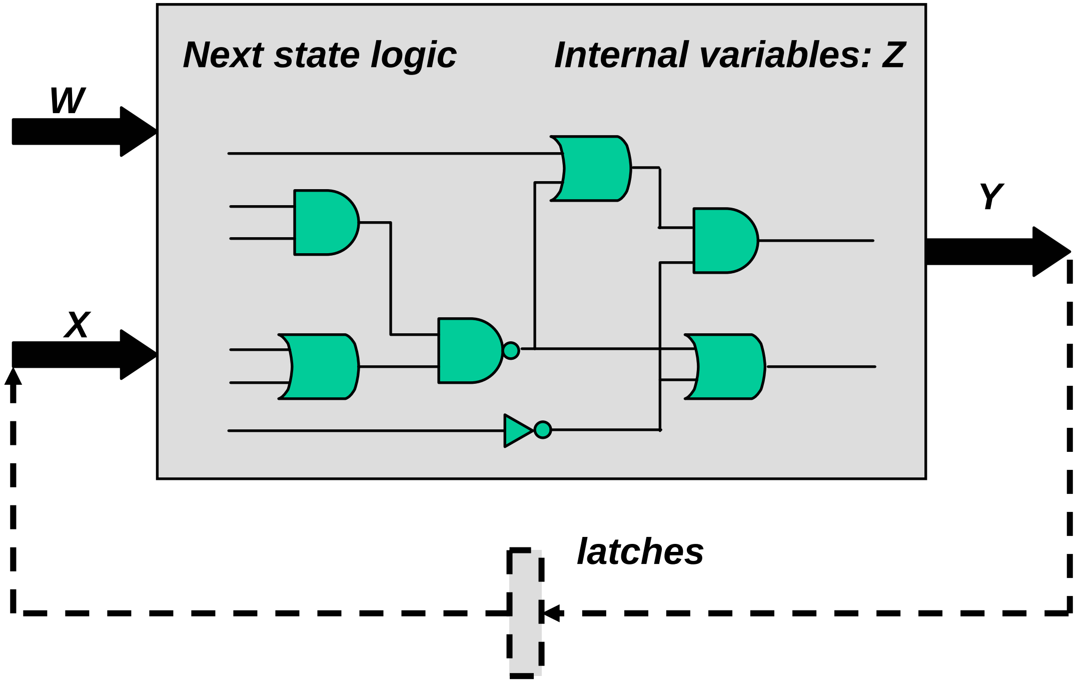


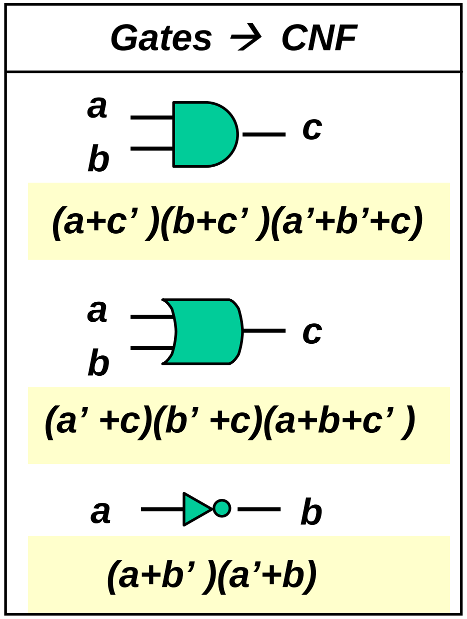
"Is there is a path from the initial state S0 to the bad state(s) where property fails?"

● ***State where the property fails***

Falsification:
Bounded Model Checking and Enhancements

Transition Relation as Circuit or CNF





Gates → CNF

$(a+c')(b+c')(a'+b'+c)$

$(a'+c)(b'+c)(a+b+c')$

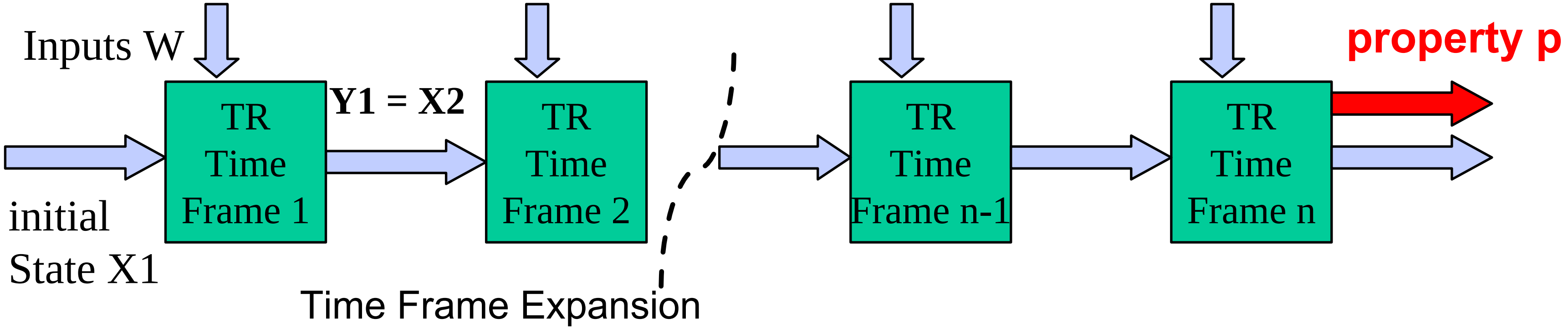
$(a+b')(a'+b)$

- ❑ **CNF** $T = \Pi_i C_i (X, W, Z, Y)$

  + linear in size of next state logic (with auxiliary variables Z)

  + fine grained conjunctive partition

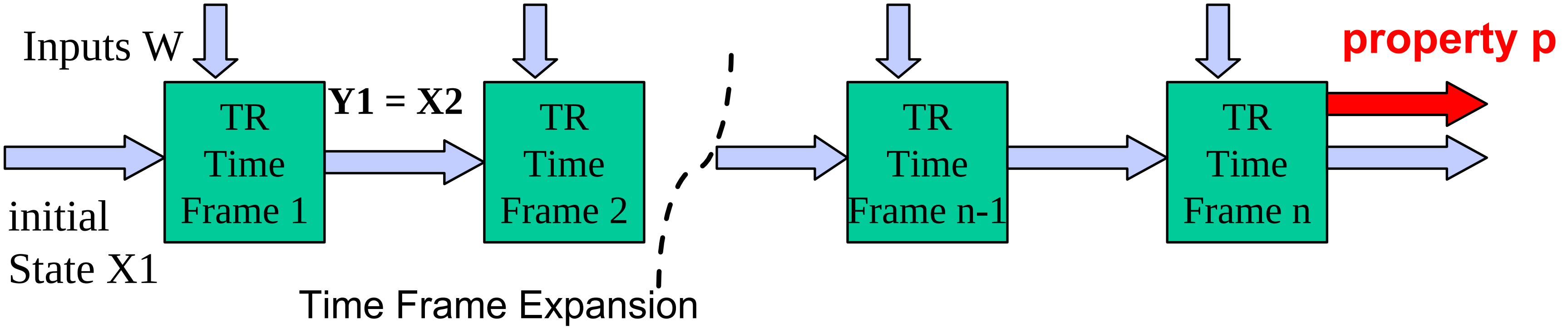
Bounded Model Checking (BMC)



- ❑ **BMC problem: Find a *k*-length counterexample for property *f***
  - Translated to a Boolean formula *B(M,f,k)* [Biere *et al.* 00]
  - Formula *B(M,f,k)* is satisfiable ⬄ a bug exists at depth *k*
  - Satisfiability of formula is checked by a standard SAT solver
    - ➢ SMT solvers are now being used for more expressive logics
- ❑ **Falsification approach**
  - Scales much better than BDD-based methods for hardware verification
    - ➢ BDDs can typically handle 100s of latches (state elements)
    - ➢ SAT can typically handle 10K latches (state elements)
  - Incomplete in practice due to large completeness threshold
    - ➢ Diameter (longest shortest path), Recurrence diameter (longest loop-free path), …
- ❑ **Main ideas**
  - Unroll transition relation up to bounded depth
  - Avoid computing sets of reachable states

Useful for finding proofs also!

BMC Translations



- ***BMC (M, f, k)***

  $= I(y_0) \wedge 1 \leq j \leq k\ [\ T(x_j, y_j, w_j) \wedge (\ y_{j-1} = x_j)\ ] \wedge 0 \leq j \leq k\ [Env(e_j)] \wedge \langle f \rangle_k$

- **Constraints**
  - **Initial state constraints**
  - **Transition relation constraints for each time frame**
  - **Latch interface propagation constraints**
  - **Environment constraints**
  - **Property constraints**

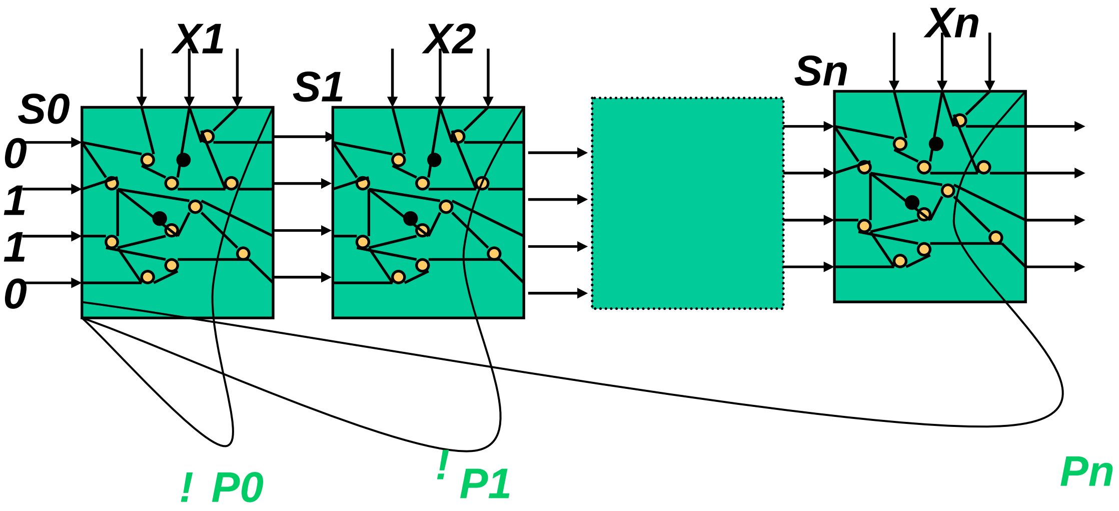
- **Many different translations for $\langle f \rangle_k$**
  - **Quadratic (worst-case cubic in *k*), linear in *k***

Falsification using SAT-based BMC

```
BMC(k, P) { // Falsify safety property P within bound k
    for (i=0; i < k; i++) {
        $P^i$ = Unroll(P, i); // Property at ith unroll frame
        if (SAT-Solve($P^i$=0) == SAT) return CE;
    }
    return NO_CE;  } // No Counter-example found
```

- ❑ **Main Tasks**
  - **Time frame unrolling of design**
  - **Construct propositional formula for the property node at depth *i***
  - **SAT check on the Boolean formula**
- ❑ **SAT problem size grows as *depth i* increases**
  - **Keep problem size small**
  - **Improve practical efficiency of SAT solver**

Improving BMC Performance

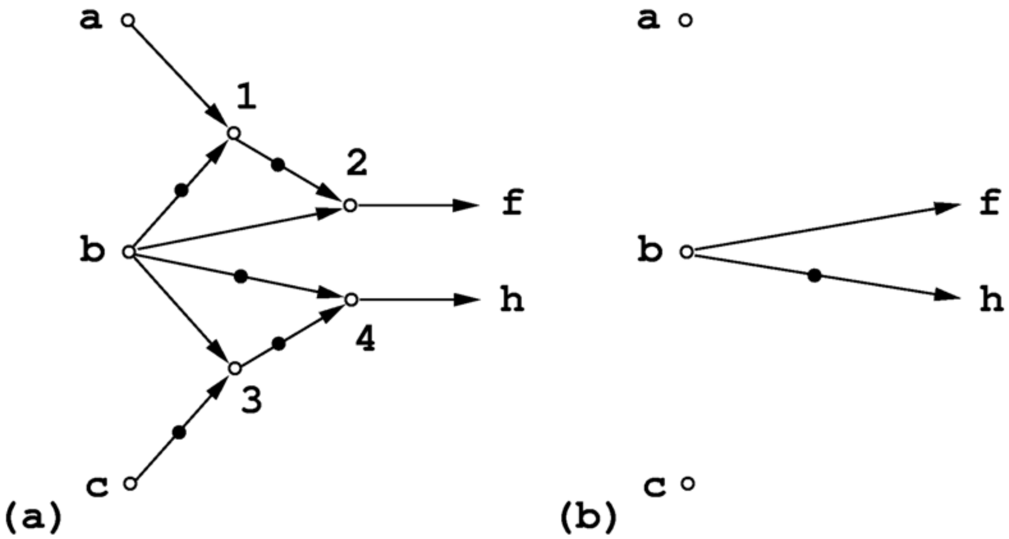


- **Dynamic circuit simplification** **[Kuehlmann & Ganai 01]**
- **Reuse of learned property constraints** **[Ganai *et al.* 02]**
- **Partitioning and incremental BMC translation** **[Ganai *et al.* 05]**
  - **Customized property translations into multiple SAT subproblems**
- **Hybrid SAT Solver** **[Ganai *et al.* 02]**
- **BDD Learning** **[Gupta *et al.* 03]**
- **BDD Constraints** **[Gupta *et al.* 03]**

BDDs work really well on small problems – use them when you can!

Circuit Representation

- ❑ **Circuit Graph [Kuehlmann Dac 96, Dac 00 …]**
  - – **2-input AND gates, with inverter edge attributes**
  - – ***On-the-fly*** **graph reduction based on functional hashing [Bryant ’88]**
  - – **Local 2-level lookup for detecting isomorphic sub-graphs**



f= !(a* !b)*b

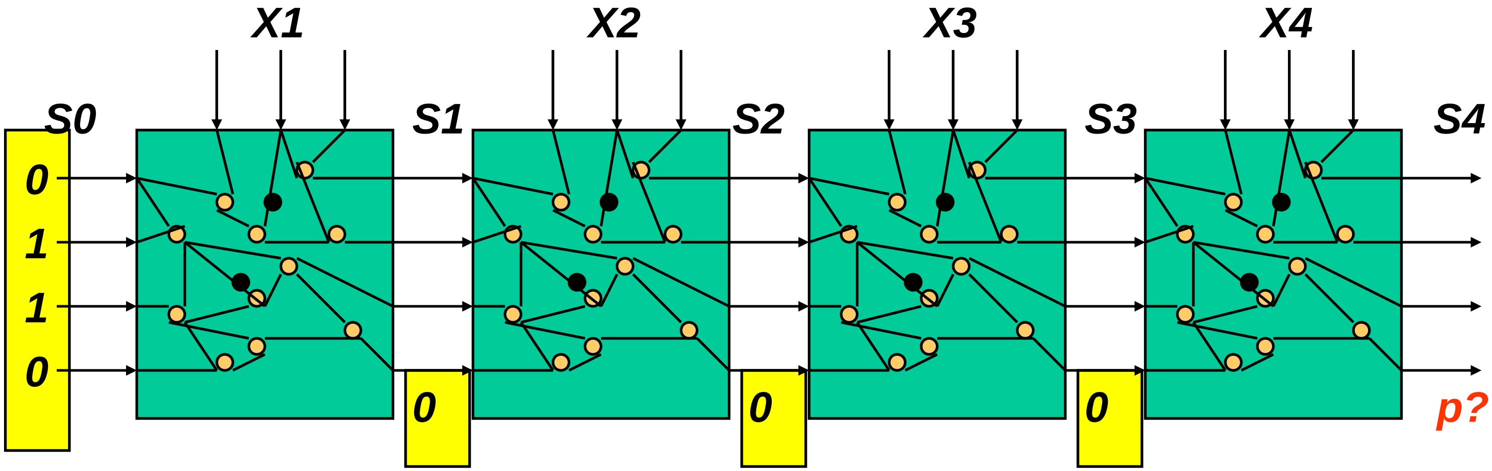
= b

h= !(b*!c)*!b

= !b

Dynamic Circuit Simplification

❑ BMC Application



- Initial state simplification by propagation of constants
- Property constraints are also learned and propagated
- Explicit unrolling provides opportunities for circuit simplification across time frames [Ganai & Aziz Vlsi02]

Hybrid SAT for BMC: Advantages

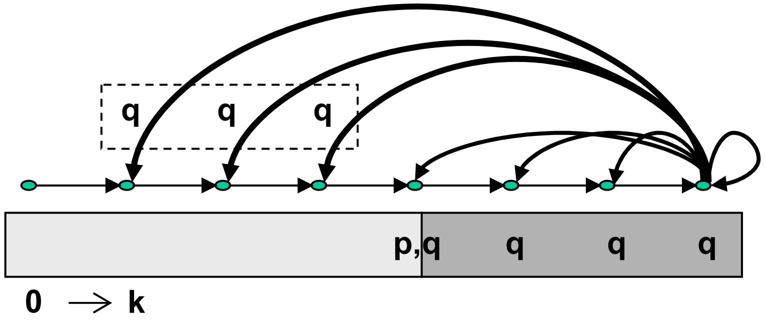
- Memory Savings
  - No need to translate circuit to CNF gate clauses
- Speed-up
  - 3X (over Chaff) typical
- Use best of CNF- and Circuit-based SAT Solvers
  - *e.g.* heuristics from Berkmin, Jerusat, Limmat, …

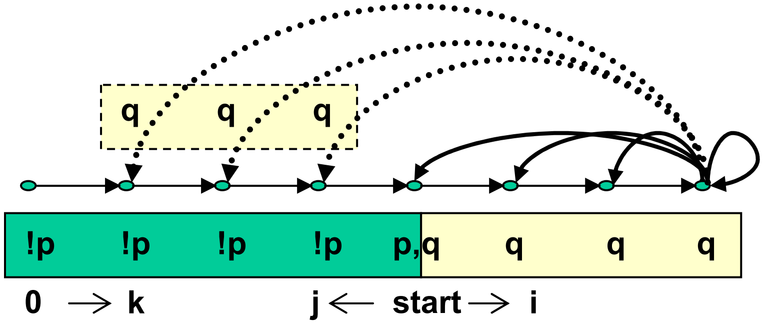
Customized Property Translations: Intuition

- **Example: G(p -> F !q)**
  - Look for a witness for F(p * G(q))

- **General Translation**

Our Translation





Monolithic SAT Formula

$$[M,f]_k = [M]_k \wedge ((\neg L_k \wedge [f]_k^0) \vee (\vee_{l=0}^{l=k} ({}_l L_k \wedge {}_l[f]_k^0)))$$

Partitioned SAT subproblems

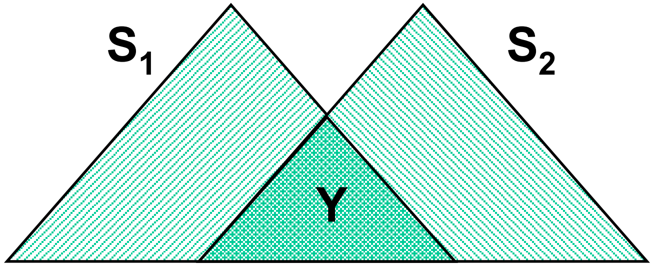
- across operators

- within and across time frames

Learning from Unsat Instances

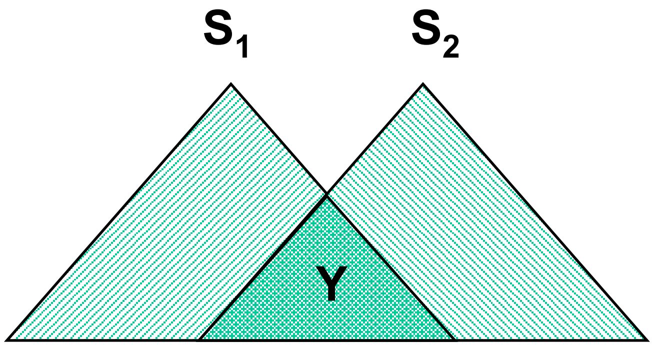
Incremental Formulation

Incremental SAT Solving Techniques



- ❑ **Main Idea**
  - Given Instance S1 and S2, let Y = S1 ∩ S2 be the set of shared clauses
  - Clauses in Y are marked
  - Conflict clauses derived from ONLY marked clauses can be reused
- ❑ **BMC Application**
  - Shared clauses arise due to circuit unrolling: circuit clauses
    - ➢ Proposed by Strichman [CAV 00, TACAS 01]
    - ➢ Mixed results
  - Our translation: property constraints are also derived incrementally
    - ➢ Leads to sharing of clauses due to property constraints
    - ➢ Mitigates the overhead of partitioning performance improvement
  - Clause Replication: conflict clause is repeated for other time frames
    - ➢ Proposed by Strichman
    - ➢ Mixed results

Incremental Learning



- ❑ **Learning from shared constraints (L1)**
  - ➢ **Reuse Learnt conflict clauses in C while solving S1 or S2**
- ❑ **Learning from satisfiable results (L2)**
  - ➢ **Use satisfiable solution of S1 as initial guess for solving S2**
- ❑ **Learning from unsatisfiable results (L3)**
  - ➢ **If S1 is unsat, one can use !S1 while solving S2**

