SAT and Bounded Model Checking

Automated Program Verification (APV)
Fall 2018

Prof. Arie Gurfinkel

UNIVERSITY OF WATERLOO

Syntax of Propositional Logic

An *atomic formula* has a form A_i , where i = 1, 2, 3 …

Formulas are defined inductively as follows:

- All atomic formulas are formulas
- For every formula F, ¬F (called not F) is a formula
- For all formulas F and G, F ∧ G (called and) and F ∨ G (called or) are formulas

Abbreviations

- use A, B, C, … instead of A_1, A_2, …
- use $F_1 \rightarrow F_2$ instead of $\neg F_1 \vee F_2$ (implication)
- use $F_1 \leftrightarrow F_2$ instead of $(F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1)$ (iff)

Syntax of Propositional Logic (PL)

$$
\begin{aligned}
\text{truth\_symbol} &::= \top(\text{true}) \mid \bot(\text{false}) \\
\text{variable} &::= p, q, r, \ldots \\
\text{atom} &::= \text{truth\_symbol} \mid \text{variable} \\
\text{literal} &::= \text{atom} \mid \neg\text{atom} \\
\text{formula} &::= \text{literal} \mid \\
&\quad \neg\text{formula} \mid \\
&\quad \text{formula} \wedge \text{formula} \mid \\
&\quad \text{formula} \vee \text{formula} \mid \\
&\quad \text{formula} \rightarrow \text{formula} \mid \\
&\quad \text{formula} \leftrightarrow \text{formula}
\end{aligned}
$$

Normal Forms: CNF and DNF

A *literal* is either an atomic proposition v or its negation ~v

A *clause* is a disjunction of literals

- e.g., (v1 || ~v2 || v3)

A formula is in *Conjunctive Normal Form* (CNF) if it is a conjunction of disjunctions of literals (i.e., a conjunction of clauses):

- e.g., (v1 || ~v2) && (v3 || v2)

$$\bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} L_{i,j} \right)$$

A formula is in *Disjunctive Normal Form* (DNF) if it is a disjuction of conjunctions of literals

$$\bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{m_i} L_{i,j} \right)$$

Boolean Satisfiability (CNF-SAT)

Let V be a set of variables

A *literal* is either a variable v in V or its negation ~v

A *clause* is a disjunction of literals

- e.g., (v1 || ~v2 || v3)

A Boolean formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses

- e.g., (v1 || ~v2) && (v3 || v2)

An *assignment* *s* of Boolean values to variables *satisfies* a clause *c* if it evaluates at least one literal in *c* to true

An assignment *s* *satisfies* a formula *C* in CNF if it satisfies every clause in *C*

Boolean Satisfiability Problem (CNF-SAT):

- determine whether a given CNF C is satisfiable

CNF Examples

CNF 1

- ~b
- ~a || ~b || ~c
- a
- sat: s(a) = True; s(b) = False; s(c) = False

CNF 2

- ~b
- ~a || b || ~c
- a
- ~a || c
- unsat

Algorithms for SAT

SAT is NP-complete

- solution can be checked in polynomial time
- no polynomial algorithms for finding a solution are known

DPLL (Davis-Putnam-Logemman-Loveland, ‘60)

- smart enumeration of all possible SAT assignments
- worst-case EXPTIME
- alternate between deciding and propagating variable assignments

CDCL (GRASP ‘96, Chaff ‘01)

- conflict-driven clause learning
- extends DPLL with
  - smart data structures, backjumping, clause learning, heuristics, restarts…
- scales to millions of variables
- N. Een and N. Sörensson, “An Extensible SAT-solver”, in SAT 2013.

Background Reading: SAT

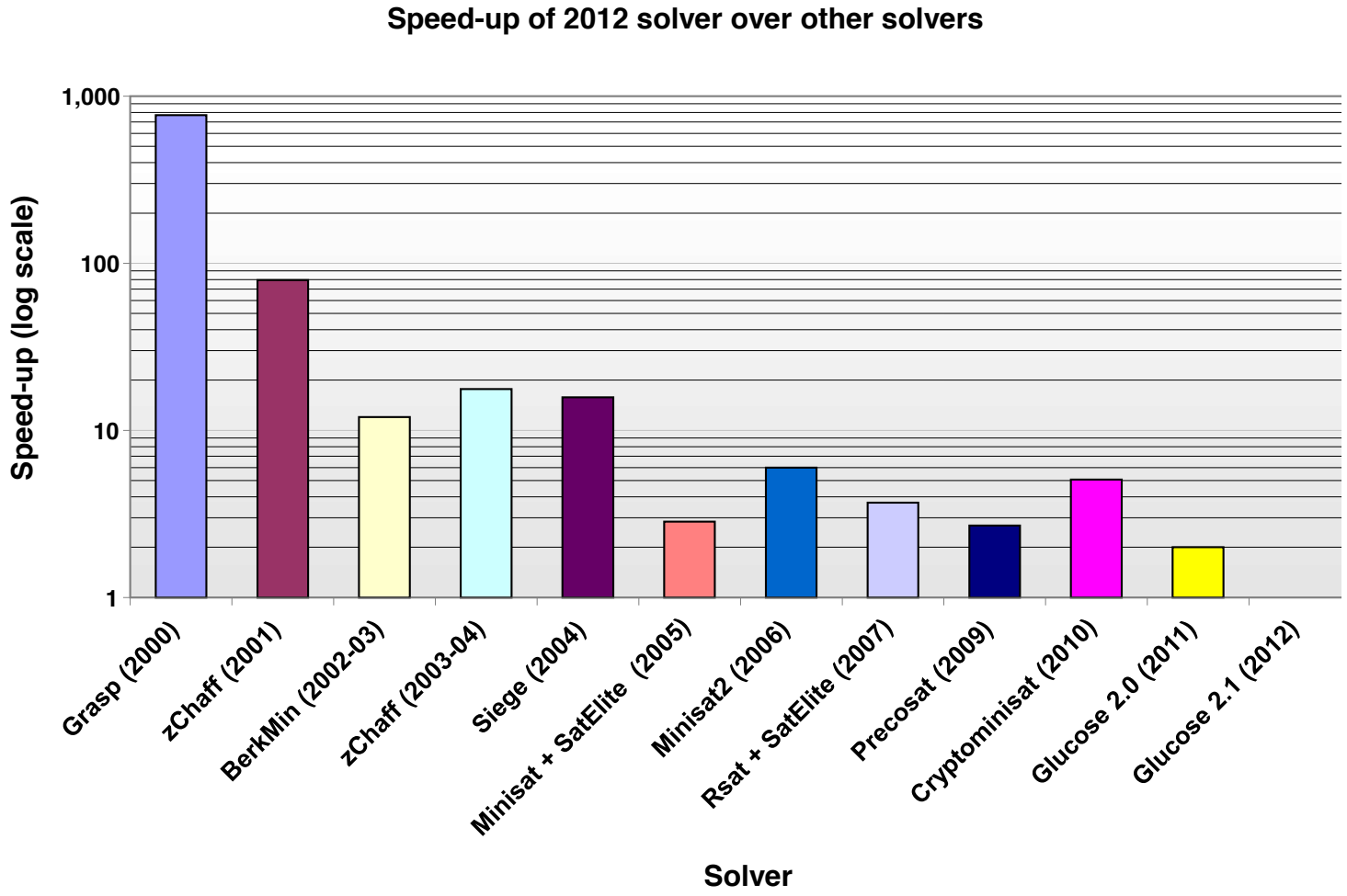
REVIEW ARTICLES

Boolean Satisfiability: From Theoretical Hardness to Practical Success

By Sharad Malik, Lintao Zhang
Communications of the ACM, Vol. 52 No. 8, Pages 76-82
10.1145/1536616.1536637

There are many practical situations where we need to satisfy several potentially conflicting constraints. Simple examples of this abound in daily life, for example, determining a schedule for a series of games that resolves the availability of players and venues, or finding a seating assignment at dinner consistent with various rules the host would like to impose. This also applies to applications in computing, for example, ensuring that a hardware/software system functions correctly with its overall behavior constrained by the behavior of its components and their

Some Experience with SAT Solving



from M. Vardi, https://www.cs.rice.edu/~vardi/papers/highlights15.pdf

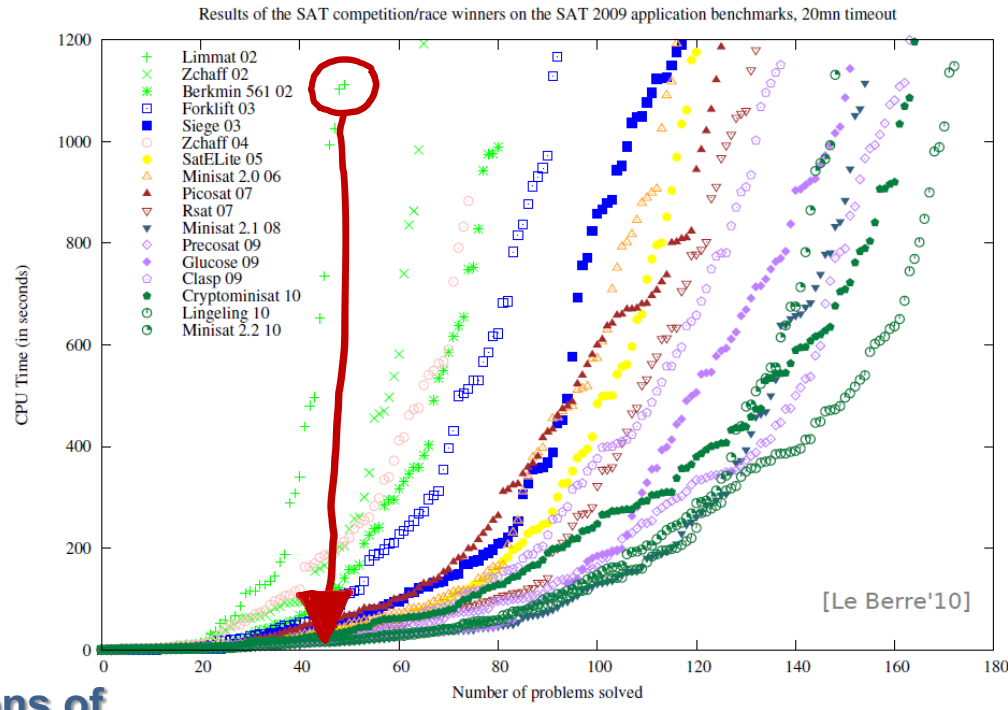
SAT - Milestones

Problems impossible 10 years ago are trivial today

year	Milestone
1960	Davis-Putnam procedure
1962	Davis-Logeman-Loveland
1984	Binary Decision Diagrams
1992	DIMACS SAT challenge
1994	SATO: clause indexing
1997	GRASP: conflict clause learning
1998	Search Restarts
2001	zChaff: 2-watch literal, VSIDS
2005	Preprocessing techniques
2007	Phase caching
2008	Cache optimized indexing
2009	In-processing, clause management
2010	Blocked clause elimination

Concept

2002 ⟷ **2010**



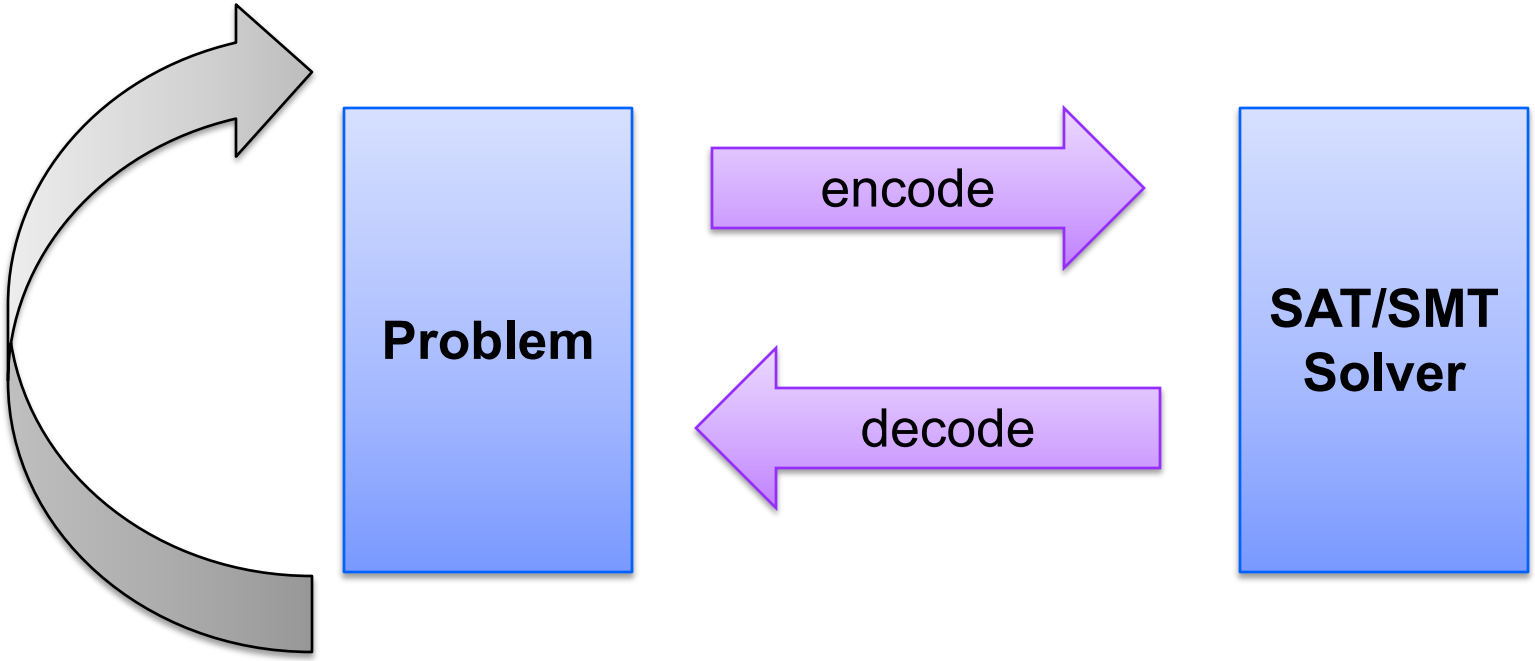
Millions of variables from HW designs

Courtesy Daniel le Berre

NP is the new P!

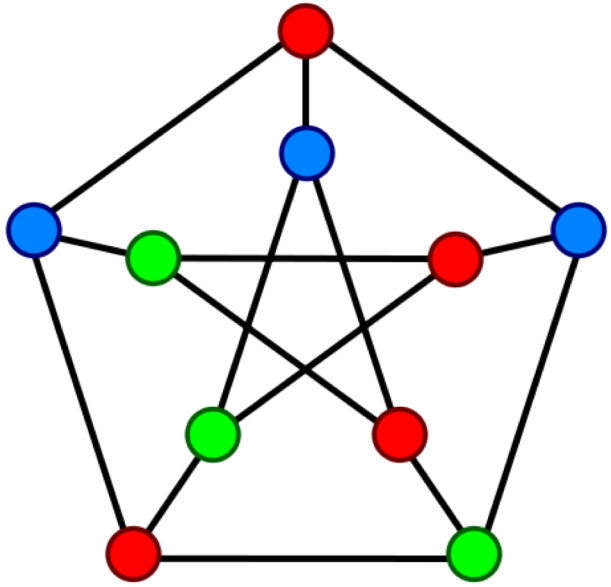
Solve any computational problem by effective reduction to SAT/SMT

- iterate as necessary



Graph k-Coloring

Given a graph *G = (V, E)*, and a natural number *k > 0* is it possible to assign colors to vertices of *G* such that no two adjacent vertices have the same color.



