Combinational Equivalence Checking: SAT Application

Virendra Singh
Associate Professor
Computer Architecture and Dependable Systems Lab
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.ee.iitb.ac.in/~viren/
E-mail: viren@ee.iitb.ac.in

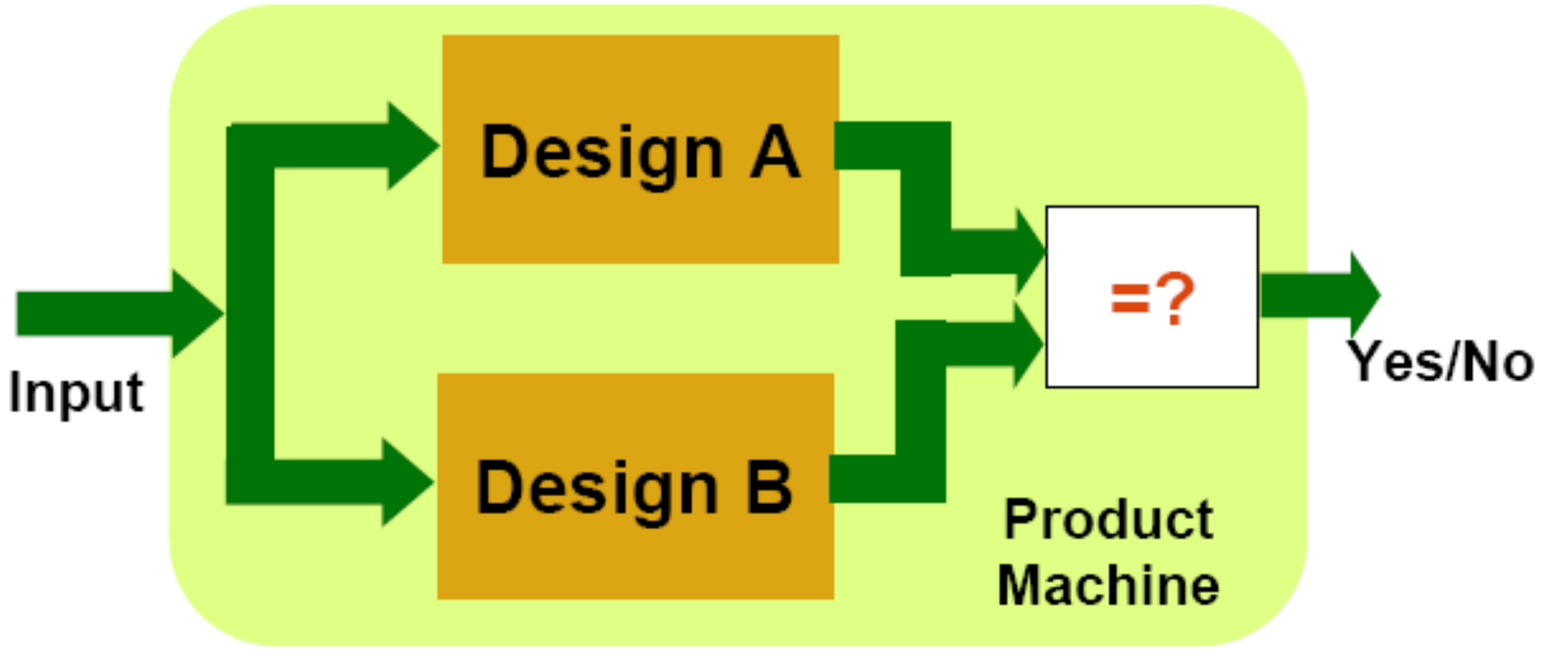
EE-709: Testing & Verification of VLSI Circuits

Lecture 3 (19 Jan 2015)

CADSL

Formal Equivalence Checking

Given two designs, prove that for all possible input stimuli their corresponding outputs are equivalent

