Introduction

Systemtheory 2

Formal Systems 2

#342201

SS 2006

Johannes Kepler Universität

Linz, Österreich

Univ. Prof. Dr. Armin Biere
Institute for Formal Models and Verification

http://fmv.jku.at/fs2

Motivation

- more and more complex systems

  Moore's Law ⇒ soon we will have 10^{30} transistors / processor

  multi-million LOC / OS

  ⇒ exploding testing costs (in general not linear in system size)

- increased dependability

  everything important depends on computers:

  stir by wire, banking, stock market, workflow, ...

  ⇒ quality concerns

- increased functionality

  security, mobility, new business processes, ...

Test and Verification

Test

standard definition: dynamic execution / simulation of a system

integration in development process necessary

extreme position: testing should actually "drive" the development process

Verification

standard definition: static checking, symbolic execution

hardware design: verification is the process of testing

⇒ our view: Test = Verification

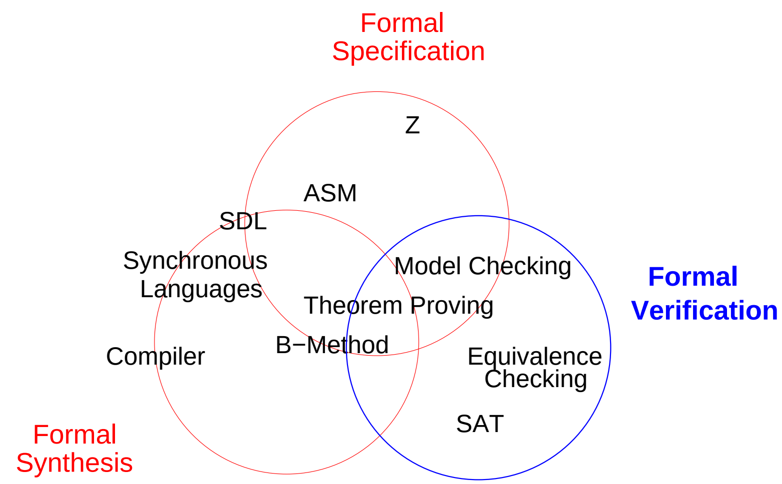
Implications

- not unusual to have more than 50% of resources allocated to testing
- testing and verification are (becoming) the bottleneck of development
- quality dilemma (drop quality for more features)
- more efficient methods for test and verification needed
  ⇒ formal verification is the most promising approach
- experts in new testing and verification methods are lacking
- long term: more formal development process not just formal verification

Formal Methods

- formal = mathematical
- mathematical models ⇒ precise semantics
- emphasizes static / symbolic reasoning about programs
  (so standard definition of verification falls into this category)
- rather narrow view in digital design: equivalence and model checking
- not esoteric: compilation in a broad sense is a formal method
  (high-level description is translated into low-level description)
- our view: use tools for reasoning (i.e. programs are formal entities)

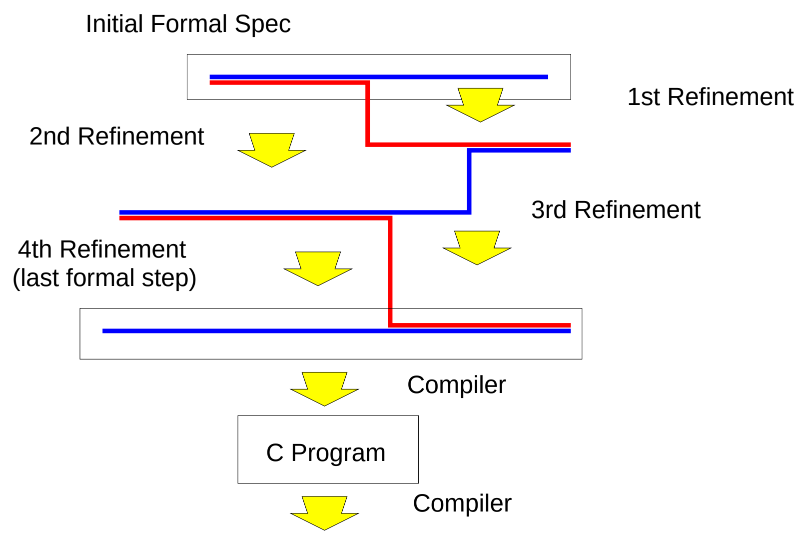
Formal Methods Classification



Formal Specification

- abstracts from unnecessary implementation details
- high-level mathematical model of the system
- very useful for high-level design
- catches ambiguous or inconsistent specifications
- formal specification per se: no tools for refinement / checking
- good example: ASM

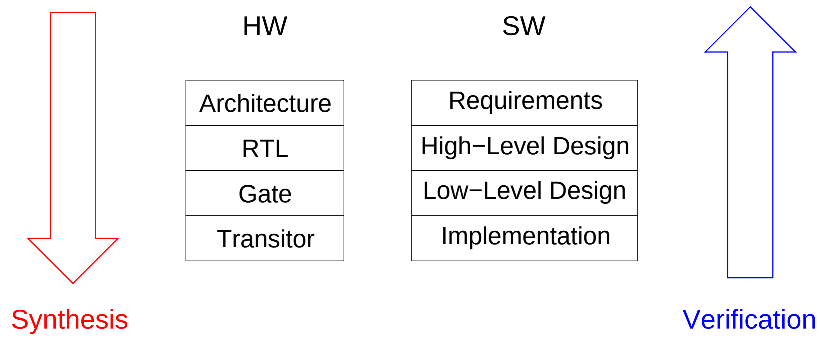
Formal Synthesis



Formal Synthesis

- integrates verification in the development process
- usually pure top-down design and incremental refinement steps
- splits large verification tasks (divide et impera) . . .
- . . . but forces dramatic change in development process
- it works but is costly
- each refinement step uses formal verification methods
  ⇒ more powerfull verification algorithms allow more automation
- good example: B-Method

Layered System Design



1. no implementation without Synthesis
2. Verification is added value (Quality)
3. both processes are incremental
4. both processes can be formal

Formal Verification

- assumptions: specification and system are given
- formal verification checks formally that system fulfills specification
- least change in development process
- full blown verification is really difficult: “post mortem verification”
- simplifications: focus on simple partial specifications
  (type safety, functional equivalence of two systems, . . .)
- methods (implemented in tools):

  simple algorithms for deducing properties directly

  complex algorithms for hard or even undecidable problems

Overview

- boolean methods:

  SAT, BDDs, ATPG, Combinational Equivalence Checking
- finite state methods:

  Bisimulation and Equivalence Checking of Automata, Model Checking
- term based methods:

  Term Rewriting, Resolution, Tableaux, Theorem Proving
- Abstraction (eg SLAM uses BDDs, Model Checking, Theorem Proving)

Focus

- how does it work?