Note: This is in addition to conflict-based learning in the SAT Solver

Customized Translation: $F(p \wedge G(q))$

[Ganai *et al.* DAC 05]

```
FG_Solve (p,q){ // L1 active always
   C=1;
   for(i=0; i<N; i++) {
     if (! is_SAT (C & $p_i$))
        C = C & !$p_i$; // L3
     // L2
     else if (G_Solve (C & $p_i$, q, i) == T) return T;}
   return undertermined;}

G_Solve (IC, q, start){ // L1 active always
   for(i=start; i<M; i++) {
     C = C & $q_i$ ;
     if (! is_SAT(C)) return $\perp$; // L2
     for(j=i; j>=start; j--) {
        if (!is_SAT(C & $FC_{ij}$)) continue; // L2
        if (is_SAT(C & $L_{ij}$ & $FC_{ij}$)) return T;
         C = C & !$L_{ij}$;} // L3
    return undetermined;}
```

Experimental Results for Customized BMC Translations

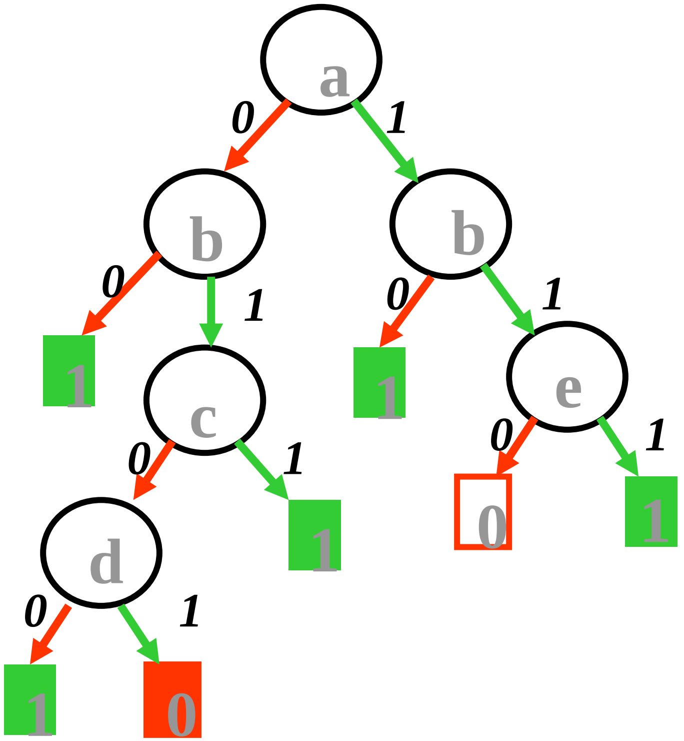
D	#FF	#PI	#G	CEX (D)	Custom (DiVer)				Std (VIS)
					NL	L1,3	L2,3	L1,2,3	
D1	2316	76	14655	19	2.3	2.2	2.3	2	77
D2	2563	88	16686	22	11.2	8.9	11.7	8	201
D3	2810	132	18740	28	730	290	862	240	2728

- D1-D3: Industry bus core designs with multiple masters/slaves
- Property: “Request should be eventually followed by ack or err”
- L1-3 Learning Schemes, NL: no learning
- VIS: monolithic BMC translation
- Customized Translation finds counter-examples quickly

Using BDDs with SAT

[Gupta *et al.* DAC 03]

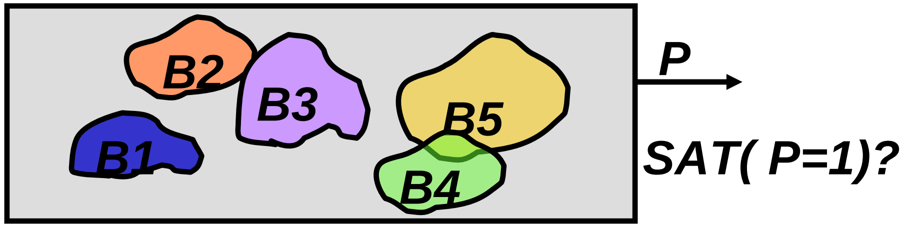
- ❑ Each path to 0 in a BDD denotes a conflict on its variables
- ❑ A BDD captures all conflicts
- ❑ Each conflict can be avoided by adding a learned clause to SAT
  - – a + b’ + c + d’
  - – a’ + b + e’
- ❑ Learning can be selective
  - – No need to add each clause
  - – Select clauses to add
- ❑ Tradeoff: usefulness vs. overhead
  - – Useful: multiple conflicts are handled simultaneously
  - – Overhead: too many learned clauses slow down BCP



- ❑ Strategy: ***Effective*** **and** ***Lightweight*** **BDD Learning**

Effective and Lightweight BDD Learning in BMC

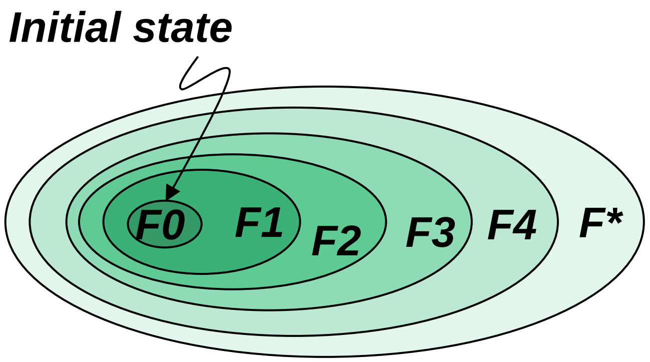
- Global BDD learning: for every circuit node
  - Impractical, wasteful
- Targeted BDD Learning: for selected circuit nodes (“Seeds”) in unrolled design



- Two Learning Schemes
  - Static BDD Learning
    - Seeds are selected statically
    - Learned clauses are added statically before starting SAT search
  - Dynamic BDD Learning
    - Seeds are selected dynamically
    - Learned clauses are added dynamically during SAT search
- Heuristics for a good balance between usefulness and overhead
- Improved search in BMC with Dynamic BDD Learning
  - Upto 73% reduction in time for same depth
  - Upto 60% more time frames searched

BDD Constraints in BMC

- ❑ BDD constraints are generated from abstract models after localization
- ❑ Forward reachability sets

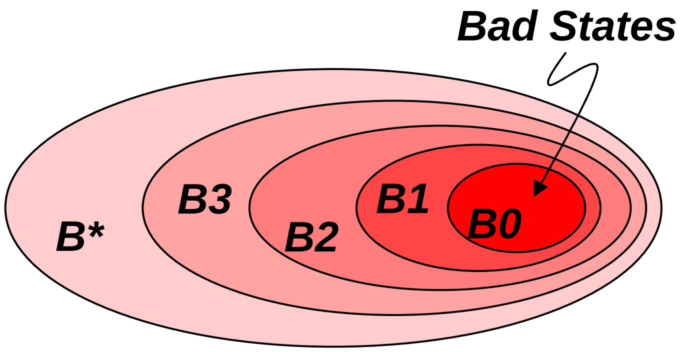


Fk: all states reachable from initial state in k steps or less

F : all reachable states*

Over-approximations for Concrete Design

- ❑ Backward reachability sets



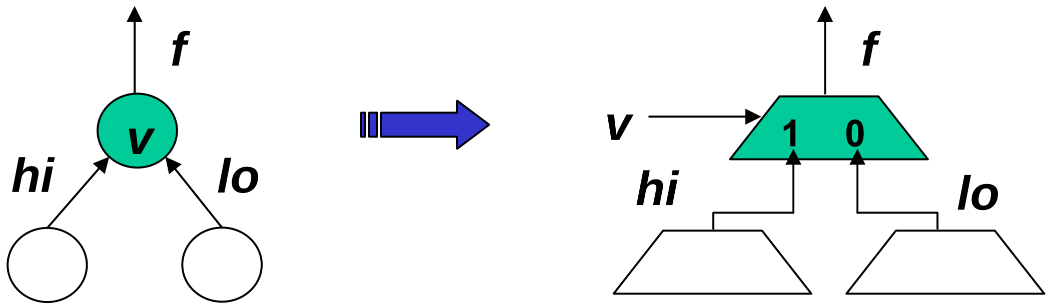
Bk: all states that can reach a bad state in k steps or less

B : all states that can reach a bad state*

Over-approximations for Concrete Design

Conversion of BDDs to CNF/Circuits

- Our approach: Convert BDD to a circuit
  - Introduce a new variable for each internal node in the BDD
  - Replace each internal node *f = (v, hi, lo)* by a multiplexor



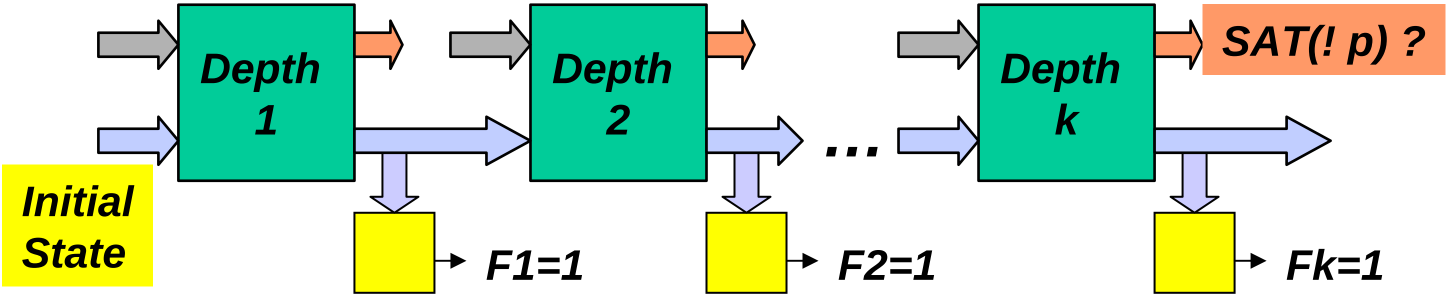
  - Size of constraint circuit is linear in size of BDD
    - ➢ Keep BDD size small by reordering or over-approximation

- Other approaches: [Cabodi *et al.* 03]
  - No new variables, but enumerate all paths to 0 as conflict clauses
  - Introduce variables for selected internal nodes, and enumerate paths between such nodes as conflict clauses

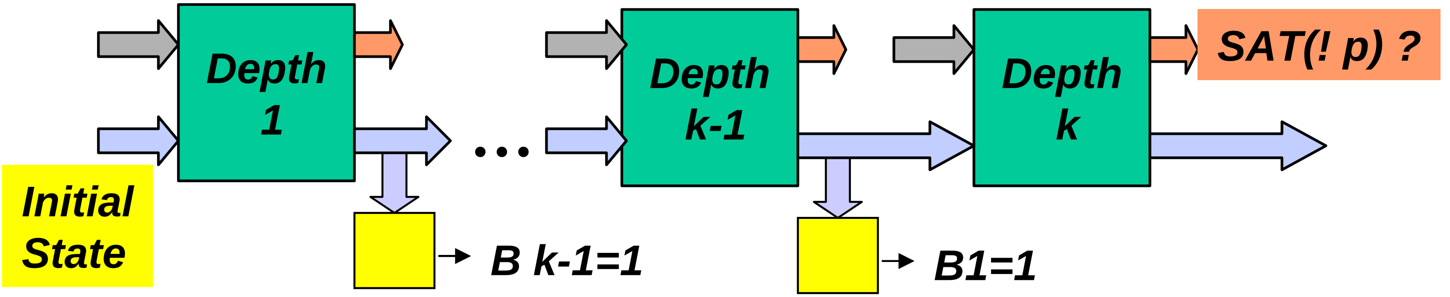
BMC Search with BDD Constraints

[Gupta *et al.* CAV 03]

- Use of forward reachability constraints



- Use of backward reachability constraints

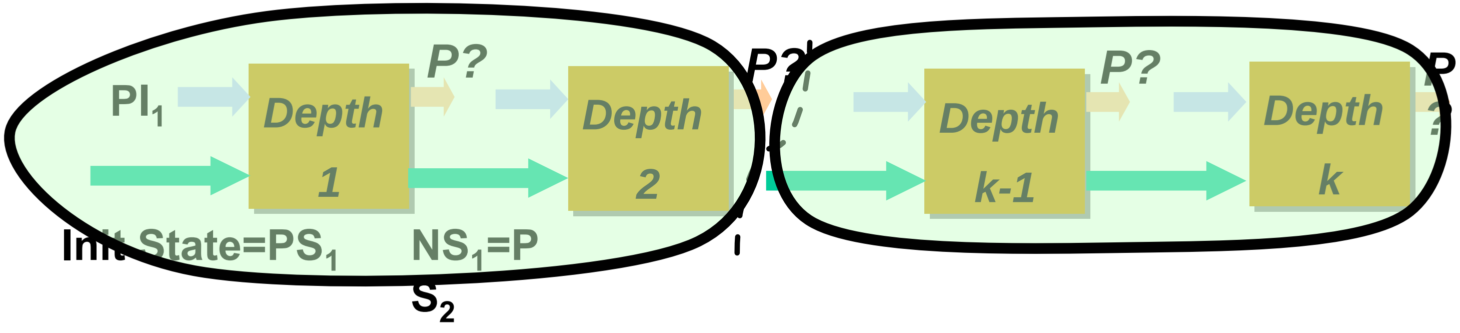


- Reachability constraints are redundant
  - Potentially useful for pruning search (like conflict clauses)
  - However, need to tradeoff usefulness vs. overhead (mixed results)

Distributed BMC (d-BMC)

SAT-based Distributed BMC

As unroll depth *k* increases, the memory requirement can exceed the memory available in a single server !

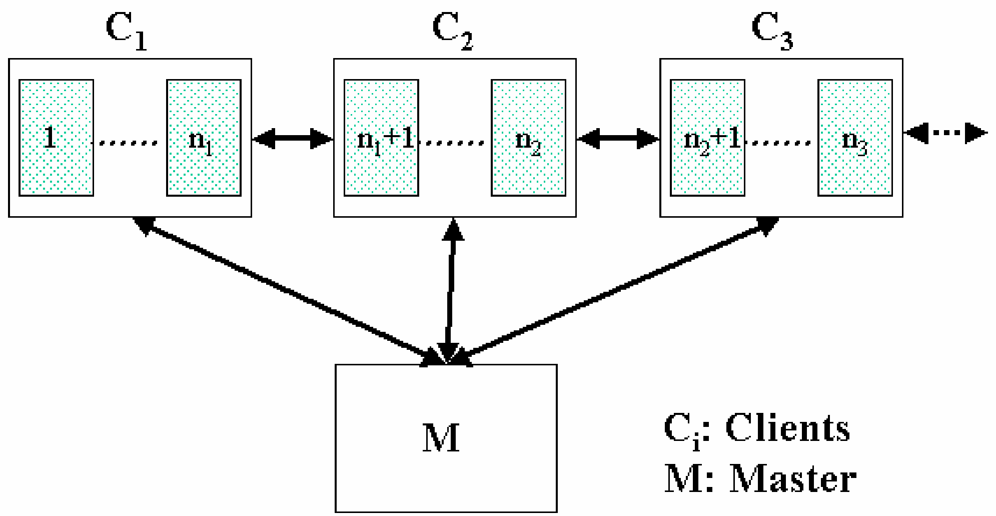


Main Idea: Partition unrolled circuit and use Distributed SAT

- ❑ Network of Workstations (NOW)
  - – Easily available, standard, cheap
- ❑ BMC problem provides a natural disjoint partitioning
  - – Need to use a distributed SAT solver

Master/Client Model for d-SAT

- ❑ Each Client C_i hosts an exclusive problem partition
- ❑ Each C_i is connected in pre-known topology
- ❑ Bi-directional FIFO (in-order) between neighbors
- ❑ Master M is connected to all Clients
- ❑ M controls the d-SAT and d-BMC



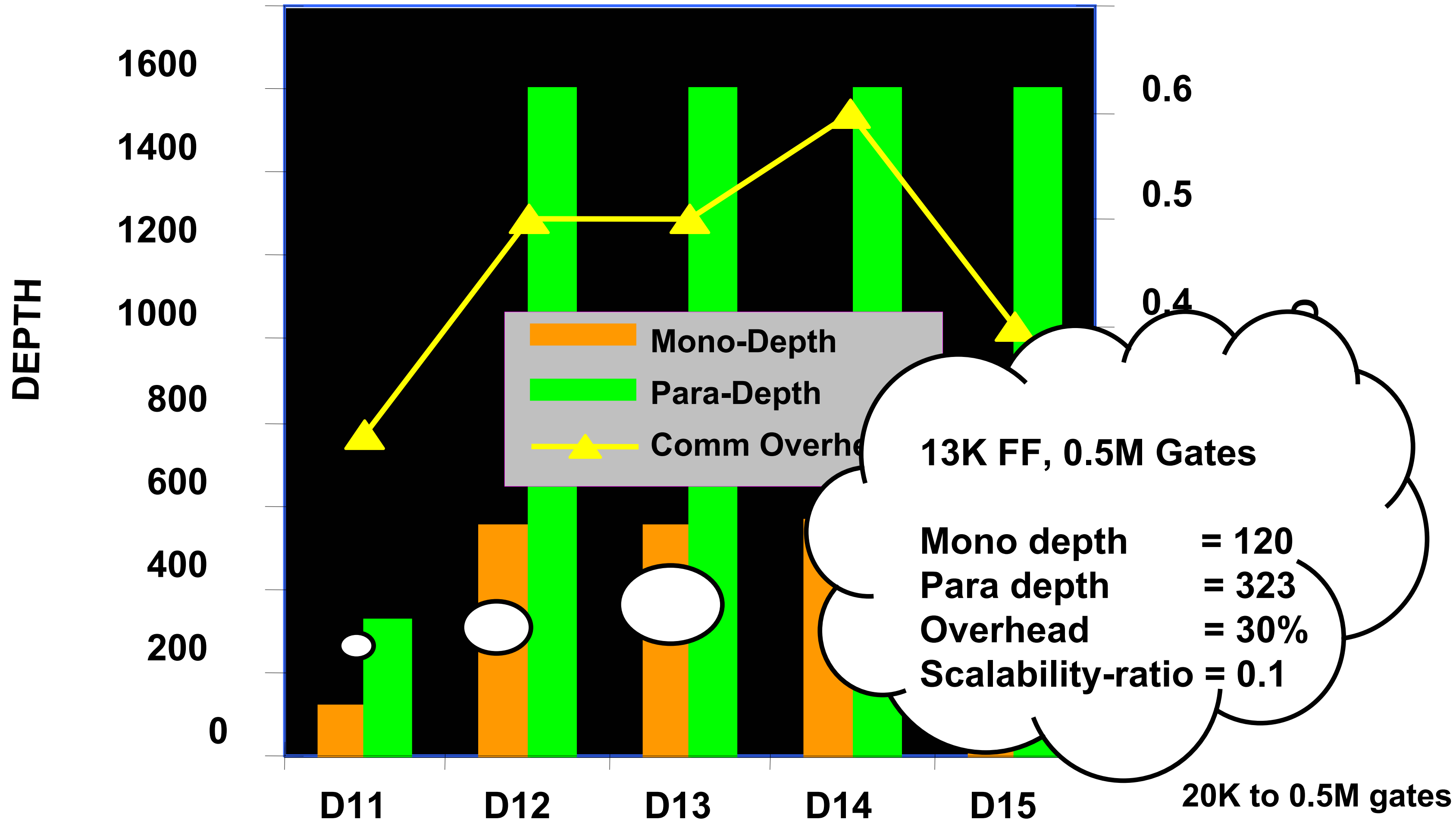
d(istributed)-SAT

```
//Master Controls the d-SAT execution
d-SATSolve( P=1) { //Check if constraint P=1 is SAT
   while (d-Decide() == SUCCESS)
          while (d-Deduce() == CONFLICT)
                 if (d-Diagnose() == FAILURE)
                           return UNSAT
   return SAT;  }
```

[Zhao 99]

- **d-Decide**
  - **Each client decides on its partition, master selects the best**
- **d-Deduce**
  - **Each client deduces on its partition, and Master deduces on (global) conflict clauses**
- **d-Diagnosis**
  - **Master performs diagnosis using global assignment stack and clients backtrack locally**