Formally:

- does there exists a function f : V → [0..k) such that
- for every edge (u, v) in E, f(u) != f(v)

Graph coloring for k > 2 is NP-complete

Problem: Encode k-coloring of G into CNF

- construct CNF *C* such that C is SAT iff G is k-colorable

https://en.wikipedia.org/wiki/Graph_coloring

k-coloring as CNF

Let a Boolean variable $f_{v,i}$ denote that vertex *v* has color *i*

- if $f_{v,i}$ is true if and only if f(v) = i

Every vertex has at least one color

$$\bigvee_{0 \leq i < k} f_{v,i} \qquad (v \in V)$$

No vertex is assigned two colors

$$\bigwedge_{0 \leq i < j < k} (\neg f_{v,i} \vee \neg f_{v,j}) \qquad (v \in V)$$

No two adjacent vertices have the same color

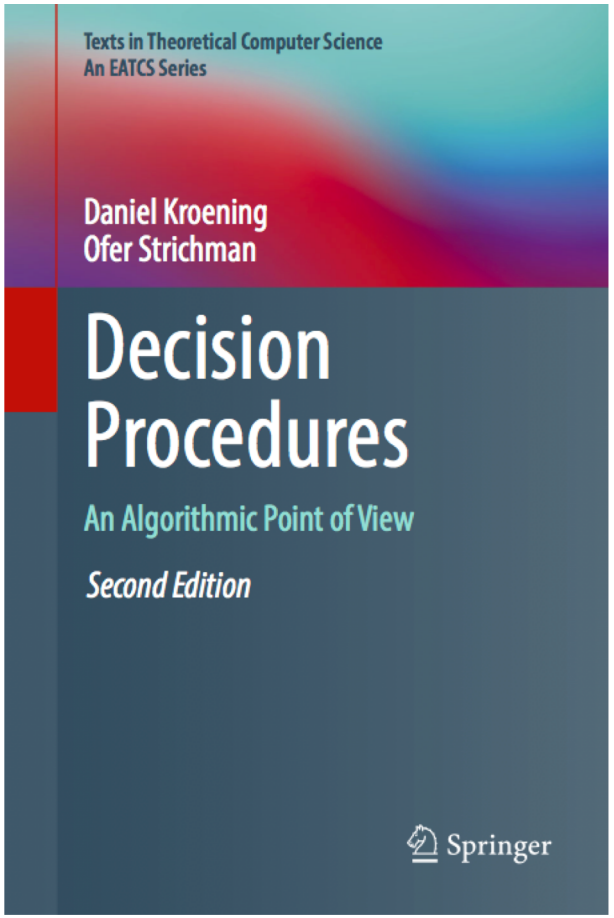
$$\bigwedge_{0 \leq i < k} (\neg f_{v,i} \vee \neg f_{u,i}) \qquad ((v,u) \in E)$$

Davis Putnam Logemann Loveland

DPLL PROCEDURE

References

Chapter 2: Decision Procedures for Propositional Logic



https://link.springer.com/book/10.1007%2F978-3-540-74105-3

Decision Procedure for Satisfiability

Algorithm that in some finite amount of computation decides if a given propositional logic (PL) formula F is satisfiable

- NP-complete problem

Modern decision procedures for PL formulae are called SAT solvers

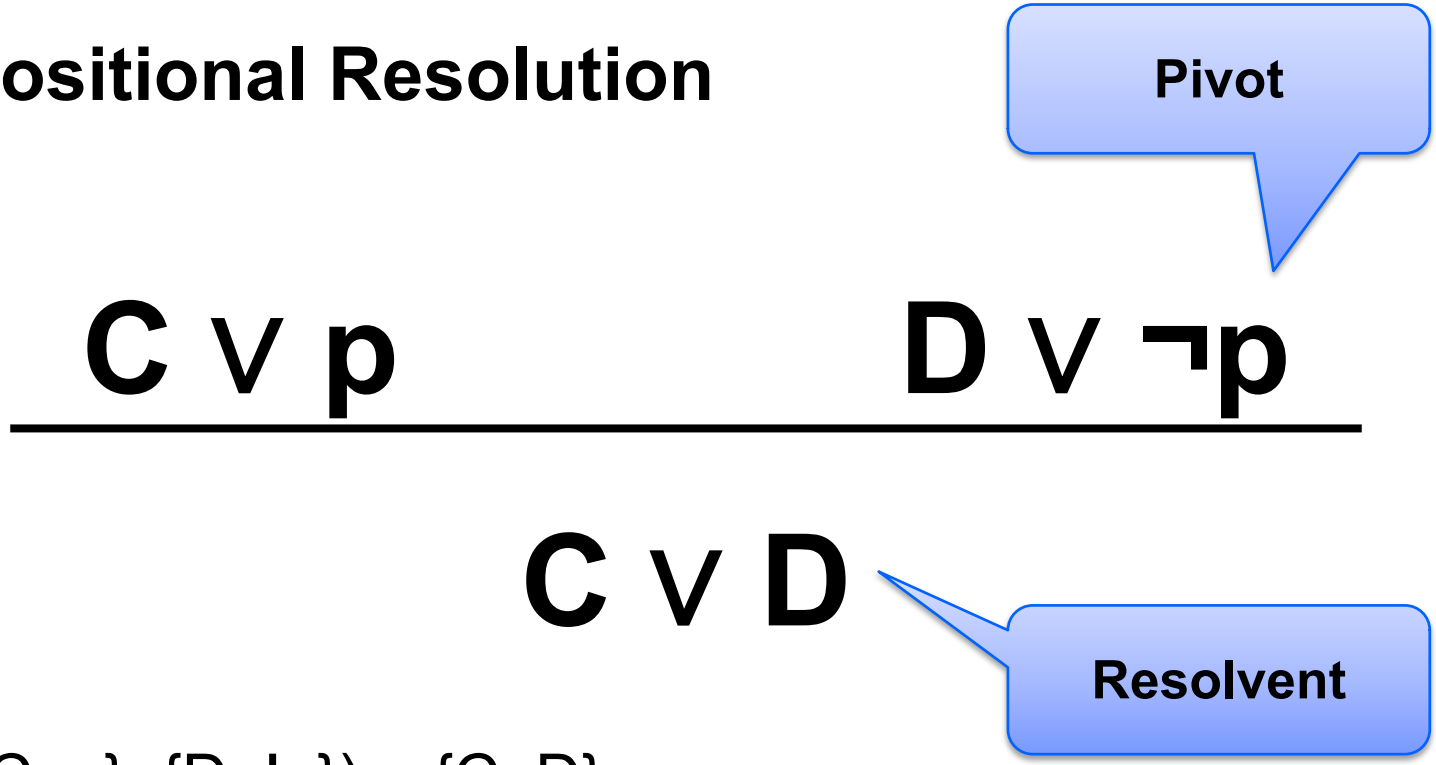
Naïve approach

- Enumerate models (i.e., truth tables)
- Enumerate resolution proofs

Modern SAT solvers

- DPLL algorithm
  - Davis-Putnam-Logemann-Loveland
- Combines model- and proof-based search
- Operates on Conjunctive Normal Form (CNF)

Propositional Resolution



Res({C, p}, {D, !p}) = {C, D}

Given two clauses {C, p} and {D, !p} that contain a literal p of different polarity, create a new clause by taking the union of literals in C and D

Resolution Lemma

Lemma:

Let F be a CNF formula. Let R be a resolvent of two clauses X and Y in F. Then, $F \cup \{R\}$ is equivalent to F

Resolution Theorem

Let F be a set of clauses

$$Res(F) = F \cup \{R \mid R \text{ is a resolvent of two clauses in } F\}$$

$$Res^0(F) = F$$

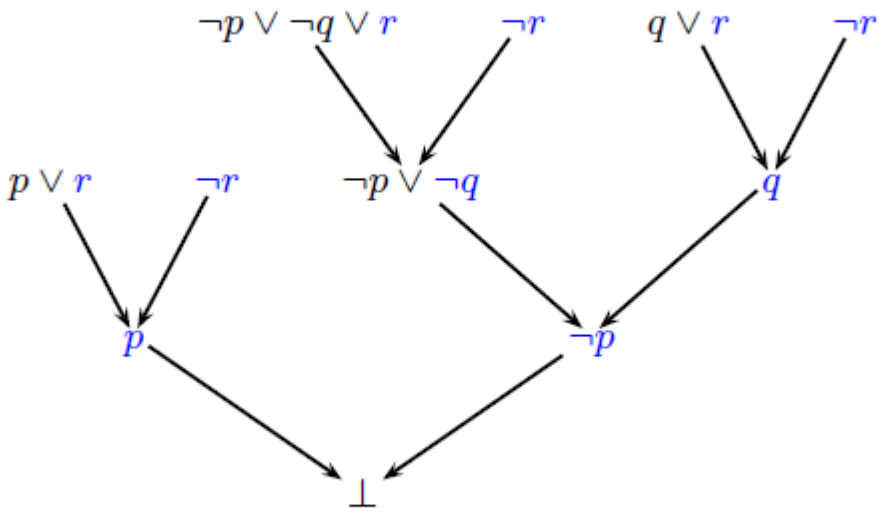
$$Res^{n+1}(F) = Res(Res^n(F)), \text{ for } n \geq 0$$

$$Res^*(F) = \bigcup_{n \geq 0} Res^n(F)$$

Theorem: A CNF F is UNAT iff Res*(F) contains an empty clause

Example of a resolution proof

A refutation of $\neg p \vee \neg q \vee r,\ p \vee r,\ q \vee r,\ \neg r$:



Resolution Proof Example

Show by resolution that the following CNF is UNSAT

$$\neg b \wedge (\neg a \vee b \vee \neg c) \wedge a \wedge (\neg a \vee c)$$

$$\dfrac{\dfrac{\dfrac{\neg a \vee b \vee \neg c \qquad a}{b \vee \neg c} \qquad b}{\neg c} \qquad \dfrac{a \qquad \neg a \vee c}{c}}{\bot}$$

Proof of the Resolution Theorem

(Soundness) By Resolution Lemma, F is equivalent to $Res^i(F)$ for any i. Let n be such that $Res^{n+1}(F)$ contains an empty clause, but $Res^n(F)$ does not. Then $Res^n(F)$ must contain to unit clauses L and ¬L. Hence, it is UNSAT.

(Completeness) By induction on the number of different atomic propositions in F.

Base case is trivial: F contains an empty clause.

IH: Assume F has atomic propositions A1, … A_{n+1}

Let F_0 be the result of replacing A_{n+1} by 0

Let F_1 be the result of replacing A_{n+1} by 1

Apply IH to F_0 and F_1 . Restore replaced literals. Combine the two resolutions.

Proof System

$$P_1, \ldots, P_n \vdash C$$

An inference rule is a tuple (P_1, …, P_n, C)

- where, P_1, …, P_n, C are formulas
- P_i are called premises and C is called a conclusion
- intuitively, the rules says that the conclusion is true if the premises are

A proof system P is a collection of inference rules

A proof in a proof system P is a tree (or a DAG) such that

- nodes are labeled by formulas
- for each node *n*, (parents(n), n) is an inference rule in P

Propositional Resolution

$$\frac{C \vee p \qquad D \vee \neg p}{C \vee D}$$

Propositional resolution is a sound inference rule

Proposition resolution system consists of a single propositional resolution rule

DP Procedure: SAT solving by resolution

Assume that input formula F is in CNF

1. Pick two clauses C_1 and C_2 in F that can be resolved
2. If the resolvent C is an empty clause, return UNSAT
3. Otherwise, add C to F and go to step 1
4. If no new clauses can be resolved, return SAT

Termination: finitely many derived clauses

DPLL: David Putnam Logemann Loveland

Combines pure resolution-based search with case splitting on decisions

Proof search is restricted to unit resolution

- can be done very efficiently (polynomial time)

Case split restores completeness

DPLL can be described by the following two rules

- F is the input formula in CNF

$$\frac{F}{F,p \quad | \quad F,\neg p}\ \textbf{split} \qquad p \text{ and } \neg p \text{ are not in } F$$

$$\frac{F,\ C \vee \ell, \neg \ell}{F, C, \neg \ell}\ \text{unit}$$

Davis, Martin; Logemann, George; Loveland, Donald (1962). "A Machine Program for Theorem Proving". *C. ACM*. 5 (7): 394–397. doi:10.1145/368273.368557

The original DPLL procedure

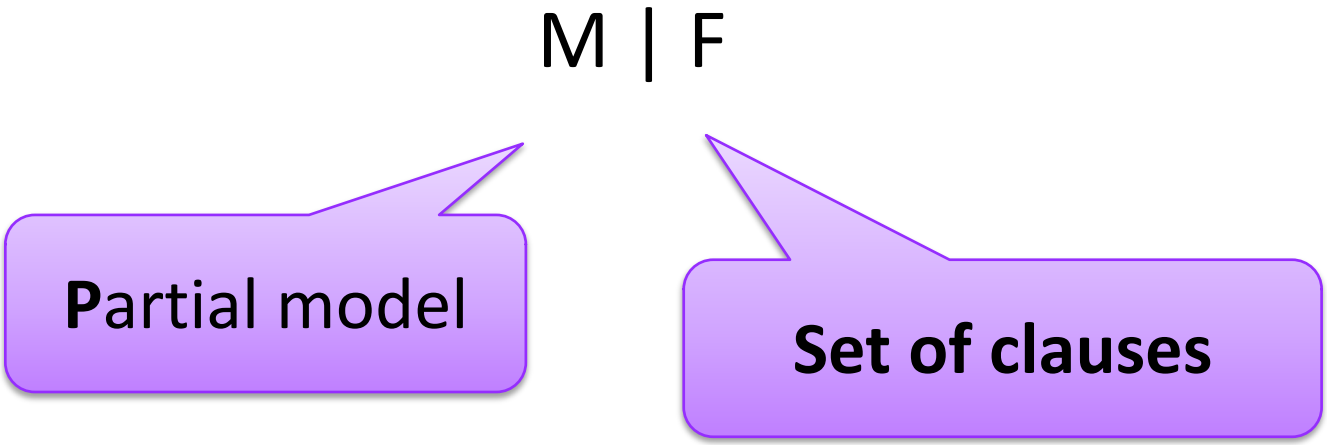
Incrementally builds a satisfying truth assignment M for the input CNF formula F

M is grown by

- deducing the truth value of a literal from M and F, or
- guessing a truth value

If a wrong guess for a literal leads to an inconsistency, the procedure backtracks and tries the opposite value

DPLL: Illustration



DPLL: Decide

Guessing (Decide)

$$p \mid p \vee q, \neg q \vee r$$

$$p, \neg q \mid p \vee q, \neg q \vee r$$

DPLL: Boolean Constraint Propagation

Deducing (Unit Propagation or BCP)

$$p \mid p \vee q, \neg p \vee s$$

$$p, s \mid p \vee q, \neg p \vee s$$

DPLL: Backtracking

Backtracking

$p, \neg s, q \mid p \vee q, s \vee q, \neg p \vee \neg q$

$p, s \mid p \vee q, s \vee q, \neg p \vee \neg q$

Pure Literals

A literal is pure if only occurs positively or negatively.