  (algorithms and data structures)

- necessary background for use of formal verification

  (and formal methods in general)

- capacity and restrictions

- first step to become an expert in a fast expanding area

SAT Example: Equivalence Checking if-then-else Chains

optimization of if-then-else chains

original C code

```
if(!a && !b) h();
else if(!a) g();
else f();
```

$\Downarrow$

```
if(!a) {
  if(!b) h();
  else g();
} else f();
```

$\Rightarrow$

optimized C code

```
if(a) f();
else if(b) g();
else h();
```

$\Uparrow$

```
if(a) f();
else {
  if(!b) h();
  else g(); }
```

How to check that these two versions are equivalent?

SAT Example cont.

1. represent procedures as *independent* boolean variables

$$\mathit{original} :=\quad \textbf{if } \neg a \wedge \neg b \textbf{ then } h \textbf{ else if } \neg a \textbf{ then } g \textbf{ else } f$$

$$\mathit{optimized} :=\quad \textbf{if } a \textbf{ then } f \textbf{ else if } b \textbf{ then } g \textbf{ else } h$$

2. compile if-then-else chains into boolean formulae

$$\text{compile}(\textbf{if } x \textbf{ then } y \textbf{ else } z) \quad\equiv\quad (x \wedge y) \vee (\neg x \wedge z)$$

3. check equivalence of boolean formulae

$$\text{compile}(\mathit{original}) \quad\Leftrightarrow\quad \text{compile}(\mathit{optimized})$$

Compilation

$$\begin{aligned}
\mathit{original} &\equiv \textbf{if } \neg a \wedge \neg b \textbf{ then } h \textbf{ else if } \neg a \textbf{ then } g \textbf{ else } f \\
&\equiv (\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge \textbf{if } \neg a \textbf{ then } g \textbf{ else } f \\
&\equiv (\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \vee a \wedge f)
\end{aligned}$$

$$\begin{aligned}
\mathit{optimized} &\equiv \textbf{if } a \textbf{ then } f \textbf{ else if } b \textbf{ then } g \textbf{ else } h \\
&\equiv a \wedge f \vee \neg a \wedge \textbf{if } b \textbf{ then } g \textbf{ else } h \\
&\equiv a \wedge f \vee \neg a \wedge (b \wedge g \vee \neg b \wedge h)
\end{aligned}$$

$$(\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \vee a \wedge f) \quad\Leftrightarrow\quad a \wedge f \vee \neg a \wedge (b \wedge g \vee \neg b \wedge h)$$

How to Check (In)Equivalence?

Reformulate it as a satisfiability (SAT) problem:

Is there an assignment to a, b, f, g, h,
which results in different evaluations of original and optimized?

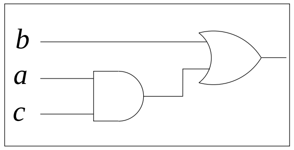
or equivalently:

Is the boolean formula $\text{compile}(\mathit{original}) \not\leftrightarrow \text{compile}(\mathit{optimized})$ satisfiable?

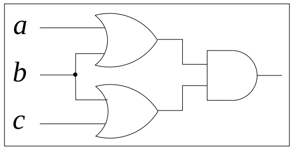
such an assignment would provide an easy to understand counterexample

Note: by concentrating on counterexamples we moved from Co-NP to NP
(this is just a theoretical note and not really important for applications)

SAT Example: Circuit Equivalence Checking



$b \vee a \wedge c$



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

equivalent?

$$b \vee a \wedge c \qquad \Leftrightarrow \qquad (a \vee b) \wedge (b \vee c)$$

SAT

SAT (Satisfiability) the classical NP complete Problem:

Given a propositional formula f over n propositional variables $V = \{x, y, \ldots\}$.

Is there are an assignment $\sigma : V \to \{0,1\}$ with $\sigma(f) = 1$?

SAT belongs to NP

There is a *non-deterministic* Touring-machine deciding SAT in polynomial time:

guess the assignment σ (linear in n), calculate $\sigma(f)$ (linear in $|f|$)

Note: on a *real* (deterministic) computer this would still require 2^n time

SAT is complete for NP (see complexity / theory class)

Implications for us:
general SAT algorithms are probably exponential in time (unless NP = P)

Conjunctive Normal Form

Definition

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

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

each clause C is a disjunction of literals

$$C = L_1 \vee \ldots \vee L_m$$

and each literal is either a plain variable x or a negated variable $\overline{x}$.

Example $(a \vee b \vee c) \wedge (\overline{a} \vee \overline{b}) \wedge (\overline{a} \vee \overline{c})$

Note 1: two notions for negation: in $\overline{x}$ and $\neg$ as in $\neg x$ for denoting negation.

Note 2: the original SAT problem is actually formulated for CNF

Note 3: SAT solvers mostly also expect CNF as input

Negation Normal Form

Assumption: we only have conjunction, disjunction and negation as operators.

a formula is in Negation Normal Form (NNF),
if negations only occur in front of variables

$\Rightarrow$ all *internal* nodes in the formula tree are either ANDs or ORs

linear algorithms for generating NNF from an arbitrary formula

often NNF generations includes elimination of other non-monotonic operators:

NNF of $f \leftrightarrow g$ is NNF of $f \wedge g \vee \overline{f} \wedge \overline{g}$

in this case the result can be exponentially larger (see parity example later).

NNF Algorithm


```
Formula
formula2nnf (Formula f, Boole sign)
{
  if (is_variable (f))
    return sign ? new_not_node (f) : f;
  if (op (f) == AND || op (f) == OR)
    {
      l = formula2nnf (left_child (f), sign);
      r = formula2nnf (right_child (f), sign);
      flipped_op = (op (f) == AND) ? OR : AND;
      return new_node (sign ? flipped_op : op (f), l, r);
    }
  else
    {
      assert (op (f) == NOT);
      return formula2nnf (child (f), !sign);
    }
}
```

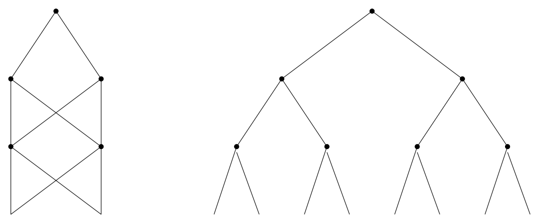

Simple Translation of Formula into CNF


```
Formula
formula2cnf_aux (Formula f)
{
  if (is_cnf (f))
    return f;
  if (op (f) == AND)
    {
      l = formula2cnf_aux (left_child (f));
      r = formula2cnf_aux (right_child (f));
      return new_node (AND, l, r);
    }
  else
    {
      assert (op (f) == OR);
      l = formula2cnf_aux (left_child (f));
      r = formula2cnf_aux (right_child (f));
      return merge_cnf (l, r);
    }
}
```


Merging two CNFs


```
Formula
formula2cnf (Formula f)
{
  return formula2cnf_aux (formula2nnf (f, 0));
}


Formula
merge_cnf (Formula f, Formula g)
{
  res = new_constant_node (TRUE);
  for (c = first_clause (f); c; c = next_clause (f, c))
    for (d = first_clause (g); d; d = next_clause (g, d))
      res = new_node (AND, res, new_node (OR, c, d));
  return res;
}
```

Why are Sharing / Circuits / DAGs important?



