Introduction to Formal Methods
Chapter 09: SAT-Based Model Checking

Roberto Sebastiani

DISI, Università di Trento, Italy – `roberto.sebastiani@unitn.it`
URL: `http://disi.unitn.it/rseba/DIDATTICA/fm2020/`
Teaching assistant: Enrico Magnago – `enrico.magnago@unitn.it`

CDLM in Informatica, academic year 2019-2020

last update: Monday 18th May, 2020, 14:49

Outline

1. Background on SAT Solving
2. SAT-based Model Checking: Generalities
3. Bounded Model Checking: Intuitions
4. Bounded Model Checking: General Encoding
5. Bounded Model Checking: Relevant Subcases
6. Bounded Model Checking: An Example
7. Computing upper bounds for *k*
8. Inductive reasoning on invariants (aka “K-Induction”)
9. K-Induction: An Example
10. Exercises

Resolution

- Search for a refutation of φ
- φ is represented as a set of clauses
- Applies iteratively the resolution rule to pairs of clauses containing a conflicting literal, until a false clause is generated or the resolution rule is no more applicable
- Many different strategies

Resolution Rule

- Resolution of two clauses with exactly one incompatible literal:

$$\frac{(\overbrace{l_1 \vee \ldots \vee l_k}^{common} \vee \overbrace{l}^{resolvent} \vee \overbrace{l'_{k+1} \vee \ldots \vee l'_m}^{C'}) \qquad (\overbrace{l_1 \vee \ldots \vee l_k}^{common} \vee \overbrace{\neg l}^{resolvent} \vee \overbrace{l''_{k+1} \vee \ldots \vee l''_n}^{C''})}{(\underbrace{l_1 \vee \ldots \vee l_k}_{common} \vee \underbrace{l'_{k+1} \vee \ldots \vee l'_m}_{C'} \vee \underbrace{l''_{k+1} \vee \ldots \vee l''_n}_{C''})}$$

- EXAMPLE:

$$\frac{(A \vee B \vee C \vee D \vee E) \qquad (A \vee B \vee \neg C \vee F)}{(A \vee B \vee D \vee E \vee F)}$$

- NOTE: many standard inference rules subcases of resolution:

$$\frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C}\ (Transit.) \qquad \frac{A \quad A \rightarrow B}{B}\ (M.\ Ponens) \qquad \frac{\neg B \quad A \rightarrow B}{\neg A}\ (M.\ Tollens)$$

Resolution Rules: unit propagation

- Unit resolution:

$$\frac{\Gamma' \wedge (l) \wedge (\neg l \vee \bigvee_i l_i)}{\Gamma' \wedge (l) \wedge (\bigvee_i l_i)}$$

- Unit subsumption:

$$\frac{\Gamma' \wedge (l) \wedge (l \vee \bigvee_i l_i)}{\Gamma' \wedge (l)}$$

- Unit propagation = unit resolution + unit subsumption

"Deterministic" rule: applied before other "non-deterministic" rules!

DPLL

- Davis-Putnam-Longeman-Loveland procedure (DPLL)
- Tries to build recursively an assignment μ satisfying φ;
- At each recursive step assigns a truth value to (all instances of) one atom.
- Performs deterministic choices first.

The DPLL Algorithm

```
function DPLL($\varphi, \mu$)
    if $\varphi = \top$                                  /* base        */
        then return True;
    if $\varphi = \bot$                                  /* backtrack */
        then return False;
    if {a unit clause ($l$) occurs in $\varphi$}         /* unit propagation   */
        then return DPLL(assign($l, \varphi$), $\mu \wedge l$);
    (...)
    $l$ := choose-literal($\varphi$);                    /* split        */
    return   DPLL(assign($l, \varphi$), $\mu \wedge l$)   or
             DPLL(assign($\neg l, \varphi$), $\mu \wedge \neg l$);
```

“Classic” chronological backtracking

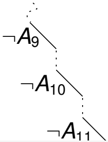
Non-recursive versions of DPLL:

- variable assignments (literals) stored in a stack
- each variable assignments labeled as “unit”, “open”, “closed”
- when a conflict is encountered, the stack is popped up to the most recent open assignment l
- l is toggled, is labeled as “closed”, and the search proceeds.

Perform “classic” chronological backtracking:
jump back to the most-recent open branching point
$\Longrightarrow$ source of large inefficiencies

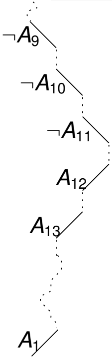
Classic chronological backtracking – example

$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$



Classic chronological backtracking – example

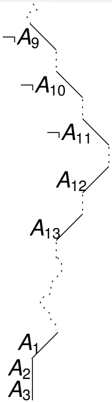
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓
$c_8 : A_1 \vee A_8$ ✓
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, ..., A_1\}$
... (branch on A_1)

Classic chronological backtracking – example

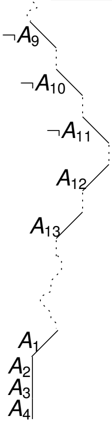
$c_1 : \neg A_1 \lor A_2$ ✓
$c_2 : \neg A_1 \lor A_3 \lor A_9$ ✓
$c_3 : \neg A_2 \lor \neg A_3 \lor A_4$
$c_4 : \neg A_4 \lor A_5 \lor A_{10}$
$c_5 : \neg A_4 \lor A_6 \lor A_{11}$
$c_6 : \neg A_5 \lor \neg A_6$
$c_7 : A_1 \lor A_7 \lor \neg A_{12}$ ✓
$c_8 : A_1 \lor A_8$ ✓
$c_9 : \neg A_7 \lor \neg A_8 \lor \neg A_{13}$
...



$\{..., \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, ..., A_1, A_2, A_3\}$
(unit A_2, A_3)

Classic chronological backtracking – example

$c_1 : \neg A_1 \vee A_2$ ✓
$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$ ✓
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓
$c_8 : A_1 \vee A_8$ ✓
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, ..., A_1, A_2, A_3, A_4\}$
(unit A_4)

Classic chronological backtracking – example

$c_1 : \neg A_1 \lor A_2$ ✓

$c_2 : \neg A_1 \lor A_3 \lor A_9$ ✓

$c_3 : \neg A_2 \lor \neg A_3 \lor A_4$ ✓

$c_4 : \neg A_4 \lor A_5 \lor A_{10}$ ✓

$c_5 : \neg A_4 \lor A_6 \lor A_{11}$ ✓

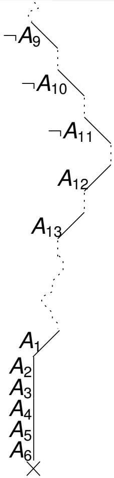
$c_6 : \neg A_5 \lor \neg A_6$ ×

$c_7 : A_1 \lor A_7 \lor \neg A_{12}$ ✓

$c_8 : A_1 \lor A_8$ ✓

$c_9 : \neg A_7 \lor \neg A_8 \lor \neg A_{13}$

...