Deeper Search using d-BMC



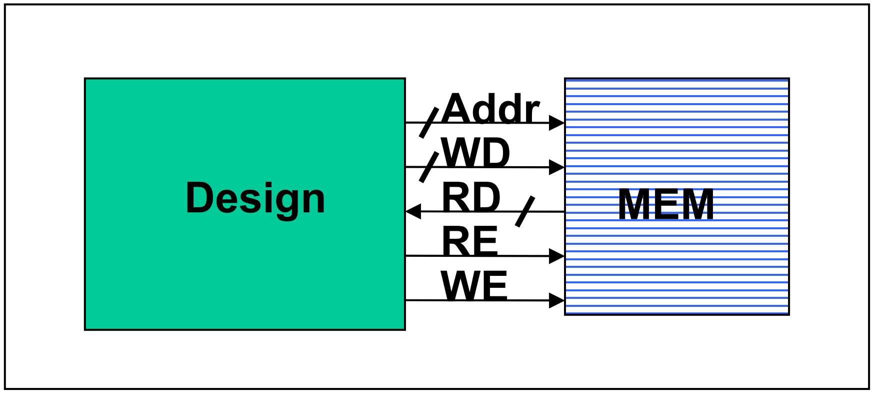
13K FF, 0.5M Gates

Mono depth = 120
Para depth = 323
Overhead = 30%
Scalability-ratio = 0.1

20K to 0.5M gates

Handling Hardware Designs with Embedded Memory

Designs with Embedded Memory



Addr: Address Bus

WD : Write Data

RD : Read Data

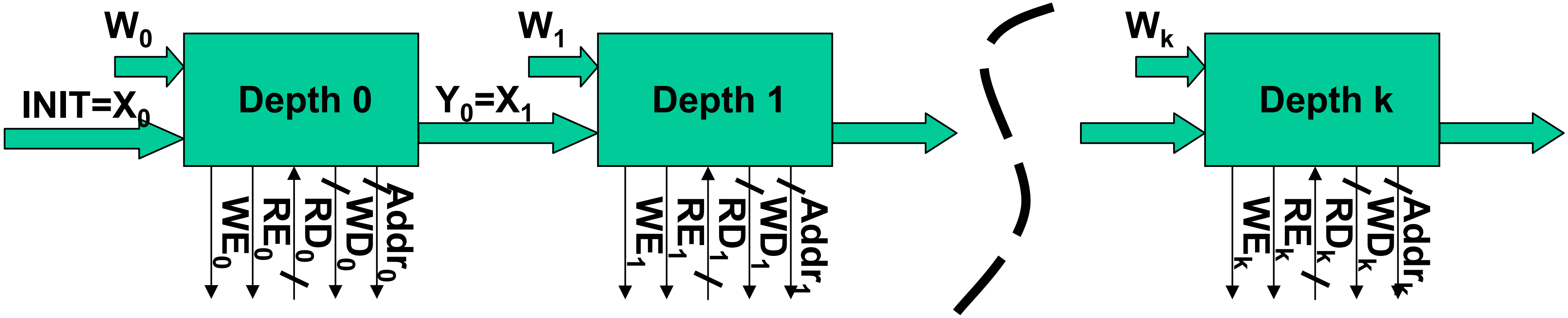
WE : Write Enable

RE : Read Enable

- Designs with embedded memories are very common
  - Multiple read and write ports
  - Arbitrary initial state
- Most formal verification techniques are inefficient or incomplete
  - Explicit memory modeling: very expensive, state space explosion
  - Remove memory: sound but not complete

    (spurious counter-examples are possible)

Efficient Memory Model (EMM) Approach



- ❑ **EMM idea:** **Remove memories, but add data forwarding constraints to SAT problem for BMC**
  - ➢ **$RD_k = WD_i$ where ($i < k$), $Addr_k = Addr_i$, and**
  - ➢ **No WRITE between i and k-th cycle at $Addr_k$**
- ❑ **Similar to theory of interpreted memories** **[Burch & Dill 94, Bryant *et al.* 00]**
  - ➢ **They used an ITE-based representation of memory constraints**
- ❑ **Arbitrary Initial State**
  - ➢ **Introduce new symbolic variables**
  - ➢ **Add constraints to capture correlation between them**

SAT-based BMC with EMM

[Ganai *et al.* CAV 04, DATE 05]

```
m-BMC(k, P) { // Falsify safety property P within bound k
    $C^{-1} = \Phi$; // Initialize memory modeling constraints
    for (i=0; i < k; i++) {
        $P^i$ = Unroll(P, i); // Property at $i^{th}$ unroll frame
        $C^i$ = EMM-Constraints(i,$C^{i-1}$); // update the constraints
        if (SAT-Solve($P^i$=0 $\wedge$ $C^i$=1) == SAT) return CE;
    }
    return NO_CE;  } // No Counter-example found
```

- □ Memory modeling constraints that capture the forwarding semantics of memory are added at every unroll
- □ Procedure EMM-Constraints
    - ➢ Adds constraints in efficient representation (CNF+gates)
- □ Extended to handle multiple memories, multiple ports

EMM Results Summary

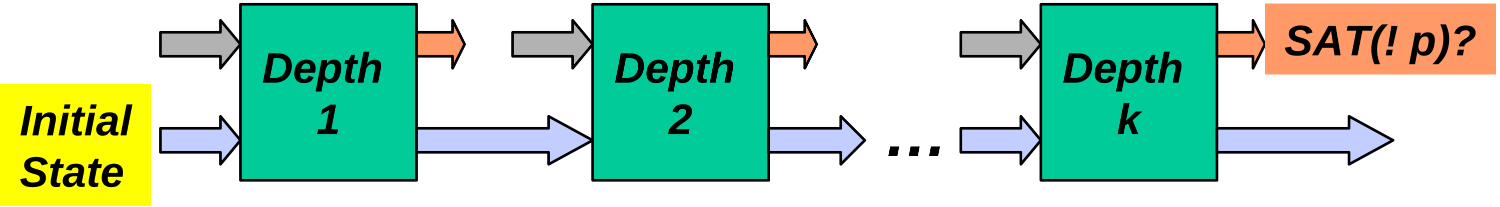
D (Prp)	Wit D	Time (sec)			Mem (MB)		
		Explicit	hITE	hESS	Explicit	hITE	hESS
3n+1 (a)	71	9903	562	590	668	82	74
3n+1 (b)	89	> 3hr	1292	1201	NA	127	113
Toh (a)	52	2587	13	10	2059	16	12
Toh (b)	444	NA	8232	6925	MO	845	569
Fib (a)	38	2835	20	15	2239	15	17
Fib (b)	207	NA	> 3hr	7330	MO	NA	461
D1	68*	10680	1264	925	2049	91	64
D1	178*	NA	> 3hr	10272	MO	NA	908

- Our approach (hESS)
  - 1-2 orders of magnitude improvement (space/time) over Explicit
  - 20-30% improvement (space/time) over hybrid-ITE approach

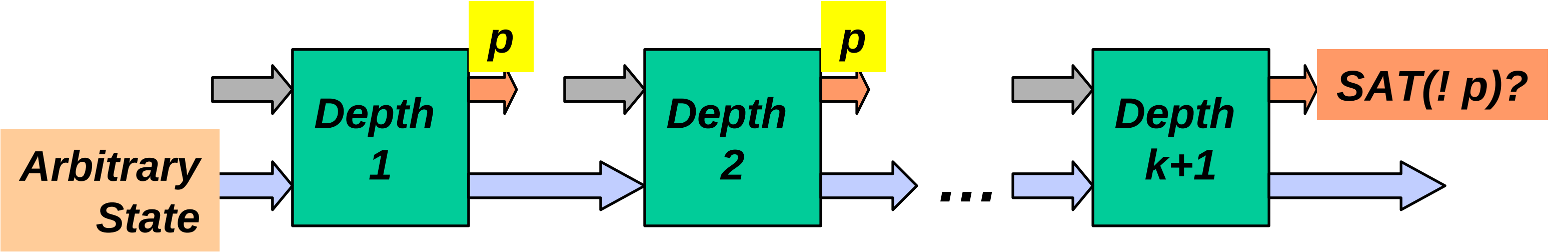
Methods for Finding Proofs of Correctness

SAT-based Proof by Induction

- **Proof by Induction with increasing depth** **[Sheeran *et al.* FMCAD 00]**
  - **Complete for safety properties by restriction to loop-free paths**
  - **Base Step: If *Sat(!p_k),* then property is false**



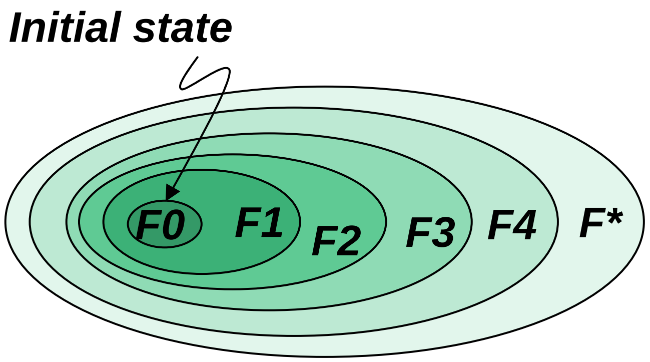
  - **Inductive Step: If *Unsat(!p_k+1),* then property is true**



  - **Else *k*++**
  - **Keep increasing *k* till conclusive result is found**
    - ***In practice, inductive step often fails: need to strengthen p!***

Recall: BDD Constraints in BMC

- BDD constraints are generated from abstract models after localization
- Forward reachability sets

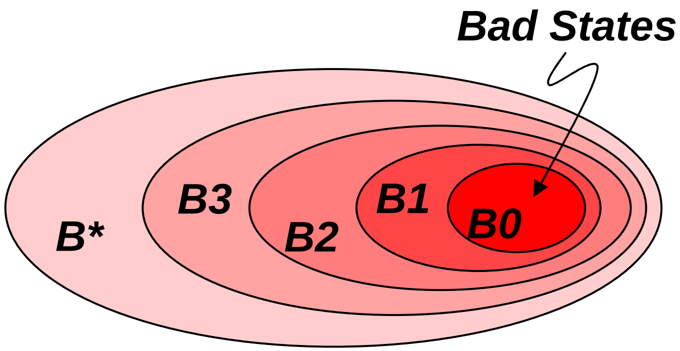


Fk: all states reachable from initial state in k steps or less

F : all reachable states*

Over-approximations for Concrete Design

- Backward reachability sets



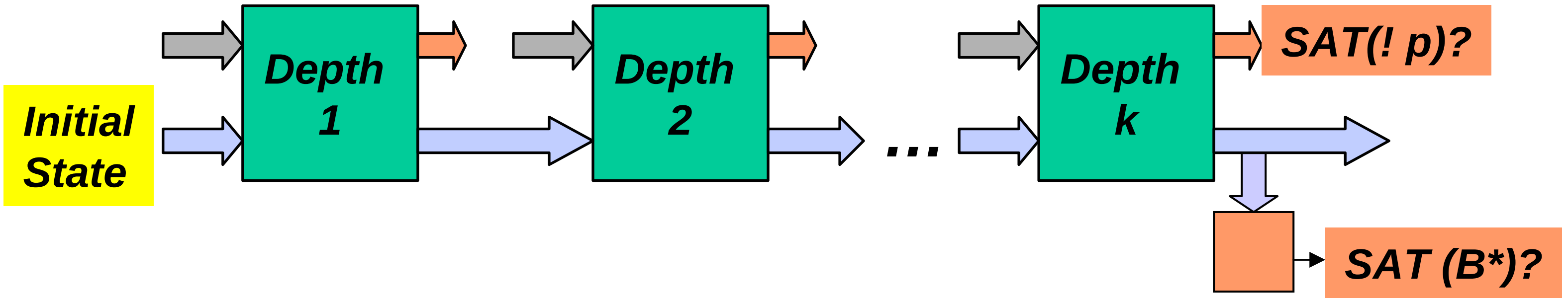
Bk: all states that can reach a bad state in k steps or less

B : all states that can reach a bad state*

Over-approximations for Concrete Design

BMC Proof with BDD Constraints

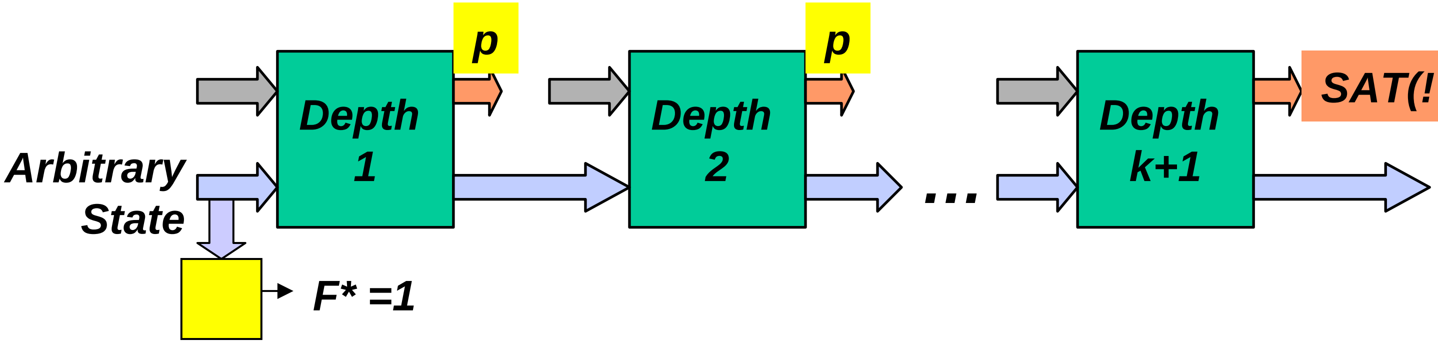
- **Base Step:**
  - If *Sat(!p_k)*, then property is false



  - Additional check: If *Unsat(B*)*, then property is true
    - *B** is not used a redundant constraint
    - Provides completeness due to conservative BDD-based model checking on abstract model

BMC Proof with BDD Constraints

- **Inductive Step:**
  - If *Unsat(!p_k),* then property is true



  - Additional constraint *F** on the arbitrary starting state
    - *F** is <u>not</u> used a redundant constraint
    - Provides an induction invariant due to over-approximate reachable state set
    - Frequently allows induction proof to succeed

Experimental Results

❑ BMC Proof with BDD Constraints

Design	BDD-based Abstract Model Analysis				BMC Proof with BDD Constraints on Concrete Design			
	#FF / #G	Time(s)	Depth	Size of F*	#FF / #G	Status	Time(s)	Mem(MB)
0in-a	41 / 462	1.6	7	131	2198 / 14702	TRUE	0.07	2.72
0in-b	115 / 1005	15.3	12	677	2265 / 16079	TRUE	0.11	2.84
0in-c	63 / 1001	18.8	18	766	2204 / 16215	TRUE	0.1	2.85

- **Despite gross over-approximation in abstract model, reachability constraints on abstract model provided strong enough induction invariants on concrete design.**

- **Though neither BDD-based method, nor SAT-based method, could complete verification, their combination allowed verification to complete very easily.**

Proof-based Abstraction

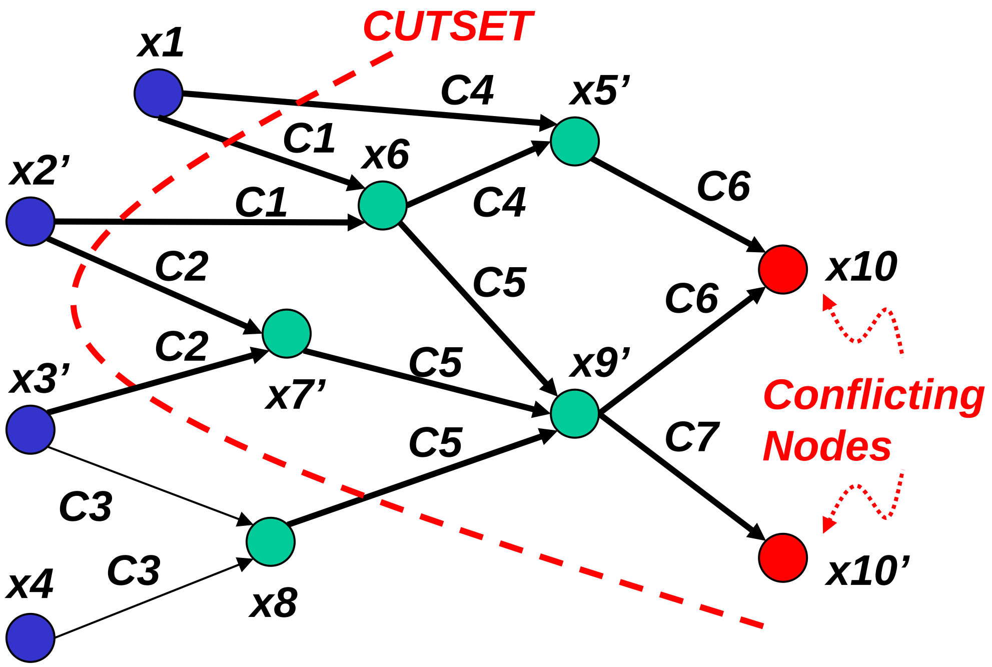
Proofs of UNSAT from SAT Solver

- ❑ **Unsatisfiable SAT Problem: Proof of Unsatisfiability**
  - **[Zhang Malik 03, Goldberg Navikov 03, McMillan Amla 03]**
  - **Conflict clause is the result of resolution on *antecedents***

Clauses:
C1: x1'+ x2 + x6
C2: x2 + x3 + x7'
C3: x3 + x4'+ x8
C4: x1'+ x6'+ x5'
C5: x6'+ x7+ x8'+ x9'
C6: x5 + x9 + x10
C7: x9 + x10'

Conflict Clause C8:
x1'+ x2 + x3 + x8'
Due to conflict (x10, x10')

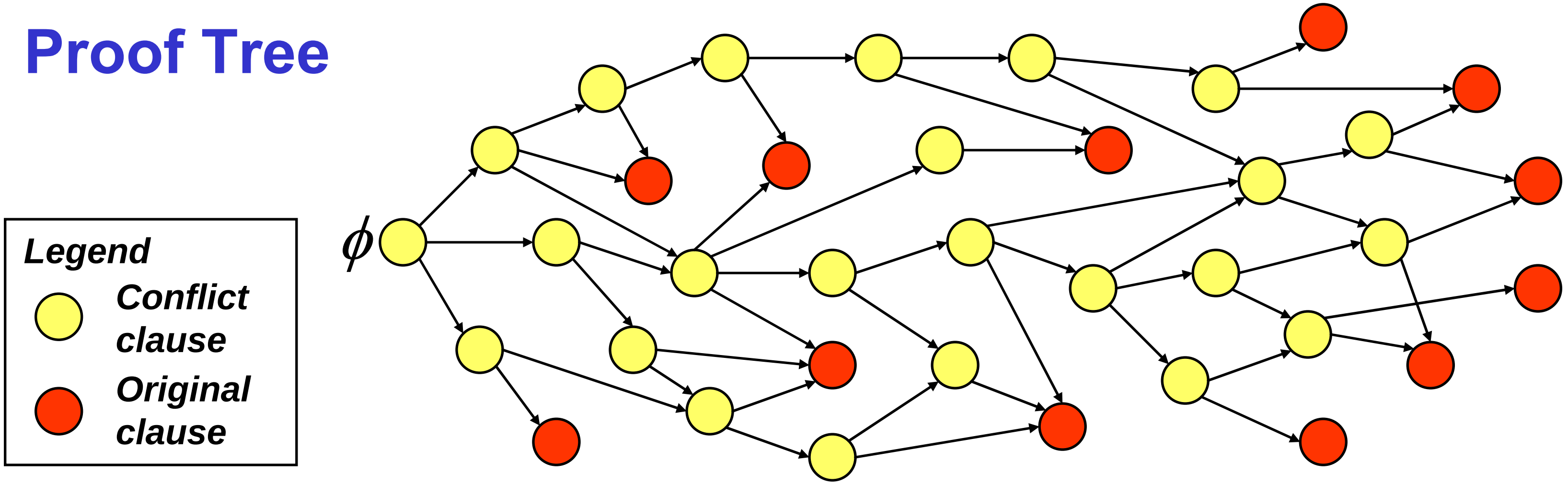
Antecedents(C8):
{C7, C6, C5, C4, C2, C1}



SAT Proof Analysis Technique

- **Unsatisfiable problem: Derive a Resolution Proof**
  - **Final conflict clause is an empty clause ϕ**
  - **Starting from ϕ, mark the recorded antecedents recursively for all conflict clauses**

Proof Tree



- **<u>Unsatisfiable Core</u>: Marked original clauses** ●
  - **sufficient for implying unsatisfiability**

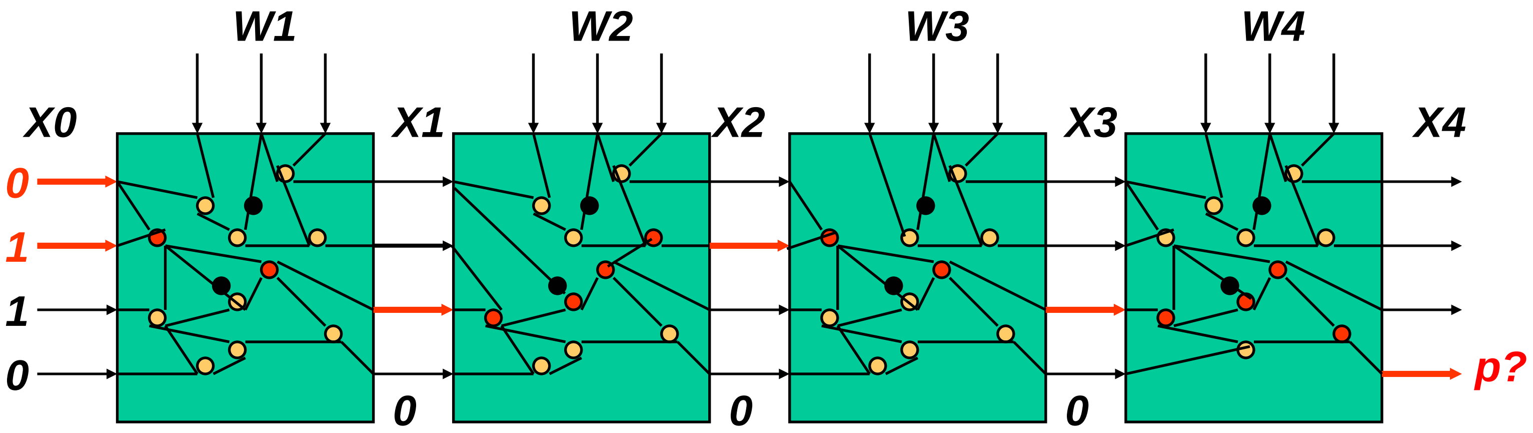
Extension to NEC Hybrid SAT Solver

- NEC Hybrid SAT Solver
  - Uses hybrid representation of Boolean problem
    - Simple gate-level representation for original circuit problem
    - CNF for learned conflict clauses
  - Hybrid BCP, Decision, and Diagnosis Engines
  - Speedup of 2-3x on most problems

- SAT Proof Analysis for Hybrid SAT Solver
  - Reasons (antecedents) for a conflict clause
    - Gates (nodes) in the circuit graph, due to circuit-based BCP
    - Clauses, due to conflict clauses or external constraints
  - Extraction of Unsatisfiable Core
    - Recursive traversal only for conflict clauses
    - Unsatisfiable Core: Set of marked nodes and clauses

BMC with SAT Proof Analysis

- BMC Problem: Is property *p* satisfiable at depth *k*?