DAG may be exponentially more succinct than expanded Tree

Examples: adder circuit, parity, mutual exclusion

Parity Example


```
Boole
parity (Boole a, Boole b, Boole c, Boole d, Boole e,
        Boole f, Boole g, Boole h, Boole i, Boole j)
{
  tmp0 = b ? !a : a;
  tmp1 = c ? !tmp0 : tmp0;
  tmp2 = d ? !tmp1 : tmp1;
  tmp3 = e ? !tmp2 : tmp2;
  tmp4 = f ? !tmp3 : tmp3;
  tmp5 = g ? !tmp4 : tmp4;
  tmp6 = h ? !tmp5 : tmp5;
  tmp7 = i ? !tmp6 : tmp6;
  return j ? !tmp7 : tmp7;
}
```

Eliminiate the `tmp`... variables through substitution.

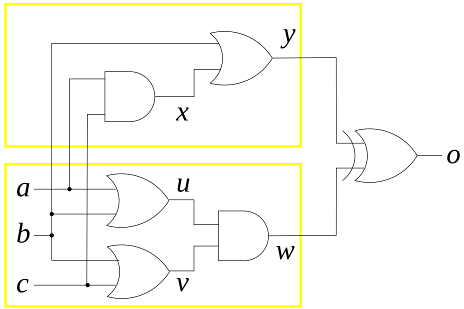
What is the size of the DAG vs the Tree representation?

How to detect Sharing

- through caching of results in algorithms operating on formulas
  (examples: substitution algorithm, generation of NNF for non-monotonic ops)
- when modeling a system: variables are introduced for subformulae
  (then these variables are used multiple times in the toplevel formula)
- structural hashing: detects structural identical subformulae
  (see Signed And Graphs later)
- equivalence extraction: eg. BDD sweeping, Stålmarcks Method
  (we will look at both techniques in more detail later)

Example of Tseitin Transformation: Circuit to CNF

CNF



$$
\begin{array}{r}
o\ \wedge \\
(x \leftrightarrow a \wedge c) \wedge \\
(y \leftrightarrow b \vee x) \wedge \\
(u \leftrightarrow a \vee b) \wedge \\
(v \leftrightarrow b \vee c) \wedge \\
(w \leftrightarrow u \wedge v) \wedge \\
(o \leftrightarrow y \oplus w)
\end{array}
$$

$$o \wedge (x \rightarrow a) \wedge (x \rightarrow c) \wedge (x \leftarrow a \wedge c) \wedge \ \ldots$$

$$o \wedge (\overline{x} \vee a) \wedge (\overline{x} \vee c) \wedge (x \vee \overline{a} \vee \overline{c}) \wedge \ \ldots$$

Algorithmic Description of Tseitin Transformation

1. for each non input circuit signal s generate a new variable x_s
2. for each gate produce complete input / output constraints as clauses
3. collect all constraints in a big conjunction

the transformation is *satisfiability equivalent*:
the result is satisfiable iff and only the original formula is satisfiable

not equivalent in the classical sense to original formula: it has new variables

extract satisfying assignment for original formula, from one of the result
(just project satisfying assignment onto the original variables)

Tseitin Transformation: Input / Output Constraints

Negation: $x \leftrightarrow \overline{y} \;\Leftrightarrow\; (x \rightarrow \overline{y}) \wedge (\overline{y} \rightarrow x)$
$\;\Leftrightarrow\; (\overline{x} \vee \overline{y}) \wedge (y \vee x)$

Disjunction: $x \leftrightarrow (y \vee z) \;\Leftrightarrow\; (y \rightarrow x) \wedge (z \rightarrow x) \wedge (x \rightarrow (y \vee z))$
$\;\Leftrightarrow\; (\overline{y} \vee x) \wedge (\overline{z} \vee x) \wedge (\overline{x} \vee y \vee z)$

Conjunction: $x \leftrightarrow (y \wedge z) \;\Leftrightarrow\; (x \rightarrow y) \wedge (x \rightarrow z) \wedge ((y \wedge z) \rightarrow x)$
$\;\Leftrightarrow\; (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{(y \wedge z)} \vee x)$
$\;\Leftrightarrow\; (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z} \vee x)$

Equivalence: $x \leftrightarrow (y \leftrightarrow z) \;\Leftrightarrow\; (x \rightarrow (y \leftrightarrow z)) \wedge ((y \leftrightarrow z) \rightarrow x)$
$\;\Leftrightarrow\; (x \rightarrow ((y \rightarrow z) \wedge (z \rightarrow y))) \wedge ((y \leftrightarrow z) \rightarrow x)$
$\;\Leftrightarrow\; (x \rightarrow (y \rightarrow z)) \wedge (x \rightarrow (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x)$
$\;\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \leftrightarrow z) \rightarrow x)$
$\;\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (((y \wedge z) \vee (\overline{y} \wedge \overline{z})) \rightarrow x)$
$\;\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \wedge z) \rightarrow x) \wedge ((\overline{y} \wedge \overline{z}) \rightarrow x)$
$\;\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (\overline{y} \vee \overline{z} \vee x) \wedge (y \vee z \vee x)$

Optimizations for Tseitin Transformation

- goal is smaller CNF (less variables, less clauses)
- extract multi argument operands (removes variables for intermediate nodes)
- half of AND, OR node constraints may be removed for *unnegated* nodes

  a node occurs negated if it has an ancestor which is a negation

  half of the constraints determine parent assignment from child assignment

  those are unnecessary if node is not used negated
- those have to be carefully applied to DAG structure
  (compare with the implementation of the BMC tool from CMU)

Davis & Putnam Procedure (DP)

- dates back to the 50ies:

  original version is *resolution based* (less successful)
- **idea:** case analysis (try $x = 0, 1$ in turn and recurse)
- most successful SAT solvers (autumn 2003)

  works for very large instances
- recent ($\leq$ 10 years) optimizations:

  backjumping, learning, UIPs, dynamic splitting heuristics, fast data structures

  (we will have a look at each of them)

Resolution

- basis for first (less successful) resolution based DP
- can be extended to first order logic
- helps to explain learning

Resolution Rule

$$\frac{C \cup \{v\} \qquad D \cup \{\neg v\}}{C \cup D} \qquad \{v, \neg v\} \cap C = \{v, \neg v\} \cap D = \emptyset$$

Read: resolving the clause $C \cup \{v\}$ with the clause $D \cup \{\neg v\}$, both above the line, on the variable v, results in the clause $D \cup C$ below the line.