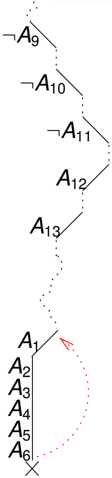


$\{..., \neg A_9, \neg A_{10}, \neg A_{1\neg A_4 1}, A_{12}, A_{13}, ..., A_1, A_2, A_3, A_4, A_5, A_6\}$

(unit A_5, A_6)$\Longrightarrow$ conflict

Classic chronological backtracking – example

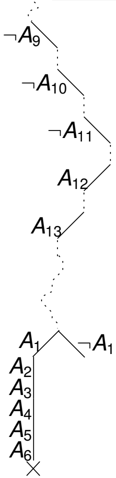
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, ...\}$
$\Longrightarrow$ backtrack up to A_1

Classic chronological backtracking – example

$c_1 : \neg A_1 \vee A_2$ ✓
$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, ..., \neg A_1\}$
(unit $\neg A_1$)

Classic chronological backtracking – example

$c_1 : \neg A_1 \vee A_2$ $\surd$

$c_2 : \neg A_1 \vee A_3 \vee A_9$ $\surd$

$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$

$c_4 : \neg A_4 \vee A_5 \vee A_{10}$

$c_5 : \neg A_4 \vee A_6 \vee A_{11}$

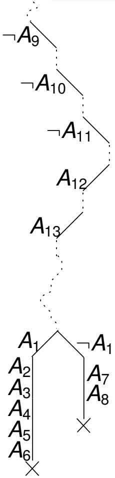
$c_6 : \neg A_5 \vee \neg A_6$

$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ $\surd$

$c_8 : A_1 \vee A_8$ $\surd$

$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$ ×

...

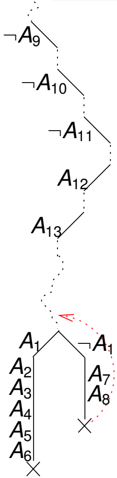


$\{..., \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, ..., \neg A_1, A_7, A_8\}$

(unit A_7, A_8) $\Longrightarrow$ conflict

Classic chronological backtracking – example

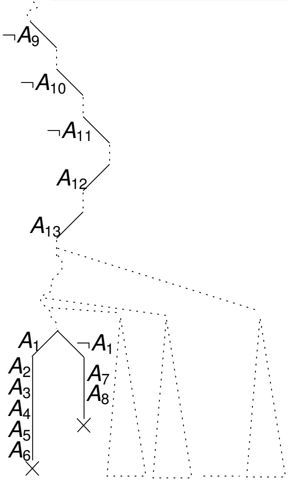
$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, ...\}$
$\Longrightarrow$ backtrack to the most recent open branching point

Classic chronological backtracking – example

$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
...



$\{..., \neg A_9, \neg A_{10}, \neg A_{11}, A_{12}, A_{13}, ...\}$
$\Longrightarrow$ lots of useless search before backtracking up to A_{13}!

Classic chronological backtracking: drawbacks

- often the branch heuristic delays the “right” choice
- chronological backtracking always backtracks to the most recent branching point, even though a higher backtrack could be possible $\Longrightarrow$ lots of useless search!

Modern DPLL implementations [Silva & Sakallah ’96, Moskewicz et al. ’01]

Conflict-Driven Clause-Learning (CDCL) DPLL solvers:

- Non-recursive: stack-based representation of data structures
- Efficient data structures for doing and undoing assignments
- Perform conflict-driven backtracking (backjumping) and learning
- May perform search restarts
- Reason on total assignments

Dramatically efficient: solve industrial-derived problems with $\approx 10^7$ Boolean variables and $\approx 10^7 - 10^8$ clauses

Conflict-directed backtracking (backjumping) and learning

- Idea: when a branch μ fails,
  (i) conflict analysis: reveal the sub-assignment $\eta \subseteq \mu$ causing the failure (conflict set η):
    - find $\eta \subseteq \mu$ by generating the conflict clause $C \stackrel{\text{def}}{=} \neg\eta$ via resolution from the falsified clause
    - by construction $\varphi \wedge \eta \models \bot$, hence $\varphi \models C$, so that $(\varphi \wedge C) \Leftrightarrow \varphi$

  (ii) learning: add the conflict clause C to the clause set

  (iii) backjumping: backtrack to the highest branching point s.t. the stack contains all-but-one literals in η, and then unit-propagate the unassigned literal on C
- may jump back up much more than one decision level in the stack $\Longrightarrow$ may avoid lots of redundant search!!.

State-of-the-art backjumping and learning: intuitions

- Conflict analysis: find $\eta \subset \mu$ (typically much smaller than μ!) s.t. assigning only the literals in η would have falsified the same clause after a chain of unit propagations
  - intuition: "η contains only the relevant assignments which caused the failure"
- Backjumping: climb up to many decision levels in the stack
  - intuition: "go back to the oldest decision where you'd have done something different if only you had known η"

  $\Longrightarrow$ may avoid lots of redundant search

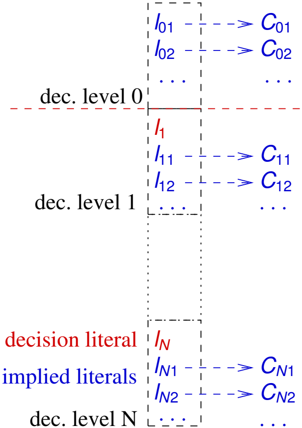
  $\Longrightarrow$ choose η s.t. all but one literals in η are as "old" as possible
- Learning: in future branches, when all-but-one literals in η are assigned, the remaining literal is assigned to false by unit-propagation:
  - intuition: "when you're about to repeat the mistake, do the opposite of the last step"

  $\Longrightarrow$ avoid finding the same conflict again

Stack-based representation of a truth assignment μ

- stack partitioned into decision levels:
  - one decision literal
  - its implied literals
  - each implied literal tagged with the clause causing its unit-propagation (antecedent clause)
- equivalent to an implication graph:
  - a node without incoming edges represent a decision literal
  - the graph contains $l_1 \xmapsto{c} l$,...,$l_n \xmapsto{c} l$ iff $c \overset{\text{def}}{=} \bigvee_{j=1}^{n} \neg l_i \vee l$ is the antecedent clause of l

representation of the dependencies between literals in μ



l_{01} ---> C_{01}

l_{02} ---> C_{02}

... ...

dec. level 0

l_1

l_{11} ---> C_{11}

l_{12} ---> C_{12}

dec. level 1 ... ...

decision literal l_N

implied literals l_{N1} ---> C_{N1}

l_{N2} ---> C_{N2}

dec. level N ... ...

Implication graph - example

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$ ✓

$c_4 : \neg A_4 \vee A_5 \vee A_{10}$ ✓

$c_5 : \neg A_4 \vee A_6 \vee A_{11}$ ✓

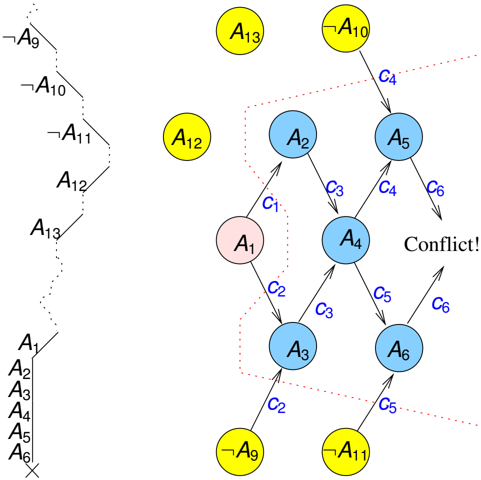
$c_6 : \neg A_5 \vee \neg A_6$ ×

$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

$c_8 : A_1 \vee A_8$ ✓

$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$

...



Building a conflict set/clause by resolution

1. $C :=$ conflicting clause
2. repeat
   (i) resolve current clause C with the antecedent clause of the last unit-propagated literal l in C

   until C verifies some given termination criteria

Idea: “Undo” unit-propagations.