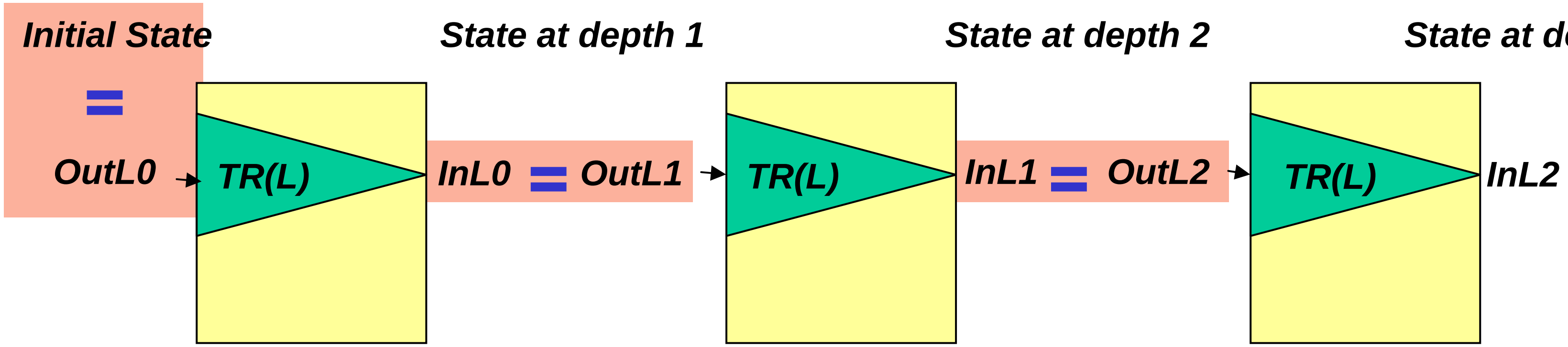


- Suppose no bug at depth *k* because *p* is unsatisfiable
  - Derive an unsatisfiable core *R(k)* using SAT solver [ZM03, MA03]
  - *R(k)* is sufficient for the original problem to be unsatisfiable

- Abstraction based on Unsat Core [MA03, GGA03]
  - Abstract model with core *R(k)* implies correctness at (up to) depth *k*
  - If *k* is sufficiently large, the abstract model may be correct for *k'* > *k*
  - Advantage: Typically *R(k)* is much smaller than entire design (10%) for shallow properties

Latch Interface Abstraction: Intuition

[Gupta *et al.* ICCAD 03]



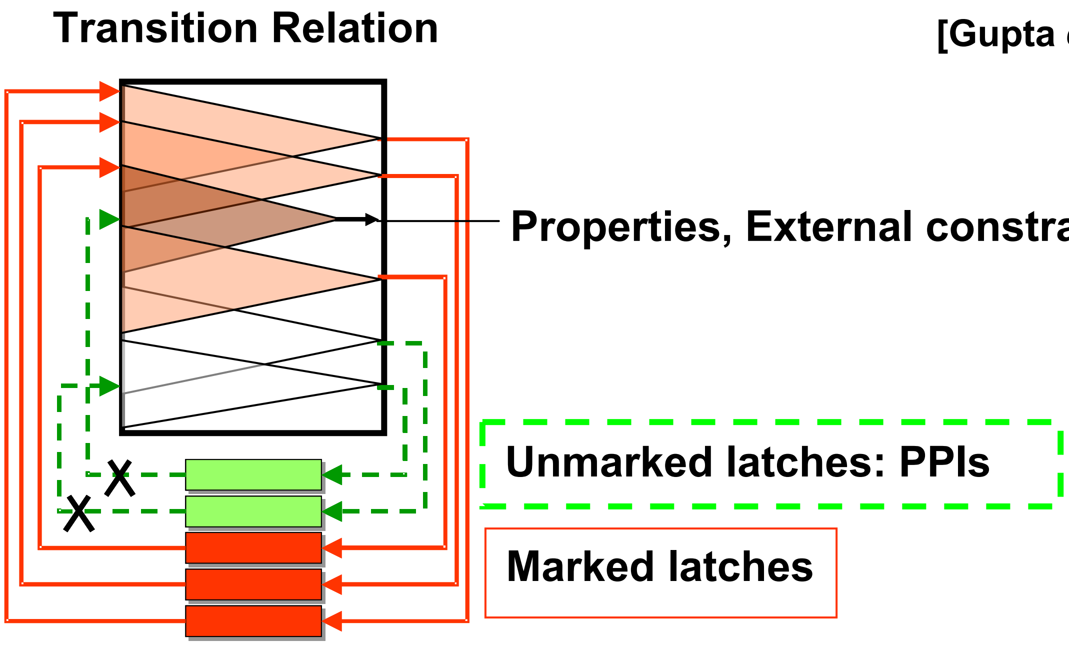
- ❑ **Latch Interface Constraints *IF(L)***
  - **Example**

    IF(L)* = *{OutL0 = initial state(L), InL0 = OutL1, InL1 = OutL2 }

- ❑ **Abstraction focuses on *Marked Latches***
  - **Some latch interface constraint belongs to Unsatisfiable Core**
  - ***Marked_Latches* = { *L* | *IF(L)* ∩ *R(k)* is not empty }**

Latch Interface Abstraction

[Gupta *et al.* ICCAD 03]



Transition Relation

Properties, External constraints

Unmarked latches: PPIs

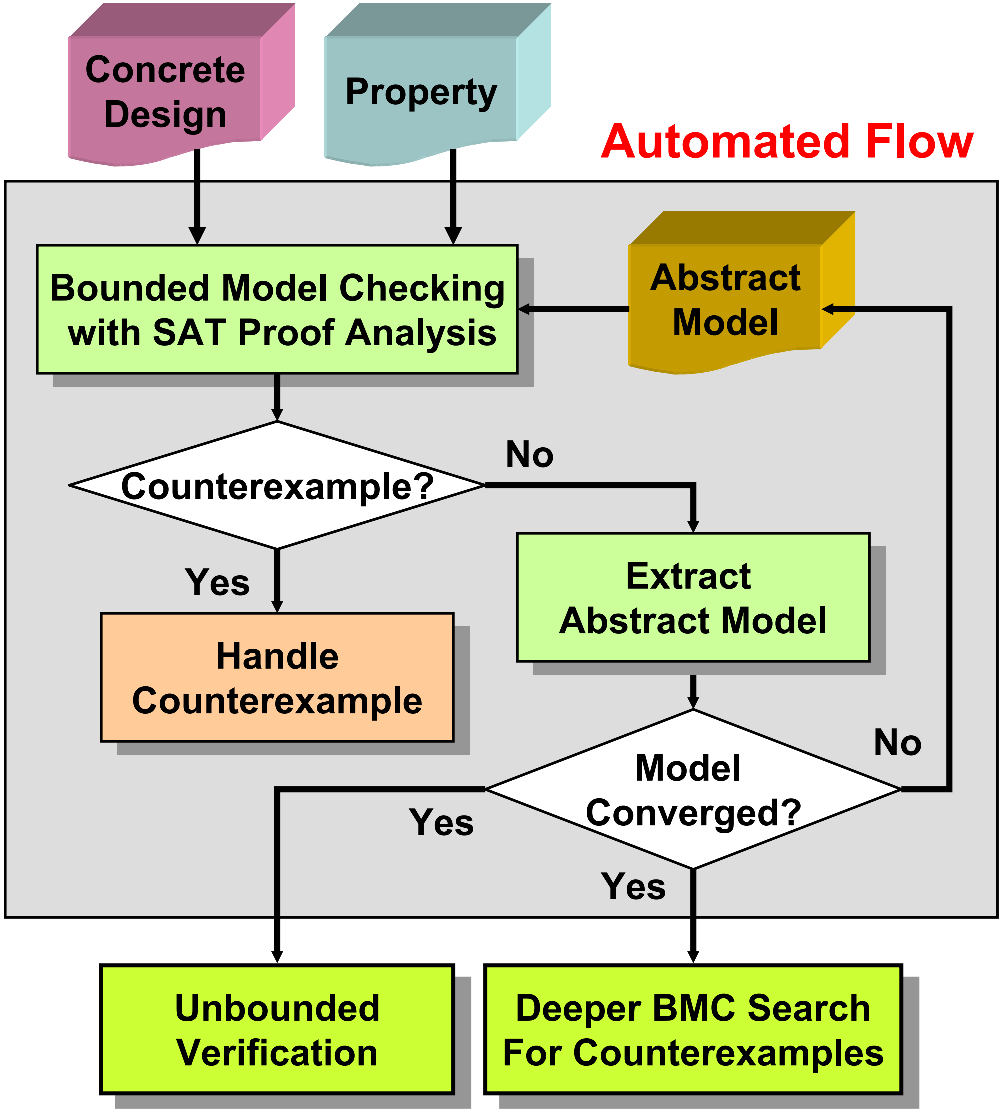
Marked latches

- ❑ **Abstract Model**
  - – Combinational fanin cones of properties and external constraints
  - – Combinational fanin cones of latches marked by SAT proof
  - – Unmarked latches are abstracted away as free inputs (pseudo-primary inputs PPIs)
- ❑ **Conservative Abstraction**
  - – A proof of correctness on abstract model guarantees proof of correctness on original design

Proof-Based Iterative Abstraction (PBIA)

- ❑ **Iterative flow**
  - **BMC with Proof Analysis**
  - **Counterexample handling**
  - **Proof-based abstraction**
  - **Iterate (up to convergence of model)**

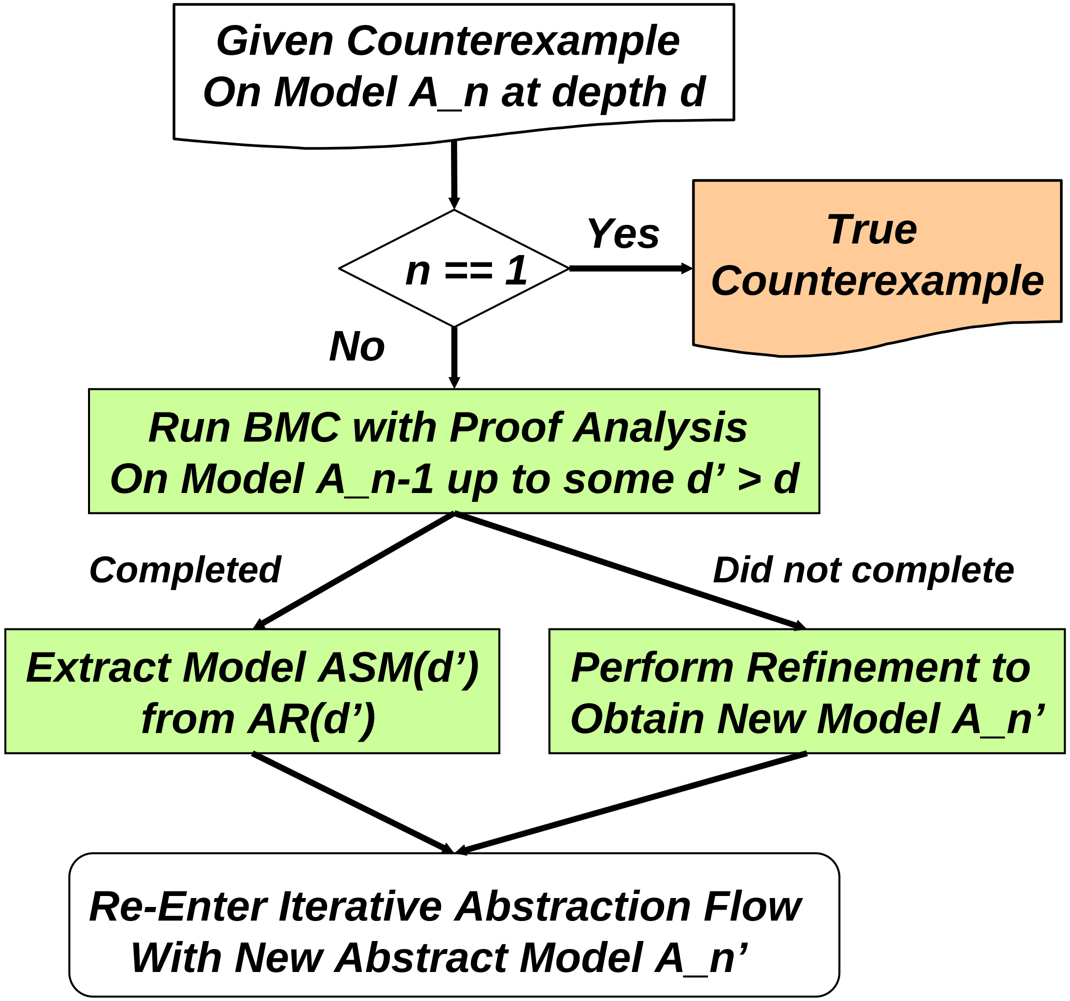
- ❑ **Abstract Models**
  - **Attempt unbounded verification**
  - **Search for bounded counterexamples**



Handling Counterexamples

- ❑ Iteration index n
- ❑ Counterexample may be spurious
- ❑ Two approaches:
  - – Deeper BMC in n-1
  - – Proof-based Refinement

    [Chauhan *et al.* 02]
- ❑ Restart iterative flow



Given Counterexample On Model A_n at depth d

n == 1

Yes

True Counterexample

No

Run BMC with Proof Analysis On Model A_n-1 up to some d' > d

Completed

Did not complete

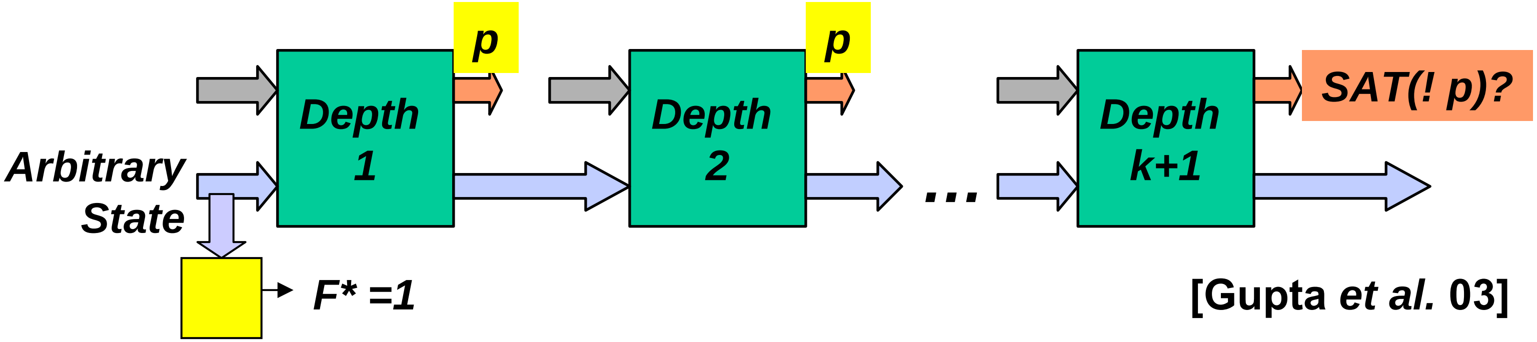
Extract Model ASM(d') from AR(d')

Perform Refinement to Obtain New Model A_n'

Re-Enter Iterative Abstraction Flow With New Abstract Model A_n'

Verification of Abstract Models

- **BDD-based Methods**
  - Traditional symbolic model checking
  - Derive reachability invariants (*F**)
- **SAT-based Methods**
  - Deeper searches for Counterexamples using BMC
  - SAT-based proof by induction, combined with invariants



[Gupta *et al.* 03]

Related Work

- **Iterative Abstraction Refinement**
  - Counterexample driven refinement [Kurshan 92, Clarke *et al.* 00]
  - CEGAR using SAT solver
    - Checking counterexamples [Clarke *et al.* 02, Wang *et al.* 03]
    - Choosing refinement candidates [Chauhan *et al.* 02]
  - Problems: Many iterations, refined model grows too large

- **Proof-based Abstractions**
  - Abstraction without counterexamples [McMillan Amla 03]
  - Interpolants for image set over-approximation [McMillan 03]
  - Problems: Need to handle large concrete models

- **Our approach**
  - Proof-based *Iterative* Abstraction + Refinement (sparingly)
  - Targeted for successive model size reduction
    - False properties: BMC search can go deeper
    - True properties: Unbounded verification methods likely to succeed
  - Iterative framework crucial in handling industry designs

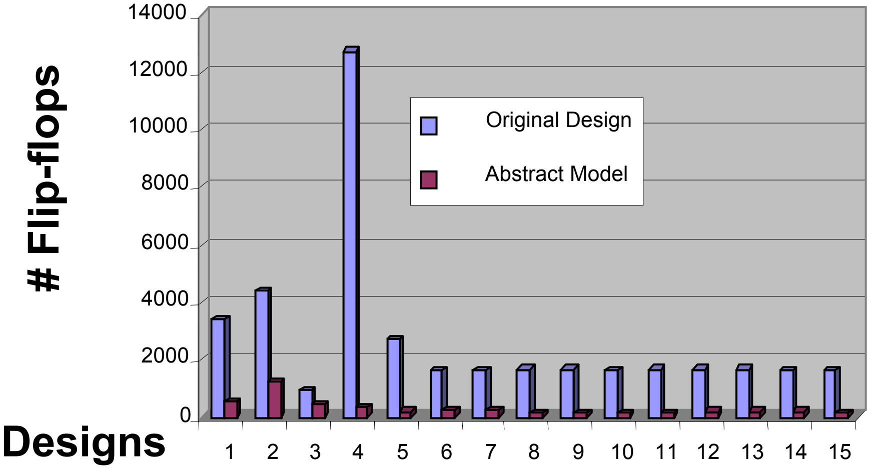
Reducing Unsat Cores

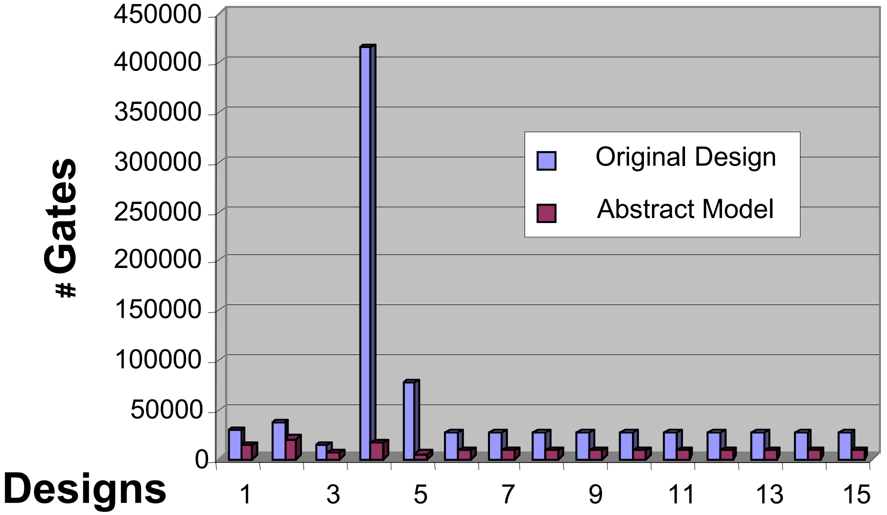
- ❑ Motivation
  - – Initial state values on latches are constants
  - – These constants get pre-processed by SAT solver before making decisions
  - – Many latches get included in Unsat Core due to these initial state values
    - ➢ They may have no impact on why property p is unsatisfiable
- ❑ Key idea: Delay the implications due to initial state values
- ❑ Naïve approach
  - – Mark these as special constraints, and do not propagate implications during Boolean constraint propagation (BCP)
  - – Problem: too much overhead in critical part of SAT solver
- ❑ Our approach: Lazy Constraints!
  - – Convert "eager" constraints to "lazy" constraints
- ❑ Example: Single literal clause (x)
  - – Eager version: (x)
    - ➢ Implications performed in pre-processing phase of SAT solver
  - – Lazy version: (x+y)(x+y')
    - ➢ Implications delayed until SAT search

Application of Lazy Constraints

- Main idea: Delaying implications
- Applications in BMC
  - Method 1: Abstract away those latches where only the initial state constraint is in Unsat Core *R(k)*
  - Method 2: Use lazy constraints for representing initial state constraints for all latches
    - To mitigate performance penalty, use it in (i>0) iterations
  - Method 3: Use lazy constraints for representing single-literal environmental constraints
- Potential benefits in proof-based abstraction
  - Methods 1 & 2: help in finding an “invariant” abstract model
  - Method 3: helps in identifying a sufficient set of environmental constraints – useful for assertion-based design methodology