Correctness of Resolution Rule

Usage of such rules: if you can derive what is above the line (premise) then you are allowed to deduce what is below the line (conclusion).

Theorem. (premise satisfiable $\Rightarrow$ conclusion satisfiable)

$$\sigma(C \cup \{v\}) = \sigma(D \cup \{\neg v\}) = 1 \quad \Rightarrow \quad \sigma(C \cup D) = 1$$

Proof.

let $c \in C$, $d \in D$ with $\quad (\sigma(c) = 1$ or $\sigma(v) = 1) \quad$ and $\quad (\sigma(d) = 1$ or $\sigma(\neg v) = 1)$

if $\quad \sigma(c) = 1$ or $\sigma(d) = 1 \quad$ conclusion follows immediately

otherwise $\quad \sigma(v) = \sigma(\neg v) = 1 \quad \Rightarrow$ contradiction

q.e.d.

Completeness of Resolution Rule

Theorem. (conclusion satisfiable $\Rightarrow$ premise satisfiable)

$$\sigma(C \cup D) = 1 \quad \Rightarrow \quad \exists \sigma' \quad \text{with} \quad \sigma'(C \cup \{v\}) = \sigma'(D \cup \{\neg v\}) = 1$$

Proof.

with out loss of generality pick $c \in C$ with $\sigma(c) = 1$

define $\quad \sigma'(x) = \begin{cases} 0 & \text{if } x = v \\ \sigma(x) & \text{else} \end{cases}$

since v and $\neg v$ do not occur in C, we still have $\sigma'(C) = 1$ and thus $\sigma'(C \cup \{v\}) = 1$

by definition $\sigma'(\neg v) = 1$ and thus $\sigma'(D \cup \{\neg v\}) = 1$

q.e.d.

Resolution Based DP

Idea: use resolution to *existentially* quantify out variables

1. if empty clause found then terminate with result **unsatisfiable**

2. find variables which only occur in one phase (only positive or negative)

3. remove all clauses in which these variables occur

4. if no clause left then terminate with result **satisfiable**

5. choose x as one of the remaining variables with occurrences in both phases

6. add results of all possible resolutions on this variable

7. remove all trivial clauses and all clauses in which x occurs

8. continue with **1.**

Example for Resolution DP

check whether XOR is weaker than OR, i.e. validity of:

$$a \vee b \;\rightarrow\; (a \oplus b)$$

which is equivalent to unsatisfiability of the negation:

$$(a \vee b) \;\wedge\; \neg(a \oplus b)$$

since negation of XOR is XNOR (equivalence):

$$(a \vee b) \;\wedge\; (a \leftrightarrow b)$$

we end up checking the following CNF for satisfiability:

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

Example for Resolution DP cont.

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

initially we can skip **1.** - **4.** of the algorithm and choose $x = b$ in **5.**

in **6.** we resolve $(\neg a \vee b)$ with $(a \vee \neg b)$ and $(a \vee b)$ with $(a \vee \neg b)$ both on b

and add the results $(a \vee \neg a)$ and $(a \vee a)$:

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

the trivial clause $(a \vee \neg a)$ and clauses with ocurrences of b are removed:

$$(a \vee a)$$

in **2.** we find a to occur only positive and in **3.** the remaining clause is removed

the test in **4.** succeeds and the CNF turns out to be **satisfiable**

(thus the original formula is invalid – not a tautology)

Correctness of Resolution Based DP

Proof. in three steps:

(A) show that termination criteria are correct

(B) each transformation preserves satisfiability

(C) each transformation preserves unsatisfiability

Ad (A):

an empty clause is an empty disjunction, which is unsatisfiable

if literals occur only in one phase assign those to $1 \Rightarrow$ all clauses satisfied

Correctness of Resolution Based DP Part (B)

CNF transformations preserve satisfiability:

removing a clause does not change satisfiability

thus only adding clauses could potentially not preserve satisfiability

the only clauses added are the results of resolution

correctness of resolution rule shows:

if the original CNF is satisfiable, then the added clause are satisfiable

(even with the same satisfying assignment)

Correctness of Resolution Based DP Part (C)

CNF transformations preserve unsatisfiability:

adding a clause does not change unsatisfiability

thus only removing clauses could potentially not preserve unsatisfiability

trivial clauses $(v \vee \neg v \vee \ldots)$ are always valid and can be removed

let f be the CNF after removing all trivial clauses (in step **7.**)

let g be the CNF after removing all clauses in which x occurs (after step **7.**)

we need to show (f unsat $\Rightarrow$ g unsat), or equivalently (g sat $\Rightarrow$ f sat)

the latter can be proven as the completeness proof for the resolution rule

(see next slide)

Correctness of Resolution Based DP Part (C) cont.

If we interpret $\cup$ as disjunction and clauses as formulae, then

$$(C_1 \vee x) \wedge \ldots \wedge (C_k \vee x) \quad \wedge \quad (D_1 \vee \neg x) \wedge \ldots \wedge (D_l \vee \neg x)$$

is, via distributivity law, equivalent to

$$(\underbrace{(C_1 \wedge \ldots \wedge C_k)}_{C} \vee x) \quad \wedge \quad (\underbrace{(D_1 \wedge \ldots \wedge D_l)}_{D} \vee \neg x)$$

and the same proof applies as for the completeness of the resolution rule.

Note: just using the completeness of the resolution rule alone does not work, since those σ' derived for multiple resolutions are formally allowed to assign different values for the resolution variable.

Problems with Resolution Based DP

- if variables have many occurences, then many resolutions are necessary
- in the worst x and $\neg x$ occur in half of the clauses ...
- ... then the number of clauses increases quadratically
- clauses become longer and longer
- unfortunately in real world examples the CNF explodes

  (we will latter see how BDDs can be used to overcome some of these problems)
- How to obtain the satisfying assignment efficiently (counter example)?

