Introduction to Software Verification

Orna Grumberg

Lectures Material
winter 2017-18

Lecture 11

- CBMC
- Efficient SAT solvers

CBMC: C Bounded Model Checker

Bounded Model Checking of C programs

Developed by Daniel Kroening

Based on slides by
Arie Gurfinkel

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 != 5,
w != 9
```

SAT

counterexample found!

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

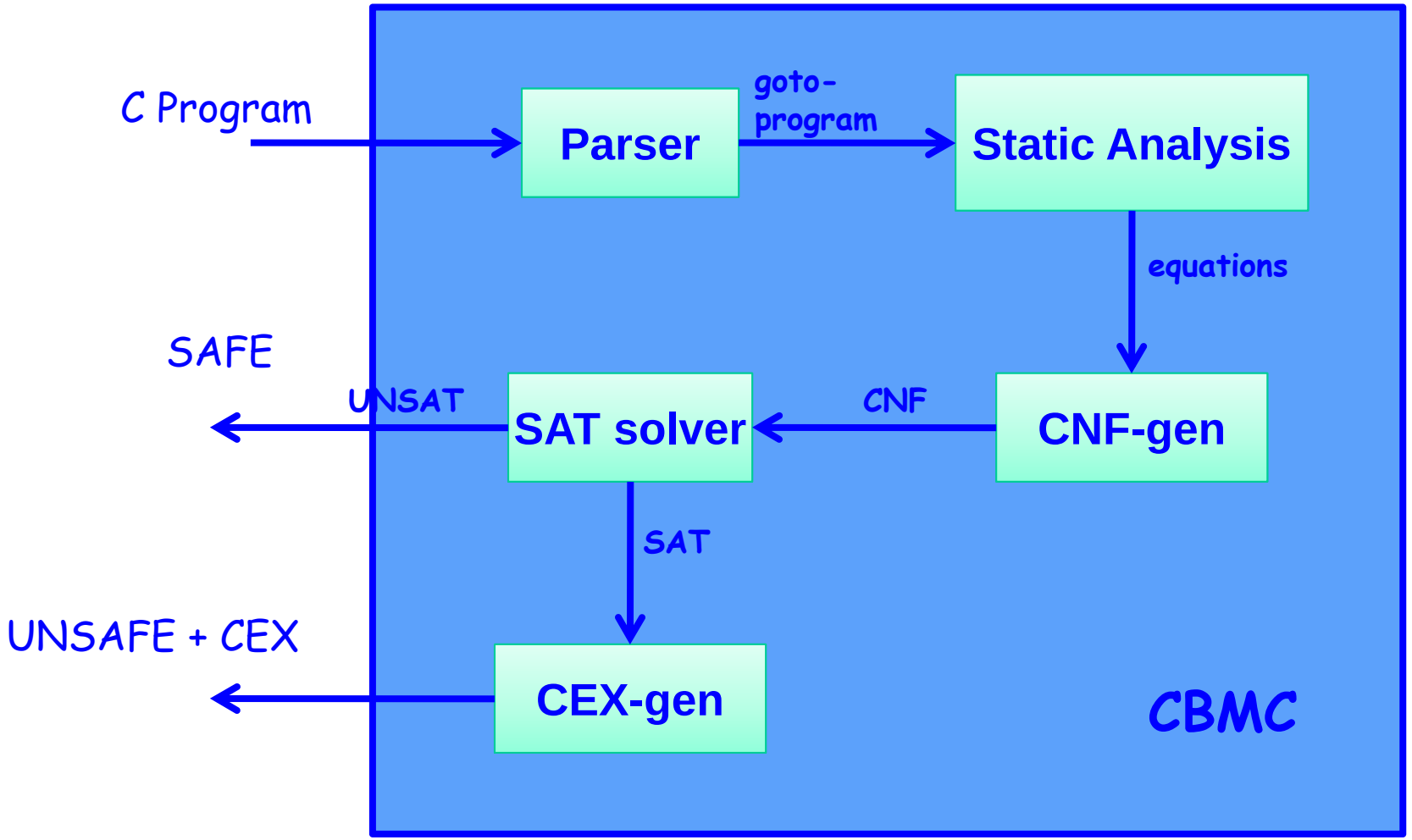
How does CBMC work

Transform a program 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



Example: Sufficient Loop Unwinding

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

unwind = 3

assert(!(j <= 2))
unwinding assertion

```
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

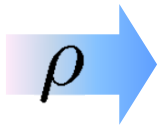
```
void f(...) {
  j = 1
  if(j <= 10) {
    j = j + 1;
    if(j <= 10) {
      j = j + 1;
      if(j <= 10) {
        j = j + 1;
        assert(!(j <= 10));
        }
      }
    }
  }
  Remainder;
}
```

Transforming Loop-Free Programs Into Equations (2)

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

Program

```
x = x+y;
x = x*2;
a[i] = 100;
```



SSA Program

$x_1 = x_0+y_0;$
$x_2 = x_1*2;$
$a_1[i_0] = 100;$

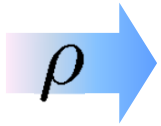
Single Static Assignment (SSA)

What about conditionals?