Decision strategy: repeat until C contains only decision literals

$$\dfrac{\neg A_1 \vee A_2 \qquad \dfrac{\neg A_1 \vee A_3 \vee A_9 \qquad \dfrac{\neg A_2 \vee \neg A_3 \vee A_4 \qquad \dfrac{\neg A_4 \vee A_5 \vee A_{10} \qquad \dfrac{\neg A_4 \vee A_6 \vee A_{11} \qquad \overbrace{\neg A_5 \vee \neg A_6}^{\textit{Conflicting cl.}}}{\neg A_4 \vee \neg A_5 \vee A_{11}}\;(A_6)}{\neg A_4 \vee A_{10} \vee A_{11}}\;(A_5)}{\neg A_2 \vee \neg A_3 \vee A_{10} \vee A_{11}}\;(A_4)}{\neg A_2 \vee \neg A_1 \vee A_9 \vee A_{10} \vee A_{11}}\;(A_3)}{\neg A_1 \vee A_9 \vee A_{10} \vee A_{11}}\;(A_2)$$

State-of-the-art in backjumping & learning

First Unique Implication Point (1st UIP) strategy:

- corresponds to consider the first clause encountered containing one literal of the current decision level (1st UIP).

$$\dfrac{\neg A_4 \vee A_5 \vee A_{10} \qquad \dfrac{\neg A_4 \vee A_6 \vee A_{11} \qquad \overbrace{\neg A_5 \vee \neg A_6}^{Conflicting\ cl.}}{\neg A_4 \vee \neg A_5 \vee A_{11}}\ (A_6)}{\underbrace{\neg A_4}_{1st\ UIP} \vee A_{10} \vee A_{11}}\ (A_5)$$

1st UIP strategy – example

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$ ✓

$c_4 : \neg A_4 \vee A_5 \vee A_{10}$ ✓

$c_5 : \neg A_4 \vee A_6 \vee A_{11}$ ✓

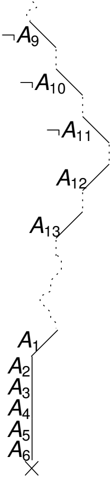
$c_6 : \neg A_5 \vee \neg A_6$ ×

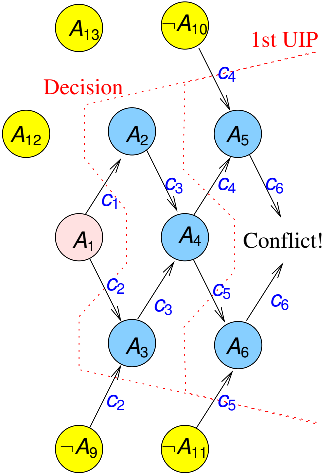
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

$c_8 : A_1 \vee A_8$ ✓

$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$

...





$\Longrightarrow$ Conflict set: $\{\neg A_{10}, \neg A_{11}, A_4\}$, learn $c_{10} := A_{10} \vee A_{11} \vee \neg A_4$

1st UIP strategy and backjumping

- The added conflict clause states the reason for the conflict
- The procedure backtracks to the most recent decision level of the variables in the conflict clause which are not the UIP.
- then the conflict clause forces the negation of the UIP by unit propagation.

E.g.: $c_{10} := A_{10} \vee A_{11} \vee \neg A_4$
$\Longrightarrow$ backtrack to A_{11}, then assign $\neg A_4$

1st UIP strategy – example (7)

$c_1 : \neg A_1 \vee A_2$ ✓

$c_2 : \neg A_1 \vee A_3 \vee A_9$ ✓

$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$ ✓

$c_4 : \neg A_4 \vee A_5 \vee A_{10}$ ✓

$c_5 : \neg A_4 \vee A_6 \vee A_{11}$ ✓

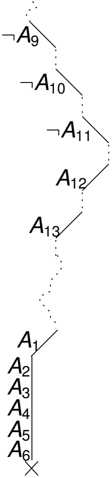
$c_6 : \neg A_5 \vee \neg A_6$ ✗

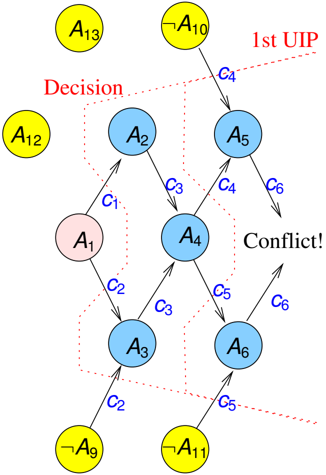
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$ ✓

$c_8 : A_1 \vee A_8$ ✓

$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$

...

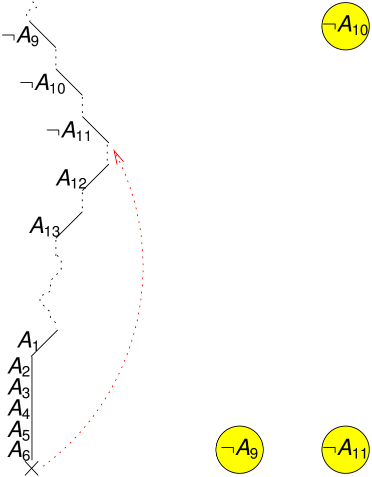




$\Longrightarrow$ Conflict set: $\{\neg A_{10}, \neg A_{11}, A_4\}$, learn $c_{10} := A_{10} \vee A_{11} \vee \neg A_4$

1st UIP strategy – example (8)

$c_1 : \neg A_1 \vee A_2$
$c_2 : \neg A_1 \vee A_3 \vee A_9$
$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$
$c_4 : \neg A_4 \vee A_5 \vee A_{10}$
$c_5 : \neg A_4 \vee A_6 \vee A_{11}$
$c_6 : \neg A_5 \vee \neg A_6$
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$
$c_8 : A_1 \vee A_8$
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$
$c_{10} : A_{10} \vee A_{11} \vee \neg A_4$
...



$\Longrightarrow$ backtrack up to A_{11} $\Longrightarrow$ $\{ ..., \neg A_9, \neg A_{10}, \neg A_{11} \}$

1st UIP strategy – example (9)

$c_1 : \neg A_1 \vee A_2$

$c_2 : \neg A_1 \vee A_3 \vee A_9$

$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$

$c_4 : \neg A_4 \vee A_5 \vee A_{10}$ ✓

$c_5 : \neg A_4 \vee A_6 \vee A_{11}$ ✓

$c_6 : \neg A_5 \vee \neg A_6$

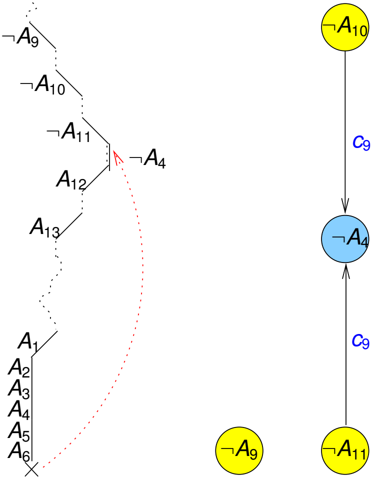
$c_7 : A_1 \vee A_7 \vee \neg A_{12}$

$c_8 : A_1 \vee A_8$

$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$

$c_{10} : A_{10} \vee A_{11} \vee \neg A_4$ ✓

...



$\Longrightarrow$ unit propagate $\neg A_4$ $\Longrightarrow$ $\{..., \neg A_9, \neg A_{10}, \neg A_{11}, A_4\}$...

Learning – example

$c_1 : \neg A_1 \vee A_2$

$c_2 : \neg A_1 \vee A_3 \vee A_9$

$c_3 : \neg A_2 \vee \neg A_3 \vee A_4$

$c_4 : \neg A_4 \vee A_5 \vee A_{10}$

$c_5 : \neg A_4 \vee A_6 \vee A_{11}$

$c_6 : \neg A_5 \vee \neg A_6$

$c_7 : A_1 \vee A_7 \vee \neg A_{12}$

$c_8 : A_1 \vee A_8$

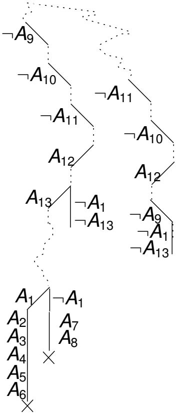
$c_9 : \neg A_7 \vee \neg A_8 \vee \neg A_{13}$ ✓

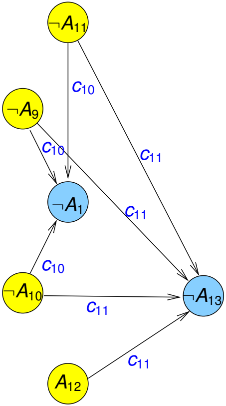
$c_{10} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_1$ ✓

$c_{11} : A_9 \vee A_{10} \vee A_{11} \vee \neg A_{12} \vee \neg A_{13}$ ✓

...