Second version of DP

- resolution based version often called DP, second version DPLL

  (DP after [DavisPutnam60] and DPLL after [DavisLogemannLoveland62])
- it eliminates variables through case analysis: time vs space
- only *unit resolution* used (also called *boolean constraint propagation*)
- case analysis is on-the-fly:

  cases are not elaborated in a predefined fixed order, but ...

  ... only remaining crucial cases have to be considered
- allows sophisticated optimizations

Unit-Resolution

a *unit clause* is a clause with a single literal

in CNF a unit clause forces its literal to be assigned to 1

unit resolution is an application of resolution, where one clause is a unit clause

also called *boolean constraint propagation*

Unit-Resolution Rule

$$\frac{C \cup \{\neg l\} \qquad \{l\}}{C} \qquad \{l, \neg l\} \cap C = \emptyset$$

here we identify $\neg\neg v$ with v for all variables v.

Unit-Resolution Example

check whether XNOR is weaker than AND, i.e. validity of:

$$a \wedge b \;\rightarrow\; (a \leftrightarrow b)$$

which is equivalent to unsatisfiability of the CNF (exercise)

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

adding clause obtained from unit resolution on a results in

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

removing clauses containing a or $\neg a$

$$b \;\wedge\; (\neg b)$$

unit resolution on b results in an empty clause and we conclude unsatisfiability.

Ad: Unit Resolution

- if unit resolution produces a unit, e.g. resolving $(a \vee \neg b)$ with b produces a, continue unit resolution with this new unit

- often this repeated application of unit resolution is also called unit resolution

- unit resolution + removal of subsumed clauses never increases size of CNF

  $$C \text{ subsumes } D \quad :\Leftrightarrow \quad C \subseteq D$$

  a unit(-clause) l subsumes all clauses in which l occurs in the same phase

- *boolean constraint propagation* (BCP): given a unit l, remove all clauses in which l occurs in the same phase, and remove all literals $\neg l$ in clauses, where it occurs in the opposite phase (the latter is unit resolution)

Basic DPLL Algorithm

1. apply repeated unit resolution and removal of all subsumed clauses (BCP)
2. if empty clause found then return **unsatisfiable**
3. find variables which only occur in one phase (only positive or negative)
4. remove all clauses in which these variables occur (pure literal rule)
5. if no clause left then return **satisfiable**
6. choose x as one of the remaining variables with occurrences in both phases
7. recursively call DPLL on current CNF with the unit clause $\{x\}$ added
8. recursively call DPLL on current CNF with the unit clause $\{\neg x\}$ added
9. if one of the recursive calls returns **satisfiable** return **satisfiable**
10. otherwise return **unsatisfiable**

DPLL Example

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

Skip **1.** - **6.**, and choose $x = a$. First recursive call:

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

unit resolution on a and removal of subsumed clauses gives

$b \wedge (\neg b)$

BCP gives empty clause, return **unsatisfiable**. Second recursive call:

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

BCP gives $\neg b$, only positive recurrence of b left, return **satisfiable**

(satisfying assignment $\{a \mapsto 0, b \mapsto 0\}$)

Expansion Theorem of Shannon

Theorem.

$$f(x) \quad \equiv \quad x \wedge f(1) \vee \overline{x} \wedge f(0)$$

Proof.

Let σ be an arbitrary assignment to variables in f including x

case $\sigma(x) = 0$:

$$\sigma(f(x)) = \sigma(f(0)) = \sigma(0 \wedge f(1) \vee 1 \wedge f(0)) = \sigma(x \wedge f(1) \vee \overline{x} \wedge f(0))$$

case $\sigma(x) = 1$:

$$\sigma(f(x)) = \sigma(f(1)) = \sigma(1 \wedge f(1) \vee 0 \wedge f(0)) = \sigma(x \wedge f(1) \vee \overline{x} \wedge f(0))$$

Correctness of Basic DPLL Algorithm

first observe: $x \wedge f(x)$ is satisfiable iff $x \wedge f(1)$ is satisfiable

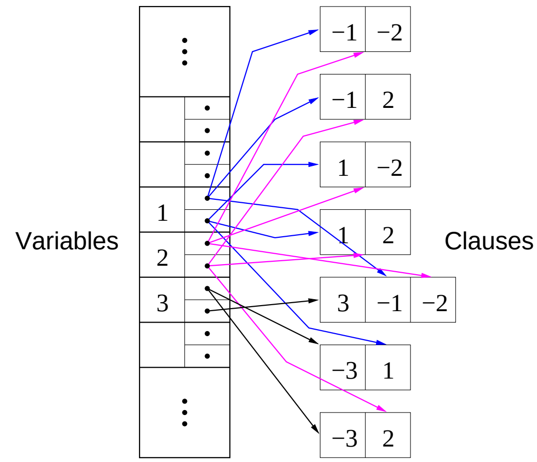
similarly, $\overline{x} \wedge f(x)$ is satisfiable iff $\overline{x} \wedge f(0)$ is satisfiable

then use expansion theorem of Shannon:

$f(x)$ satisfiable iff $\overline{x} \wedge f(0)$ or $x \wedge f(1)$ satisfiable iff $\overline{x} \wedge f(x)$ or $x \wedge f(x)$ satisfiable

rest follows along the lines of the the correctness proof for resolution based DP

Simple Data Structures in DP Implementation



BCP Implementation Details

- each variable is marked as *unassigned*, *false*, or *true* ($\{X, 0, 1\}$)

- no explicit resolution:
  - when a literal is assigned visit all clauses where its negation occurs
  - find those clauses which have all but one literal assigned to false
  - assign remaining non false literal to *true* and continue

- decision:
  - heuristically find a variable that is still unassigned
  - heuristically determine phase for assignment of this variable

More Implementation Details

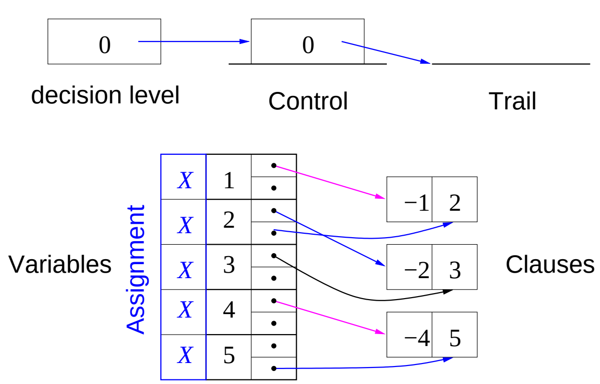
- *decision level* is the depth of recursive calls (= #nested decisions)