Program

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

w = x;
```

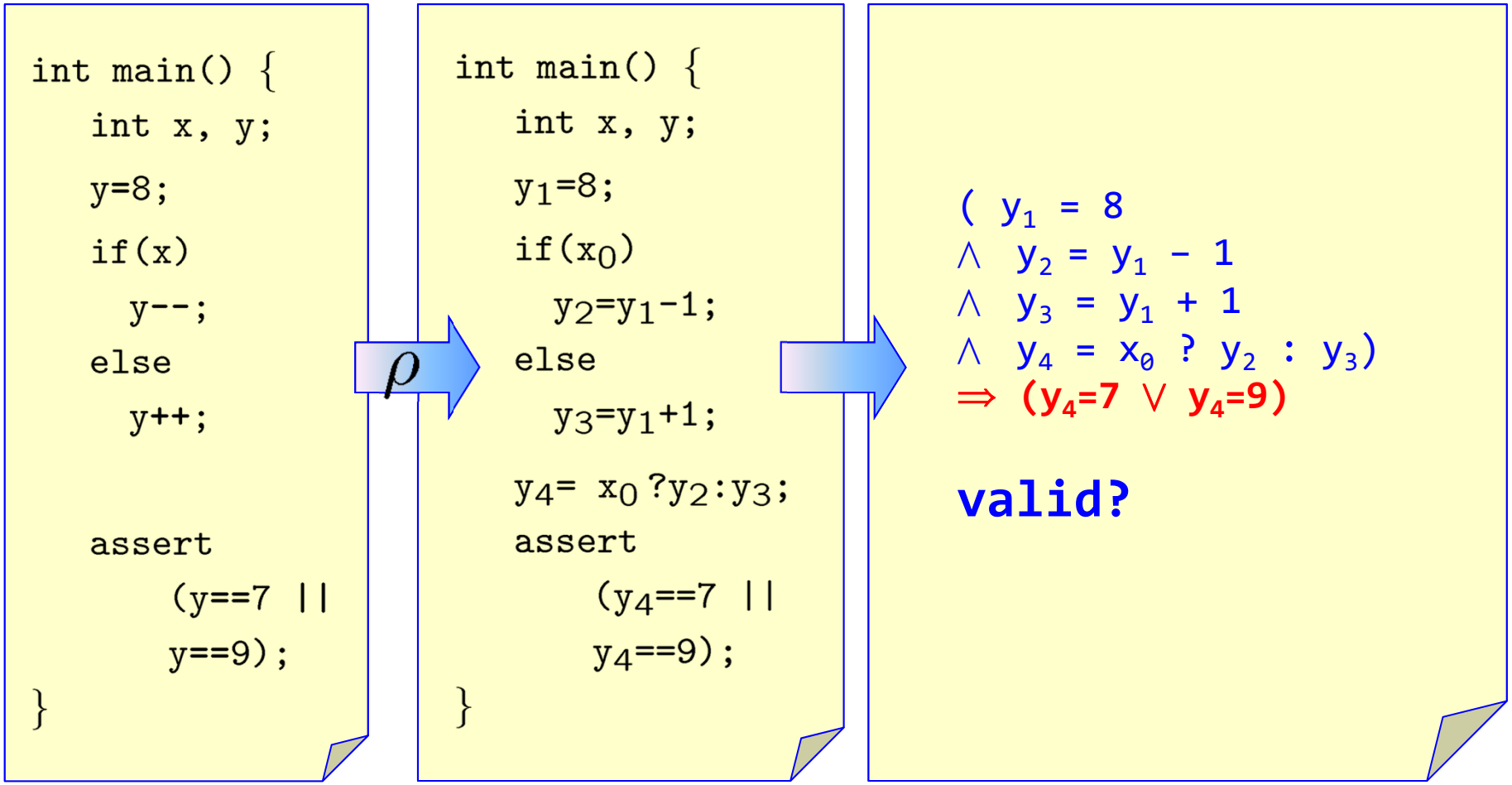


SSA Program

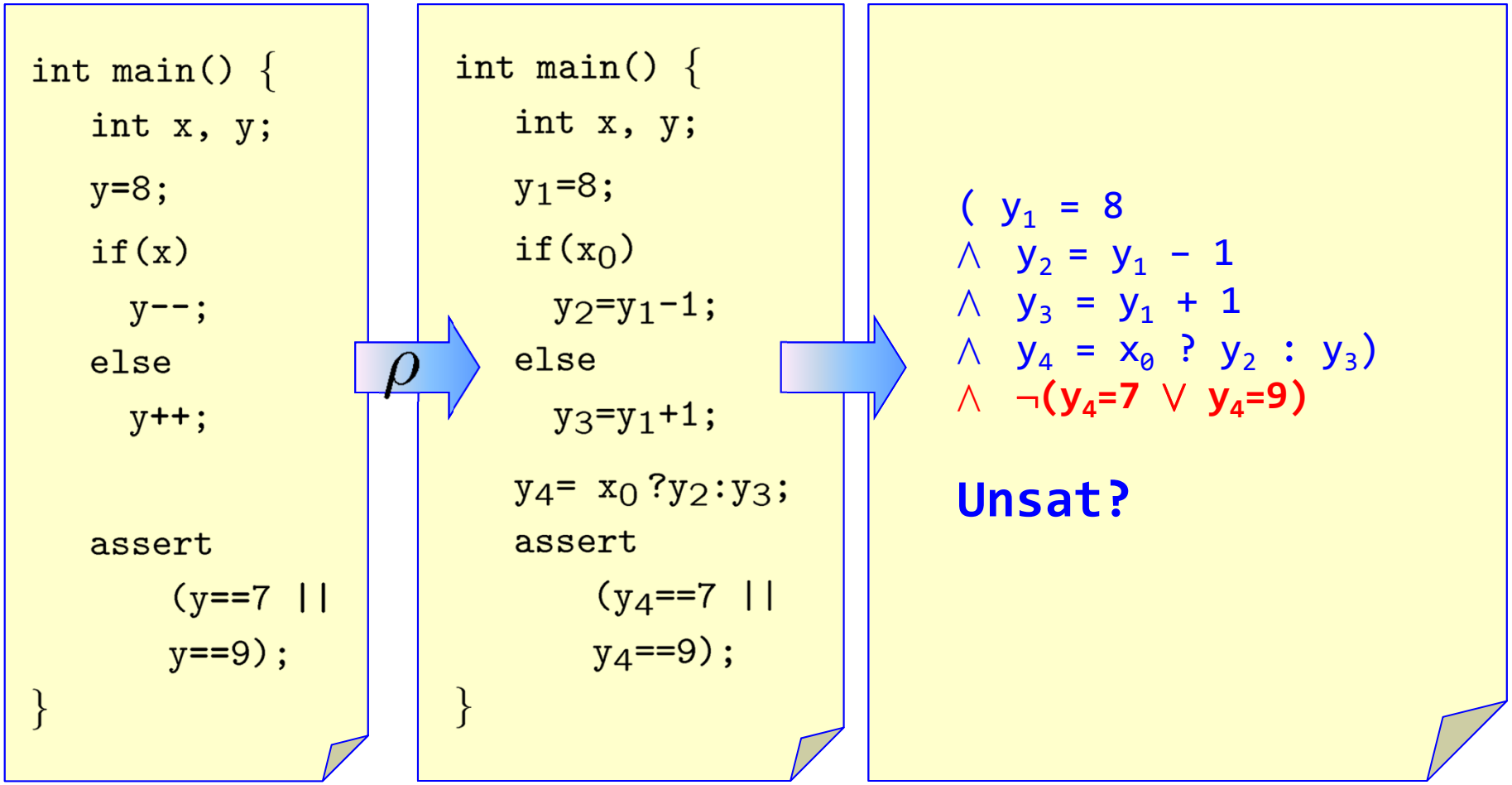
```
if (v0)
  x1 = y0;
else
  x2 = z0 ;
x3 = v0 ? x1 : x2;
w1 = x3;
```

For each join point, add new variables with selectors

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)
valid?

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)
Unsat?

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

Non-determinism

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

Assumptions

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

Assume-Guarantee Reasoning (1)

Is sort correct?

Check by splitting on the arguments of sort

```
void sort (int* p,int n) { … }
void main(void) {
  …
  sort(a1, 10);
  …
  sort(a2, 7);
  …
}
```

Assume-Guarantee Reasoning (2)

(Assume) Is sort correct assuming p is not NULL?

```
void sort (int* p, int n) {
      assume(p!=NULL);
       …    /* sort code */
      assert(sorted(p,n));
}
```

Assume-Guarantee Reasoning (2)

(Guarantee) Is `sort` guaranteed to be called with a **non-NULL argument**?