Example :

$\varphi = (\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_2) \wedge (x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$

$\neg x_1$ and x_3 are pure literals

Pure literal rule :

Clauses containing pure literals can be removed from the formula (i.e. just satisfy those pure literals)

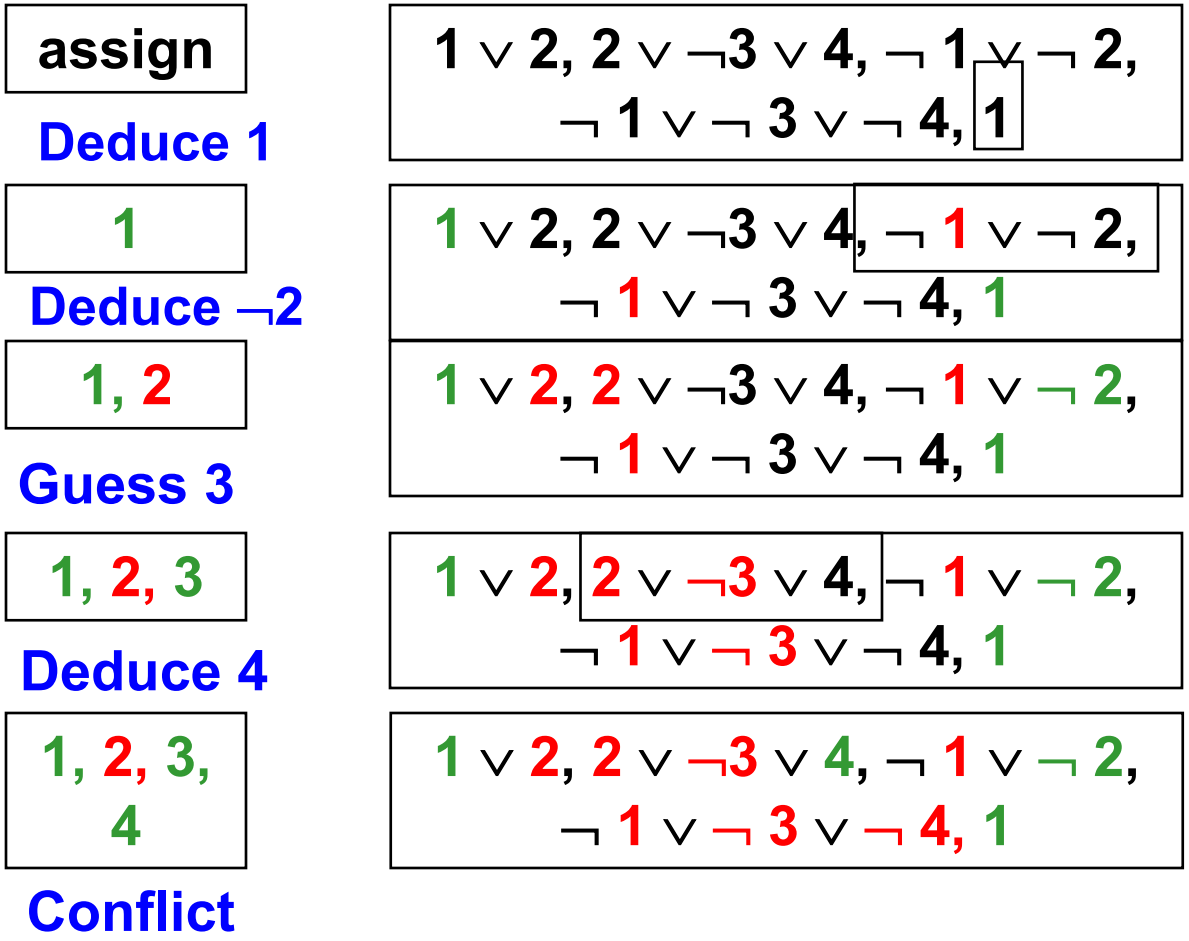
$$\varphi_{\neg x_1, x_3} = (x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$$

Preserve satisfiability, not logical equivalency !

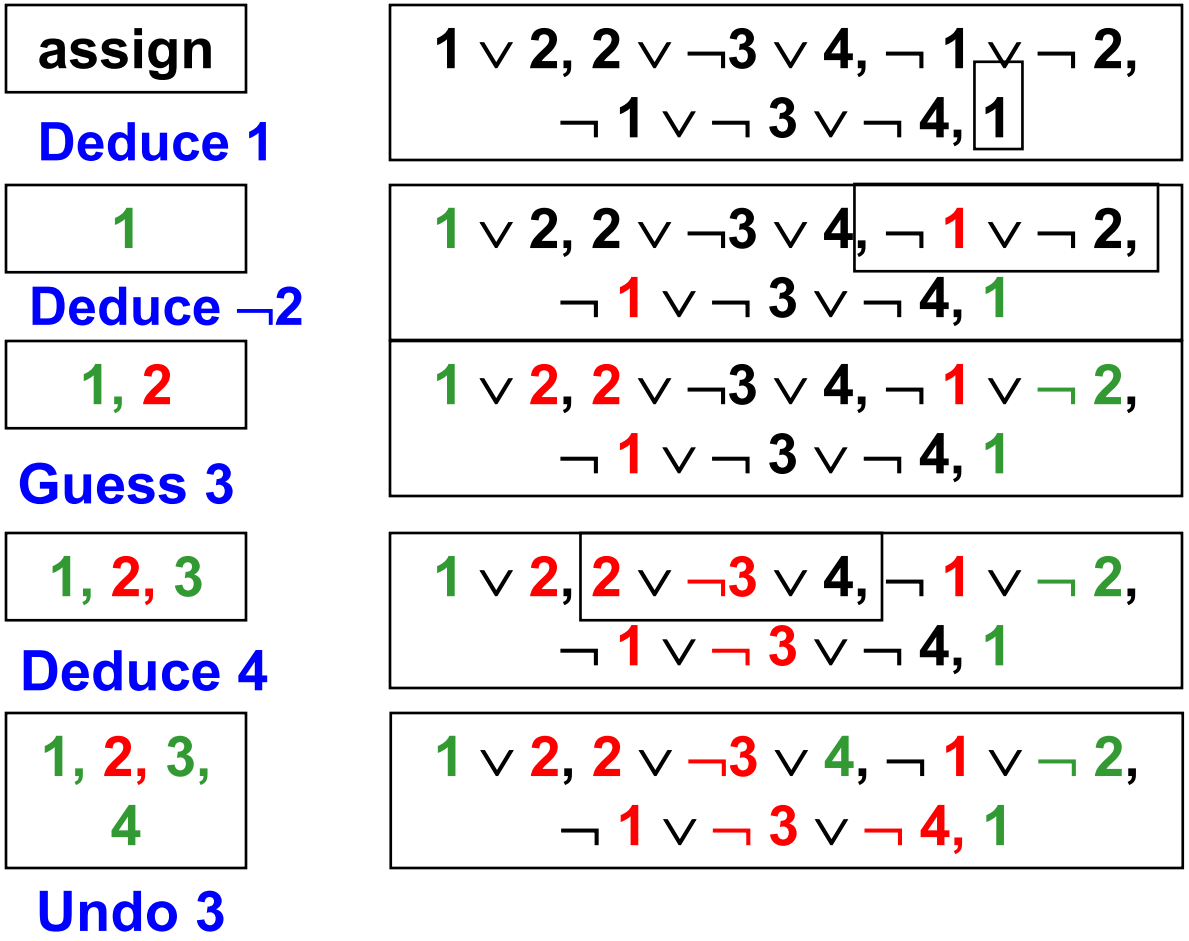
DPLL Procedure

- Standard backtrack search
- DPLL(F) :
  - Apply unit propagation
  - If conflict identified, return UNSAT
  - Apply the pure literal rule
  - If F is satisfied (empty), return SAT
  - Select decision variable x
    - If DPLL($F \wedge x$)=SAT return SAT
    - return DPLL($F \wedge \neg x$)

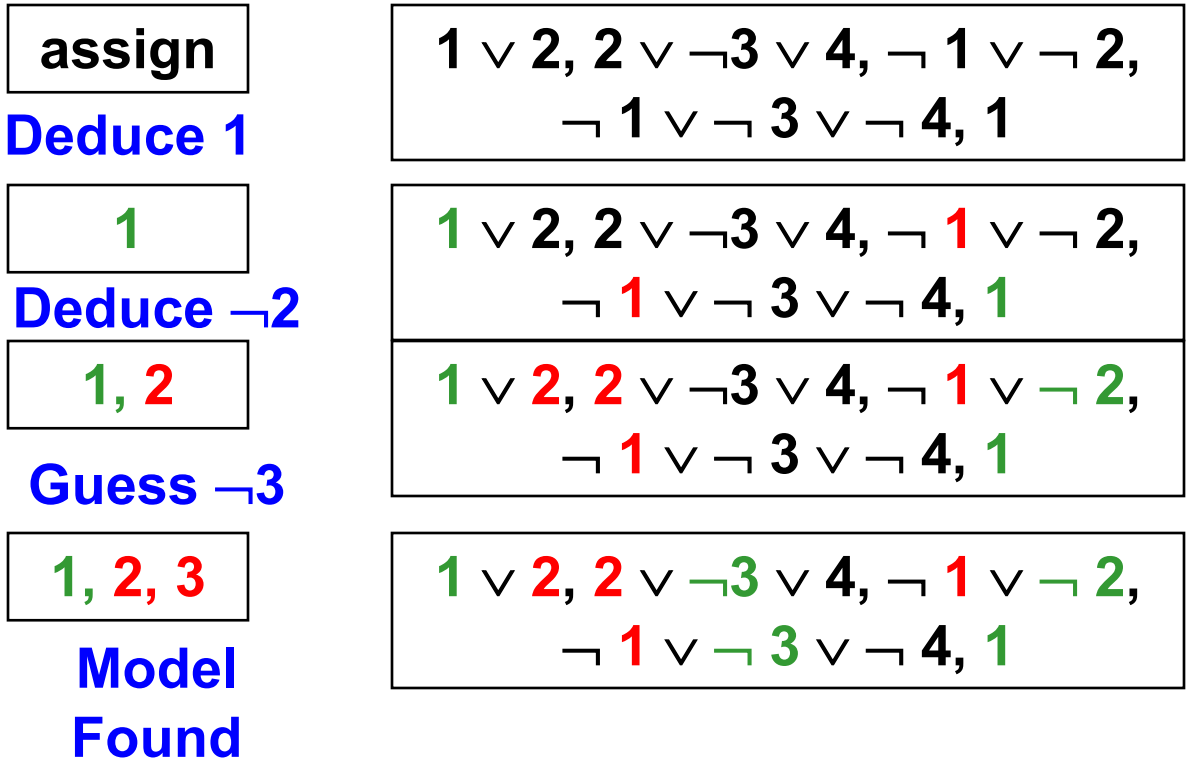
The Original DPLL Procedure – Example



The Original DPLL Procedure – Example



The Original DPLL Procedure – Example



An Abstract Framework for DPLL

The DPLL procedure can be described declaratively by simple sequent-style calculi

Such calculi, however, cannot model meta-logical features such as backtracking, learning, and restarts

We model DPLL and its enhancements as transition systems instead

A transition system is a binary relation over states, induced by a set of conditional transition rules

An Abstract Framework for DPLL

State

- **fail** or M ‖ F
- where
  - F is a CNF formula, a set of clauses, and
  - M is a sequence of annotated literals denoting a partial truth assignment

Initial State

- ∅ ‖ F, where F is to be checked for satisfiability

Expected final states:

- **fail** if F is unsatisfiable
- M ‖ G
  where
  - M is a model of G
  - G is logically equivalent to F

Transition Rules for DPLL

Extending the assignment:

UnitProp $M \parallel F, C \vee l \rightarrow M\ l \parallel F, C \vee l$ $\left\{ \begin{array}{l} M \models \neg C \\ l \text{ is undefined in } M \end{array} \right.$

Decide $M \parallel F, C \rightarrow M\ l^d \parallel F, C$ $\left\{ \begin{array}{l} l \text{ or } \neg l \text{ occur in } C \\ l \text{ is undefined in } M \end{array} \right.$

Notation: l^d is a decision literal

Transition Rules for DPLL

Repairing the assignment:

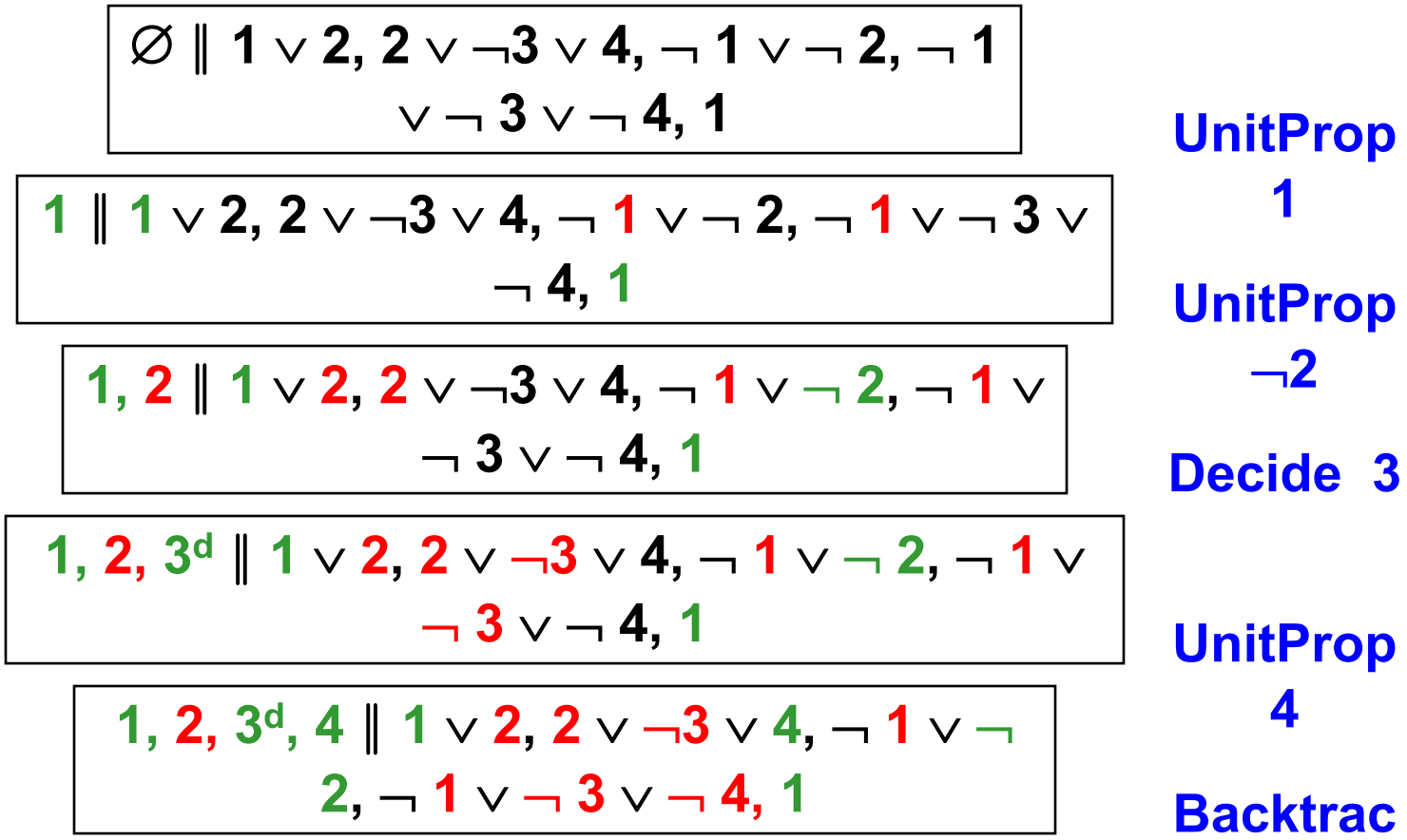
Fail $M \parallel F, C \rightarrow \text{fail}$