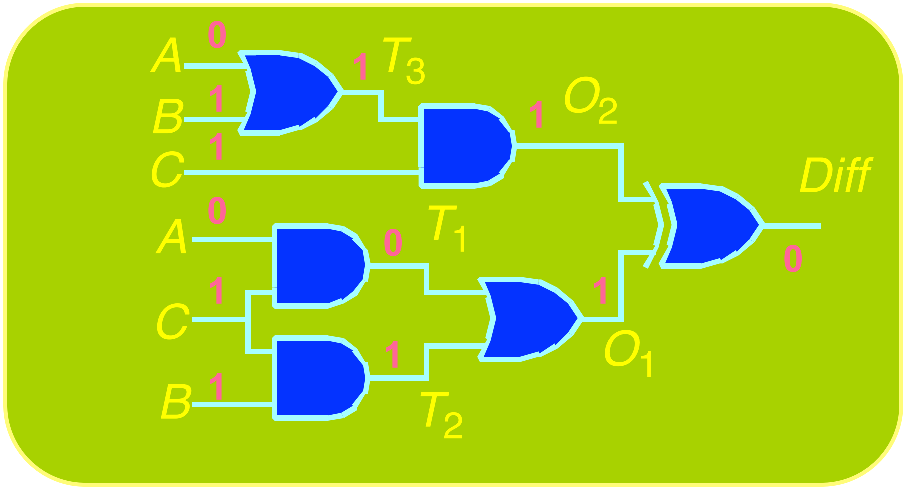


Variants of Decision Diagrams

- Multiterminal BDDs (MTBDD) – Pseudo Boolean functions B^n □ N, terminal nodes are integers
- Ordered Kronecker Functional Decision Diagrams (OKFDD) – uses XOR in OBDDs
- Binary Moment Diagrams (BMD) – good for arithmetic operations and word-level representation
- Zero-suppressed BDD (ZDD) – good for representing sparse sets
- Partitioned OBDDs (POBDD) – highly compact representation which retains most of the features of ROBDDs
- BDD packages –
  - CUDD from Univ. of Colorado, Boulder,
  - CMU BDD package from Carnegie Mellon Univ.
  - In addition, companies like Intel, Fujitsu, Motorola etc. have their own internal BDD packages

Formal Equivalence Checking

- Satisfiability Formulation
  - Search for input assignment giving different outputs
- Branch & Bound
  - Assign input(s)
  - Propagate forced values
  - Backtrack when cannot succeed



- Challenge
  - Must prove all assignments fail
    - Co-NP complete problem
  - Typically explore significant fraction of inputs
  - Exponential time complexity

SAT Problem definition

Given a CNF formula, f :

- A set of variables, *V* — *(a,b,c)*
- Conjunction of clauses — (C_1,C_2,C_3)
- Each clause: disjunction of literals over *V*

Does there exist an assignment of Boolean values to the variables, V which sets at least one literal in each clause to '1' ?

Example : $\underbrace{(a+b+\bar{c})}_{C_1}\underbrace{(\bar{a}+c)}_{C_2}\underbrace{(a+\bar{b}+c)}_{C_3}$ $a = b = c = 1$

DPLL algorithm for SAT

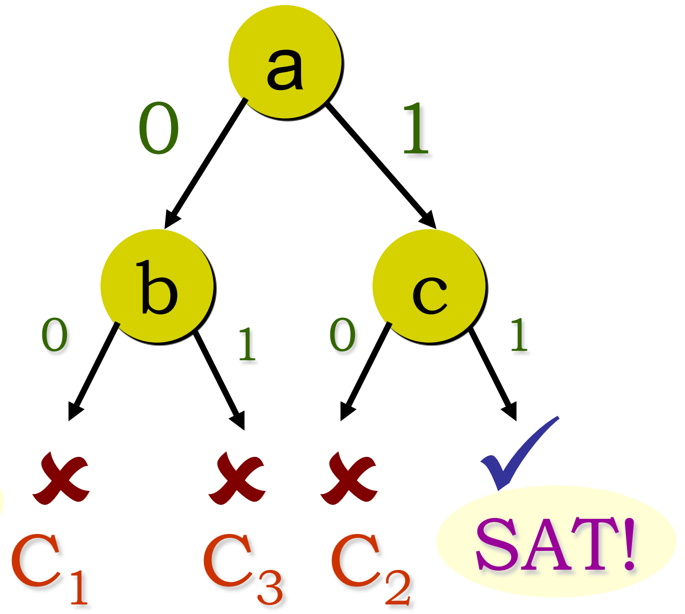
[Davis, Putnam, Logemann, Loveland 1960,62]

Given : CNF formula $f(v_1,v_2,..,v_k)$, and an ordering function Next_Variable

Example :

$$\underbrace{(a+b)}_{C_1}\underbrace{(\bar{a}+c)}_{C_2}\underbrace{(a+\bar{b})}_{C_3}$$

CONFLICT!