- the *trail* is a stack to remember order in which variables are assigned

- for each decision level the old trail height is saved on the *control stack*

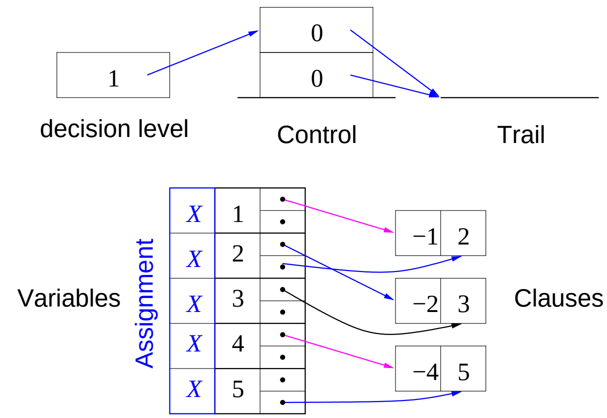
- undoing assignments in backtracking:
  - get old trail height from control stack
  - unassign all variables up to the old trail height

BCP Example



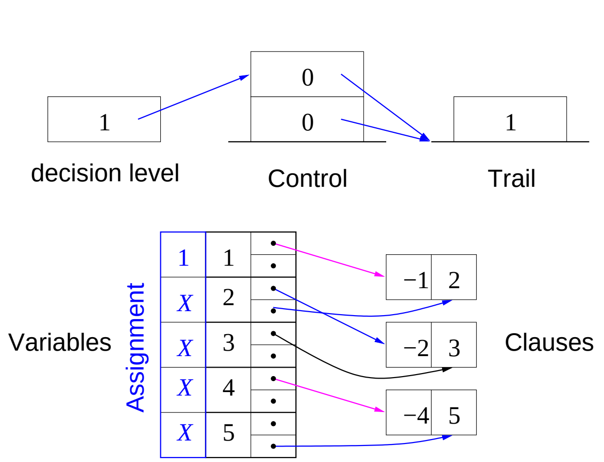
Example cont.

Decide



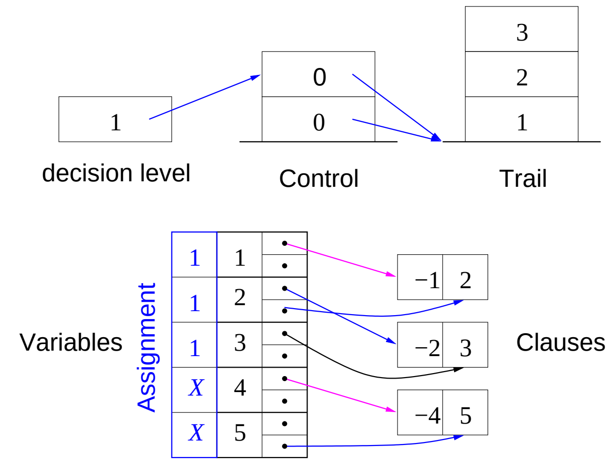
Example cont.

Assign



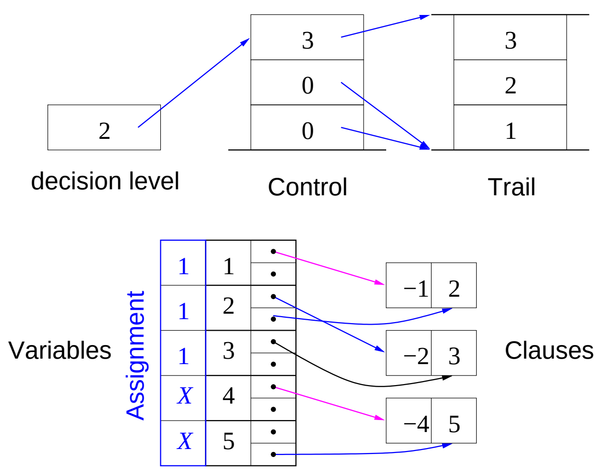
Example cont.

BCP



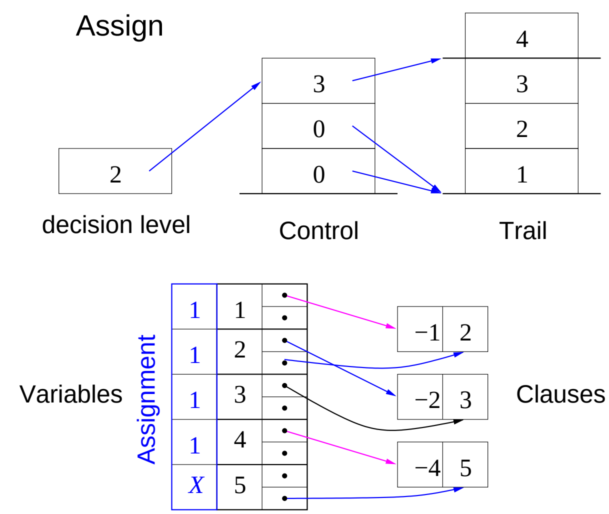
Example cont.

Decide

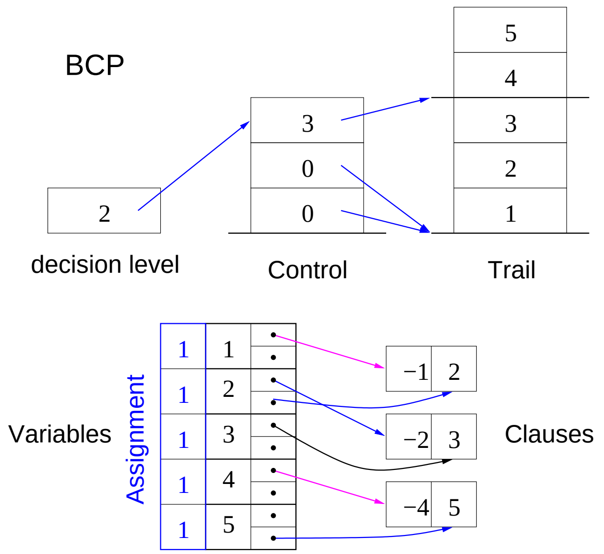


Example cont.

Assign



Example cont.