- $M \models \neg C$
- M does not contain decision literals

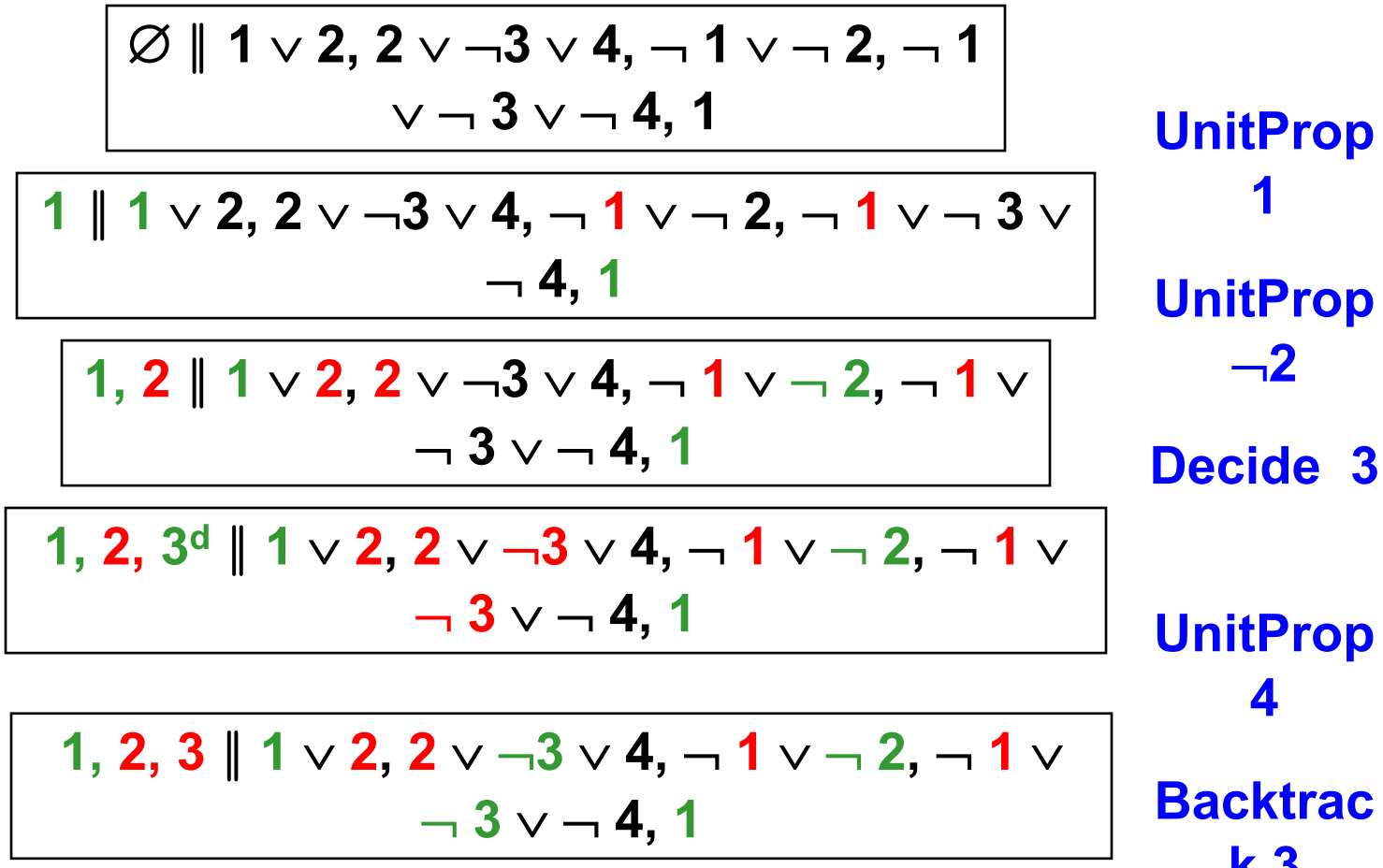
Backtrack $M\ l^d\ N \parallel F, C \rightarrow M \neg l \parallel F, C$

- $M\ l^d\ N \models \neg C$
- l is the last decision literal

Transition Rules DPLL – Example



Transition Rules DPLL – Example



Transition Rules for DPLL (on one slide)

UnitProp $M \parallel F, C \vee l \rightarrow M\ l \parallel F, C \vee l$ if $M \models \neg C$ and l is undefined in M

Decide $M \parallel F, C \rightarrow M\ l^d \parallel F, C$ if l or $\neg l$ occur in C and l is undefined in M

Fail $M \parallel F, C \rightarrow fail$ if $M \models \neg C$ and M does not contain decision literals

Backtrack $M\ l^d\ N \parallel F, C \rightarrow M \neg l \parallel F, C$ if $M\ l^d\ N \models \neg C$ and l is the last decision literal

The DPLL System – Correctness

Some terminology

- Irreducible state: state to which no transition rule applies.
- Execution: sequence of transitions allowed by the rules and starting with states of the form $\emptyset \parallel F$.
- Exhausted execution: execution ending in an irreducible state

Proposition (Strong Termination) Every execution in DPLL is finite

Proposition (Soundness) For every exhausted execution starting with $\emptyset \parallel F$ and ending in $M \parallel F$, $M \models F$

Proposition (Completeness) If F is unsatisfiable, every exhausted execution starting with $\emptyset \parallel F$ ends with fail

Maintained in more general rules + theories

Modern DPLL: CDCL

Conflict **D**riven **C**lause **L**earning

- two watched literals – efficient index to find clauses that can be used in unit resolution
- periodically restart backtrack search
- activity-based decision heuristic to choose decision variable
- **conflict resolution via clausal learning**

We will briefly look at clausal learning

More details on CDCL are available in

- Chapter 2 of Decision Procedures book
- ECE750 with Vijay Ganesh

Conflict Directed Clause Learning

Lemma learning

$\neg t, \mathbf{p}, \mathbf{q}, \mathbf{s} \mid t \vee \neg p \vee q, \neg q \vee s, \mathbf{\neg p \vee \neg s}$

$\neg t, p, q, s \mid t \vee \neg p \vee q, \neg q \vee \mathbf{s}, \neg p \vee \neg s \mid \mathbf{\neg p \vee \neg s}$

$\neg t, p, q, s \mid t \vee \neg p \vee \mathbf{q}, \neg q \vee s, \neg p \vee \neg s \mid \mathbf{\neg p \vee \neg q}$

$\neg t, p, q, s \mid t \vee \neg p \vee q, \neg q \vee s, \neg p \vee \neg s \mid \mathbf{\neg p \vee t}$

Learned Clause by Resolution

A new clause is learned by resolving the conflict clause with clauses deduced from the last decision

$\neg t, p, q, s \mid t \vee \neg p \vee q, \neg q \vee s, \neg p \vee \neg s$

$$\dfrac{\dfrac{t \vee \neg p \vee q \qquad \neg q \vee s}{t \vee \neg p \vee s} \qquad \neg p \vee \neg s}{\neg p \vee t}$$

Learned Clause by Resolution

A new clause is learned by resolving the conflict clause with clauses deduced from the last decision

$$\neg t, p, q, s \mid t \vee \neg p \vee q, \neg q \vee s, \neg p \vee \neg s$$

$$\dfrac{\dfrac{\neg p \vee \neg s \qquad \neg q \vee s}{\neg p \vee \neg q} \qquad t \vee \neg p \vee q}{t \vee \neg p}$$

Trivial Resolution: at every resolution step, at least one clause is an input clause

Modern CDCL: Abstract Rules

Initialize	$\epsilon \mid F$	F *is a set of clauses*
Decide	$M \mid F \Longrightarrow M, \ell \mid F$	ℓ *is unassigned*
Propagate	$M \mid F, C \vee \ell \Longrightarrow M, \ell^{C \vee \ell} \mid F, C \vee \ell$	C *is false under* M
Sat	$M \mid F \Longrightarrow M$	F *true under* M
Conflict	$M \mid F, C \Longrightarrow M \mid F, C \mid C$	C *is false under* M
Learn	$M \mid F \mid C \Longrightarrow M \mid F, C \mid C$	
Unsat	$M \mid F \mid \emptyset \Longrightarrow Unsat$	
Backjump	$MM' \mid F \mid C \vee \ell \Longrightarrow M\ell^{C \vee \ell} \mid F$	$\bar{C} \subseteq M, \neg\ell \in M'$
Resolve	$M \mid F \mid C' \vee \neg\ell \Longrightarrow M \mid F \mid C' \vee C$	$\ell^{C \vee \ell} \in M$
Forget	$M \mid F, C \Longrightarrow M \mid F$	C is a learned clause
Restart	$M \mid F \Longrightarrow \epsilon \mid F$	

Model

Proof

Conflict Resolution

[Nieuwenhuis, Oliveras, Tinelli J.ACM 06] customized

Conjuctive Normal Form

$$
\begin{array}{lll}
\varphi \leftrightarrow \psi & \Rightarrow_{\mathrm{CNF}} & \varphi \rightarrow \psi \wedge \psi \rightarrow \varphi \\
\varphi \rightarrow \psi & \Rightarrow_{\mathrm{CNF}} & \neg\varphi \vee \psi \\
\neg(\varphi \vee \psi) & \Rightarrow_{\mathrm{CNF}} & \neg\varphi \wedge \neg\psi \\
\neg(\varphi \wedge \psi) & \Rightarrow_{\mathrm{CNF}} & \neg\varphi \vee \neg\psi \\
\neg\neg\varphi & \Rightarrow_{\mathrm{CNF}} & \varphi \\
(\varphi \wedge \psi) \vee \xi & \Rightarrow_{\mathrm{CNF}} & (\varphi \vee \xi) \wedge (\psi \vee \xi)
\end{array}
$$

Every propositional formula can be put in CNF

PROBLEM: (potential) exponential blowup of the resulting formula

Tseitin Transformation – Main Idea

Introduce a fresh variable e_i for every subformula G_i of F

- intuitively, e_i represents the truth value of G_i

Assert that every e_i and G_i pair are equivalent

- $e_i \leftrightarrow G_i$
- and express the assertion as CNF

Conjoin all such assertions in the end

Formula to CNF Conversion

```
def cnf (ϕ):
  p, F = cnf_rec (ϕ)
  return p ∧ F

def cnf_rec (ϕ):
  if is_atomic (ϕ): return (ϕ, True)
  elif ϕ == ψ ∧ ξ:
    q, F1 = cnf_rec (ψ)
    r, F2 = cnf_rec (ξ)

    p = mk_fresh_var ()
    # C is CNF for p↔(q∧r)
    C = (¬p∨q)∧(¬p∨r)∧(p∨¬q∨¬r)
    return (p, F1∧F2∧C)
  elif ϕ == ψ∨ξ:
    …
```

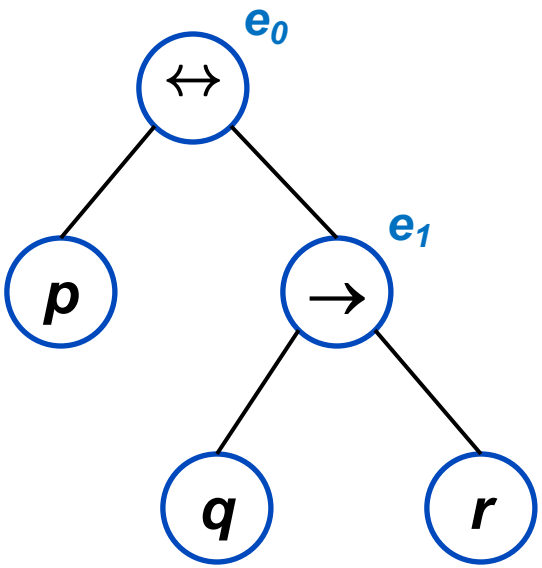
`mk_fresh_var()` returns a fresh variable not used anywhere before

Exercise: Complete cases for

$\phi == \psi \vee \xi$, $\phi == \neg\psi$, $\phi == \psi \leftrightarrow \xi$

Tseitin Transformation: Example

$G : p \leftrightarrow (q \rightarrow r)$

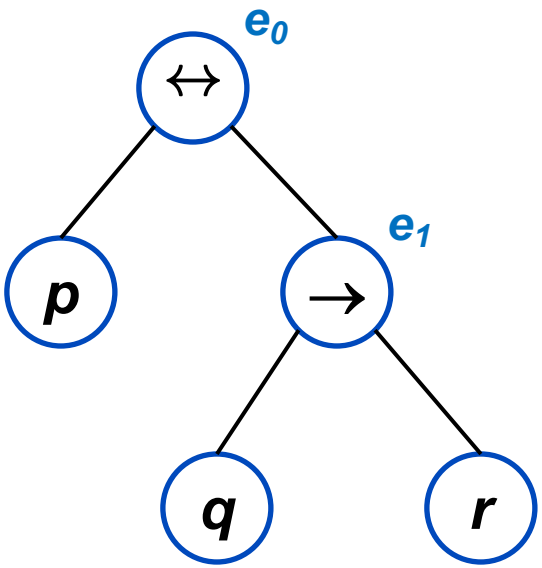


$G : e_0 \wedge (e_0 \leftrightarrow (p \leftrightarrow e_1)) \wedge (e_1 \leftrightarrow (q \rightarrow r))$

$$
\begin{aligned}
& e_1 \leftrightarrow (q \rightarrow r) \\
= \; & (e_1 \rightarrow (q \rightarrow r)) \wedge ((q \rightarrow r) \rightarrow e_1) \\
= \; & (\neg e_1 \vee \neg q \vee r) \wedge ((\neg q \vee r) \rightarrow e_1) \\
= \; & (\neg e_1 \vee \neg q \vee r) \wedge (\neg q \rightarrow e_1) \wedge (r \rightarrow e_1) \\
= \; & (\neg e_1 \vee \neg q \vee r) \wedge (q \vee e_1) \wedge (\neg r \vee e_1)
\end{aligned}
$$

Tseitin Transformation: Example

$G : p \leftrightarrow (q \rightarrow r)$

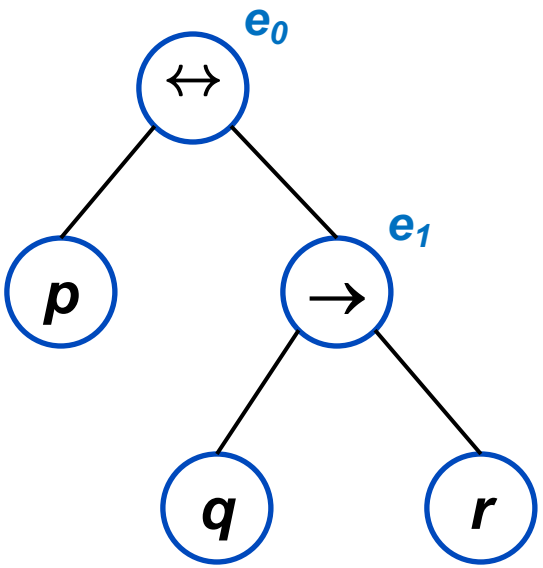


$G : e_0 \wedge (e_0 \leftrightarrow (p \leftrightarrow e_1)) \wedge (e_1 \leftrightarrow (q \rightarrow r))$

$$
\begin{aligned}
& e_0 \leftrightarrow (p \leftrightarrow e_1) \\
= \quad & (e_0 \rightarrow (p \leftrightarrow e_1)) \wedge ((p \leftrightarrow e_1)) \rightarrow e_0) \\
= \quad & (e_0 \rightarrow (p \rightarrow e_1)) \wedge (e_0 \rightarrow (e_1 \rightarrow p)) \wedge \\
& (((p \wedge e_1) \vee (\neg p \wedge \neg e_1)) \rightarrow e_0) \\
= \quad & (\neg e_0 \vee \neg p \vee e_1) \wedge (\neg e_0 \vee \neg e_1 \vee p) \wedge \\
& (\neg p \vee \neg e_1 \vee e_0) \wedge (p \vee e_1 \vee e_0)
\end{aligned}
$$

Tseitin Transformation: Example

$G : p \leftrightarrow (q \rightarrow r)$



$G : e_0 \wedge (e_0 \leftrightarrow (p \leftrightarrow e_1)) \wedge (e_1 \leftrightarrow (q \rightarrow r))$

$G : e_0 \wedge (\neg e_0 \vee \neg p \vee e_1) \wedge (\neg e_0 \vee p \vee \neg e_1) \wedge (e_0 \vee p \vee e_1) \wedge (e_0 \vee \neg p \vee \neg e_1) \wedge$
$(\neg e_1 \vee \neg q \vee r) \wedge (e_1 \vee q) \wedge (e_1 \vee \neg r)$