DPLL algorithm: Unit clause rule

Rule: Assign to *true* any single literal clauses.

$$\underset{\substack{\| \\ 0}}{(a} + \underset{\substack{\| \\ 0}}{b} + c) \Longrightarrow \boxed{c = 1}$$

Apply Iteratively: *Boolean Constraint Propagation (BCP)*

$$a(\bar{a}+c)(\bar{b}+c)(a+b+\bar{c})(\bar{c}+e)(\bar{d}+e)(c+d+\bar{e})$$

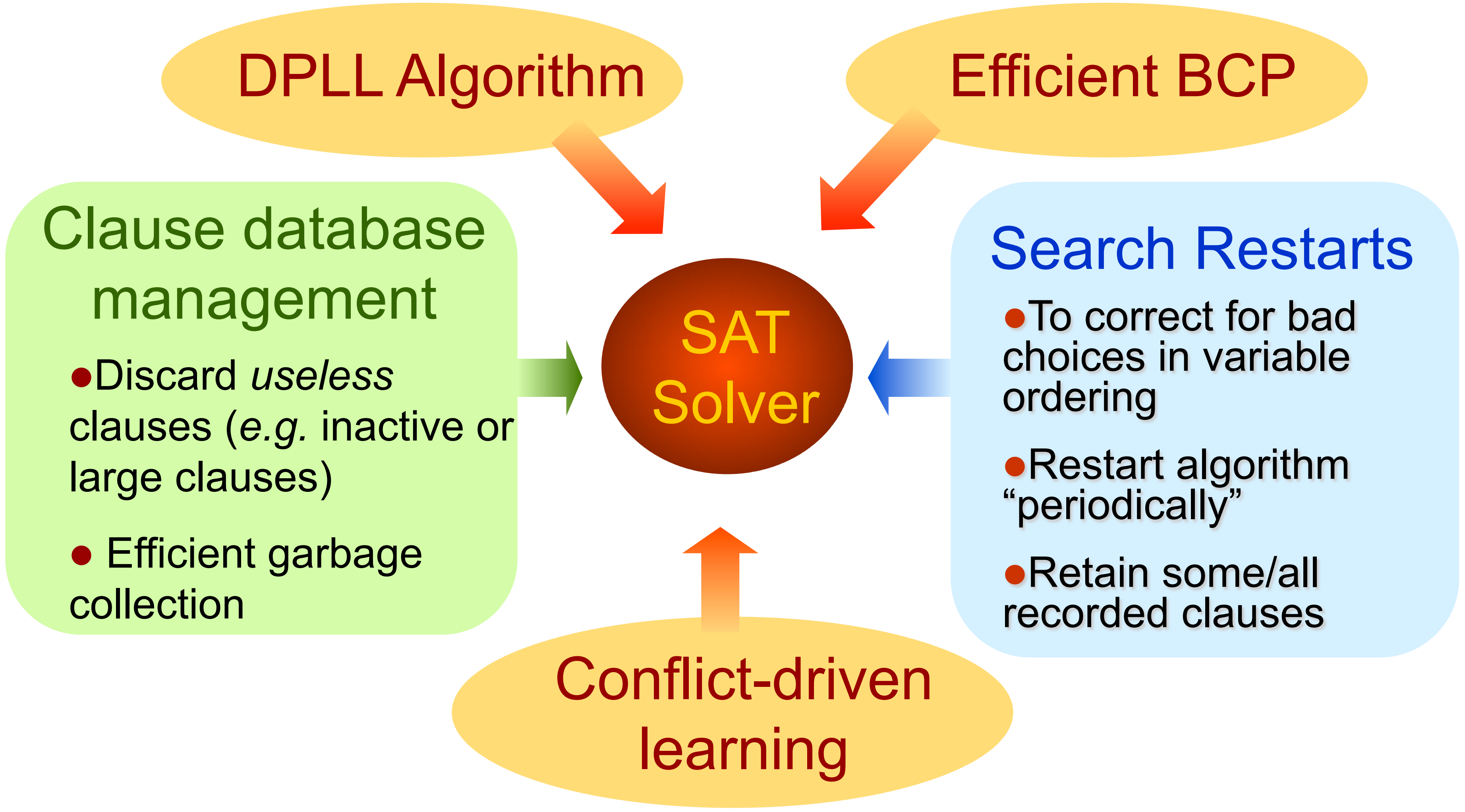
$$\Downarrow$$

$$c(\bar{b}+c)(\bar{c}+e)(\bar{d}+e)(c+d+\bar{e})$$

$$\Downarrow$$

$$e(\bar{d}+e)$$

Anatomy of a modern SAT solver



DPLL Algorithm

Efficient BCP

Clause database management

- Discard *useless* clauses (*e.g.* inactive or large clauses)
- Efficient garbage collection

SAT Solver

Search Restarts

- To correct for bad choices in variable ordering
- Restart algorithm "periodically"
- Retain some/all recorded clauses

Conflict-driven learning

Pure Literal Rule

- A variable is *pure* if its literals are either all positive or all negative
- Satisfiability of a formula is unaffected by assigning pure variables the values that satisfy all the clauses containing them

$$\phi = (a + c)(b + c)(b + \neg d)(\neg a + \neg b + d)$$

- Set c to 1; if ϕ becomes unsatisfiable, then it is also unsatisfiable when c is set to 0.