$\Longrightarrow$ Unit: $\{\neg A_1, \neg A_{13}\}$





Remark: the “quality” of conflict sets

- Different ideas of “good” conflict set
    - Backjumping: if causes the highest backjump (“local” role)
    - Learning: if causes the maximum pruning (“global” role)
- Many different strategies implemented

Drawbacks of Learning

- Prunes drastically the search.
- Problem: may cause a blowup in space

⟹ techniques to drop learned clauses when necessary
  - according to their size
  - according to their activity.

Definition

A clause is currently active if it occurs in the current implication graph (i.e., it is the antecedent clause of a literal in the current assignment).

Property

In order to guarantee correctness, completeness & termination of a CDCL solver, it suffices to keep each clause until it is active.
⟹ CDCL solvers require polynomial space

Incremental SAT solving [Een & Sorenson'03]

- Many CDCL solvers provide a stack-based incremental interface
  - it is possible to push/pop ϕ_i into a stack of formulas $\Phi \stackrel{\text{def}}{=} \{\phi_1, ..., \phi_k\}$
  - check incrementally the satisfiability of $\bigwedge_{i=1}^{k} \phi_i$.
- Maintains the status of the search from one call to the other
  - in particular it records the learned clauses (plus other information) keeping track efficiently of their dependencies on the ϕ_i's

$\Longrightarrow$ reuses search from one call to another

- Essential in many applications (in particular in FV)

Many applications of SAT Solvers

- Many successful applications of SAT:
  - Boolean circuits
  - (Bounded) Planning
  - (Bounded) Model Checking
  - Cryptography
  - Scheduling
  - ...
- All NP-complete problem can be (polynomially) converted to SAT.
- Key issue: find an efficient encoding.

SAT-based Model Checking

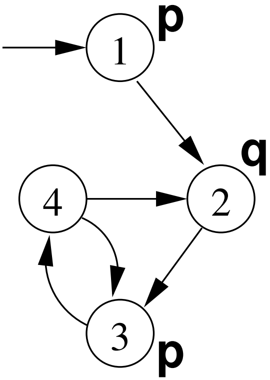
- Key problems with BDD's:
  - they can explode in space
  - an expert user can make the difference (e.g. reordering, algorithms)
- A possible alternative:
  - Propositional Satisfiability Checking (SAT)
  - SAT technology is very advanced
- Advantages:
  - reduced memory requirements
  - limited sensitivity: one good setting, does not require expert users
  - much higher capacity (more variables) than BDD based techniques
- Various techniques: Bounded Model Checking, K-induction, Interpolant-based, IC3/PDR,...

SAT-based Bounded Model Checking & K-Induction

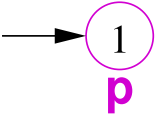
Key Ideas:

- BMC: look for counter-example paths of increasing length k
  $\Longrightarrow$ oriented to finding bugs
- K-Induction: look for an induction proofs of increasing length k
  $\Longrightarrow$ oriented to prove correctness
- BMC [resp. K-induction]: for each k, build a Boolean formula that is satisfiable [resp. unsatisfiable] iff there is a counter-example [resp. proof] of length k
  - can be expressed using $k \cdot |\mathbf{s}|$ variables
  - formula construction is not subject to state explosion
- satisfiability of the Boolean formulas is checked using a SAT solver
  - can manage complex formulae on several 100K variables
  - returns satisfying assignment (i.e., a counter-example)

Bounded Model Checking: Example

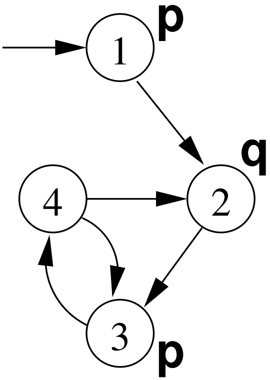


- LTL Formula: $\mathbf{G}(p \rightarrow \mathbf{F}q)$
- Negated Formula (violation): $\mathbf{F}(p \wedge \mathbf{G}\neg q)$
- $k = 0$:

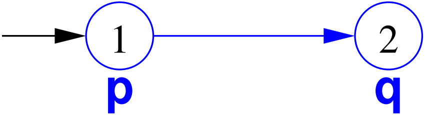


- No counter-example found.

Bounded Model Checking: Example

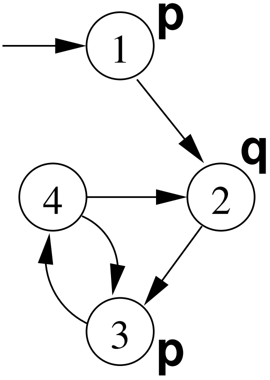


- LTL Formula: $\mathbf{G}(p \rightarrow \mathbf{F}q)$
- Negated Formula (violation): $\mathbf{F}(p \wedge \mathbf{G}\neg q)$
- $k = 1$:

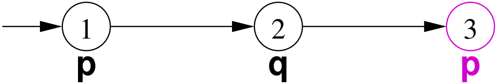


- No counter-example found.

Bounded Model Checking: Example

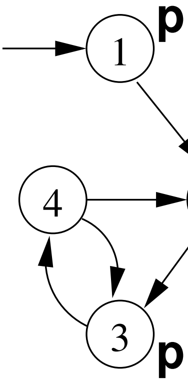


- LTL Formula: $\mathbf{G}(p \rightarrow \mathbf{F}q)$
- Negated Formula (violation): $\mathbf{F}(p \wedge \mathbf{G}\neg q)$
- $k = 2$:

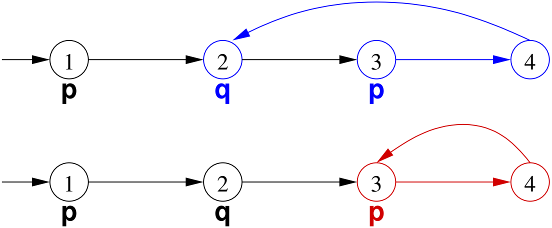


- No counter-example found.

Bounded Model Checking: Example



- LTL Formula: $\mathbf{G}(p \rightarrow \mathbf{F}q)$
- Negated Formula (violation): $\mathbf{F}(p \wedge \mathbf{G}\neg q)$
- $k = 3$:



- The 2nd trace is a counter-example!

The problem [Biere et al, 1999]

Ingredients:

- A system written as a Kripke structure $M := \langle S, I, T, \mathcal{L} \rangle$
- A property f written as a LTL formula:
- an integer $k \geq 0$ (bound)

Problem

Is there a (possibly-partial) execution path π of M of length k satisfying the temporal property f?

- the check is repeated for increasing values of $k = 1, 2, 3, ...$

The encoding

Equivalent to the satisfiability problem of a Boolean formula $[[M,f]]_k$ defined as follows:

$$[[M,f]]_k \quad := \quad [[M]]_k \wedge \; [[f]]_k \tag{1}$$

$$[[M]]_k \quad := \quad I(s^0) \wedge \bigwedge_{i=0}^{k-1} R(s^i, s^{i+1}), \tag{2}$$

$$[[f]]_k \quad := \quad (\neg \bigvee_{l=0}^{k} R(s^k, s^l) \wedge \; [[f]]_k^0) \vee \bigvee_{l=0}^{k} (R(s^k, s^l) \wedge \; {}_l[[f]]_k^0), \tag{3}$$

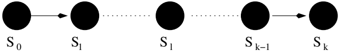
- the vector s of propositional variables is replicated k+1 times s^0, s^1, ..., s^k
- $[\![M]\!]_k$ encodes the fact that the k-path is an execution of M
- $[\![f]\!]_k$ encodes the fact that the k-path satisfies f

The Encoding [cont.]