Results: Derivation of Abstract Models





Order of Magnitude Reductions

- ***Number of Flip-flops***
- ***Number of Gates***

Reduction Across Iterations

Design D4	
Iteration	**#FF Abstract Model**
1	**12716**
2	**330**
3	**187**
4	**84**
5	**73**
6	**71**
7	**71**

Lazy Constraints in PBIA

- Experimental Results

D	Concrete Model		Abstract Model in First Iteration, i =1				Final Abstract Model Generated by Iterative Abstraction												
							No Lazy			With Lazy PPI Constraints									
			Set 1	Set 2	Set 4	Best	Set 1			No LLC, Set 2			LLC, i>1, Set 3			LLC, i>0, Set 4			Best
	#FF	# G	#FF	#FF	#FF	%R	#FF	# I	T(s)	#FF	# I	T(s)	#FF	# I	T(s)	#FF	# I	T(s)	%R
D1	3378	28384	481	480	322	33%	522	9	60476	516	9	50754	294	4	11817	294	4	8993	44%
D2	4367	36471	1190	1190	1146	4%	1223	8	80630	1233	5	39573	1119	9	64361	1136	9	70029	9%
D3	910	13997	507	437	364	28%	433	5	11156	355	9	32520	166	10	29249	196	6	32291	62%
D4	12716	416182	404	330	TO*	18%	369	4	1099	71	6	1203	71	6	1310			TO*	81%
D5	2714	77220	187	137	3	98%	187	2	17	3	5	22	3	3	21	3	2	17	98%
D6	1635	26059	116	111	17	85%	228	6	5958	225	4	5324	148	3	4102	146	2	7	36%
D7	1635	26084	110	110	23	79%	244	3	3028	240	2	3039	155	5	2768	146	2	85	40%
D8	1670	26729	30	30	19	37%	149	3	25	149	3	28	148	3	28	148	2	41	1%
D9	1670	26729	115	115	22	81%	162	3	40	162	3	43	147	3	44	149	2	43	9%
D10	1635	26064	38	38	16	58%	159	2	12	158	3	29	146	3	30	145	2	6	9%
D11	1670	26729	30	30	19	37%	149	3	25	149	3	28	148	3	28	148	2	40	1%
D12	1670	26729	104	98	75	28%	183	4	2119	182	4	2316	182	4	2376	180	2	653	2%
D13	1670	26729	62	61	52	16%	180	2	63	179	2	68	154	3	71	174	3	61	14%
D14	1635	26085	74	71	15	79%	190	3	1352	192	3	1515	154	5	1480	142	3	10	25%
D15	1635	26060	27	27	27	0%	153	3	125	153	3	149	153	3	142	151	3	73	1%

Notes: (a) LLC denotes Lazy Latch Constraints (b) TO* denotes time out in first iteration

- **Average reduction in #FFs in Unsatisfiable Core: 45%**
- **Potentially useful in other applications, e.g. Interpolants**

Results: Final Verification of Abstract Models

Design	Original Abstraction				+Lazy Constraints				+Sufficient External Constraints			
	# FF	# Env	Proof?	Time (s)	# FF	# Env	Proof?	Time (s)	# FF	# Env	Proof ?	Time (s)
D1	522	142	No	TO	294	142	No	TO	163	11	Yes	58
D2	1223	142	No	TO	1119	142	No	TO	994	23	No	TO
D3	433	0	No	TO	166	0	No	TO	166	0	No	TO
D4	369	0	No	TO	71	0	Yes	29	71	0	Yes	29
D5	187	0	No	TO	3	0	Yes	1	3	0	Yes	1
D6	228	264	No	TO	146	264	No	TO	17	87	Yes	18
D7	244	264	No	TO	146	264	No	TO	23	93	Yes	26
D8	149	264	No	TO	148	264	No	TO	18	86	Yes	1
D9	162	264	No	TO	147	264	No	TO	20	89	Yes	21
D10	159	264	No	TO	145	264	No	TO	16	87	Yes	4
D11	149	264	No	TO	148	264	No	TO	18	86	Yes	18
D12	183	264	No	TO	180	264	No	TO	76	112	Yes	70
D13	180	264	No	TO	154	264	No	TO	29	91	Yes	98
D14	190	264	No	TO	142	264	No	TO	14	88	Yes	22
D15	153	264	No	TO	151	264	No	TO	28	93	Yes	22

- **None of 15 difficult industry designs could be proved correct, even after significant reduction in size of abstract model**
- **With additional techniques (use of lazy constraints, identification of sufficient external constraints), *13 of 15* designs proved correct**

Proofs for Designs with Embedded Memory

Extensions of EMM+BMC for Proofs

[Ganai et al. DATE 05]

- Provide SAT-based inductive proofs
  - Precise modeling of arbitrary initial memory state
    - By introducing new variables for each depth of unrolling
    - But constraining them when there is no write
  - This can provide proofs in addition to falsification with EMM
- Combine EMM with Proof-based Iterative Abstraction
  - Identify relevant memory modules/ports
    - If the control latch for a memory port is not marked in Unsat Core, then that memory module/port can be abstracted away
  - This generates smaller abstract models for verification
- Improvement (space/time)
  - 1-2 orders of magnitude over explicit modeling

SAT-based
Unbounded Model Checking (UMC)

Symbolic Model Checking

[McM 90, BCL+94]

X: present state variables
Y: next state variables
W: input variables

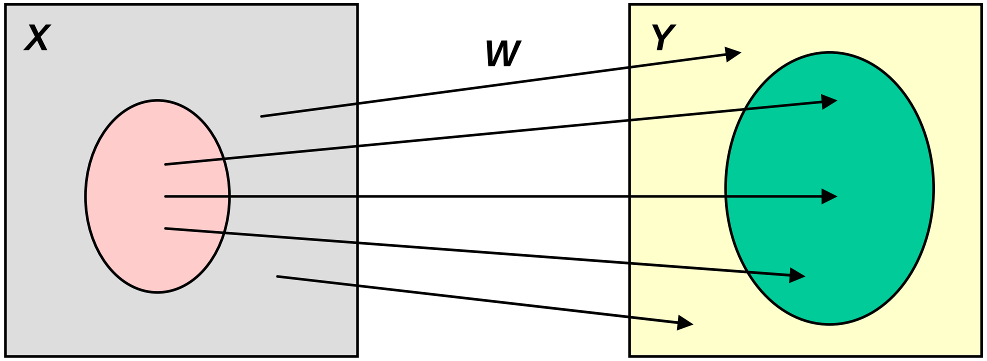
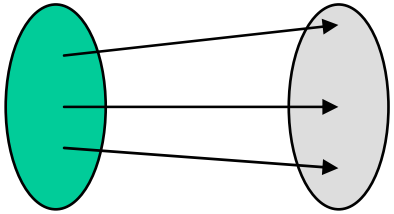


Image Computation

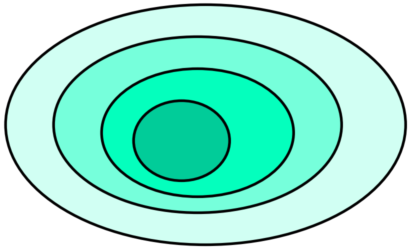
$$\mathit{Image}(Y) = \exists\ X, W.\ \ T(X,W,Y) \wedge \mathit{From}(X)$$

- **Related operations**

Pre-Image Computation



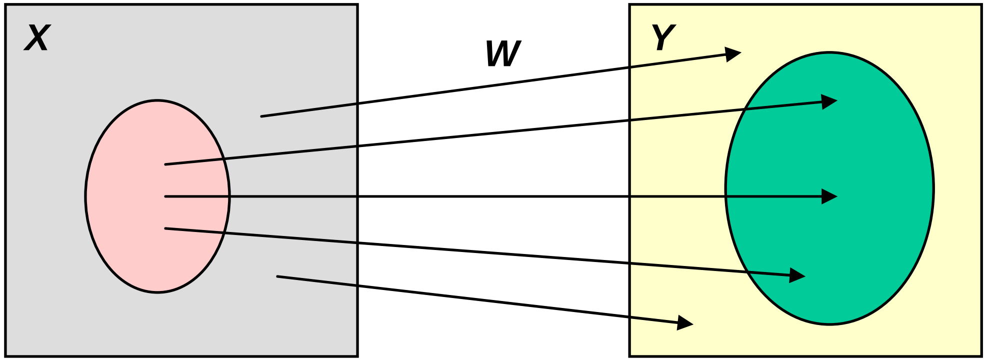
Fixpoint Computation



- **Core steps of many applications**
  - **equivalence checking, reachability analysis, model checking …**

SAT+BDD Image Computation

X: present state variables
Y: next state variables
W: input variables



$$Image(Y) = \exists\ X, W.\ \ T(X,W,Y) \wedge\ From(X)$$

	Standard	**SAT+BDD Approach**
State Sets	**BDDs**	**BDDs**
Transition Relation	**BDDs**	**CNF Formula**
Conjunction+Quantification	**BDD operations**	**SAT + BDD operations**

[BCL+94, CBM 89]

- **Other Approaches** **[Abdulla *et al.* 00, Clarke *et al.* 00]**
  - **Perform explicit quantification, use RBCs or BEDs**

Image Computation: SAT+BDDs

[Gupta *et al.* FMCAD 00]

$$\overbrace{Image(Y)}^{BDD(s)} = \underset{\downarrow}{\boxed{\exists\ X, W.}}\ \overbrace{T(X,W,Y)}^{CNF}\ \underset{\downarrow}{\boxed{\wedge}}\ \overbrace{From(X)}^{BDD(s)}$$

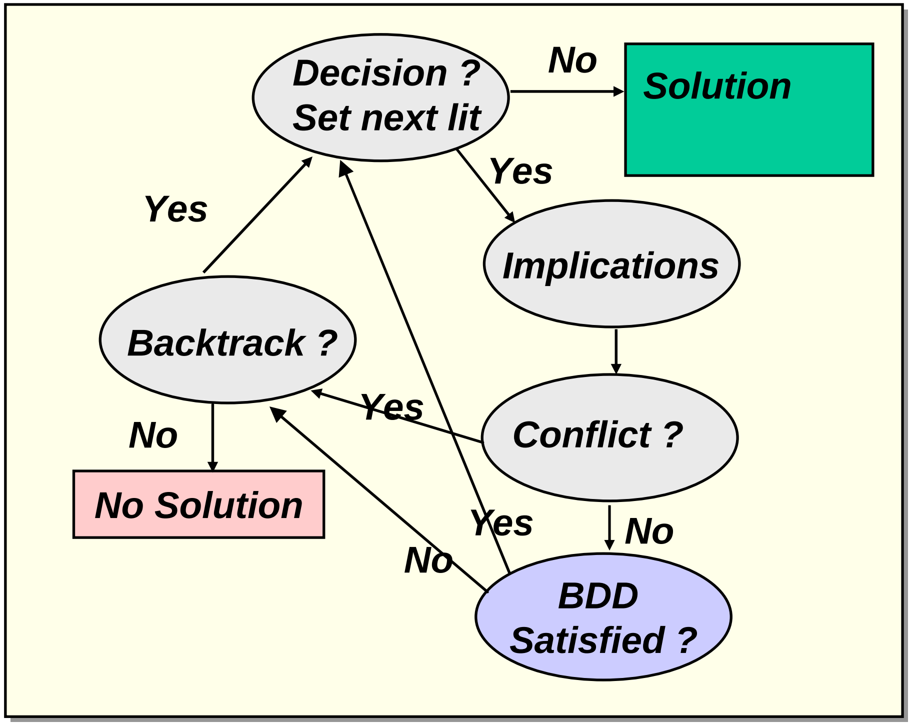
Enumerating all SAT solutions over Y *BDD Bounding*

- ❑ **Representation Framework**
  - – **BDDs for From(X), Image(Y)**
    - ➢ **can also use a list of disjunctively decomposed BDDs when a monolithic BDD gets too large**
  - – **CNF for Transition Relation**

- ❑ **Operations**
  - – **BDD Bounding**
  - – **Enumeration of all SAT solutions for T (on Y)**

BDD Bounding in SAT

- **Main idea**
  - A BDD can be used to *constrain* the variables of a SAT search space
  - If partial assignment in SAT satisfies BDD, then continue, else backtrack
- **Advantages in Image Computation**
  - Leads to early pruning of search space due to From set
  - Can be used to avoid re-enumerating solutions for Image set

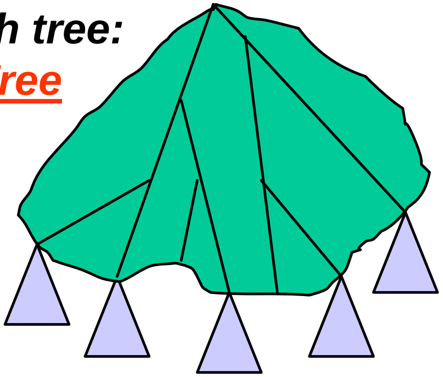


Enumerating All Solutions

- Search space: all values of variables (X, W, Z, Y)

	BDD DAGs	*SAT Decision Tree*
Flexibility	*Low (fixed ordering)*	*High (no restriction on decisions)*
Solution Sharing	*High (canonical)*	*Low (non-canonical)*

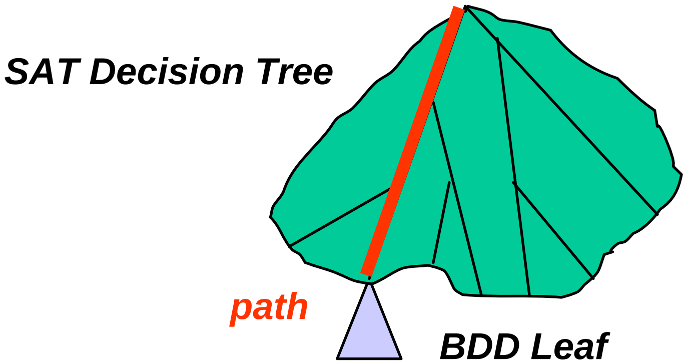
- Strategy: keep the flexibility, but avoid cube enumeration



Top level search tree:
SAT Decision Tree

Leaves of SAT search tree:
BDD sub-problems

BDDs at SAT Leaves



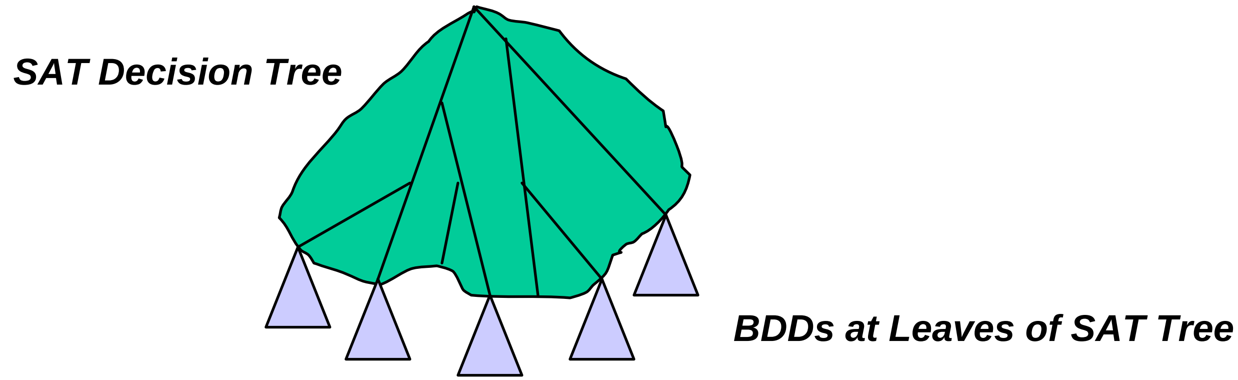
- Leaf sub-problem at the end of a partial assignment *path*
  - convert unsatisfied clauses in CNF to BDDs
  - cofactor each of them with the partial assignment along the path
  - cofactor From(X) with the partial assignment along the path
  - solve the following problem:

Standard BDD-based image computation

$$\mathit{Solution}(Y) = \mathit{path}(Y') \wedge \boxed{\exists\, X'', W'', Z''.\ \left(\Pi_i\ \mathit{cof\text{-}Unsat\text{-}C}_i\,(X'',W'',Z'',Y'')\right) \wedge \mathit{cof\text{-}From}(X'')}$$

Fine-grained conjunctive partition provides greater scope for early quantification

When to trigger BDD Leaves?



- SAT decisions provide a disjunctive decomposition of the problem
  - Similar to BDD-based disjunctive decomposition approaches

    [CBM 89, CCQ 99, MKR+ 00]
- Boundary between SAT and BDDs allows a time vs. space tradeoff
- Adaptive triggering of BDD sub-problems
  - Heuristics based on number of unassigned variables, BDD sizes etc.
  - Timeout mechanism: If BDD sub-problem blows up, go back to SAT for further splitting

Experimental Results (1)

Setup

- VIS, SAT+BDD experiments run on Sun Ultra 297MHz, 1 GB machine
- dynamic reordering turned on
- 10 hours time limit (usually)

Good performance on relatively "easy" circuits

Name	L	PI	PO	Comb	Vars	CNF	Step	Reached	Moon	VIS	SAT+BDD		
								States	Time(s)	Time(s)	Time(s)	Leaves	BB
s1269	37	18	10	606	456	1244	10 c	1.31E+09	891	3374	2688	1814	1258
s1512	57	29	21	837	496	1301	1024 c	1.66E+12	2016	2362	5753	3069	3
s3271	116	26	14	1688	1183	3219	17 c	1.32E+31	4833	17933	14793	2415	633
s3330	132	40	73	1921	846	2114	9 c	7.28E+17	10316	20029	3967	574	42

Experimental Results (2)

Name	L	PI	PO	Comb	Vars	CNF	Step	Reached	VIS	SAT+BDD			*X*
								States	Time(s)	Time(s)	Leaves	BB	
prolog	136	36	73	1737	1027	2607	4	1.73E+17	25003	490	40	16	***50***
							9 c	7.28E+17	TT > 10h	5927	167	17	
s5378	164	35	49	2943	1012	2819	8	2.24E+17	57986	5957	73	117	***10***
							45 c	3.17E+19	SS > 30h	60500	1358	932	
s1423	74	17	5	731	574	1464	11	7.99E+09	7402	2322	308	114	***3***
							13	7.96E+10	TT > 10h	16724	528	127	
s3384	183	43	26	1868	1187	2853	4	4.41E+26	24875	787	834	28	***30***
							5	8.19E+30	TT > 10h	2882	1178	30	
s9234.1	211	36	39	5808	2316	6548	7	2.33E+13	2360	8030	112	96	***1/3***
							9	6.47E+14	11577	TT > 10h			
s13207.1	638	62	152	8589	3464	8773	9	6.45E+25	3210	12944	47	59	***1/4***
							14	2.14E+29	28600	TT > 10h			

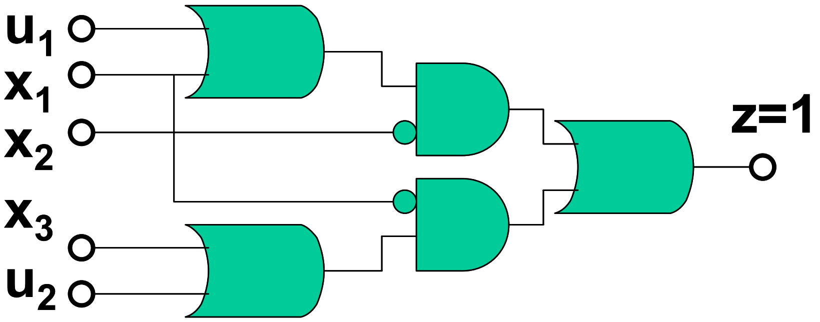
- **Completed traversal on prolog, and s5378**

Purely SAT-based Unbounded Model Checking (UMC)

SAT-based Pre-Image Computation

```
SAT-EQ($f$,A,B) { // calculate $\exists_B f(A,B)$
  C=$\emptyset$; // initialize constraint
  while (SAT_Solve ($f$=1$\wedge$C=0)=SAT) {
    $\alpha$=get_assignment_cube();
    c=get_enumerated_cube($\alpha$,A); // obtain, $\exists_B \alpha$
    C=C$\vee$c;
  }else return C; } // return when no more solution
```

[McMillan CAV 02]

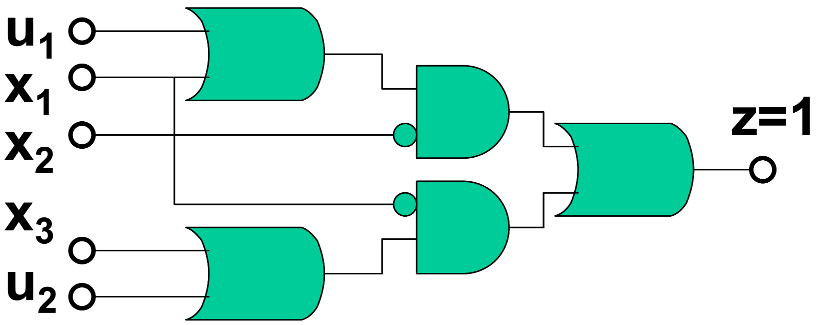


u_1,u_2	: input variables
x_1,x_2,x_3	: state variables
F	$= x_1'(x_3+u_2)+x_2'(x_1+u_1)$
Goal	: $\exists u_1 u_2$ F (All state cube solutions)
Solution	: $x_1' + x_2'$

Steps of cube-wise enumeration (Example)

1. First Enumeration: $u_1=1$, $x_2=0$, $u_2=?$
2. Blocking constraint: x_2
3. Second Enumeration: $x_1=0$, $x_2=1$, $u_2=1$
4. Blocking constraint: $x_2 \cdot (x_1+x_2')=x_2 \cdot x_1$

Motivation



u_1,u_2 : input variables

x_1,x_2,x_3 : state variables

$F = x_1'(x_3+u_2)+x_2'(x_1+u_1)$

Goal : $\exists u_1 u_2 F$

Find all state cube solutions

(solution: $x_1' + x_2'$)

Steps of Blocking Clause (BC) Approach

1. First Enumeration: $u_1=1, x_2=0, u_2=?$
2. Blocking constraint: x_2
3. Second Enumeration: $x_1=0, x_2=1, u_2=1$
4. Blocking constraint:
   $x_2 * (x_1 + x_2') = x_2 * x_1$

Number of Enumerations: 2

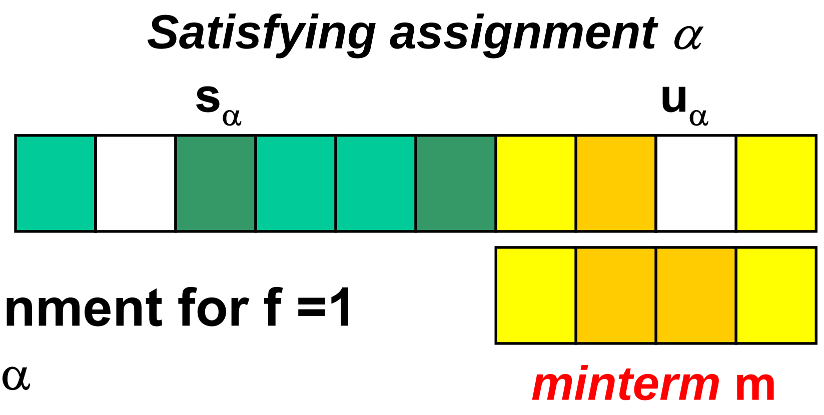
- Can we capture more new solutions per enumeration than by cube-wise enumeration approach?
- Can we efficiently represent the solutions to mitigate the space-out problem?
- Can we use better SAT solver that uses circuit information efficiently?