```
void main(void) {
  …
  assert (a1!=NULL);     // sort(a1,10)
  for (c = 0; c < 10; c++)
     a1[c] = nondet_int();
  assume(sorted(a1,10));
  …
  assert (a2!=NULL);     // sort(a2,7);
  for (c = 0; c < 7; c++)
     a2[c] = nondet_int();
  assume(sorted(a2,7));
  …}
```

Dangers of unrestricted assumptions

Assumptions can lead to vacuous satisfaction

This program is passed by CBMC!

```
if (x > 0) {
        assume (x < 0);
        x = -2;
        assert (x!=-2);
}
```

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

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

How does CBMC work

Transform a program 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

Efficient SAT solvers

The SAT Problem

- Given a propositional formula $\varphi(\bar{v})$, is there a satisfying *assignment* A for $\bar{v}$
- An assignment is a function from $\bar{v}$ to *{true, false}*
- A is a satisfying assignment if $\varphi(A(\bar{v}))$= *true*
- A is called a *solution* for $\varphi(\bar{v})$
- A partial assignment assigns a subset of $\bar{v}$

CNF representation of $\varphi(\vec{v})$

- $\varphi(\vec{v})$ is a conjunction of clauses: $\varphi(\vec{v}) = cl_1 \wedge cl_2 \wedge \ldots \wedge cl_n$
- A clause is a disjunction of literals: $cl_i = (lit_1 \vee \ldots \vee lit_l)$
- A literal is an atomic proposition or its negation

- **Example:**
  $(a \vee c) \wedge (b \vee c) \wedge (\neg a \vee \neg b \vee d)$

- A satisfies $\varphi(\vec{v})$ iff A satisfies all its clauses

- **Example:**
  $(a \vee c) \wedge (b \vee c) \wedge (\neg a \vee \neg b \vee d)$

- Satisfying assignments:
  - A_1 = (a=true, b=true, d=true, c=false)
  - A_2 = (c=true, a=false, b=true, d=false)

- CNF formulas
  - Clause does not contain a, $\neg a$
  - No repetition of literals in clause

- CNF formulas can be represented as a set of sets of literals

Searching for a satisfying assignment

- Inefficient way:
  check each one of the 2^n assignments where $|\vec{v}| = n$

- The basis for efficient SAT solving:
  Davis, Putnam,Logeman, Loveland (DPLL)
  1960, 1962

First idea: Unit Clause

Given