The encoding for a formula *f* with *k* steps, $[[f]]_k$ is the disjunction of

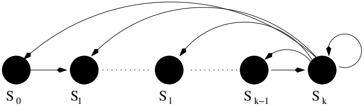
- the constraints needed to express a model without loopback:

$$(\neg(\bigvee_{l=0}^{k} R(s^k, s^l)) \wedge \ [[f]]_k^0)$$



  - $[[f]]_k^i,\ i \in [0,k]$: encodes the fact that *f* holds in s^i under the assumption that $s^0, ..., s^k$ is a no-loopback path
- the constraints needed to express a given loopback, for all possible points of loopback:

$$\bigvee_{l=0}^{k}(R(s^k, s^l) \wedge \ {}_l[[f]]_k^0)$$



  - ${}_l[[f]]_k^i,\ i \in [0,k]$: encodes the fact that *f* holds in s^i under the assumption that $s^0, ..., s^k$ is a path with a loopback from s^k to s^l

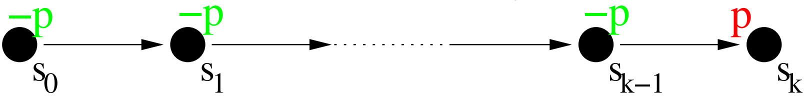
The encoding of $[[f]]_k^i$ and ${}_l[[f]]_k^i$

f	$[[f]]_k^i$	${}_l[[f]]_k^i$
p	p_i	p_i
$\neg p$	$\neg p_i$	$\neg p_i$
$h \wedge g$	$[[h]]_k^i \wedge [[g]]_k^i$	${}_l[[h]]_k^i \wedge {}_l[[g]]_k^i$
$h \vee g$	$[[h]]_k^i \vee [[g]]_k^i$	${}_l[[h]]_k^i \vee {}_l[[g]]_k^i$
$\mathbf{X}g$	$[[g]]_k^{i+1}$ *if* $i < k$ $\bot$ *otherwise.*	${}_l[[g]]_k^{i+1}$ *if* $i < k$ ${}_l[[g]]_k^{l}$ *otherwise.*
$\mathbf{G}g$	$\bot$	$\bigwedge_{j=min(i,l)}^{k} {}_l[[g]]_k^j$
$\mathbf{F}g$	$\bigvee_{j=i}^{k} [[g]]_k^j$	$\bigvee_{j=min(i,l)}^{k} {}_l[[g]]_k^j$
$h\mathbf{U}g$	$\bigvee_{j=i}^{k} \left([[g]]_k^j \wedge \bigwedge_{n=i}^{j-1} [[h]]_k^n \right)$	$\bigvee_{j=i}^{k} \left({}_l[[g]]_k^j \wedge \bigwedge_{n=i}^{j-1} {}_l[[h]]_k^n \right) \vee$ $\bigvee_{j=l}^{i-1} \left({}_l[[g]]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[[h]]_k^n \wedge \bigwedge_{n=l}^{j-1} {}_l[[h]]_k^n \right)$
$h\mathbf{R}g$	$\bigvee_{j=i}^{k} \left([[h]]_k^j \wedge \bigwedge_{n=i}^{j} [[g]]_k^n \right)$	$\bigwedge_{j=min(i,l)}^{k} {}_l[[g]]_k^j \vee$ $\bigvee_{j=i}^{k} \left({}_l[[h]]_k^j \wedge \bigwedge_{n=i}^{j} {}_l[[g]]_k^n \right) \vee$ $\bigvee_{j=l}^{i-1} \left({}_l[[h]]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[[g]]_k^n \wedge \bigwedge_{n=l}^{j} {}_l[[g]]_k^n \right)$

Example: **F**p (reachability)

- $f := \mathbf{F}p$, s.t. p Boolean:
  is there a reachable state in which p holds?
- a finite path can show that the property holds
- $[[M, f]]_k$ is:

$$I(s^0) \wedge \bigwedge_{i=0}^{k-1} R(s^i, s^{i+1}) \wedge \bigvee_{j=0}^{k} p^j$$

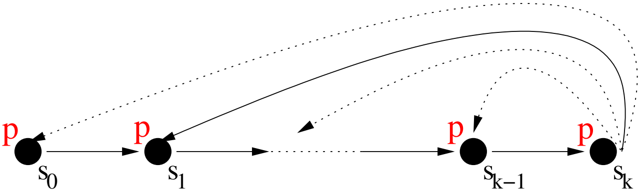


Important: incremental encoding

if done for increasing value of k, then it suffices that $[[M, f]]_k$ is:

Example: **G**p

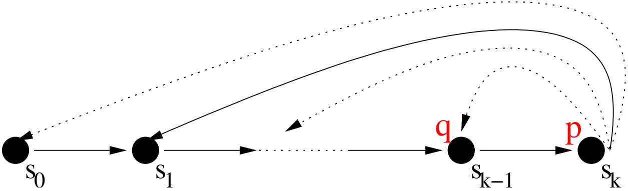
- $f := \mathbf{G}p$, s.t. p Boolean: is there a path where p holds forever?
- We need to produce an infinite behaviour, with a finite number of transitions
- We can do it by imposing that the path loops back



- $[[M, f]]_k$ is:

Example: **GF**q (fair states)

- $f := \mathbf{GF}q$, s.t. q Boolean: does q hold infinitely often?
- Again, we need to produce an infinite behaviour, with a finite number of transitions

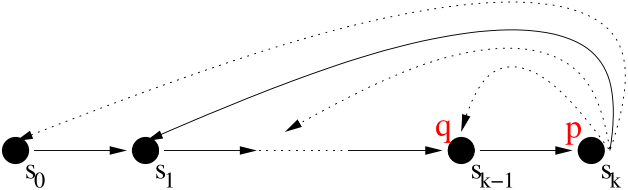


- $[[M, f]]_k$ is:

$k-1 \quad k \; (\quad k \;)$

Example: $\mathbf{GF}q \wedge \mathbf{F}p$ (fair reachability)

- $f := \mathbf{GF}q \wedge \mathbf{F}p$, s.t. p, q Boolean: provided that q holds infinitely often, is there a reachable state in which p holds?
- Again, we need to produce an infinite behaviour, with a finite number of transitions



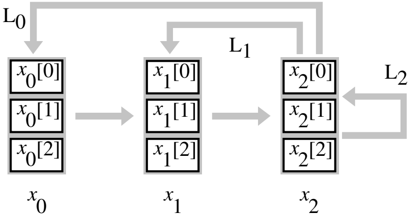
- $[[M, f]]_k$ is:

Example: a bugged 3-bit shift register

- System *M*:
  - $I(x) := \top$ (arbitrary initial state)
  - Correct *R*: $R(x, x') := (x'[0] \leftrightarrow x[1]) \wedge (x'[1] \leftrightarrow x[2]) \wedge (x'[2] \leftrightarrow 0)$
  - Bugged *R*: $R(x, x') := (x'[0] \leftrightarrow x[1]) \wedge (x'[1] \leftrightarrow x[2]) \wedge (x'[2] \leftrightarrow 1)$
- Property: $\mathbf{AF}(\neg x[0] \wedge \neg x[1] \wedge \neg x[2])$
- BMC Problem: is there an execution π of $\mathcal{M}$ of length *k* s.t. $\pi \models \mathbf{G}((x[0] \vee x[1] \vee x[2]))$?