Tseitin Transformation [1968]

Used in practice

- No exponential blow-up
- CNF formula size is linear with respect to the original formula

Does not produce an equivalent CNF

However, given F, the following holds for the computed CNF F':

- F' is equisatisfiable to F
- Every model of F' can be translated (i.e., projected) to a model of F
- Every model of F can be translated (i.e., completed) to a model of F'

No model is lost or added in the conversion

DIMACS CNF File Format

Textual format to represent CNF-SAT problems

```
c start with comments
c
c
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

Format details

- comments start with `c`
- header line: `p cnf nbvar nbclauses`
  - `nbvar` is # of variables, `nbclauses` is # of clauses
- each clause is a sequence of distinct numbers terminating with 0
  - positive numbers are variables, negative numbers are negations

BOUNDED MODEL CHECKING

SAT-based Model Checking

Main idea

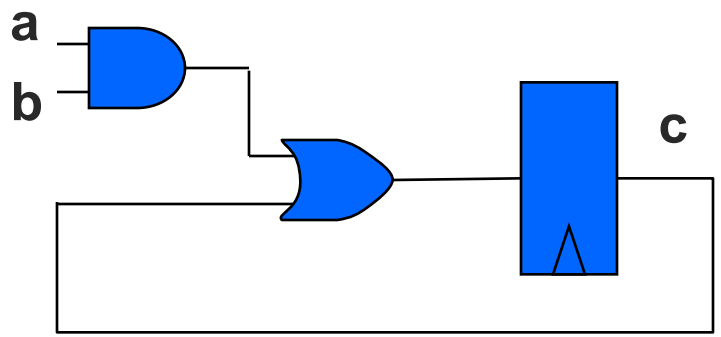
Translate the model and the specification to propositional formulas

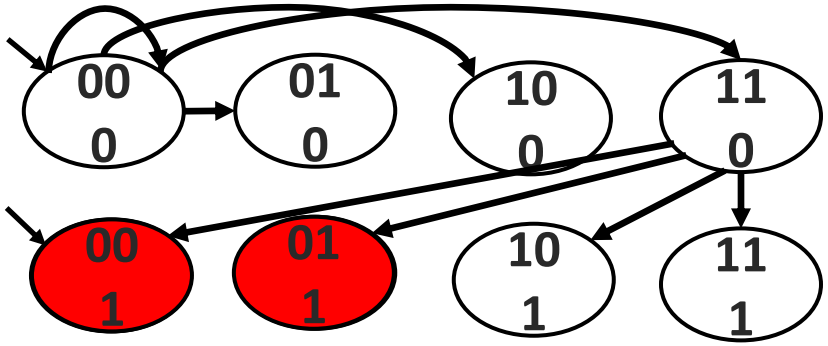
(p, ¬p, p∨q, p∧q, p→q…)

Reduce the model checking problem to satisfiability of propositional formulas

Use efficient tools (SAT solvers) for solving the satisfiability problem

Modeling with Propositional Formulas





Finite-State System is modeled as (V, INIT, T):

- **V – finite set of Boolean variables**
  - **Boolean variables: a b c ➔ 8 states: 000,001,…**
- **INIT(V) – describes the set of initial states**
  - **INIT = ¬a ∧ ¬b**
- **T(V,V') – describes the set of transitions**
  - **T(a,b,c,a',b',c') = (c' ↔ (a ∧ b) ∨ c)**

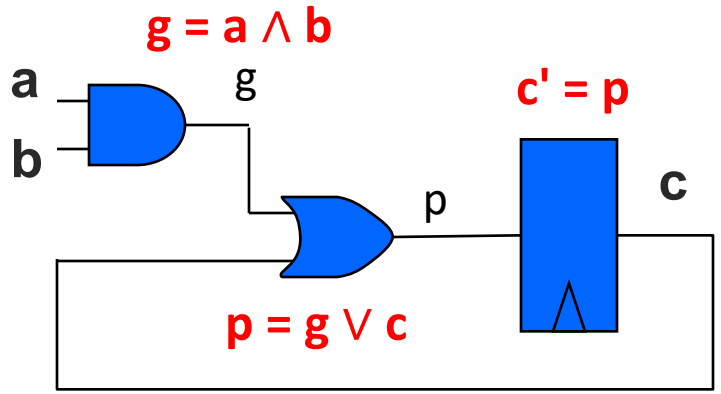
state = valuation to variables

note: $c = c_t$ and $c' = c_{t+1}$

Property:

- **p(V) - describes the set of states satisfying p**
  - **p = a ∨ ¬c (Bad = ¬p = ¬a ∧ c)**

Modeling in CNF (Tseitin encoding)



$T(a,b,c,g,p,a',b',c') =$

$g \leftrightarrow a \wedge b,$

$p \leftrightarrow g \vee c,$

$c' \leftrightarrow p$

Each circuit element is a constraint

Bounded model checking (BMC) for checking AGp

Given

- A finite transition system M= (V, INIT(V), T(V,V'))
- A safety property AG p, where p = p(V)
- A bound k

Determine

- Does M contain a counterexample to p of *k transitions (or fewer)* ?

*** BMC can handle all of LTL formulas**

A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu, DAC'99

Bounded model checking for checking AGp

Unwind the model for k levels, i.e., construct all computations of length k

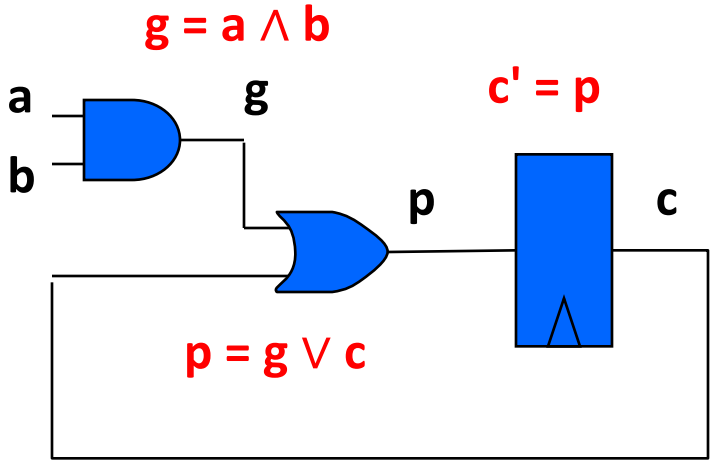
If a state satisfying $\neg p$ is encountered, then produce a counterexample

The method is suitable for **falsification**, not verification

Can be translated to a SAT problem

Bounded model checking with SAT

Construct a formula $f_{M,k}$ describing all possible computations of M of length k



T(a,b,c,a',b',c') =

$g \leftrightarrow a \wedge b,$

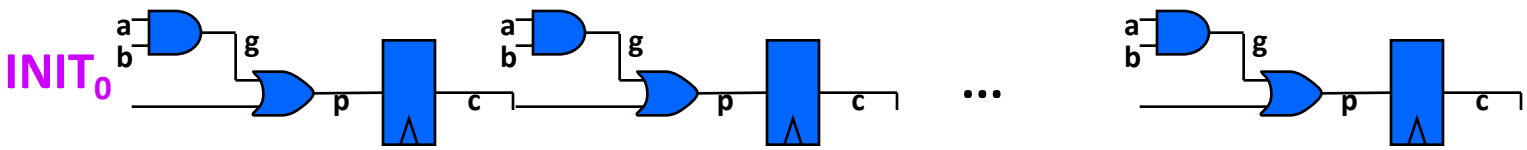
$p \leftrightarrow g \vee c,$

$c' \leftrightarrow p$

Bounded model checking with SAT

Construct a formula $f_{M,k}$ describing all possible computations of M of length k

$$f_{M,k} = INIT_0 \wedge T_0 \wedge T_1 \wedge \ldots \wedge T_{k-1}$$



$a_0,b_0,c_0,$ $a_1,b_1,c_1,$ $a_{k-1},b_{k-1},c_{k-1},$ $a_k,b_k,c_k,$

g_0,p_0 g_1,p_1 g_{k-1},p_{k-1} g_k,p_k

$INIT_0 = INIT(V_0)$

$T_i = T(V_i,V_{i+1})$

Bounded model checking with SAT

Construct a formula $\mathbf{f_{M,k}}$ describing all possible computations of M of length k

Construct a formula $\mathbf{f_{\varphi,k}}$ expressing that φ=**EF**$\neg$**p** holds within k computation steps

$\mathbf{f_{\varphi,k}} = V_{i=0,..k} (\neg \mathbf{p_i})$ [Sometimes $\mathbf{f_{\varphi,k}} = \neg \mathbf{p_k}$]

$\mathbf{p_i = p(V_i)}$

Bounded model checking with SAT

Construct a formula $f_{M,k}$ describing all possible computations of M of length k

Construct a formula $f_{\varphi,k}$ expressing that $\varphi = \mathbf{EF}\neg\mathbf{p}$ holds within k computation steps

Check whether $f = f_{M,k} \wedge f_{\varphi,k}$ is satisfiable

If f is satisfiable then $M \not\models \mathbf{AGp}$

The satisfying assignment is a **counterexample**

BMC for checking AG p with SAT

Unfold the model k times:

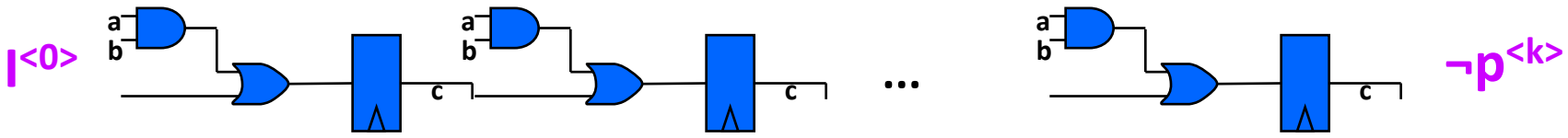
- $U = T^{<0>} \wedge T^{<1>} \wedge \ldots \wedge T^{<k-1>}$

Biere, et al. TACAS99

$I^{<0>} = I(V_0)$

$T^{<i>} = T(V_i, V_{i+1})$

$p^{<k>} = p(V_k)$



- Use SAT solver to check satisfiability of

  $I^{<0>} \wedge U \wedge \neg p^{<k>}$

- If satisfiable: the satisfying assignment describes a counterexample of length k
- If unsatisfiable: property has no counterexample of length k

Example – shift register

Shift register of 3 bits: <x, y, z>

Transition relation:

$T(x,y,z,x',y',z') = x' \leftrightarrow y \wedge y' \leftrightarrow z \wedge z'=1$

error

Initial condition:

$INIT(x,y,z) = x=0 \vee y=0 \vee z=0$

Specification: $AG (x=0 \vee y=0 \vee z=0)$

Propositional formula for k=2

$$f_{M,2} = (x_0=0 \vee y_0=0 \vee z_0=0) \wedge$$
$$(x_1 \leftrightarrow y_0 \wedge y_1 \leftrightarrow z_0 \wedge z_1=1) \wedge$$
$$(x_2 \leftrightarrow y_1 \wedge y_2 \leftrightarrow z_1 \wedge z_2=1)$$

$$\text{INIT} = x=0 \vee y=0 \vee z=0$$

$$T = x' \leftrightarrow y \wedge y' \leftrightarrow z \wedge z'=1$$

$$f_{\varphi,2} = V_{i=0,..2} (x_i=1 \wedge y_i=1 \wedge z_i=1)$$

$$P = x=0 \vee y=0 \vee z=0$$

Satisfying assignment: 101 011 111

This is a counterexample!

A remark

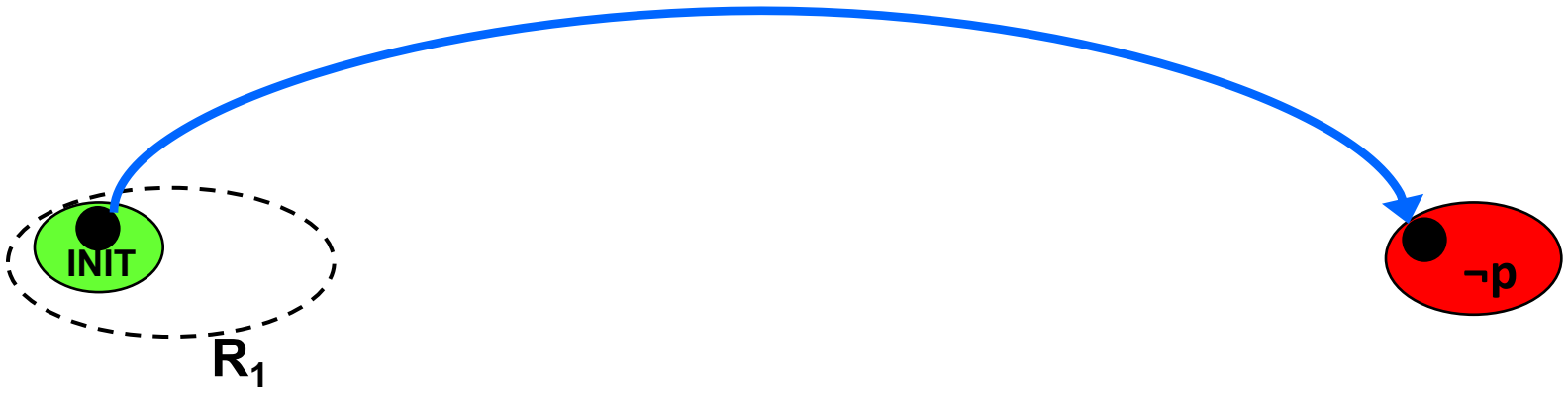
In order to describe a computation of length k by a propositional formula we need k+1 copies of the state variables.

With BDDs we use only two copies: for current and next states.

BMC for checking φ=AGp

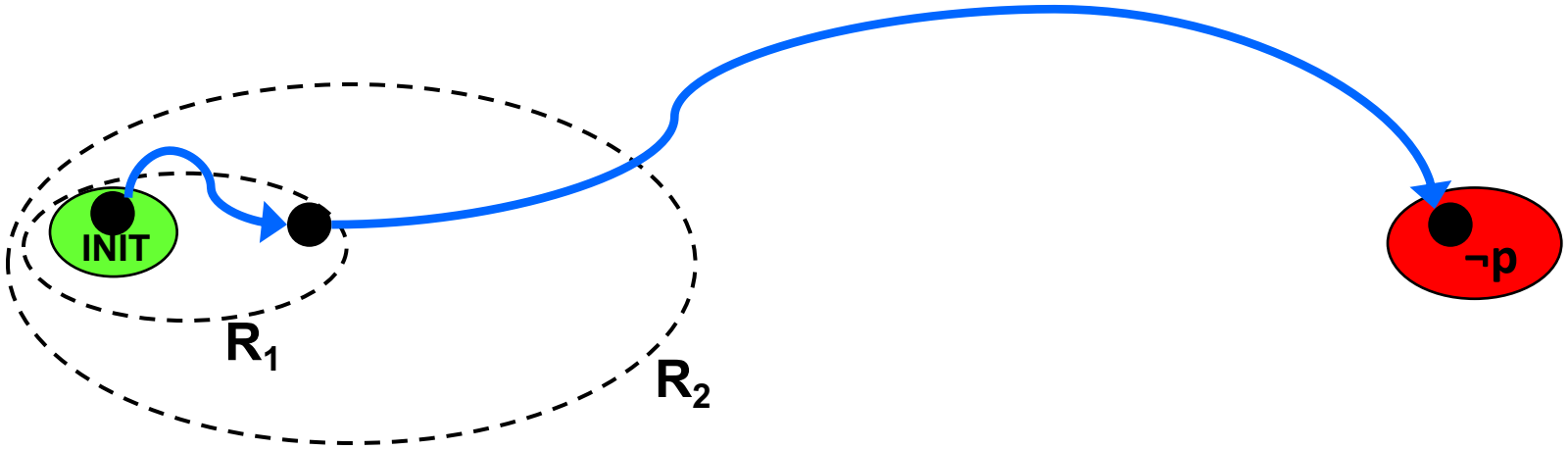
1. k=1
2. Build a propositional formula f_M^k describing all prefixes of length k of paths of M from an initial state
3. Build a propositional formula f_φ^k describing all prefixes of length k of paths satisfying F¬p
4. If $(f_M^k \wedge f_\varphi^k)$ is **satisfiable**,
   return the satisfying assignment as a **counterexample**
5. Otherwise, increase k and return to 2.

Bounded Model Checking



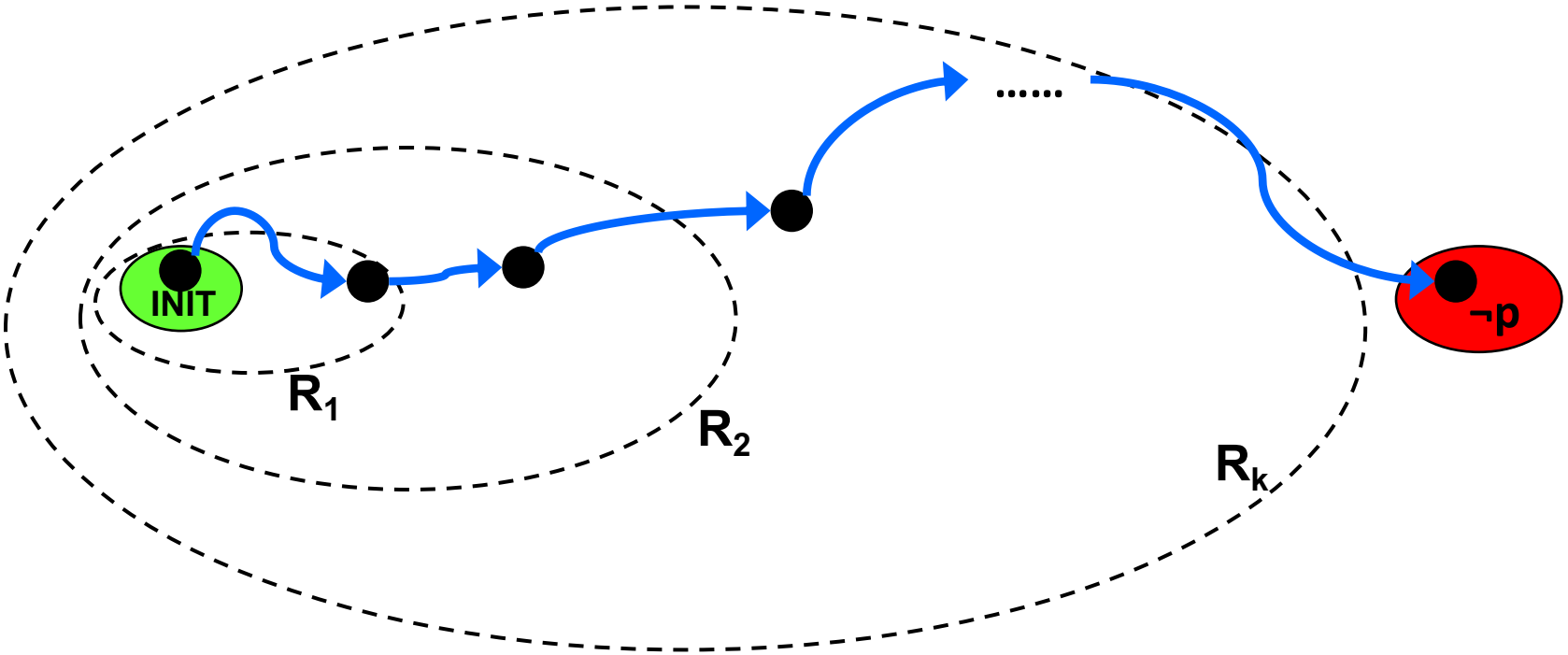
$$INIT(V^0) \wedge T(V^0,V^1) \wedge \neg p(V^1)$$

Bounded Model Checking



$$INIT(V^0) \wedge T(V^0,V^1) \wedge T(V^1,V^2) \wedge \neg p(V^2)$$

Bounded Model Checking



$$INIT(V^0) \wedge T(V^0,V^1) \wedge \ldots \wedge T(V^{k-1},V^k) \wedge \neg p(V^k)$$

BMC for checking AFp (φ=EG¬p)

Is there an infinite path in M

- From an initial state
- all of its states satisfying ¬**p**
- Over k+1 states ?

If exists, there must also exist a lasso

BMC for checking AFp ($\varphi = EG\neg p$)

An infinite path in M, from an initial state, over k+1 states, all satisfying $\neg p$:

$$f_M^k(V_0,\ldots,V_k) = INIT(V_0) \wedge \bigwedge_{i=0,\cdots k-1} T(V_i,V_{i+1}) \wedge \bigvee_{i=0,\cdots k-1} (V_k = V_i)$$

- $V_k = V_i$ means bitwise equality: $\bigwedge_{j=0,\ldots n} (v_{kj} \leftrightarrow v_{ij})$

$$f_\varphi^k(V_0,\ldots,V_k) = \bigwedge_{i=0,\cdots k} \neg p(V_i)$$

Remark: BMC can handle all of LTL formulas

Bounded model checking

Can handle all of **LTL** formulas

Can be used for **verification** by choosing k which is large enough

- Need bound on length of the shortest counterexample.
  - *diameter* bound. The diameter is the maximum length of the shortest path between any two states.

Using such k is often **not practical** due to the size of the model

- Worst case diameter is exponential. Obtaining better bounds is sometimes possible, but generally intractable.

Bounded Model Checking

Terminates

- with a counterexample or
- with time- or memory-out

=> The method is suitable for **falsification**, not verification

Can be used for **verification** by choosing k which is large enough

- Need bound on length of the shortest counterexample.
  - *diameter* bound. The diameter is the maximum length of the shortest path between any two states.

Using such k is often **not practical**

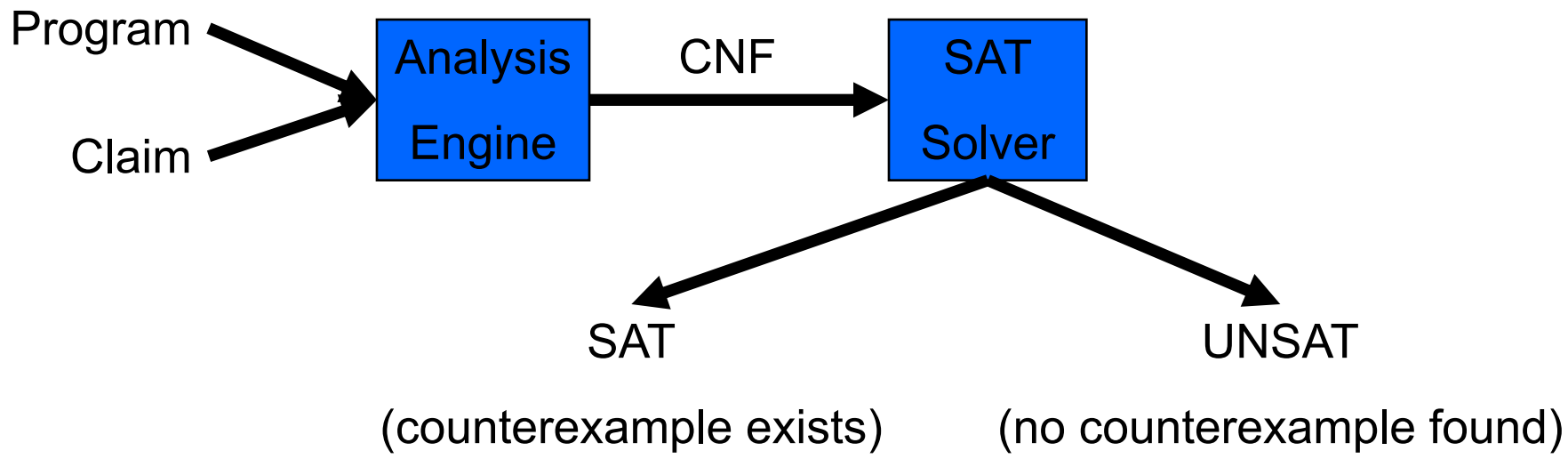
  - Worst case diameter is exponential. Obtaining better bounds is sometimes possible, but generally intractable.

Bounded Model Checker for C

CBMC

Bug Catching with SAT-Solvers

Main Idea: Given a program and a claim use a SAT-solver to find whether there exists an execution that violates the claim.



Programs and Claims

- Arbitrary ANSI-C programs
  - With bitvector arithmetic, dynamic memory, pointers, …
- Simple Safety Claims
  - Array bound checks (i.e., buffer overflow)
  - Division by zero
  - Pointer checks (i.e., NULL pointer dereference)
  - Arithmetic overflow
  - User supplied assertions (i.e., `assert (i > j)`)
  - etc

Why use a SAT Solver?

- SAT Solvers are very efficient
- Analysis is completely automated
- Analysis as good as the underlying SAT solver
- Allows support for many features of a programming language
  - bitwise operations, pointer arithmetic, dynamic memory, type casts

A (very) simple example (1)

Program