- a propositional formula $\varphi(\vec{v})$, and
- a partial assignment *A*

A Unit Clause is a clause with

  - exactly one unassigned literal, while
  - all other literals are false

- *Asserts* the value of the unassigned variable

$$\begin{array}{l} a = ? \\ b = 1 \\ d = 0 \end{array} \qquad cl = (a \vee \neg b \vee d) \qquad \Rightarrow \qquad a = 1$$

- a=1 is implied by b=1 and d=0

Boolean Constraint Propagation (BCP)

- BCP: For $\varphi(\vec{v})$ and a partial assignment A, computes all possible implications
  - Based on unit clauses
- Conflict: a variable gets both 0 and 1 under A

DPLL algorithm

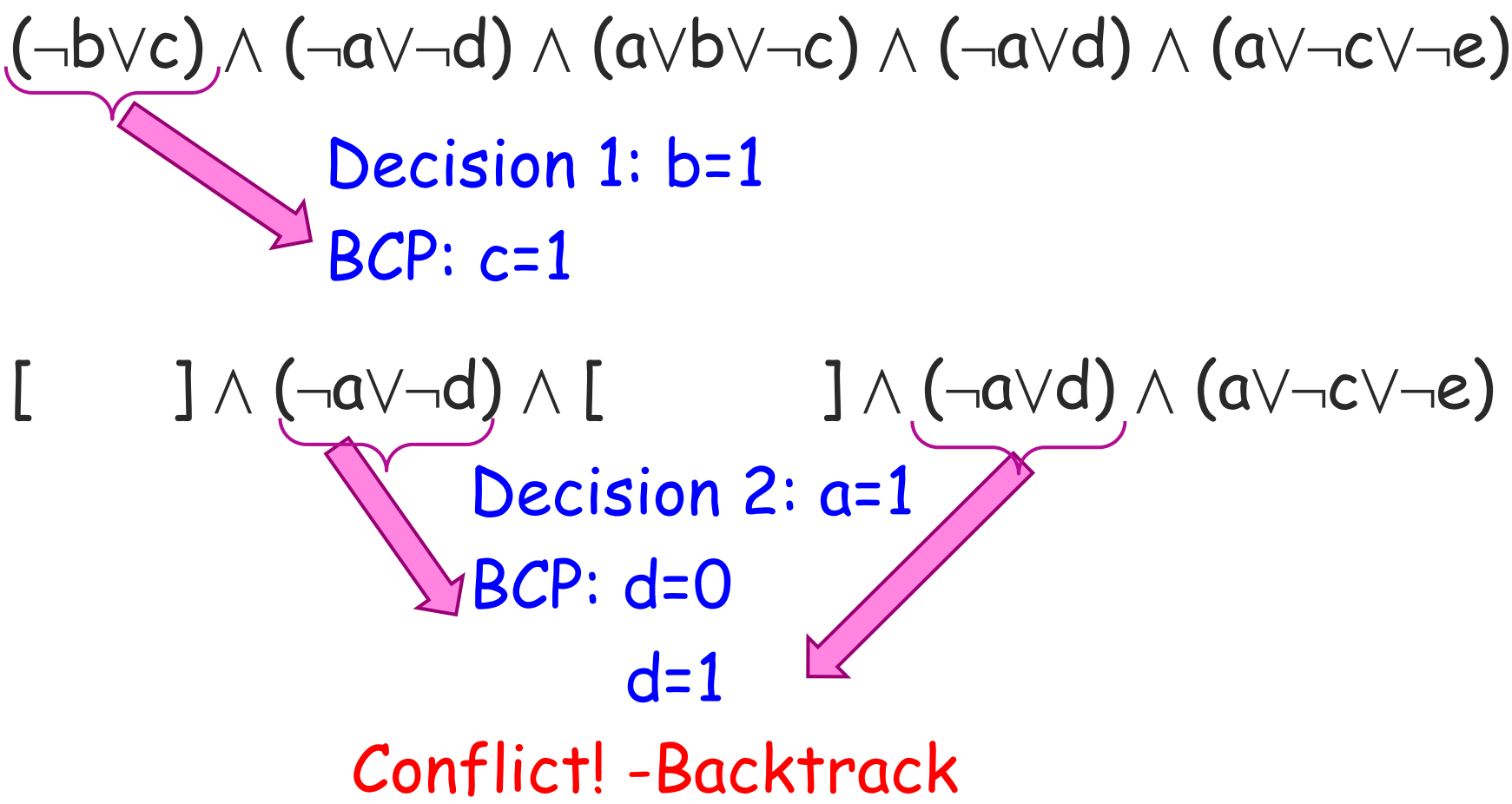
1. Start with an empty partial assignment A
2. If A is complete (no new decision to make), return (SAT, A)
3. Otherwise, extend A with a decision: D=(variable, value)
4. BCP: extend A with all implications of D
5. If (no conflict) go to 2
6. If (conflict), apply backtracking:

   Let D=(v,b) be the last decision s.t. (v, !b) has not been checked yet

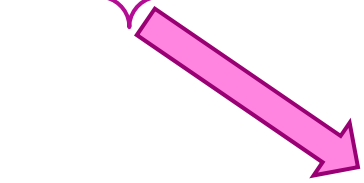
   - Flip decision D: Remove (v,b), extend A with (v, !b)
   - Undo all implications from D=(v,b) and from flips made after D
   - Return to 4
7. If no decision to flip, return UNSAT

DPLL algorithm

- Termination
  - No unassigned variable – SAT
  - No decision variable to flip – UNSAT


(¬b∨c) ∧ (¬a∨¬d) ∧ (a∨b∨¬c) ∧ (¬a∨d) ∧ (a∨¬c∨¬e)
Decision 1: b=1
BCP: c=1
[ ] ∧ (¬a∨¬d) ∧ [ ] ∧ (¬a∨d) ∧ (a∨¬c∨¬e)
Decision 2: a=1
BCP: d=0
d=1
Conflict! -Backtrack

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



Decision 1: b=1

BCP: c=1

$[\quad\quad] \wedge (\neg a \vee \neg d) \wedge [\quad\quad\quad\quad] \wedge (\neg a \vee d) \wedge (a \vee \neg c \vee \neg e)$

Decision 2: a=0

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

Decision 1: b=1

BCP: c=1

[ ] ∧ [ ] ∧ [ ] ∧ [ ] ∧ $(a \vee \neg c \vee \neg e)$

Decision 2: a=0

BCP: e=0

Partial satisfying assignment:
b=1,c=1,a=0,e=0

Other solutions to the state-explosion problem

Small models replace the full, concrete model:

- Abstraction
- Compositional verification
- Partial order reduction
- Symmetry