Basic Idea (1/2)

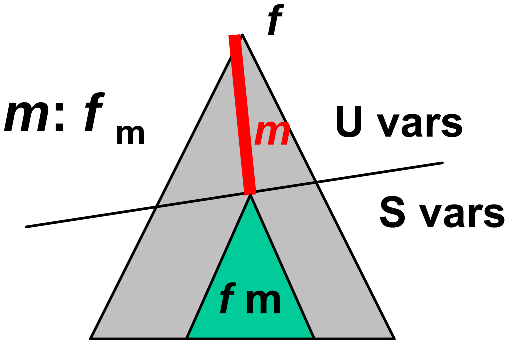
Theorem 1

Let,

- **$\alpha : V_\alpha \rightarrow \{0,1\}$ be the satisfying assignment for f =1**
- **s_α be the satisfying <u>state cube</u> for α**
- **u_α be the satisfying <u>input cube</u> for α**



Consider function *f* cofactored by input minterm *m*: f_m
If *m* is satisfying ($\in u_\alpha$), then $s_\alpha \subseteq f_m$



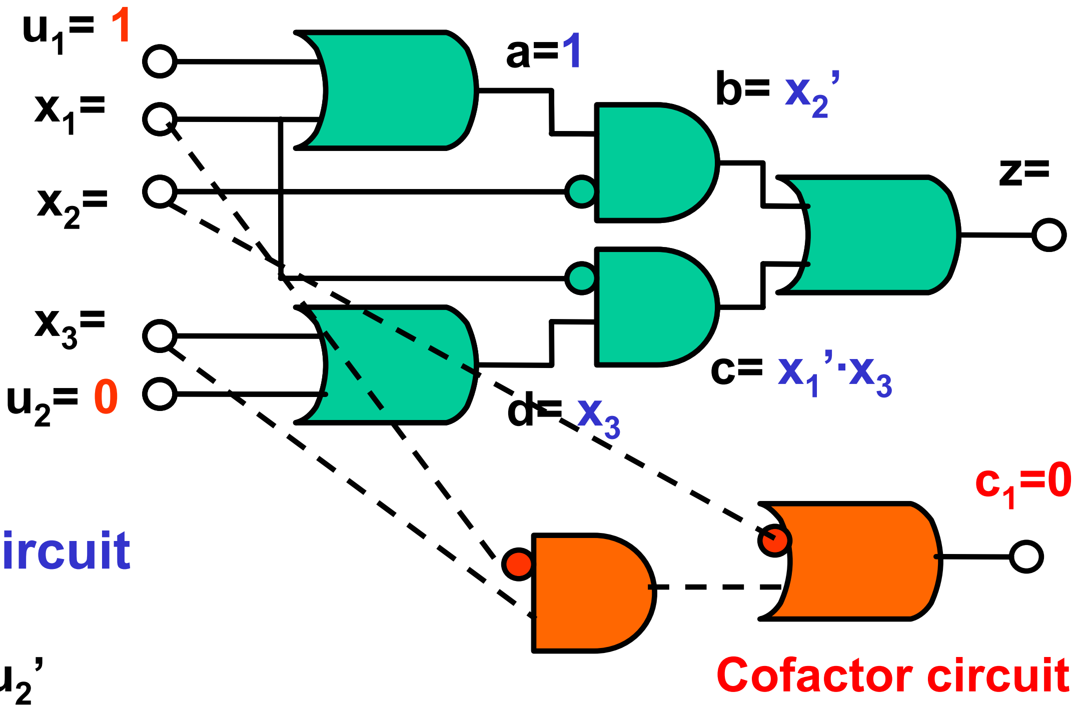
Claims

- **Cofactor f_m subsumes satisfying solutions captured by cube s_α**
- **Therefore, cofactor-based enumeration requires fewer SAT solver enumerations than a cube-based enumeration**

Basic Idea (1/2) - Example

Solution to f =1

$f = x_1'(x_3+u_2)+x_2'(x_1+u_1)$
$s_\alpha = x_1 \cdot x_2' \cdot x_3$
$u_\alpha = u_1 \cdot u_2'$



Our approach: Cofactor circuit

- **Pick a minterm, m = $u_1 \cdot u_2'$**
- **Cofactor, $f_m = c_1 = x_1' \cdot x_3 + x_2'$**
  Note f_m captures more than one cube

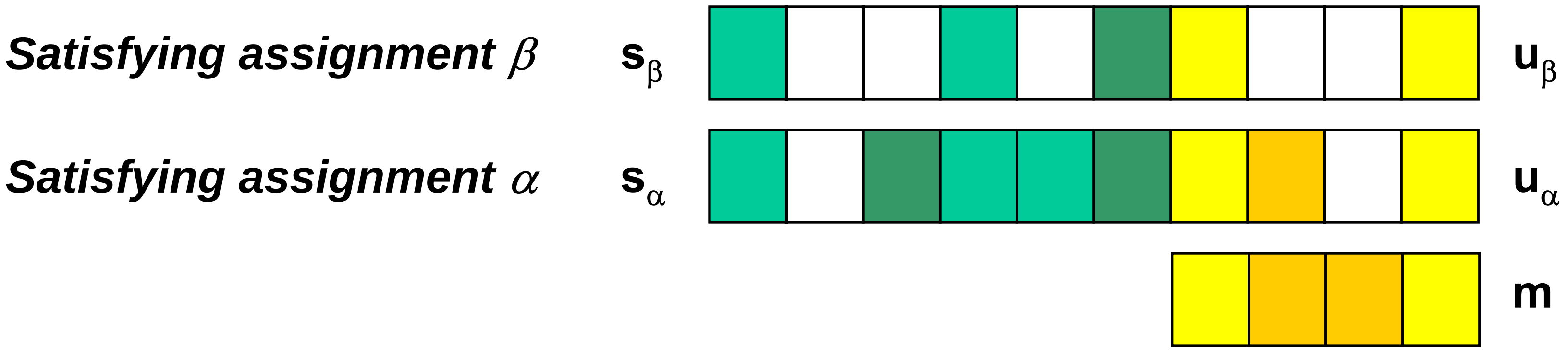
Clearly, $s_\alpha \subseteq f_m$

Basic Idea (2/2)

Theorem 2

Let

- **α and β be two satisfying assignments for *f*=1**
- **β represents a solution enlargement of α**


Satisfying assignment β
s_β
u_β
Satisfying assignment α
s_α
u_α
m

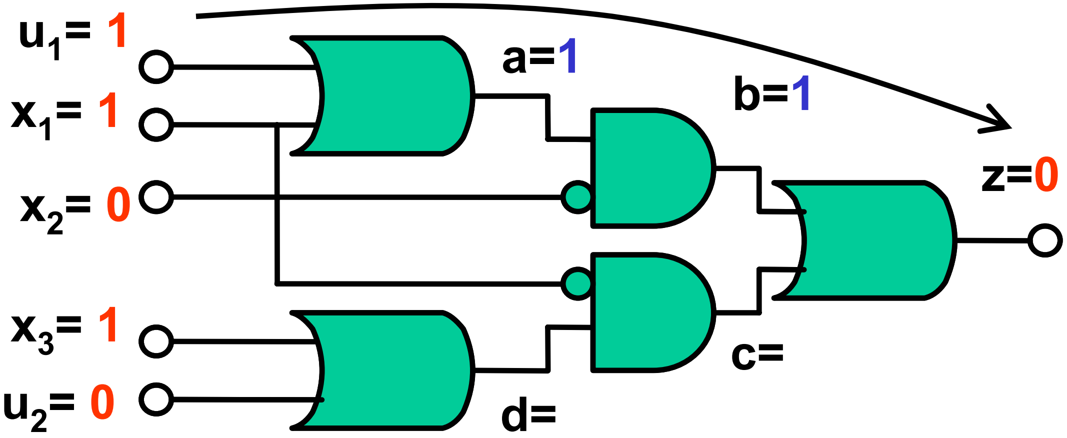
- **If input minterm m $\in u_\alpha$, then $s_\beta \subseteq f_m$**
  i.e. a cofactor subsumes all state cube enlargements

Enlargement of state cubes is unnecessary!

Basic Idea (2/2) – Example

Solution to f=1

$f = x_1'(x_3+u_2)+x_2'(x_1+u_1)$
$s_\alpha = x_1 \cdot x_2' \cdot x_3$
$u_\alpha = u_1 \cdot u_2'$



Cube enlargement (redrawing implication graph [McMillan CAV'02])

1. **Constraints:** $z=0, u_1=1, u_2=0, x_1=1, x_2=0, x_2=1$
2. **Implication:** $u_1=1 \rightarrow a=1, (x_2=0, a=1) \rightarrow b=1, b=1 \rightarrow z=1$ (conflict)
3. **Conflict Analysis:** $u_1=1, x_2=0$ (reasons for conflict)

Enlarged cube: $s_\beta = x_2'$, $u_\beta = u_1$

Our approach (cofactor circuit) : $f_m = x_1'.x_3 + x_2'$

Clearly, $s_\beta \subseteq f_m$

Cofactoring-based Quantification using SAT

[Ganai *et al.* ICCAD 04]

```
mSAT-EQ($f$,A,B) { // calculate $\exists B\ f(A,B)$
  C = $\emptyset$; // initialize constraint
  while (SAT_Solve($f$=1$\wedge$C=0) = SAT) {
    $\alpha$ = get_assignment_cube();
    m = get_satisfying_input_minterm($\alpha$,B);
    $f_m$= cofactor_cube($f$, m);
    C = C $\vee$ $f_m$; // add cofactor blocking constraint
} else return C; } // return when no more solution
```

Efficient Hybrid (circuit+CNF) SAT Solver

Efficient state space representation using reduced circuit graphs

Iteration #1

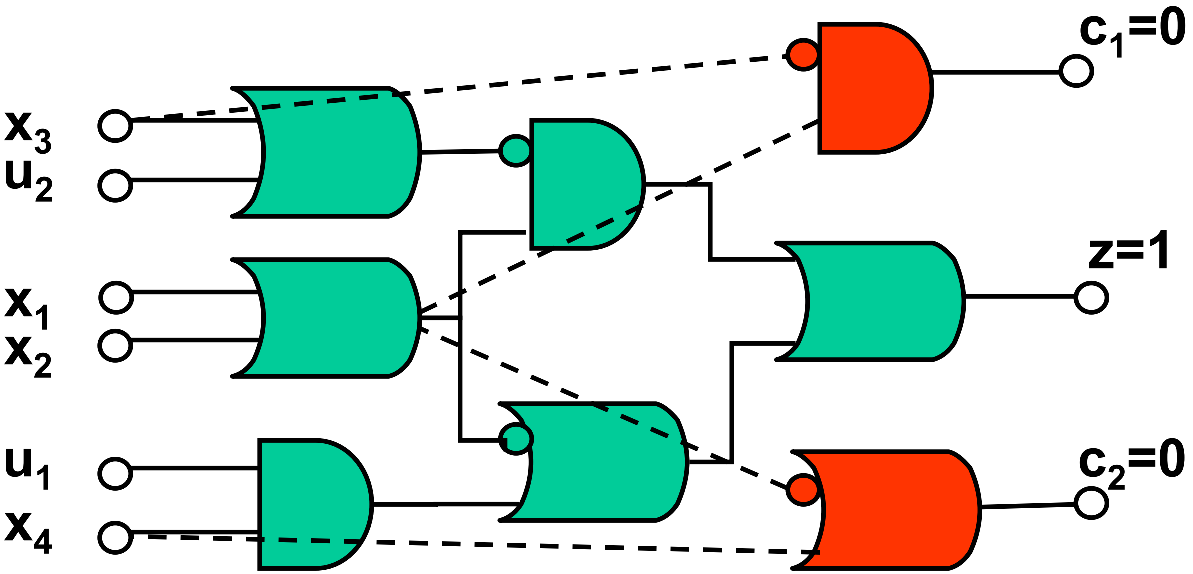
- Sol^n: $x_1 \cdot x_3' \cdot u_2'$
- Pick: $u_1=0$

Cofactor: $c_1=x_3' \cdot (x_1+x_2)$

Iteration #2

- Sol^n: $x_3 \cdot u_1 \cdot x_4$
- Pick: $u_2=0$

Cofactor: $c_2= x_1' \cdot x_2'+x_4$

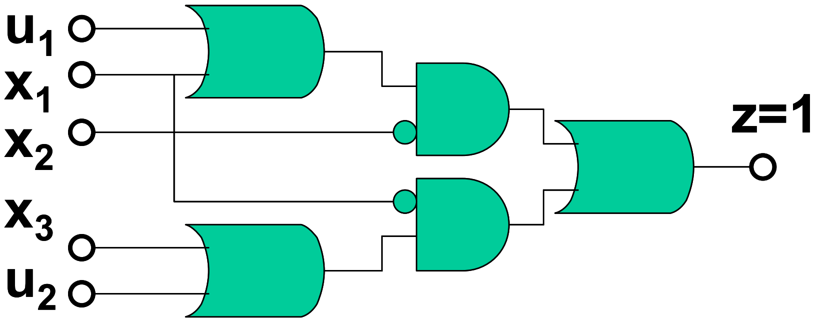


Heuristics for Choosing Input Minterms

Input minterm choice makes a difference

1. First enumeration: u_1=1, x_2=0, u_2=?
2. Pick u_2=0 (instead of u_2=1)
3. Cofactor F with u_1=1,u_2=0
   $F(u_1=1,u_2=0) = x_1'x_3+x_2'$
4. Blocking constraint: $(x_1+x_3')\cdot x_2$

Need one more enumeration to complete



$F = x_1'(x_3+u_2)+x_2'(x_1+u_1)$

Goal : $\exists u_1u_2\, F\ (= x_1' + x_2')$

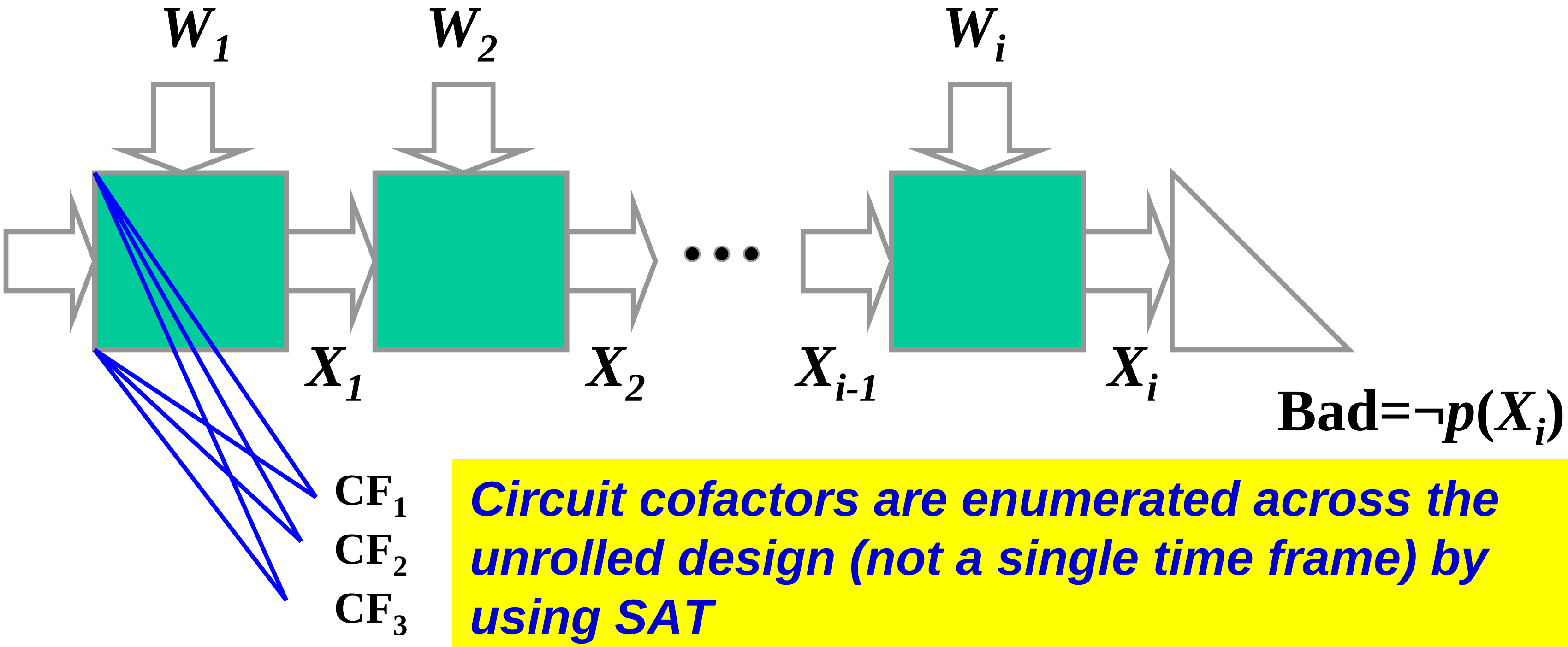
Proposed Heuristics

- Hr: uses a minterm chosen randomly
- H1: uses structure information of the circuit like fanouts
- H2: uses SAT justification frontiers
- H3: uses SAT justification frontiers and fanout information
- H4: uses SAT justification frontiers and latch frontiers information
- H5: uses SAT justification frontiers, fanout and latch frontiers

SAT-based UMC using Circuit Cofactoring (CC)

[Ganai *et al.* 04]

- ❑ Symbolic backward traversal using unrolled TR



Circuit cofactors are enumerated across the unrolled design (not a single time frame) by using SAT

- ❑ Issues in practice
  - State sets (represented as circuit cofactors) may blow up
  - Performance is not as good as SAT-based BMC (search for bugs), which avoids computation of state sets
- ❑ Complementary to BDD-based UMC for deriving proofs