```
int x;
int y=8,z=0,w=0;
if (x)
  z = y - 1;
else
  w = y + 1;
assert (z == 7 ||
        w == 9)
```

Constraints

y = 8,

z = x ? y – 1 : 0,

w = x ? 0 :y + 1,

z != 7,

w != 9

UNSAT

no counterexample

assertion always holds!

A (very) simple example (2)

Program

```
int x;
int y=8,z=0,w=0;
if (x)
  z = y - 1;
else
  w = y + 1;
assert (z == 5 ||
        w == 9)
```

Constraints

$$y = 8,$$
$$z = x \ ? \ y - 1 : 0,$$
$$w = x \ ? \ 0 : y + 1,$$
$$z \neq 5,$$
$$w \neq 9$$

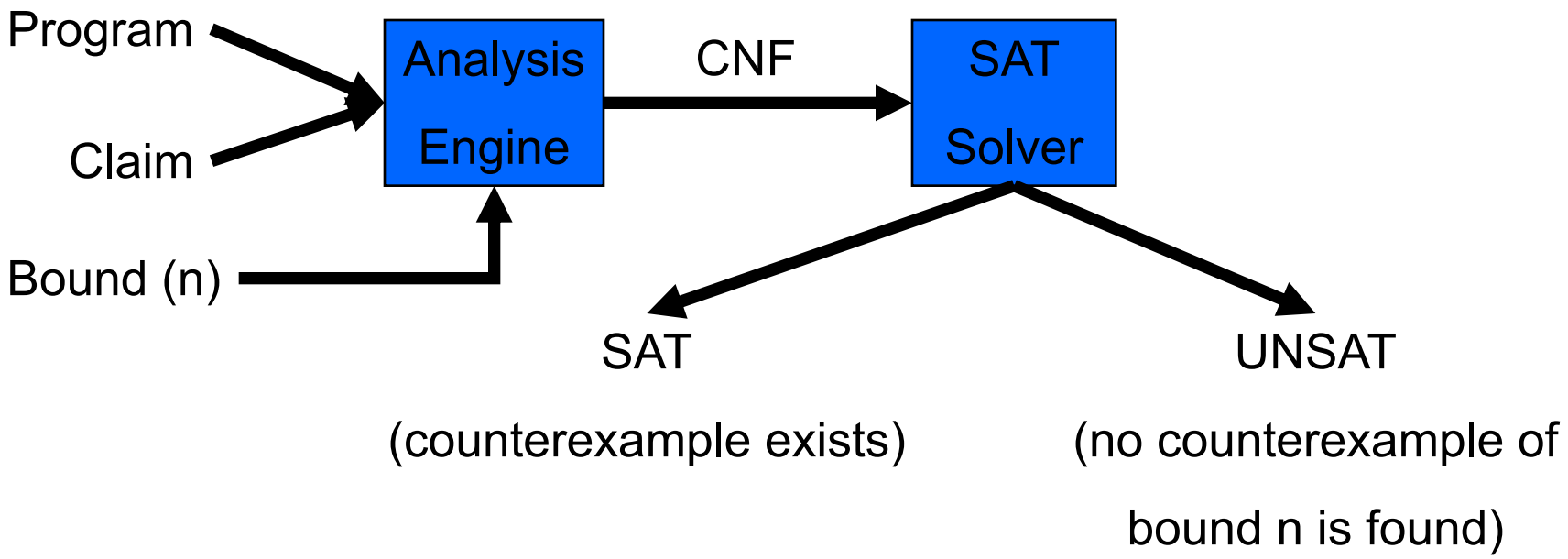
SAT

counterexample found!

$y = 8, x = 1, w = 0, z = 7$

What about loops?!

- SAT Solver can only explore finite length executions!
- Loops must be bounded (i.e., the analysis is incomplete)



CBMC: C Bounded Model Checker

- Developed at CMU by Daniel Kroening and Ed Clarke
- Available at: http://www.cprover.org/cbmc
  - On Ubuntu: `apt-get install cbmc`
  - `with source code`
- Supported platforms: Windows, Linux, OSX
- Has a command line, Eclipse CDT, and Visual Studio interfaces

- Scales to programs with over 30K LOC
- Found previously unknown bugs in MS Windows device drivers

CBMC: Supported Language Features

ANSI-C is a low level language, not meant for verification but for efficiency

Complex language features, such as

- Bit vector operators (shifting, and, or,…)
- Pointers, pointer arithmetic
- Dynamic memory allocation: malloc/free
- Dynamic data types: `char s[n]`
- Side effects
- `float` / `double`
- Non-determinism

DEMO

Using CBMC from Command Line

- To see the list of claims