Example: a bugged 3-bit shift register [cont.]

$k = 2$:



$[[M]]_2$: $\left(\begin{array}{l}(x_1[0] \leftrightarrow x_0[1]) \wedge (x_1[1] \leftrightarrow x_0[2]) \wedge (x_1[2] \leftrightarrow 1) \wedge \\ (x_2[0] \leftrightarrow x_1[1]) \wedge (x_2[1] \leftrightarrow x_1[2]) \wedge (x_2[2] \leftrightarrow 1)\end{array}\right) \wedge$

$\bigvee_{l=0}^{2} L_l$: $\left(\begin{array}{l}((x_0[0] \leftrightarrow x_2[1]) \wedge (x_0[1] \leftrightarrow x_2[2]) \wedge (x_0[2] \leftrightarrow 1))\vee \\ ((x_1[0] \leftrightarrow x_2[1]) \wedge (x_1[1] \leftrightarrow x_2[2]) \wedge (x_1[2] \leftrightarrow 1))\vee \\ ((x_2[0] \leftrightarrow x_2[1]) \wedge (x_2[1] \leftrightarrow x_2[2]) \wedge (x_2[2] \leftrightarrow 1))\end{array}\right) \wedge$

$\bigwedge_{i=0}^{2}(x \neq 0)$: $\left(\begin{array}{l}(x_0[0] \vee x_0[1] \vee x_0[2]) \wedge \\ (x_1[0] \vee x_1[1] \vee x_1[2]) \wedge \\ (x_2[0] \vee x_2[1] \vee x_2[2])\end{array}\right)$

$\Longrightarrow$ SAT: $x_i[j] := 1 \ \forall i, j$

Bounded Model Checking: summary

- incomplete technique:
  - if you find all formulas unsatisfiable, it tells you nothing
  - computing the maximum k (diameter) possible but extremely hard
- very efficient for some problems (typically debugging)
- lots of enhancements
- current symbolic model checkers embed a SAT based BMC tool

Efficiency Issues in Bounded Model Checking

- Caching different problems:
  - can we exploit the similarities between problems at k and $k+1$?
- Simplification of encodings
  - Reduced Boolean Circuits (RBC)
  - Boolean Expression Diagrams (BED)
  - And-Inverter Graphs (AIG)
  - Simplification based on Binary-Clauses Reasoning
- When can we stop increasing the bound k if we don't find violations?

Basic bounds for k

Theorem [Biere et al. TACAS 1999]

Let f be a LTL formula. $M \models \mathbf{E}f \iff M \models_k \mathbf{E}f$ for some $k \leq |M| \cdot 2^{|f|}$.

- $|M| \cdot 2^{|f|}$ is always a bound of k.
  - $|M|$ huge!

  $\Longrightarrow$ not so easy to compute in a symbolic setting.

$\Longrightarrow$ need to find better bounds!

Note: [Biere et al. TACAS 1999] use "$M \models \mathbf{E}f$" as "there exists a path of M verifying f", so that $M \not\models \mathbf{A}\neg f \iff M \models \mathbf{E}f$

Other bounds for k

ACTL & ECTL

- ACTL is a subset of CTL in which "**A**..." (resp. "**E**...") sub-formulas occur only positively (resp. negatively) in each formula.
  e.g. $\mathbf{AG}(p \rightarrow \mathbf{AGAF}q)$
- ECTL is a subset of CTL in which "**E**..." (resp. "**A**...") sub-formulas occur only positively (resp. negatively) in each formula.
  e.g. $\mathbf{EF}(p \wedge \mathbf{EFEG}\neg q)$
- ECTL is the dual subset of ACTL: $\phi \in ECTL \Longleftrightarrow \neg\phi \in ACTL$.
- Many frequently-used LTL properties $\neg f$ have equivalent ACTL representations $\mathbf{A}\neg f'$ (e.g. $\mathbf{G}(p \rightarrow \mathbf{GF}q)$ wrt. $\mathbf{AG}(p \rightarrow \mathbf{AGAF}q)$)

Theorem [Biere et al. TACAS 1999]

Let f be an ECTL formula. $M \models \mathbf{E}f \Longleftrightarrow M \models_k \mathbf{E}f$ for some $k \leq |M|$.

Other bounds for k (cont)

Theorem [Biere et al. TACAS 1999]

Let p be a Boolean formula and d be the diameter of M. Then $M \models \mathbf{EF}p \Longleftrightarrow M \models_k \mathbf{EF}p$ for some $k \leq d$.

Theorem [Biere et al. TACAS 1999]

Let f be an ECTL formula and d be the recurrence diameter of M. Then $M \models \mathbf{E}f \Longleftrightarrow M \models_k \mathbf{E}f$ for some $k \leq d$.

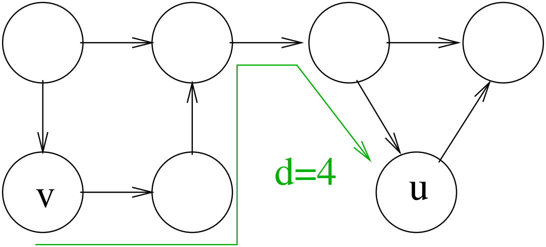
The diameter

Definition: diameter

Given M, the diameter of M is the smallest integer d s.t. for every path $s_0, ..., s_{d+1}$ there exist a path $t_0, ..., t_l$ s.t. $l \leq d$, $t_0 = s_0$ and $t_l = s_{d+1}$.

- Intuition: if u is reachable from v, then there is a path from v to u of length d or less.

$\Longrightarrow$ it is the maximum distance between two states in M.



The diameter: computation

- d is the smallest integer d which makes the following formula true:

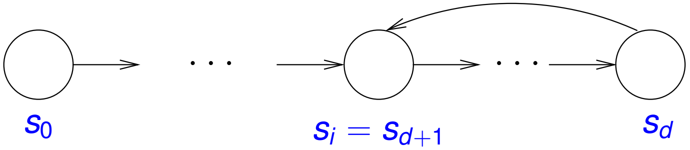
$$\forall s_0, \ldots, s_{d+1} . \exists t_0, \ldots, t_d .$$
$$\underbrace{\bigwedge_{i=0}^{d} T(s_i, s_{i+1})}_{s_0,\ldots,s_{d+1} \text{ is a path}} \rightarrow \underbrace{\left(t_0 = s_0 \wedge \bigwedge_{i=0}^{d-1} T(t_i, t_{i+1}) \wedge \bigvee_{i=0}^{d} t_i = s_{d+1} \right)}_{t_0,\ldots,t_i \text{ is another path from } s_0 \text{ to } s_{d+1} \text{ for some } i}$$

- Quantified Boolean formula (QBF): much harder than NP-complete!

The recurrence diameter

Definition: recurrence diameter

Given M, the recurrence diameter of M is the smallest integer d s.t. for every path $s_0, ..., s_{d+1}$ there exist $j \leq d$ s.t. $s_{d+1} = s_j$.



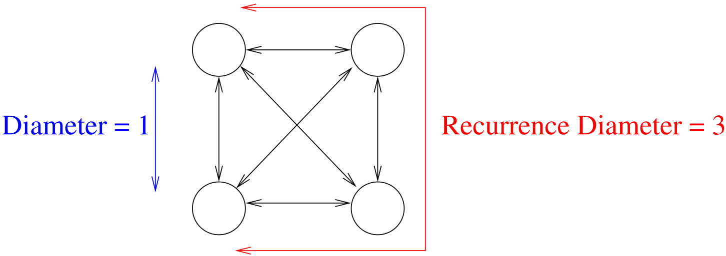
- Intuition: the maximum length of a non-loop path

The recurrence diameter: computation

- d is the smallest integer d which makes the following formula true:

$$\forall s_0, ..., s_{d+1}. \underbrace{\bigwedge_{i=0}^{d} T(s_i, s_{i+1})}_{s_0,\ldots,s_{d+1} \textit{ is a path}} \rightarrow \underbrace{\bigvee_{i=0}^{d} s_i = s_{d+1}}_{s_0,\ldots,s_{d+1} \textit{ contains a cicle}}$$

- Validity problem: coNP-complete (solvable by SAT).
- Possibly much longer than the diameter!