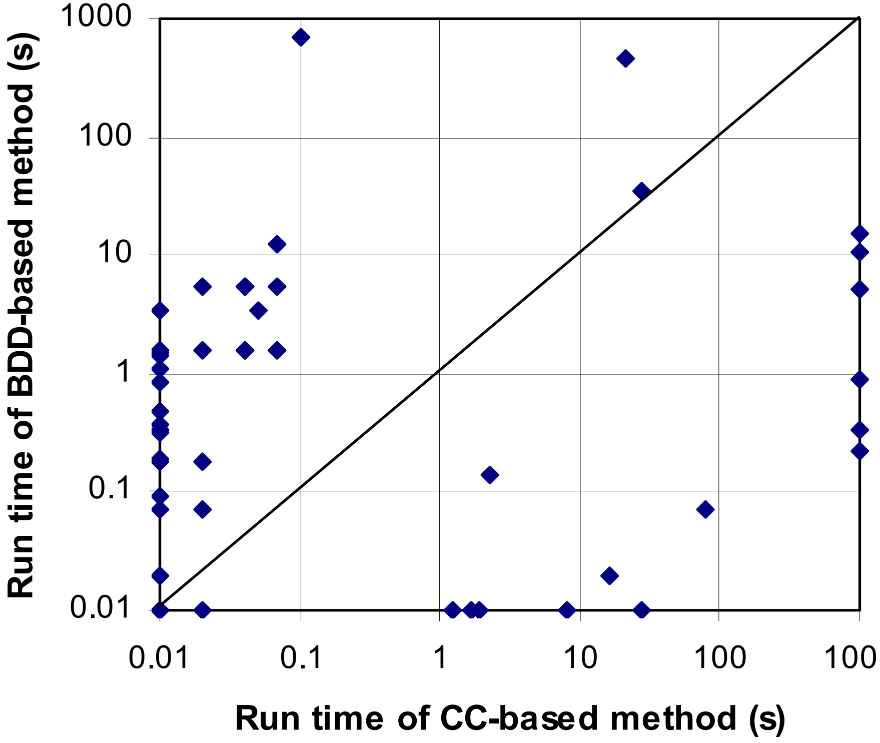
Experiments for CC-based UMC

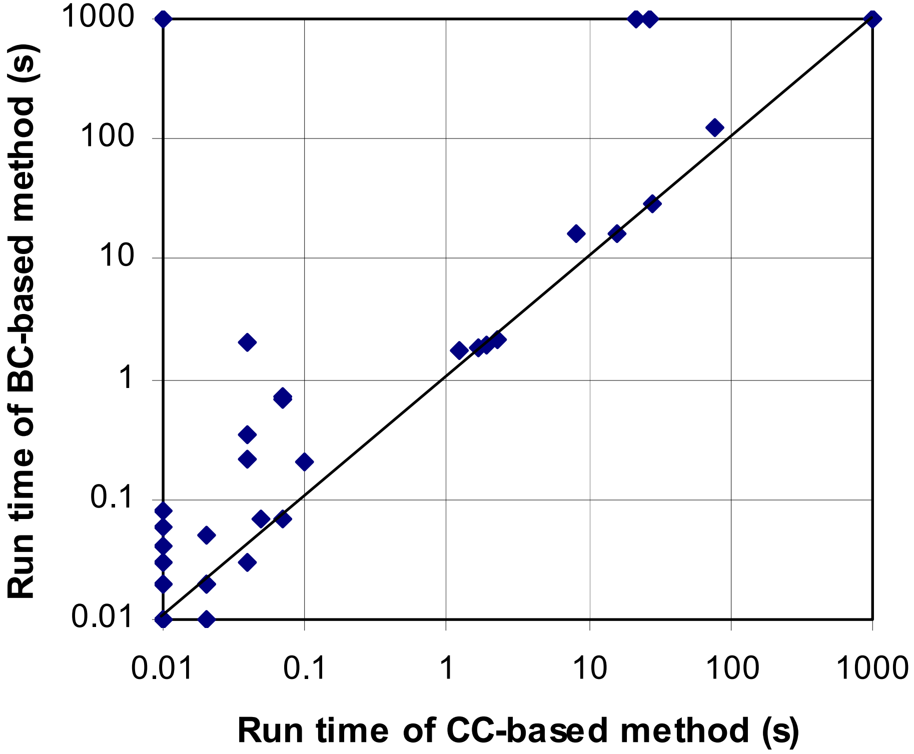
🟨 Blocking clause with redrawing of implication graph using hybrid solver (BC)

🟦 Circuit Cofactoring (CC)

k	P	D1 #FF=168 #Gates=2.5k		D2 (w/ Env) #FF=294 #Gates=9.6k		D3 (w/ Env) #FF=1k #Gates=16k		D4 #FF=1.7k #Gates=16k		D5 #FF=1.7k #Gates=15k	
1	#E	10	1	6	1	~~54k*~~	1	870	1	981	1
	T(s)	0	0	0	0	~~>3H~~	0	116	0	105	0
	MB	3	3	4	4	~~30*~~	5	9	5	8	5
2	#E	24	1	582	4	-	1	~~27k*~~	1	~~36k*~~	1
	T(s)	0	0	12	0	-	0	~~>3H~~	0	~~>3H~~	0
	MB	3	3	5	5	-	7	~~50*~~	6	~~39*~~	6
3	#E	~~86K*~~	1	38k	7	-	10	-	2	-	2
	T(s)	~~>3H~~	0	2268	0	-	3	-	1	-	1
	MB	~~19*~~	3	9	5	-	12	-	8	-	8
4	#E	-	92	8K	19	-	69	-	4	-	3
	T(s)	-	0	3080	0	-	73	-	3	-	3
	MB	-	3	11	7	-	48	-	10	-	10

Comparison of Circuit Cofactoring (CC) w/ BDDs and w/ Blocking Clauses (BC)



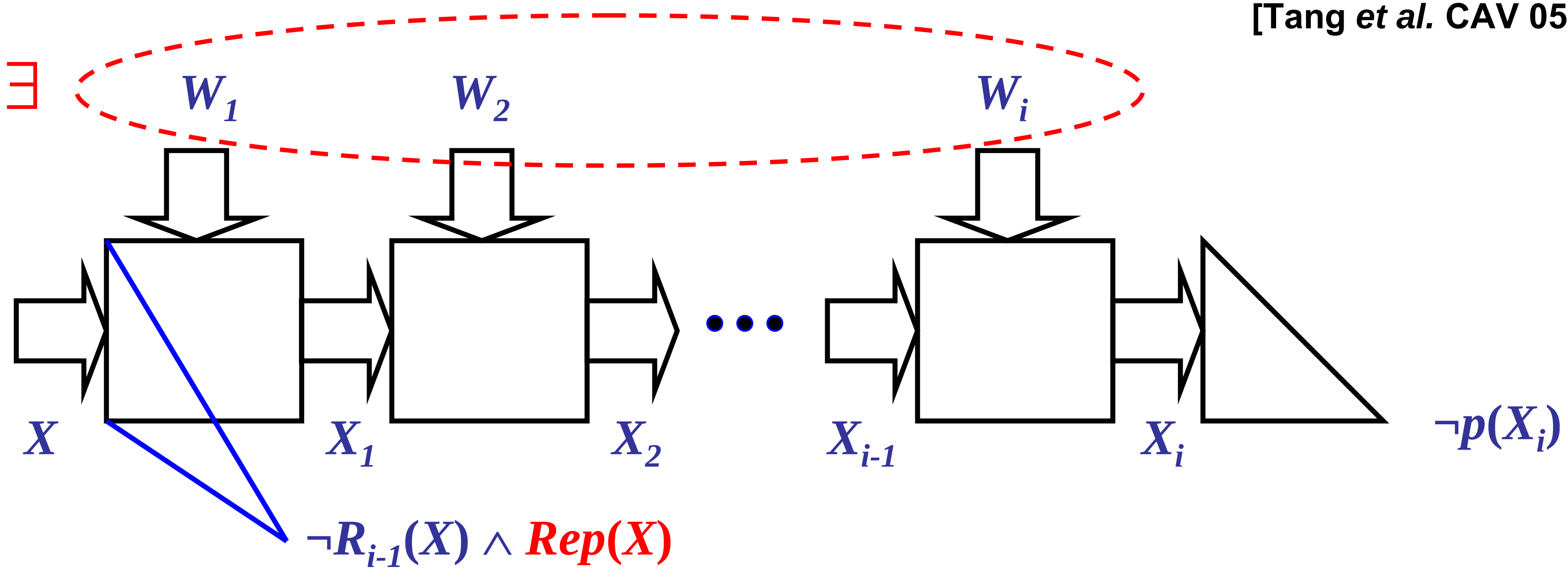


VIS Benchmarks: 102 safety properties

- 68 cases CC does better, 16 cases BDD does better (Note the complimentary strengths)
- CC does better than BC in almost all cases

Symmetry Reduction with SAT-based UMC

[Tang *et al.* CAV 05]



- ❑ Used the Representative Predicate Rep(X) to constrain pre-images
- ❑ Reduced number of cofactor enumerations
  - – Non-representative states are not enumerated
- ❑ Simplified SAT problems
  - – More constrained search space for SAT solver

CC Approach Summary

- **Cofactoring-based quantification using SAT**
  - Guaranteed to require fewer enumerations compared to cube-wise enumerations (order-of-magnitude better in practice)
  - Captures more newer states compared to cube-wise approach
  - Uses efficient representation for states
  - Uses efficient hybrid SAT solver

- **Improved SAT-based UMC**
  - Performs quantification on unrolled designs
  - Orders of magnitude improvement in performance on industry designs and public benchmarks compared to cube-wise enumeration
  - Successfully proved correctness of property on an industry design for which all other approaches failed

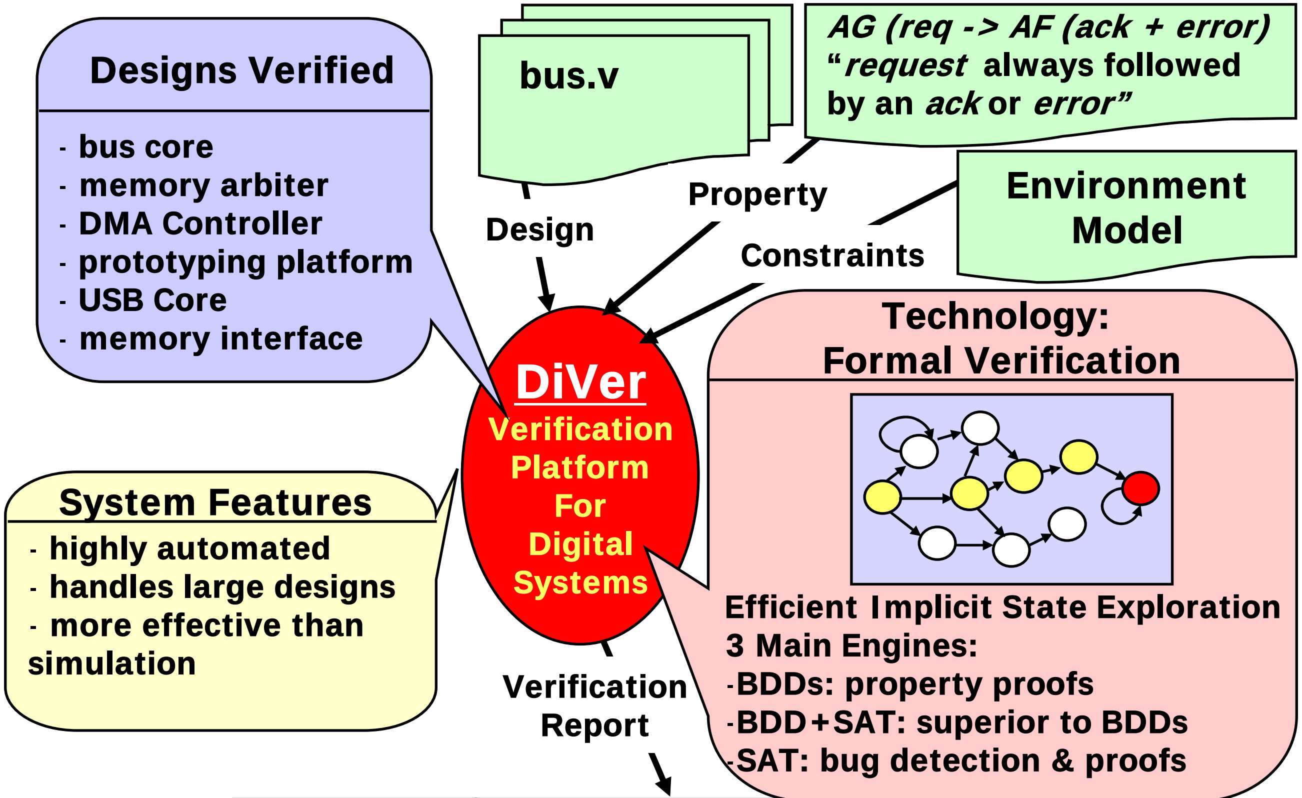
- **Future work: Combine this method with interpolation-based approach (*McMillan CAV'03*)**

SAT-based UMC Summary

Work	Solver	Quantification Strategy	State	Strengths / Weakness
Gupta et al. *FMCAD ‘00*	CNF-based SAT	Enumeration of solution cubes and BDD quantification at intermediate SAT sub-tree. Uses BDDs to block solution.	BDD	❑ Control over BDD quantification based on size of subproblem ❑ BDDs do not scale, not robust
McMillan *CAV ‘02*	CNF-based SAT	Enumeration of solution cubes. Uses blocking clauses to prevent same solution.	CNF	❑ Redrawing of implication graph for solution enlargement ❑ Captures only one solution cube ❑ Representation is inefficient
Kang et al. *DAC ‘03*	CNF-based SAT	Enumeration of solution cubes. Uses blocking clauses.	CNF	❑ Logic minimizer to reduce size of blocking clauses ❑ Captures only one solution cube
Sheng et al. *DAC ‘03*	ATPG (PODEM)	Decisions on inputs. Uses satisfying cut-set to prevent same solution.	BDD	❑ Reduces number of backtracks ❑ Captures only one solution cube ❑ BDDs do not scale, not robust
McMillan *CAV’03*	CNF	Uses interpolants derived from SAT proofs.	CNF	❑ No Quantification ❑ Over-approximated set of states
Ganai et al. *ICCAD’04*	Hybrid	Uses circuit cofactoring to capture solutions.	Red. graph	❑ Circuit-cofactor captures more than one solution cube ❑ Order-of-magnitude improvement

NEC’s DiVer (VeriSol)
Hardware Verification Platform

DiVer Hardware Verification Platform



Designs Verified

- bus core
- memory arbiter
- DMA Controller
- prototyping platform
- USB Core
- memory interface

bus.v

AG (req -> AF (ack + error)
"*request* always followed by an *ack* or *error*"

Environment Model

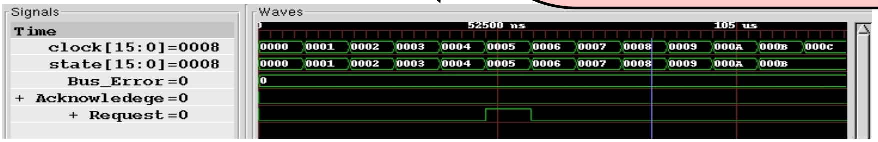
DiVer
Verification Platform For Digital Systems

System Features

- highly automated
- handles large designs
- more effective than simulation

Technology: Formal Verification

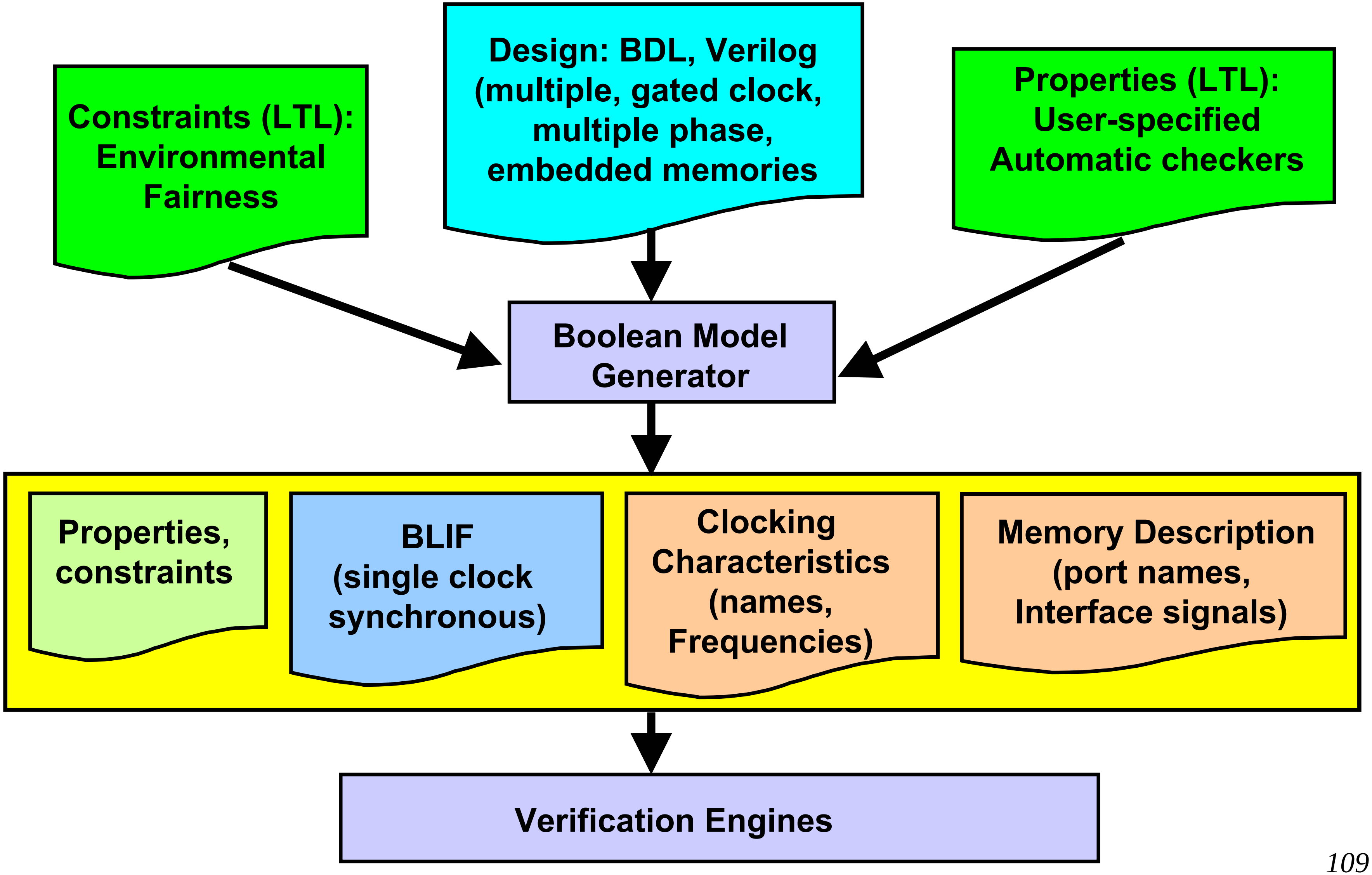
Efficient Implicit State Exploration
3 Main Engines:
-BDDs: property proofs
-BDD+SAT: superior to BDDs
-SAT: bug detection & proofs



```
Signals                       Waves
Time                          0            52500 ns                 105 us
    clock[15:0]=0008          0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C
    state[15:0]=0008          0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B
      Bus_Error=0             0
+ Acknowledege=0
      + Request=0
```

Bug found in bus design (1735 flip-flops) in 10 sec

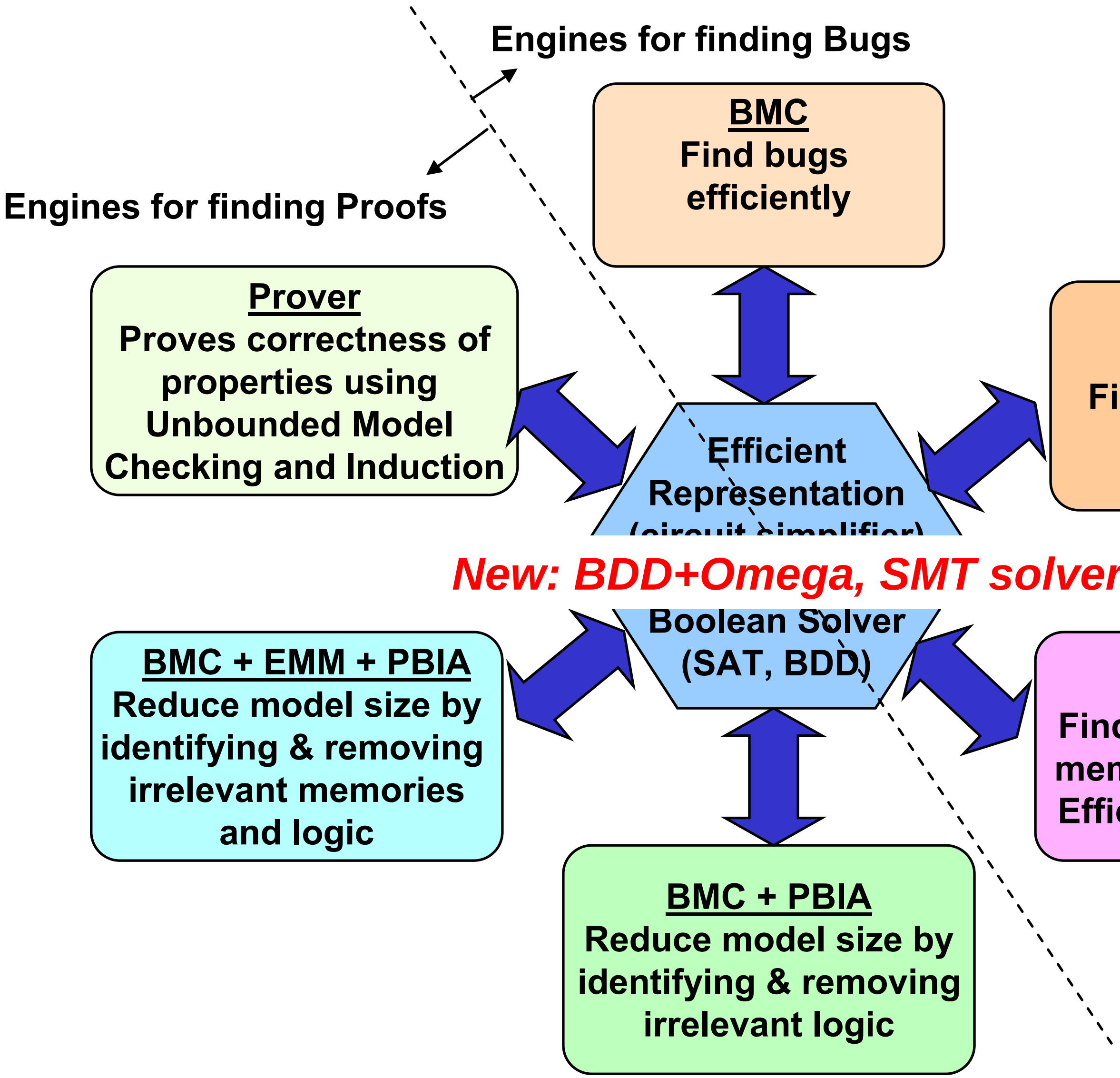
DiVer Front-end


Constraints (LTL): Environmental Fairness
Design: BDL, Verilog (multiple, gated clock, multiple phase, embedded memories
Properties (LTL): User-specified Automatic checkers
Boolean Model Generator
Properties, constraints
BLIF (single clock synchronous)
Clocking Characteristics (names, Frequencies)
Memory Description (port names, Interface signals)
Verification Engines

VeriSol (DiVer) Engines

Interesting large problems are within reach!