```
cbmc --show-claims -I include file.c
```

- To check a single claim

```
cbmc --unwind n --claim x –I include file.c
```

- For help
  - cbmc --help

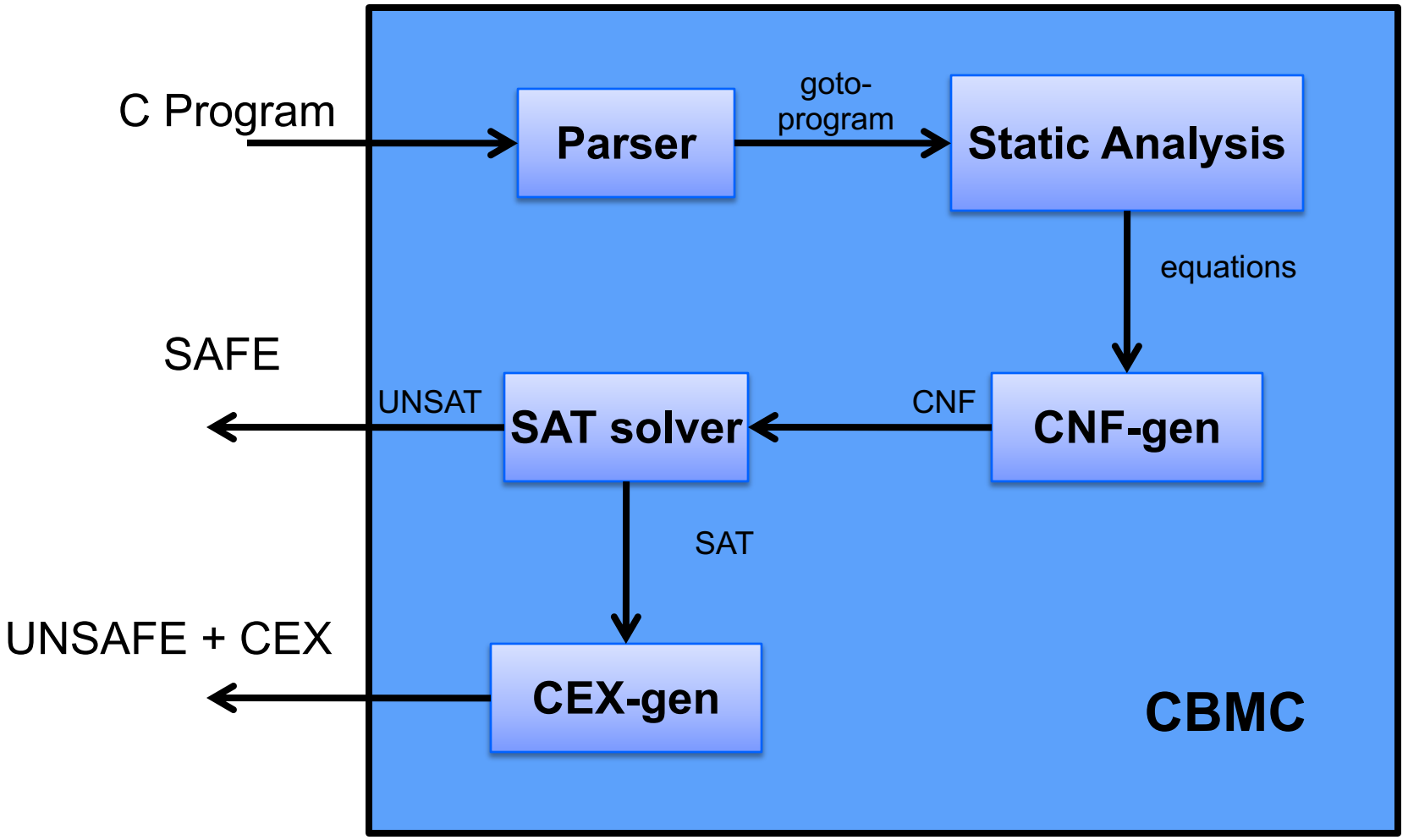
How does it work

Transform a programs into a set of equations

1. Simplify control flow
2. Unwind all of the loops
3. Convert into Single Static Assignment (SSA)
4. Convert into equations
5. Bit-blast
6. Solve with a SAT Solver
7. Convert SAT assignment into a counterexample

CBMC: Bounded Model Checker for C

A tool by D. Kroening/Oxford and Ed Clarke/CMU



Control Flow Simplifications

- All side effect are removed
  - `e.g., j=i++` becomes `j=i;i=i+1`
- Control Flow is made explicit
  - `continue`, `break` replaced by `goto`
- All loops are simplified into one form
  - `for`, `do while` replaced by `while`

Loop Unwinding

- All loops are unwound
  - can use different unwinding bounds for different loops
  - to check whether unwinding is sufficient special “unwinding assertion” claims are added
- If a program satisfies all of its claims and all unwinding assertions then it is correct!
- Same for backward `goto` jumps and recursive functions

Loop Unwinding

```
void f(...) {
  ...
  while(cond) {
    Body;
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Loop Unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    while(cond) {
      Body;
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Loop Unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond) {
        Body;
      }
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Unwinding assertion

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        while(cond) {
          Body;
        }
      }
    }
  }
  Remainder;
}
```

- while() loops are unwound iteratively
- Break / continue replaced by goto
- Assertion inserted after last iteration: violated if program runs longer than bound permits

Unwinding assertion

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        assert(!cond);

        }
      }
    }
  }
  Remainder;
}
```

Unwinding assertion

- while() loops are unwound iteratively
- Break / continue replaced by goto
- Assertion inserted after last iteration: violated if program runs longer than bound permits
- Positive correctness result!

Example: Sufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 2)
    j = j + 1;
  Remainder;
}
```

unwind = 3

```
void f(...) {
  j = 1
  if(j <= 2) {
    j = j + 1;
    if(j <= 2) {
      j = j + 1;
      if(j <= 2) {
        j = j + 1;
        assert(!(j <= 2));
        }
      }
    }
  }
  Remainder;
}
```

Example: Insufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 10)
    j = j + 1;
  Remainder;
}
```

unwind = 3

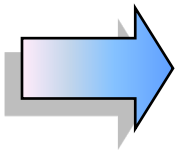
```
void f(...) {
  j = 1
  if(j <= 10) {
    j = j + 1;
    if(j <= 10) {
      j = j + 1;
      if(j <= 10) {
        j = j + 1;
        assert(!(j <= 10));
        }
      }
    }
  }
  Remainder;
}
```

Transforming Loop-Free Programs Into Equations (1)

Easy to transform when every variable is only assigned once!

Program

```
x = a;
y = x + 1;
z = y - 1;
```



Constraints

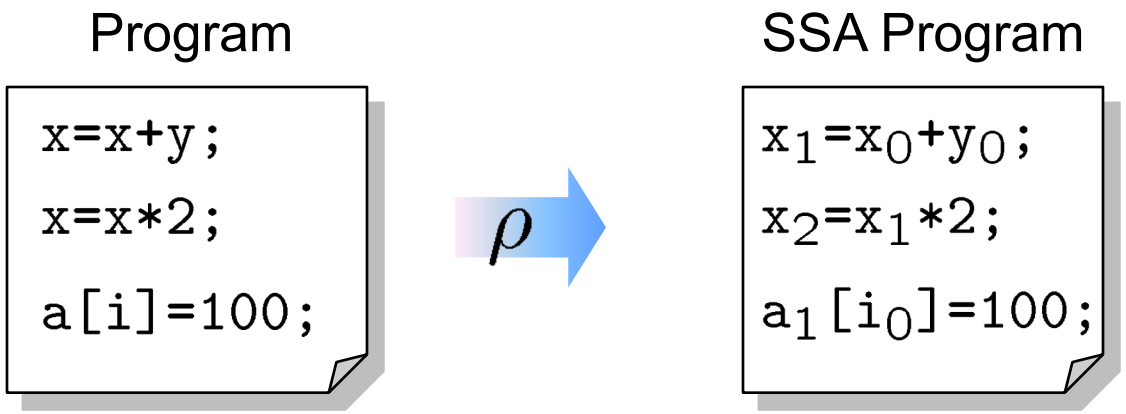
x = a &&

y = x + 1 &&

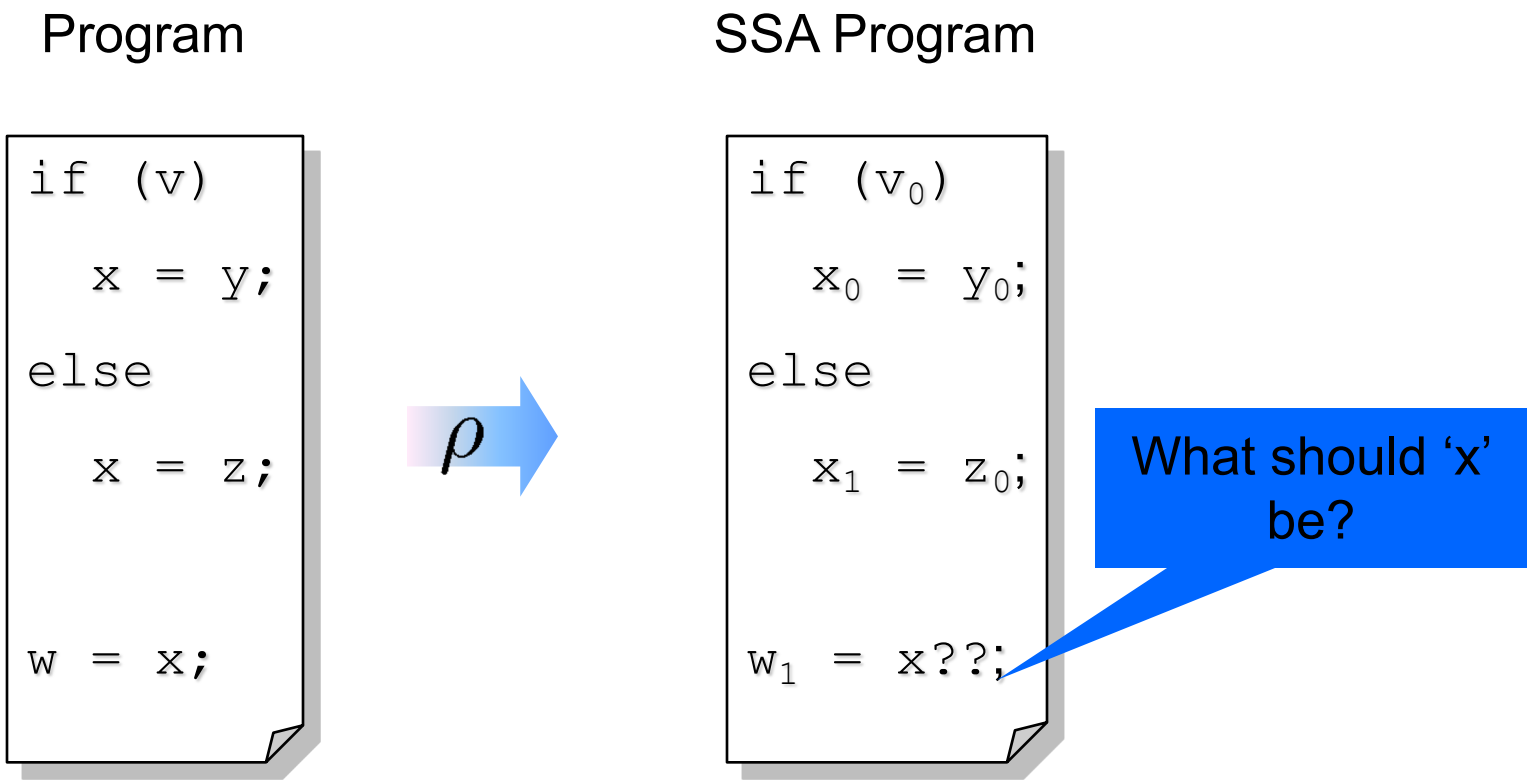
z = y – 1 &&

Transforming Loop-Free Programs Into Equations (2)

When a variable is assigned multiple times,
use a new variable for the RHS of each assignment



What about conditionals?



What about conditionals?

Program

```
if (v)
  x = y;
else
  x = z;


w = x;
```

SSA Program

```
if (v0)
  x0 = y0;
else
  x1 = z0;
x2 = v0 ? x0 : x1;
w1 = x2
```

For each join point, add new variables with selectors

Adding Unbounded Arrays

$$v_\alpha[a] = e \quad \rho \quad v_\alpha = \lambda i : \begin{cases} \rho(e) & : \ i = \rho(a) \\ v_{\alpha-1}[i] & : \ \text{otherwise} \end{cases}$$

Arrays are updated “whole array” at a time

A[1] = 5; A_1=λ i : i == 1 ? 5 : A_0[i]

A[2] = 10; A_2=λ i : i == 2 ? 10 : A_1[i]

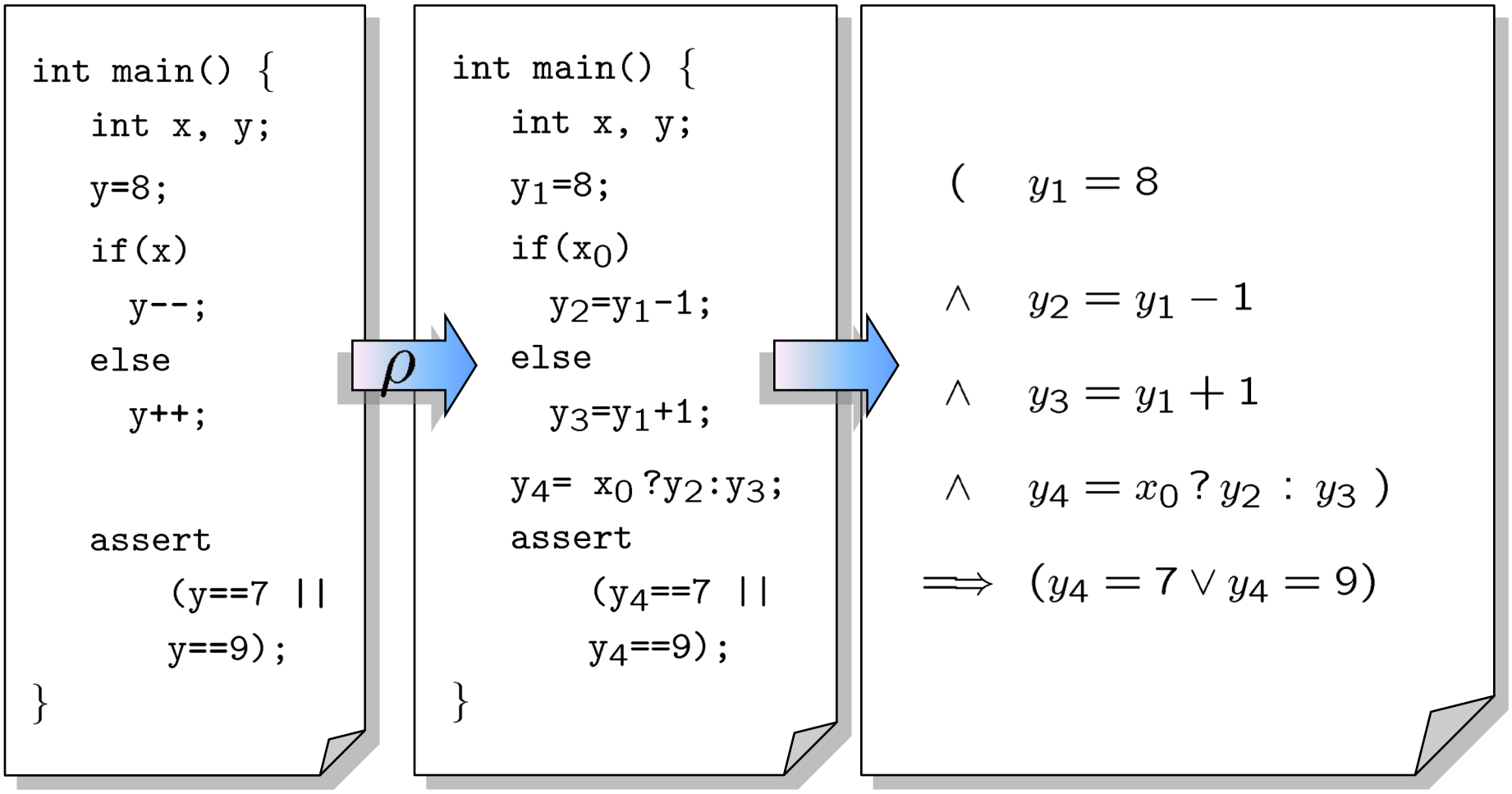
A[k] = 20; A_3=λ i : i == k ? 20 : A_2[i]

Examples: $A_2[2] == 10$ $A_2[1]==5$ $A_2[3] == A_0[3]$

$A_3[2] == (k==2 \ ? \ 20 : 10)$

Uses only as much space as there are uses of the array!