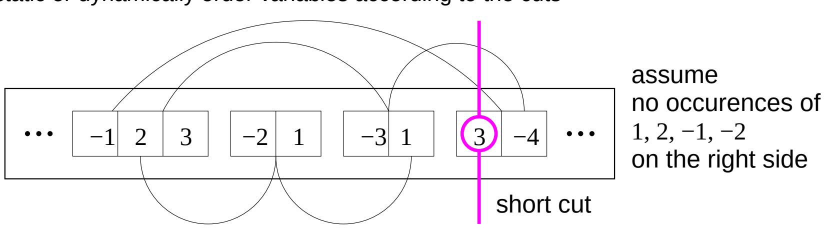
Decision Heuristics

- **static heuristics:**
  - one *linear* order determined before solver is started
  - usually quite fast, since only calculated once
  - can also use more expensive algorithms
- **dynamic heuristics**
  - typically calculated from number of occurences of literals (in unsatisfied clauses)
  - rather expensive, since it requires traversal of all clauses (or more expensive updates in BCP)
  - recently, *second order* dynamic heuristics (Chaff)

Cut Width Heuristics

- view CNF as a graph:

  clauses as nodes, edges between clauses with same variable
- a *cut* is a set of variables that splits the graph in two parts
- recursively find short cuts that cut of parts of the graph
- static or dynamically order variables according to the cuts



Cut Width Algorithm

```
int
sat (CNF cnf)
{
  SetOfVariables cut = generate_good_cut (cnf);
  CNF assignment, left, right;

  left = cut_off_left_part (cut, cnf);
  right = cut_off_right_part (cut, cnf);

  forall_assignments (assignment, cut)
  {
    if (sat (apply (assignment, left)) && sat (apply (assignment, right)))
      return 1;
  }

  return 0;
}
```

Cut Width Heuristics cont.

- resembles cuts in circuits when CNF is generated with Tseitin transformation
- ideally cuts have constant or logarithmic size . . .
  - for instance in tree like circuits
  - so the problem is *reconvergence*:
    the same signal / variable is used multiple times
- . . . then satisfiability actually becomes polynomial (see exercise)

CNF in Horn Form

A clause is called *positive* if it contains a positive literal.

A clause is called *negative* if all its literals are negative.

A clause is a *Horn* clause if contains at most one positive literal.

CNF is in *Horn Form* iff all clauses are Horn clause (Prolog without negation)

Order assignments point-wise: $\sigma \leq \sigma'$ iff $\sigma(x) \leq \sigma'(x)$ for all $x \in V$

Horn Form with only positive clauses has minimal satisfying assignment.

Minimal satisfying assignment is obtained by BCP (polynomial).

A Horn Form is satisfiable iff the minimal assignments of its positive part satisfies all its negative clauses as well.

DP and Horn Form

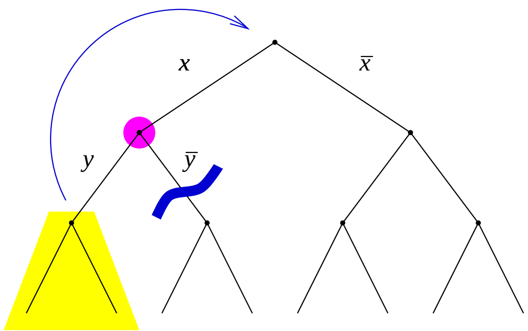
- CNF in Horn Form: use above specialized fast algorithm
- non Horn: split on literals which occurs positive in non Horn clauses
  - actually choose variable which occurs most often in such clauses
- this gradually transforms non Horn CNF into Horn Form
- main heuristic in SAT solver SATO
- **Note:** In general, BCP in DP prunes search space by avoiding assignments incompatible to minimal satisfying assingment for the Horn part of the CNF.

non Horn part of CNF	Horn part of CNF

Other popular Decision Heuristics

- Dynamic Largest Individual Sum (DLIS)
  - fastest dynamic first order heuristic (eg GRASP solver)
  - choose literal (variable + phase) which occurs most often
  - ignore satisfied clauses
  - requires explicit traversal of CNF (or more expensive BCP)
- look-forward heuristics (eg SATZ solver)
  - do trial assignments and BCP for all unassigned variables (both phases)
  - if BCP leads to conflict, force toggled assignment of current trial decision
  - skip trial assignments implied by previous trial assignments
    (removes a factor of $|V|$ from the runtime of one decision search)

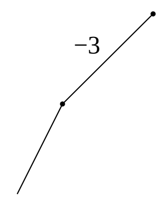
Backjumping



If y has never been used to derive a conflict, then skip $\overline{y}$ case.

Immediately *jump back* to the $\overline{x}$ case – assuming x was used.

Backjumping Example



~~(−3 1)~~

~~(−3 2)~~

$(-1\ -2\ \cancel{3})$

$(-1\ -2)$

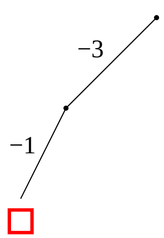
$(-1\ 2)$

$(1\ -2)$

$(1\ 2)$

Split on -3 first (bad decision).

Backjumping Example



~~(−3 1)~~

~~(−3 2)~~

~~$(-1\ -2\ \cancel{3})$~~

~~(−1 −2)~~

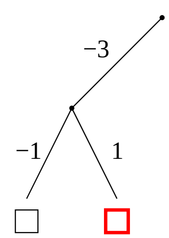
~~(−1 2)~~

$(\cancel{1}\ -2)$

$(\cancel{1}\ 2)$

Split on -1 and get first conflict.

Backjumping Example



~~(−3 1)~~

~~(−3 2)~~

$(\cancel{-1}\ -2\ \cancel{3})$

$(\cancel{-1}\ -2)$

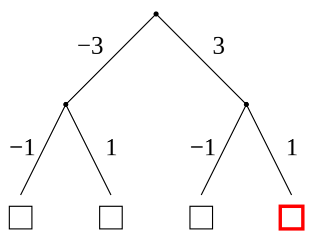
$(\cancel{-1}\ 2)$

~~(1 −2)~~

~~(1 2)~~

Regularly backtrack and assign 1 to get second conflict.

Backjumping Example



$\cancel{(-3\ 1)}$

$\cancel{(-3\ 2)}$

$(\cancel{-1}\ -2\ \cancel{3})$

$(\cancel{-1}\ -2)$

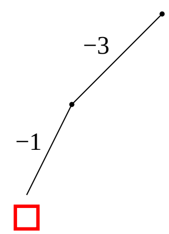
$(\cancel{-1}\ 2)$

$\cancel{(1\ -2)}$

$\cancel{(1\ 2)}$

Backtrack to root, assign 3 and derive same conflicts.

Backjumping Example



$\cancel{(-3\ 1)}$

$\cancel{(-3\ 2)}$

$\cancel{(-1\ -2\ \cancel{3})}$

$\cancel{(-1\ -2)}$

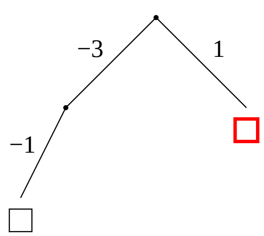
$\cancel{(-1\ 2)}$

$(\cancel{1}\ -2)$

$(\cancel{1}\ 2)$

Assignment -3 does not contribute to conflict.