Inductive Reasoning on Invariants

Invariant: “**AG***Good*”, *Good* being a Boolean formula

(i) If all the initial states are good,

(ii) and if from good states we only go to good states

then we can conclude that the system is correct for all reachable states.

SAT-based Inductive Reasoning on Invariants

(i) If all the initial states are good

- $I(s^0) \rightarrow Good(s^0)$ is valid (i.e. its negation is unsatisfiable)

(ii) if from good states we only go to good states

- $(Good(s^{k-1}) \wedge R(s^{k-1}, s^k)) \rightarrow Good(s^k)$ is valid (i.e. its negation is unsatisfiable)

then we can conclude that the system is correct for all reachable states

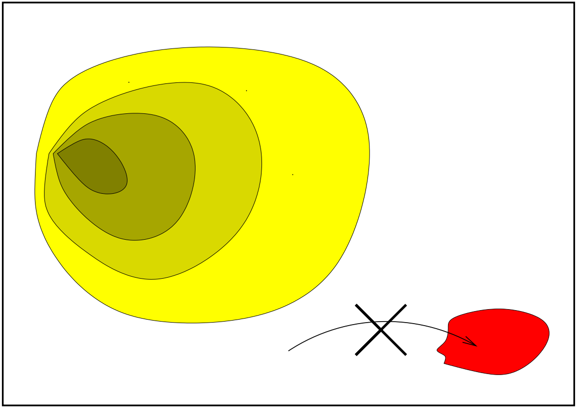
⇒ Check for the (un)satisfiability of the Boolean formulas:

$$(I(s^0) \wedge \neg Good(s^0));$$
$$(Good(s^{k-1}) \wedge R(s^{k-1}, s^k)) \wedge \neg Good(s^k)\ )$$

(iii) N.B: “$(I(s^0) \wedge \neg Good(s^0))$” is step-0 incremental BMC encoding for $\mathbf{F}\neg Good$.

Strengthening of Invariants

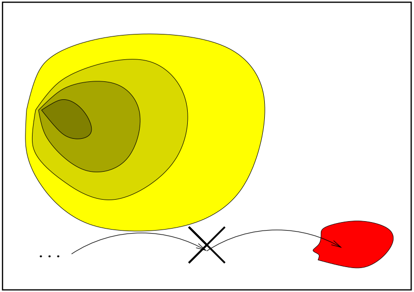
- Problem: Induction may fail because of unreachable states:
  - if $(Good(s^{k-1}) \wedge R(s^{k-1}, s^k)) \rightarrow Good(s^k)$ is not valid, this does not mean that the property does not hold
  - both s^{k-1} and s^k might be unreachable



Strengthening of Invariants [cont.]

Solution (once you know you cannot reach $\neg Good$ in up to 1 step):

- increase the depth of induction
  $(Good(s^{k-2}) \wedge R(s^{k-2}, s^{k-1}) \wedge Good(s^{k-1}) \wedge R(s^{k-1}, s^{k}) \wedge \neg(s^{k-2} = s^{k-1})) \rightarrow Good(s^{k})$



- force loop freedom with $\neg(s^i = s^j)$ for every $i \neq j$ s.t. $i, j \leq k$
- performed after step-1 BMC step returns “unsat”:
  $I(s^0) \wedge (R(s^0, s^1) \wedge Good(s^0)) \wedge \neg Good(s^1)$

Strengthening of Invariants [cont.]

$\Longrightarrow$ Check for the [un]satisfiability of the Boolean formulas:

$I(s^0) \wedge \neg Good(s^0)$; $[BMC_0]$

$(Good(s^{k-1}) \wedge R(s^{k-1}, s^k)) \wedge \neg Good(s^k)$; $[Kind_0]$

$I(s^0) \wedge (R(s^0, s^1) \wedge Good(s^0)) \wedge \neg Good(s^1)$; $[BMC_1]$

$(Good(s^{k-2}) \wedge R(s^{k-2}, s^{k-1}) \wedge Good(s^{k-1}) \wedge R(s^{k-1}, s^k)) \wedge \neg Good(s^k)$
$\wedge \neg(s^{k-2} = s^{k-1})$; $[Kind_1]$

$I(s^0) \wedge (R(s^0, s^1) \wedge Good(s^0) \wedge (R(s^1, s^2) \wedge Good(s^1)) \wedge \neg Good(s^2)$; $[BMC_2]$

...

- repeat for increasing values of the gap $1, 2, 3, 4, ...$
- intuition: increasingly tighten the constraint for “spurious” counterexamples: a spurious counterexample must be a chain $s_{k-n}, ..., s_k$ of unreachable and different states s.t. $\neg Good(s_k)$ and $R(s_i, s_{i+1}), \forall i$.
- dual to –and interleaved with– bounded model checking steps
- K-Induction steps can be shifted ($k \stackrel{\text{def}}{=} 0$) to share the subformulas: $\bigwedge_{i=0}^{k-1} (R(s^i, s^{i+1}) \wedge Good(s^i)) \wedge \neg Good(s^{k-2})$

Mixed BMC & K-Induction [Sheeran et al. 2000]

$$\begin{array}{lcl} Base_n & := & I(\mathbf{s}_0) \wedge \bigwedge_{i=0}^{n-1} (R(\mathbf{s}_i, \mathbf{s}_{i+1}) \wedge \varphi(\mathbf{s}_i)) \wedge \neg\varphi(\mathbf{s}_n) \\ Step_n & := & \bigwedge_{i=0}^{n} (R(\mathbf{s}_i, \mathbf{s}_{i+1}) \wedge \varphi(\mathbf{s}_i)) \wedge \neg\varphi(\mathbf{s}_{n+1}) \\ Unique_n & := & \bigwedge_{0 \le i \le j \le n} \neg(\mathbf{s}_i = \mathbf{s}_{j+1}) \end{array}$$

Algorithm

```
function CHECK_PROPERTY $(I, R, \varphi)$
    for $n := 0, 1, 2, 3, ....$ do
        if (DPLL($Base_n$) == SAT)
            then return PROPERTY_VIOLATED;
        else if (DPLL($Step_n \wedge Unique_n$) == UNSAT)
            then return PROPERTY_VERIFIED;
    end for;
```

$\Longrightarrow$ reuses previous search if DPLL is incremental!!

Example: a correct 3-bit shift register

- System M:
    - $I(x) := (\neg x[0] \land \neg x[1] \land \neg x[2])$
    - $R(x, x') := ((x'[0] \leftrightarrow x[1]) \land (x'[1] \leftrightarrow x[2]) \land (x'[2] \leftrightarrow 0))$
- Property: $\mathbf{AG}\neg x[0]$

Example: a correct 3-bit shift register [cont.]

- Init (BMC Step 0): $((\neg x^0[0] \wedge \neg x^0[1] \wedge \neg x^0[2]) \wedge x^0[0]) \Longrightarrow$ unsat
- K-Induction Step 1:

$$\left(\begin{array}{l} (\neg x^0[0] \wedge ((x^1[0] \leftrightarrow x^0[1]) \wedge (x^1[1] \leftrightarrow x^0[2]) \wedge (x^1[2] \leftrightarrow 0))) \\ \quad \wedge\, x^1[0] \end{array} \right)$$

$\Longrightarrow$ (partly by unit-propagation)

sat: $\left\{ \begin{array}{ccc} \neg x^0[0], & x^0[1], & x^0[2], \\ x^1[0], & x^1[1], & \neg x^1[2] \end{array} \right\}$

$\Longrightarrow$ not proved

Remark

Both $\{\neg x^0[0], \quad x^0[1], \quad x^0[2])\}$ and $\{ \quad x^1[0], \quad x^1[1], \neg x^1[2]\}$ are non-reachable.