Example


int main() {
int x, y;
y=8;
if(x)
y--;
else
y++;
assert
(y==7 ||
y==9);
}
ρ
int main() {
int x, y;
y1=8;
if(x0)
y2=y1-1;
else
y3=y1+1;
y4= x0 ?y2:y3;
assert
(y4==7 ||
y4==9);
}
(y1 = 8
∧ y2 = y1 − 1
∧ y3 = y1 + 1
∧ y4 = x0 ? y2 : y3)
⟹ (y4 = 7 ∨ y4 = 9)

Pointers

While unwinding, record right hand side of assignments to pointers

This results in very precise points-to information

- Separate for each pointer
- Separate for each instance of each program location

Dereferencing operations are expanded into case-split on pointer object (not: offset)

- Generate assertions on offset and on type

Pointer data type assumed to be part of bit-vector logic

- Consists of pair <object, offset>

Dynamic Objects

Dynamic Objects:

- `malloc` / `free`
- Local variables of functions

Auxiliary variables for each dynamically allocated object:

- Size (number of elements)
- Active bit
- Type

`malloc` sets size (from parameter) and sets active bit

`free` asserts that active bit is set and clears bit

Same for local variables: active bit is cleared upon leaving the function

Modeling with CBMC

UNIVERSITY OF
WATERLOO

From Programming to Modeling

Extend C programming language with 3 modeling features

Assertions

- assert(e) – aborts an execution when e is false, no-op otherwise

```
void assert (_Bool b) { if (!b)  exit(); }
```

Non-determinism

- nondet_int() – returns a non-deterministic integer value

```
int nondet_int () { int x; return x; }
```

Assumptions

- assume(e) – “ignores” execution when e is false, no-op otherwise

```
void assume (_Bool e) { while (!e) ;  }
```

Example

```
int x, y;
void main (void)
{
  x = nondet_int ();

  assume (x > 10);
  y = x + 1;

  assert (y > x);
}
```

possible overflow

assertion fails

Using nondet for modeling

Library spec:

"foo is given non-deterministically, but is taken until returned"

CMBC stub:

```
int nondet_int ();
int is_foo_taken = 0;
int grab_foo () {
  if (!is_foo_taken)
    is_foo_taken = nondet_int ();
  return is_foo_taken; }
```

```
void return_foo ()
{ is_foo_taken = 0; }
```

Assume-Guarantee Reasoning (1)

Is `foo` correct?

Check by splitting on the argument of `foo`

```
int foo (int* p) { … }

void main(void) {

  …

  foo(x);

  …

  foo(y);

  …

}
```

Assume-Guarantee Reasoning (2)

(A) Is `foo` correct assuming `p` is not NULL?

```
int foo (int* p) { __CPROVER_assume(p!=NULL); … }
```

(G)Is `foo` guaranteed to be called with a non-NULL argument?

```
void main(void) {
  …
  assert (x!=NULL);// foo(x);
  …
  assert (y!=NULL); //foo(y);
  …}
```

Dangers of unrestricted assumptions

Assumptions can lead to vacuous satisfaction

This program is passed by CMBMC!

```
if (x > 0) {
  __CPROVER_assume (x < 0);
  assert (0); }
```

Assume must either be checked with assert or used as an idiom:

```
x = nondet_int ();
y = nondet_int ();
__CPROVER_assume (x < y);
```

Example: Prophecy variables

```
int x, y, v;
void main (void)
{
  v = nondet_int ();
  x = v;

  x = x + 1;
  y = nondet_int ();
  assume (v == y);

  assert (x == y + 1);
}
```

v is a *prophecy* variable

it guesses the future value of y

assume blocks executions with a wrong guess

syntactically: x is changed *before* y

semantically: x is changed *after* y

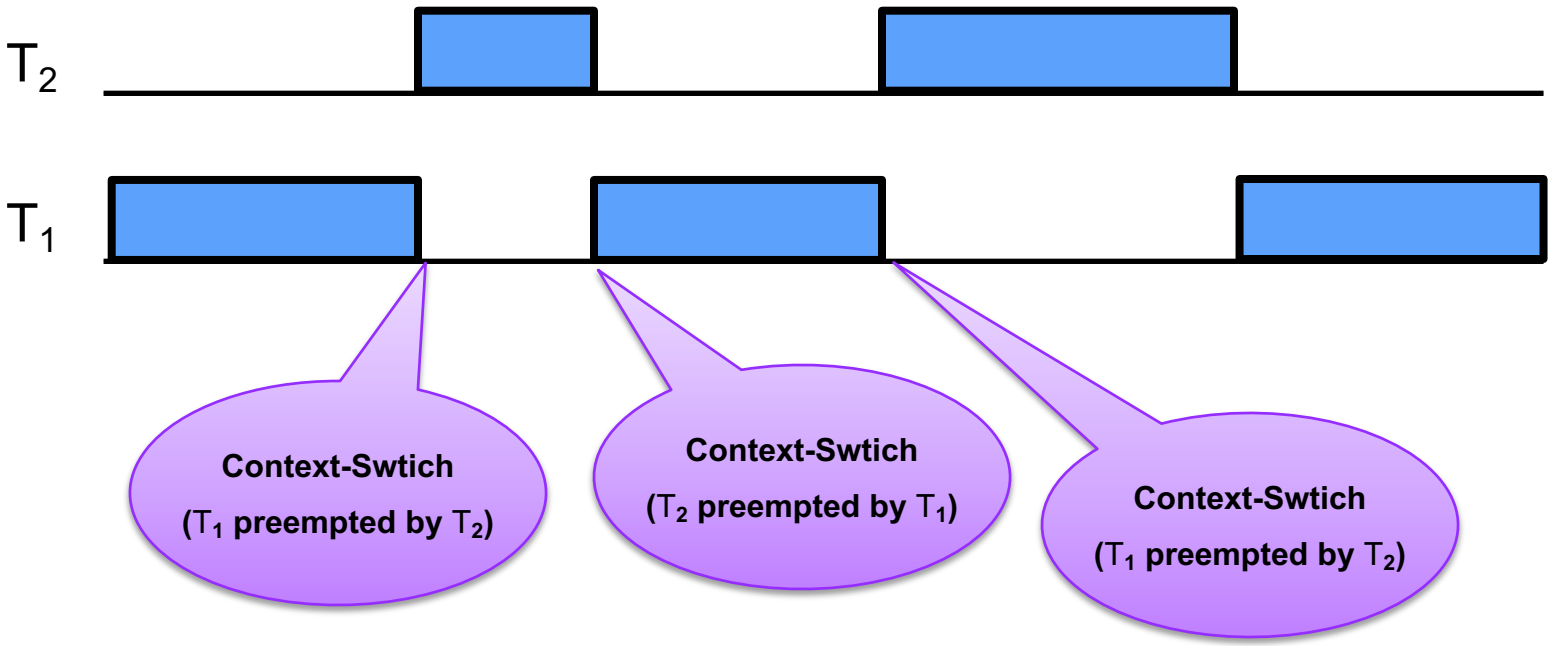
Context-Bounded Analysis with CBMC

UNIVERSITY OF WATERLOO

Akash Lal and Tom Reps. “Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis”

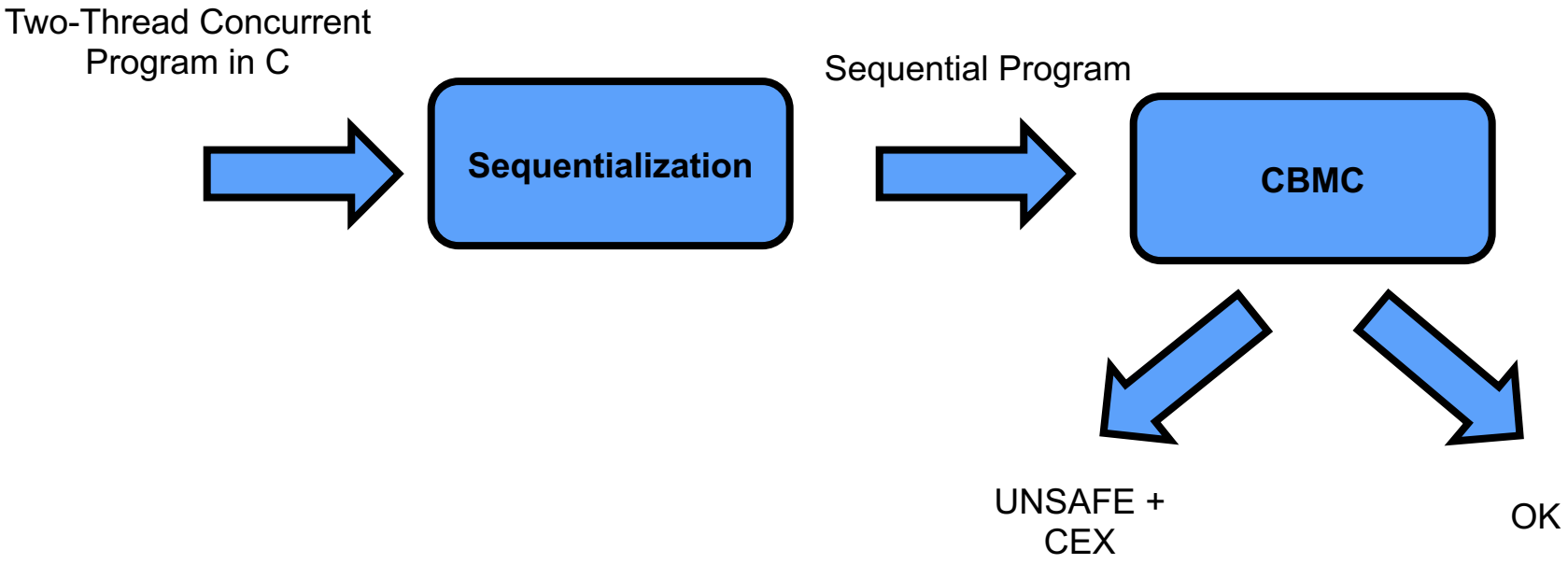
Context-Bounded Analysis (CBA)

Explore all executions of TWO threads that have at most R context-switches (per thread)



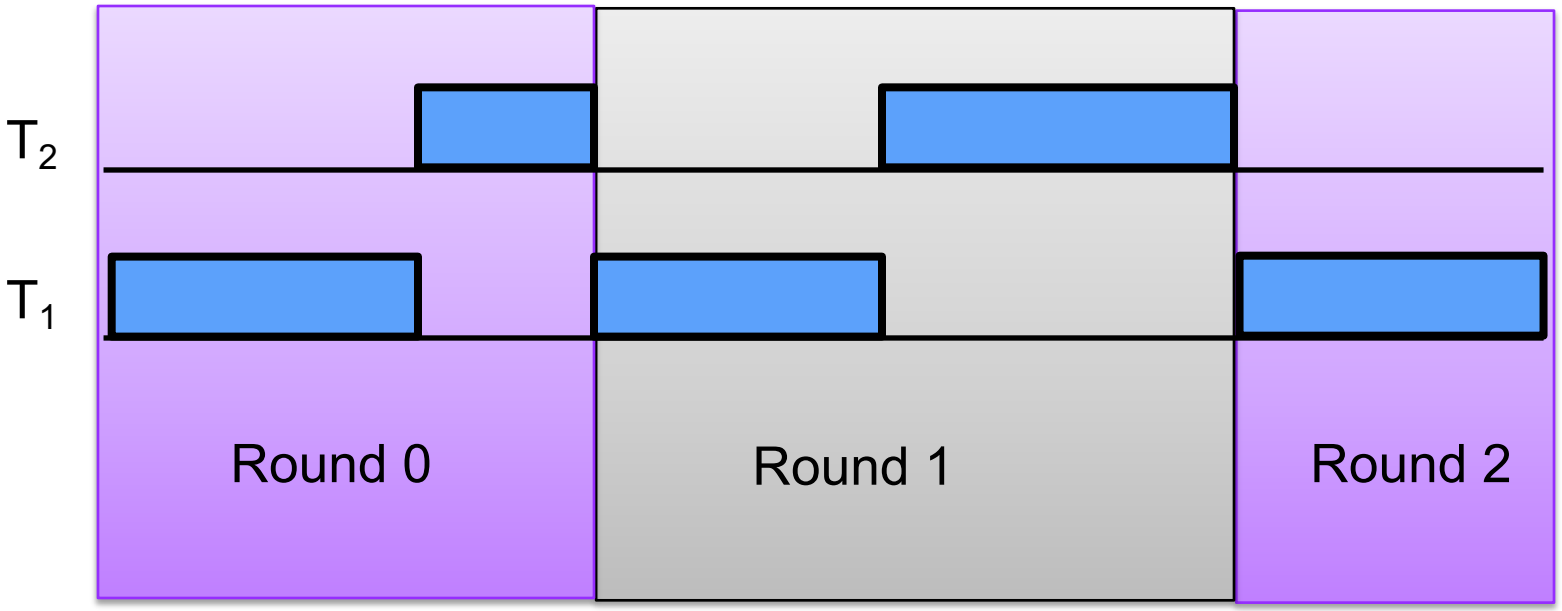
CBA via Sequentialization

1. Reduce concurrent program P to a sequential (non-deterministic) program P' such that "P has error" iff "P' has error"
2. Check P' with CBMC

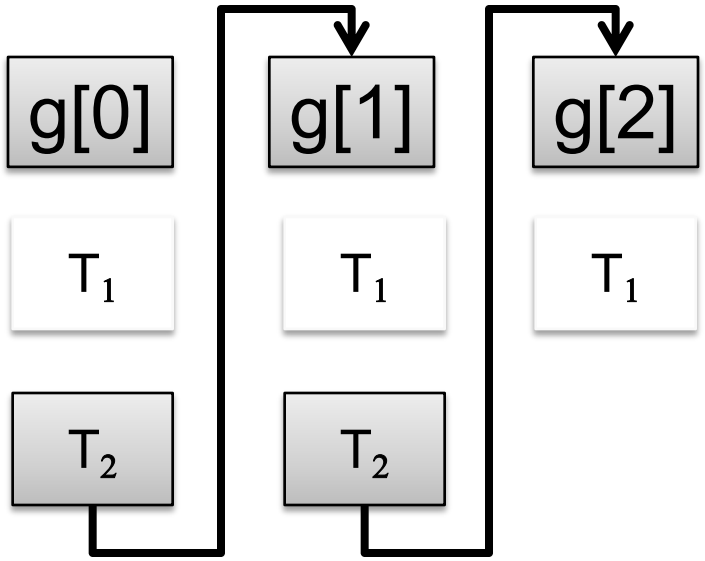


Key Idea

1. Divide execution into rounds based on context switches
2. Execute executions of each context separately, starting from a symbolic state
3. Run all parts of Thread 1 first, then all parts of Thread 2
4. Connect executions from Step 2 using assume-statements



Sequentialization in Pictures



Guess initial value of each global in each round

Execute task bodies

- T_1
- T_2

Check that initial value of round i+1 is the final value of round i

CBA Sequentialization in a Nutshel

Sequential Program for execution of R rounds (i.e., context switches):

1. for each global variable g, let g[r] be the value of g in round r
2. execute thread bodies sequentially
   - first thread 1, then thread 2
   - for global variables, use g[r] instead of g when running in round r
   - non-deterministically decide where to context switch
   - at a context switch jump to a new round (i.e., inc r)
3. check that initial value of round r+1 is the final value of round r
4. check user assertions

CBA Sequentialization


```
var
  int round;                          // current round
  int g[R], i_g[R];                   // global and initial global
  Bool saved_assert = 1;              // local assertions
```

```
void main()
  initShared();
  initGlobals();

  for t in [0,N) : // for each thread
       round = 0;
       T't();
  checkAssumptions();
  checkAssertions();
```

```
initShared()
 for each global var g, g[0] = init_value(g);
```

```
initGlobals()
  for r in [1,R): //for each round
    for each global g: g[r] = i_g[r] = nondet();
```

```
checkAssumtpions()
  for r in [0,R-1):
    for each global g:
      assume (g[r] == i_g[r+1]);
```

```
checkAssertions()
  assert (saved_assert);
```

CBA Sequentialization: Task Body 2/2

```
void T'_t ()
  Same as T_t, but each statement 'st' is replaced with:
     contextSwitch(); st[g ← g[round]];
  and 'assert(e)' is replaced with:
     saved_assert = e;
```

```
void contextSwitch()
  int oldRound;

  if (nondet()) return;  // non-det do not context switch

  oldRound = round;
  round = nondet_int();
  assume (oldRound < round <= R-1);
```

For more details, see

Akash Lal and Tom Reps. "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis",

in Proceedings of Computer Aided Verification, 2008.

Checking user-specified claims

Assert, assume, and non-determinism + Programming can be used to specify many interesting claims

Use CBMC to check that this loop has a non-terminating execution

```
int dir=1;
while (x>0) {
    x = x + dir;
    if (x>10) {dir = -1*dir;}
    if (x<5) {dir = -1*dir;}
}
```