[Ganai *et al.* TACAS 05]



Engines for finding Bugs

Engines for finding Proofs

BMC
Find bugs efficiently

Prover
Proves correctness of properties using Unbounded Model Checking and Induction

Distributed BMC
Find bugs on network of workstations

Efficient Representation (circuit simplifier)

New: BDD+Omega, SMT solvers

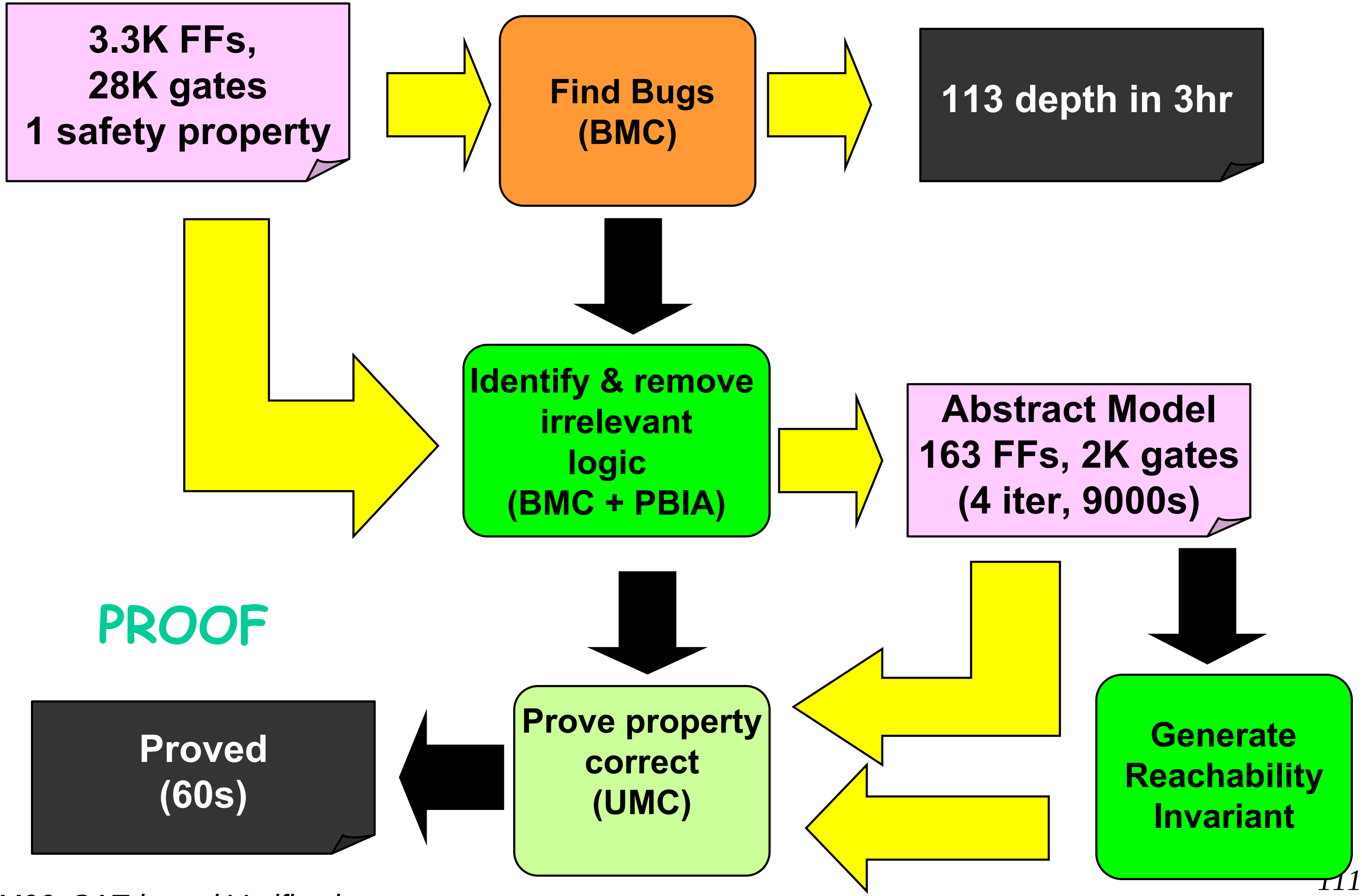
Boolean Solver (SAT, BDD)

BMC + EMM + PBIA
Reduce model size by identifying & removing irrelevant memories and logic

BMC + EMM
Find bugs in embedded memory systems using Efficient Memory Model

BMC + PBIA
Reduce model size by identifying & removing irrelevant logic

Case Study: Multiple Verification Engines



3.3K FFs,
28K gates
1 safety property

Find Bugs
(BMC)

113 depth in 3hr

Identify & remove
irrelevant
logic
(BMC + PBIA)

Abstract Model
163 FFs, 2K gates
(4 iter, 9000s)

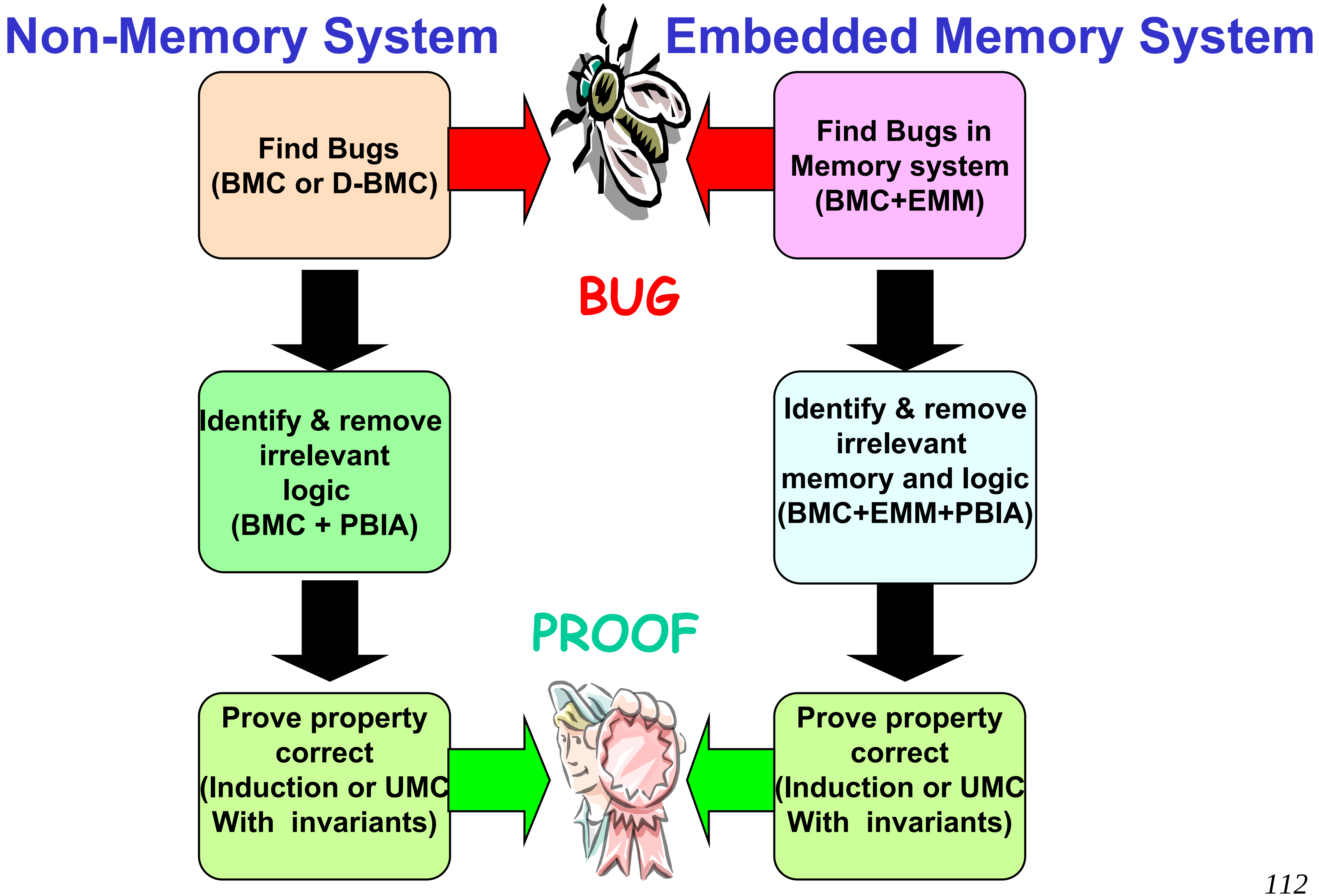
PROOF

Generate
Reachability
Invariant

Prove property
correct
(UMC)

Proved
(60s)

Standard Verification Flows



Non-Memory System

Embedded Memory System

Find Bugs
(BMC or D-BMC)

Find Bugs in
Memory system
(BMC+EMM)

BUG

Identify & remove
irrelevant
logic
(BMC + PBIA)

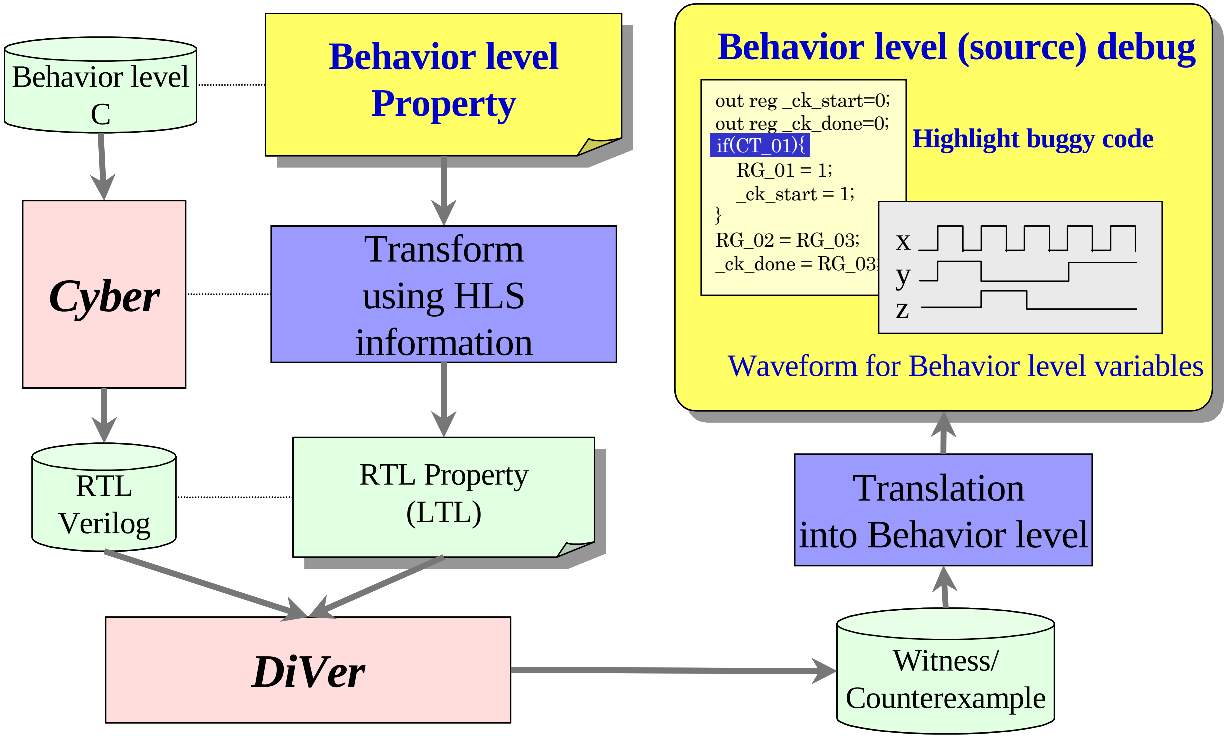
Identify & remove
irrelevant
memory and logic
(BMC+EMM+PBIA)

PROOF

Prove property
correct
(Induction or UMC
With invariants)

Prove property
correct
(Induction or UMC
With invariants)

NEC’s Behavioral Synthesis Design Flow


Behavior level C
Behavior level Property
Behavior level (source) debug
Highlight buggy code
x
y
z
Waveform for Behavior level variables
Cyber
Transform using HLS information
RTL Verilog
RTL Property (LTL)
Translation into Behavior level
DiVer
Witness/ Counterexample

```
out reg _ck_start=0;
out reg _ck_done=0;
if(CT_01){
   RG_01 = 1;
   _ck_start = 1;
}
RG_02 = RG_03;
_ck_done = RG_03
```

- ❑ **Cyber Work Bench (CWB)**
  - **Developed by NEC Japan (Wakabayashi et al.)**
  - **Automatically translates behavioral level design (C-based) to RTL design (Verilog)**
  - **Generates property monitors for RTL design automatically**
- ❑ **DiVer is integrated within CWB**
  - **Provides verification of RTL designs**
  - **Has been used successfully to find bugs by in-house design groups**

Applications in Software Verification

Model Checking Software Programs

C Program

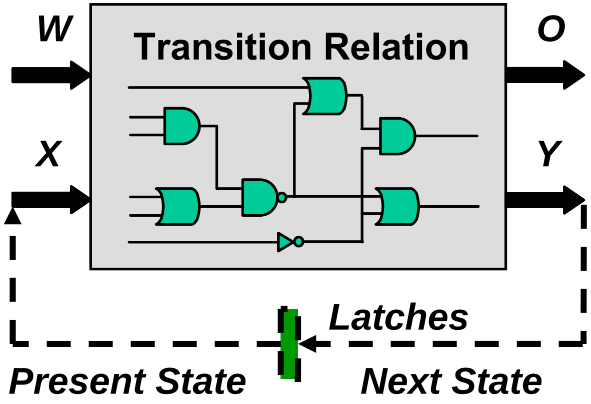
```
void bar() {
    int x = 3 , y = x-3 ;
    while ( x <= 4 ) {
        y++ ;
        x = foo(x);
    }
    y = foo(y);
}

int foo ( int l ) {
    int t = l+2 ;
    if ( t>6 )
        t - = 3;
    else
        t --;
    return t;
}
```

Huge gap !

Finite state circuit model

M = (S,s0,TR,L)



X: present state variables
Y: next state variables
W: input variables

Challenges:

- Rich data types
- Structures and arrays
- Pointers and pointer arithmetic
- Dynamic memory allocation
- Procedure boundaries and recursion
- Concurrent programs

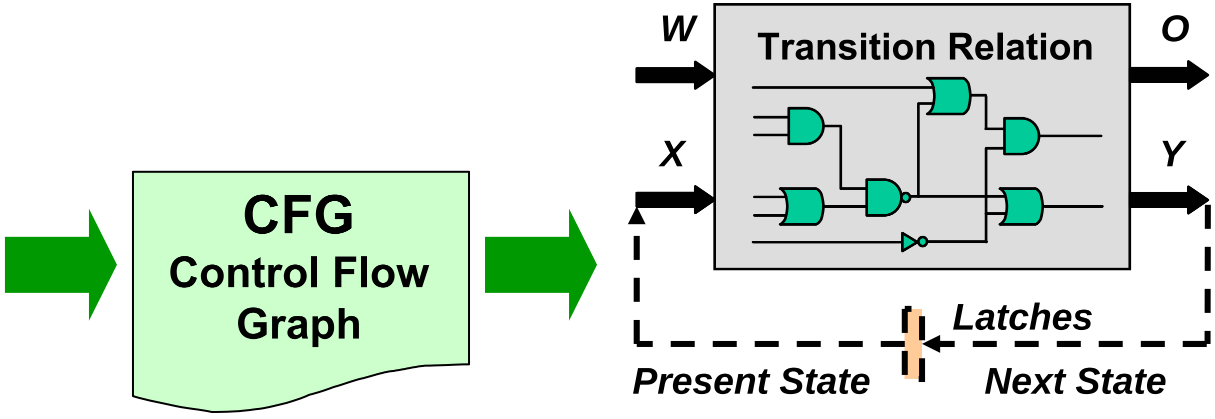
Intermediate Representation

C Program

```
void bar() {
    int x = 3 , y = x-3 ;
    while ( x <= 4 ) {
        y++ ;
        x = foo(x);
    }
    y = foo(y);
}

int foo ( int l ) {
    int t = l+2 ;
    if ( t>6 )
        t - = 3;
    else
        t --;
    return t;
}
```

M = (S,s0,TR,L)

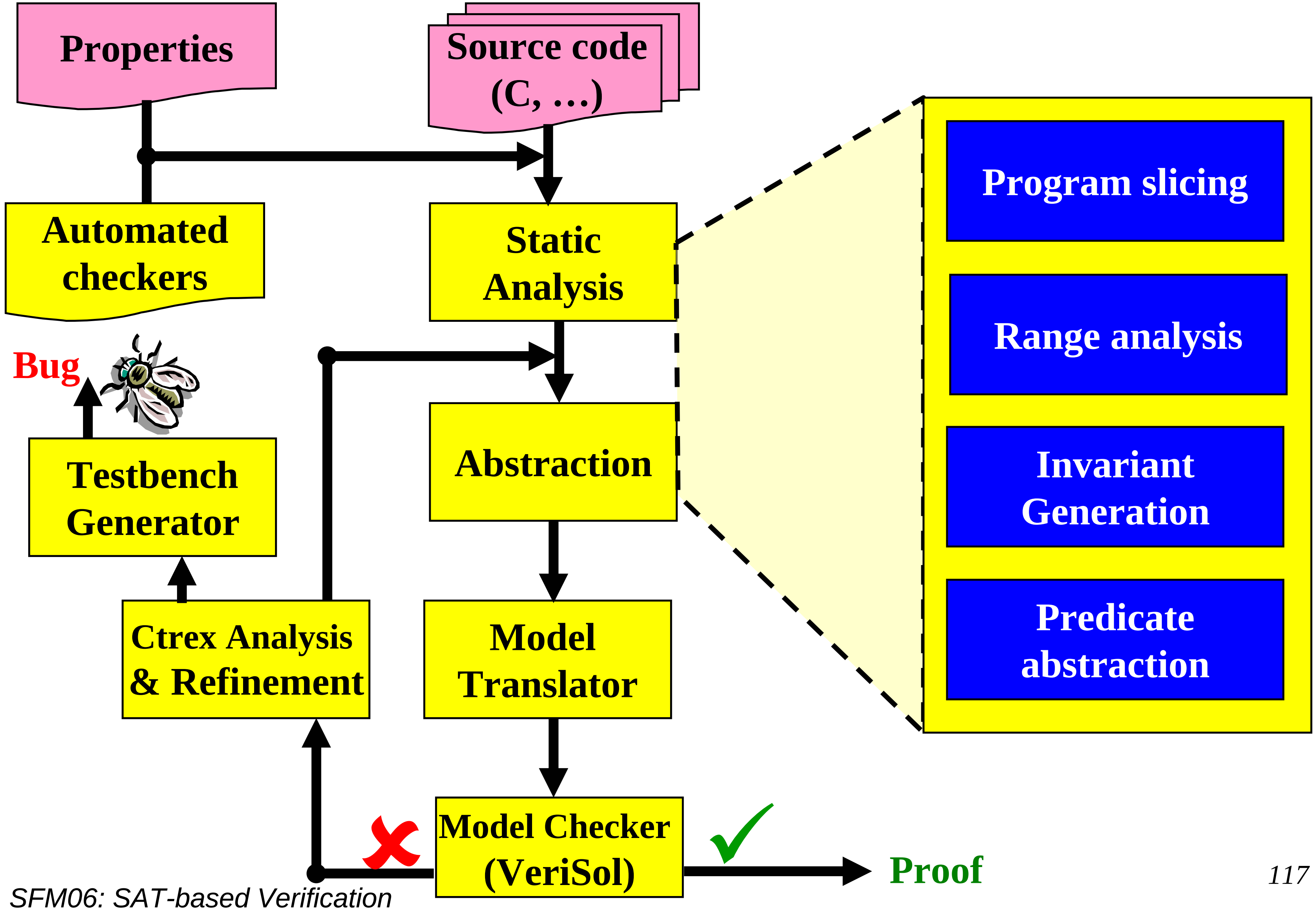


X: present state variables
Y: next state variables
W: input variables

- **Control Flow Graph**
  - Language-independent intermediate representation
  - Provides the basis for several optimizations (compilers, program analysis)
  - Allows separation of model building phase from model checking phase

F-Soft Software Verification Platform

[Ivancic *et al.* CAV 05, ICCD 05]



Thank you !