Example: a correct 3-bit shift register [cont.]

- BMC Step 1: (...)$\Longrightarrow$ unsat
- K-Induction Step 2:

$$\left(\begin{array}{l} (\neg x^0[0] \wedge ((x^1[0] \leftrightarrow x^0[1]) \wedge (x^1[1] \leftrightarrow x^0[2]) \wedge (x^1[2] \leftrightarrow 0))\wedge \\ \quad \neg x^1[0] \wedge ((x^2[0] \leftrightarrow x^1[1]) \wedge (x^2[1] \leftrightarrow x^1[2]) \wedge (x^2[2] \leftrightarrow 0)) \\ \ ) \wedge x^2[0] \end{array}\right)$$

$$\wedge\neg((x^1[0] \leftrightarrow x^0[0]) \wedge (x^1[1] \leftrightarrow x^0[1]) \wedge (x^1[2] \leftrightarrow x^0[2]))$$

$$\Longrightarrow \text{ sat: } \left\{\begin{array}{rrr} \neg x^0[0], & \neg x^0[1], & x^0[2] \\ \neg x^1[0], & x^1[1], & \neg x^1[2] \\ x^2[0], & \neg x^2[1], & \neg x^2[2] \end{array}\right\} \Longrightarrow \text{not proved}$$

Remark

$\{\neg x^0[0], \neg x^0[1], \ x^0[2]\}$, $\{\neg x^1[0], \ x^1[1], \neg x^1[2]\}$, and $\{\ x^2[0], \neg x^2[1], \neg x^2[2]\}$ are non-reachable.

Example: a correct 3-bit shift register [cont.]

- BMC Step 2: (...) $\Longrightarrow$ unsat
- K-Induction Step 3:

$$
\begin{pmatrix}
(\neg x^0[0] \wedge ((x^1[0] \leftrightarrow x^0[1]) \wedge (x^1[1] \leftrightarrow x^0[2]) \wedge (x^1[2] \leftrightarrow 0))\wedge \\
\quad \neg x^1[0] \wedge ((x^2[0] \leftrightarrow x^1[1]) \wedge (x^2[1] \leftrightarrow x^1[2]) \wedge (x^2[2] \leftrightarrow 0))\wedge \\
\quad \neg x^2[0] \wedge ((x^3[0] \leftrightarrow x^2[1]) \wedge (x^3[1] \leftrightarrow x^2[2]) \wedge (x^3[2] \leftrightarrow 0)) \\
\ ) \wedge x^3[0]
\end{pmatrix}
$$

$$
\begin{array}{l}
\wedge\neg((x^1[0] \leftrightarrow x^0[0]) \wedge (x^1[1] \leftrightarrow x^0[1]) \wedge (x^1[2] \leftrightarrow x^0[2])) \\
\wedge\neg((x^2[0] \leftrightarrow x^0[0]) \wedge (x^2[1] \leftrightarrow x^0[1]) \wedge (x^2[2] \leftrightarrow x^0[2])) \\
\wedge\neg((x^2[0] \leftrightarrow x^1[0]) \wedge (x^2[1] \leftrightarrow x^1[1]) \wedge (x^2[2] \leftrightarrow x^1[2]))
\end{array}
$$

$\Longrightarrow$ (unit-propagation) $\{x^3[0], x^2[1], x^1[2]\}$

$\Longrightarrow$ unsat

$\Longrightarrow$ proved!

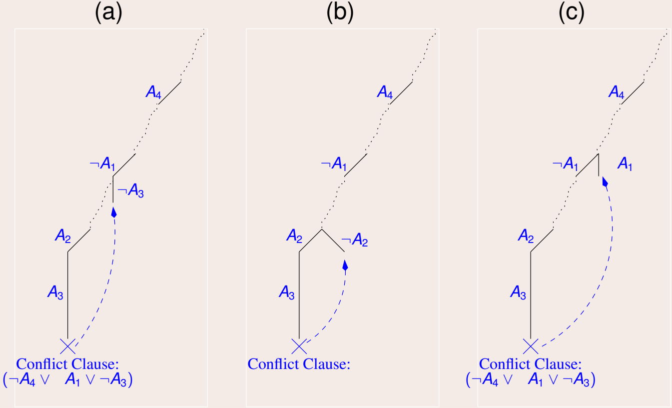
Other Successful SAT-based (UNbounded) MC Techniques

- Counter-example guided abstraction refinement (CEGAR)
  [Clarke et al. CAV 2002]
- Interpolant-based MC
  [Mc Millan, TACAS 2005]
- IC3/PDR
  [Bradley, VMCAI 2011]
- ...

For a survey see e.g.
[Amla et al., CHARME 2005, Prasad et al. STTT 2005].

Ex: CDCL SAT Solving

Which of the following figures may correspond to a modern DPLL 1st-UIP backjumping step?



[Solution: The correct answer is (a). (b) represents standard chronological backtracking, whilst (c) is nonsense.]

Ex: Bounded Model Checking

Given the symbolic representation of a FSM M, expressed in terms of the two Boolean formulas: $I(x,y) \stackrel{\text{def}}{=} \neg x \wedge y$, $T(x,y,x',y') \stackrel{\text{def}}{=} (x' \leftrightarrow (x \leftrightarrow \neg y)) \wedge (y' \leftrightarrow \neg y)$, and the LTL property: $\varphi \stackrel{\text{def}}{=} \neg \mathbf{F}(x \wedge y)$,

1. Write a Boolean formula whose solutions (if any) represent executions of M of length 2 which violate φ.

   [Solution: The question corresponds to the Bounded Model Checking problem $M \models_2 \mathbf{E}\,\mathbf{F}f$, s.t. $f(x,y) \stackrel{\text{def}}{=} (x \wedge y)$. Thus we have:

$$\begin{array}{lcl}
\neg x_0 \wedge y_0 & \wedge & // \; I(x_0,y_0) \wedge \\
(x_1 \leftrightarrow (x_0 \leftrightarrow \neg y_0)) \wedge (y_1 \leftrightarrow \neg y_0) & \wedge & // \; T(x_0,y_0,x_1,y_1) \wedge \\
(x_2 \leftrightarrow (x_1 \leftrightarrow \neg y_1)) \wedge (y_2 \leftrightarrow \neg y_1) & \wedge & // \; T(x_1,y_1,x_2,y_2) \wedge \\
((x_0 \wedge y_0) & \vee & // \; (f(x_0,y_0) \vee \\
(x_1 \wedge y_1) & \vee & // \; f(x_1,y_1) \vee \\
(x_2 \wedge y_2)) & & // \; f(x_2,y_2))
\end{array}$$

   ]

2. Is there a solution? If yes, find the corresponding execution; if no, show why.

   [Solution: Yes: $\{\neg x_0, y_0, x_1, \neg y_1, x_2, y_2\}$, corresponding to the execution: $(0,1) \rightarrow (1,0) \rightarrow (1,1)$]

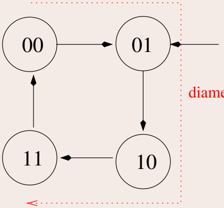
Ex: Bounded Model Checking

3. From the solutions to question #1 and #2 we can conclude that:
   - (a) $M \models \varphi$
   - (b) $M \not\models \varphi$
   - (c) we can conclude nothing.

   [Solution: b)]

4. What are the diameter and the recurrence diameter of this system?

   [Solution:



diameter = recurrence diameter = 3

]