Resolution (original DP)

- Iteratively apply resolution (consensus) to eliminate one variable each time
  - i.e., consensus between all pairs of clauses containing *x* and ¬*x*
  - formula satisfiability is preserved
- Stop applying resolution when,
  - Either empty clause is derived → instance is unsatisfiable
  - Or only clauses satisfied or with pure literals are obtained → instance is satisfiable

$\phi = (a + c)(b + c)(d + c)(\neg a + \neg b + \neg c)$ Eliminate variable *c*

$\phi_1 = (a + \neg a + \neg b)(b + \neg a + \neg b)(d + \neg a + \neg b)$
$= (d + \neg a + \neg b)$

Instance is SAT !

Stallmarck's Method (SM) in CNF

- Recursive application of the branch-merge rule to each variable with the goal of identifying common conclusions

$$j = (a + b)(\neg a + c)(\neg b + d)(\neg c + d)$$

Try $a = 0$: $(a = 0) \rightarrow (b = 1) \rightarrow (d = 1)$ $C(a = 0) = \{a = 0, b = 1, d = 1\}$

Try $a = 1$: $(a = 1) \rightarrow (c = 1) \rightarrow (d = 1)$ $C(a = 1) = \{a = 1, c = 1, d = 1\}$

$C(a = 0) \cap C(a = 1) = \{d = 1\}$

Any assignment to variable a implies $d = 1$.
Hence, $d = 1$ is a necessary assignment !

Recursion can be of arbitrary depth

Recursive Learning (RL) in CNF

- Recursive evaluation of clause satisfiability requirements for identifying common assignments

$$= (a + b)(\neg a + d)(\neg b + d)$$

Try $a = 1$:	$(a = 1) \rightarrow (d = 1)$	$C(a = 1) = \{a = 1, d = 1\}$
Try $b = 1$:	$(b = 1) \rightarrow (d = 1)$	$C(b = 1) = \{b = 1, d = 1\}$

$C(a = 1) \cap C(b = 1) = \{d = 1\}$

Every way of satisfying $(a + b)$ implies $d = 1$.
Hence, $d = 1$ is a necessary assignment !

Recursion can be of arbitrary depth

SM vs. RL

- Both complete procedures for SAT
- Stallmarck's method:
  - hypothetic reasoning based on variables
- Recursive learning:
  - hypothetic reasoning based on clauses
- Both can be integrated into backtrack search algorithms

Local Search

- Repeat *M* times:
  - Randomly pick complete assignment
  - Repeat *K* times (and while exist unsatisfied clauses):
    - Flip variable that will satisfy largest number of unsat clauses

$j = (a + b)(\neg a + c)(\neg b + d)(\neg c + d)$ Pick random assignment

$j = (a + b)(\neg a + c)(\neg b + d)(\neg c + d)$ Flip assignment on *d*

$j = (a + b)(\neg a + c)(\neg b + d)(\neg c + d)$ Instance is satisfied !

Comparison

- Local search is incomplete
  - If instances are known to be SAT, local search can be competitive
- Resolution is in general impractical
- Stallmarck's Method (SM) and Recursive Learning (RL) are in general slow, though robust
  - SM and RL can derive too much unnecessary information
- For most EDA applications backtrack search (DP) is currently the most promising approach **!**
  - **Augmented with techniques for inferring new clauses/ implicates (i.e. learning) !**

Techniques for Backtrack Search

- Conflict analysis
  - Clause/implicate recording
  - Non-chronological backtracking
- Incorporate and extend ideas from:
  - Resolution
  - Recursive learning
  - Stallmarck's method
- Formula simplification & Clause inference [Li,AAAI00]
- Randomization & Restarts [Gomes&Selman,AAAI98]

Clause Recording

- During backtrack search, for each conflict create clause that explains and prevents recurrence of same conflict

φ = (*a* + *b*)(¬*b* + *c* + *d*) (¬*b* + *e*)(¬*d* + ¬*e* + *f*)…

Assume (decisions) *c* = 0 and *f* = 0

Assign *a* = 0 and imply assignments

A conflict is reached: (¬*d* + ¬*e* + *f*) is unsat

$$(a=0)\wedge(c=0)\wedge(f=0)\Rightarrow(\varphi=0)$$

$$(\varphi=1)\Rightarrow(a=1)\vee(c=1)\vee(f=1)$$

φ create new clause: (*a* + *c* + *f*)

Thank You