Backjumping Example



$\cancel{(-3\ 1)}$

$(-3\ 2)$

$(\cancel{-1}\ -2\ 3)$

$(\cancel{-1}\ -2)$

$(\cancel{-1}\ 2)$

$\cancel{(1\ -2)}$

$\cancel{(1\ 2)}$

So just *backjump* to root before assigning 1.

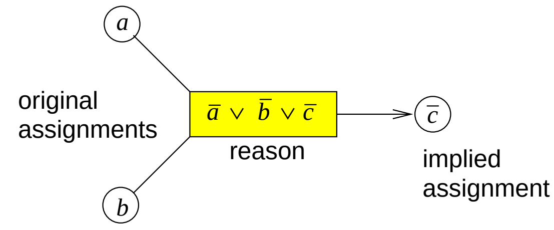
Backjumping

- backjumping helps to *recover* from bad decisions
  - bad decisions are those that do not contribute to conflicts
  - without backjumping same conflicts are generated in second branch
  - with backjumping the second branch of bad decisions is just skipped
- particularly useful for unsatisfiable instances
  - in satisfiable instances good decisions will guide us to the solution
- with backjumping many bad decisions increase search space roughly quadratically instead of exponentially with the number of bad decisions

Implication Graph

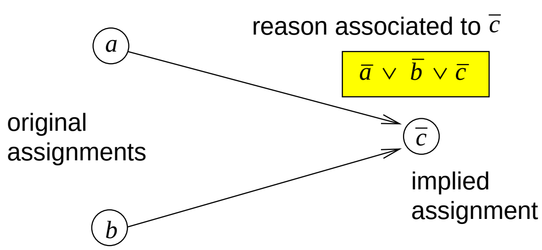
- the implication graph maps inputs to the result of resolutions
- backward from the empty clause all contributing clauses can be found
- the variables in the contributing clauses are contributing to the conflict
- important optimization, since we only use unit resolution
  - generate graph only for resolutions that result in unit clauses
  - the assignment of a variable is result of a decision or a unit resolution
  - therefore the graph can be represented by saving the *reasons* for assignments with each assigned variable

General Implication Graph as Hyper-Graph



(edges of directed hyper graphs may have multiple source and target nodes)

Optimized Implication Graph for Unit Resolution in DP



- graph becomes an ordinary (non hyper) directed graph
- simplifies implementation:
  - store a pointer to the reason clause with each assigned variable
  - decision variables just have a null pointer as reason
  - decisions are the roots of the graph

Learning

- can we *learn* more from a conflict?
  - backjumping does not *fully* avoid the occurrence of the same conflict
  - the same (partial) assignments may generate the same conflict
- generate *conflict clauses* and add them to CNF
  - the literals contributing to a conflict form a partial assignment
  - this partial assignment is just a conjunction of literals
  - its negation is a clause (implied by the original CNF)
  - adding this clause avoids this partial assignment to happen again

Conflict Driven Backtracking/Backjumping

- observation: current decision always contributes to conflict
  - otherwise BCP would have generated conflict one decision level lower
  - conflict clause has (exactly one) literal assigned on current decision level
- instead of backtracking
  - generate and add conflict clause
  - undo assignments as long conflict clause is empty or unit clause
    (in case conflict clause is the empty clause conclude unsatisfiability)
  - resulting assignment from unit clause is called *conflict driven assignment*

CNF for following Examples


```
-3 1 2 0
3 -1 0
3 -2 0
-4 -1 0
-4 -2 0
-3 4 0
3 -4 0
-3 5 6 0
3 -5 0
3 -6 0
4 5 6 0
```

We use a version of the DIMACS format.

Variables are represented as positive integers.

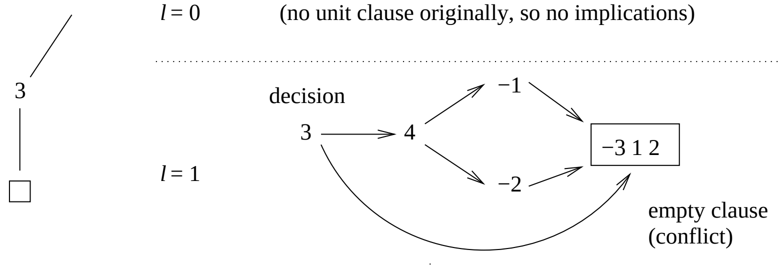
Integers represent literals.

Subtraction means negation.

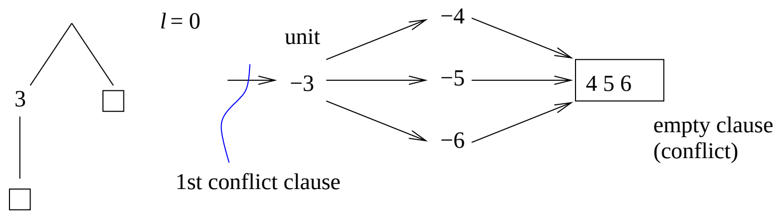
A clause is a zero terminated list of integers.

CNF has a good cut made of variables 3 and 4 (cf Exercise 4 + 5).
(but we are going to apply DP with learning to it)

DP with Learning Run 1 (3 as 1st decision)

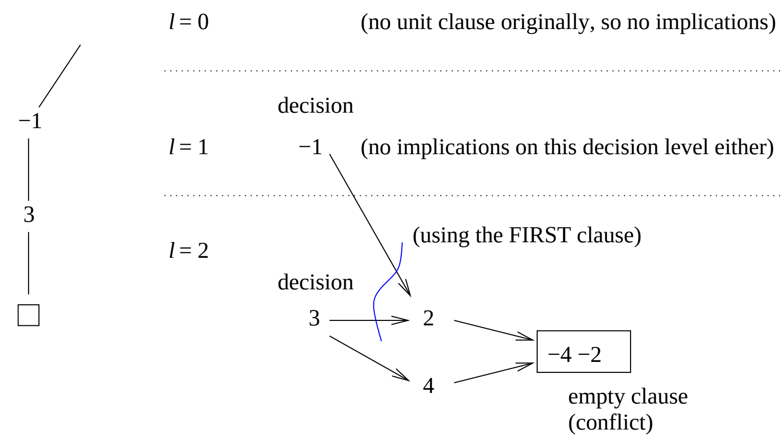


unit clause −3 is generated as learned clause and we backtrackt to $l = 0$



since −3 has a real unit clause as reason, an empty conflict clause is learned

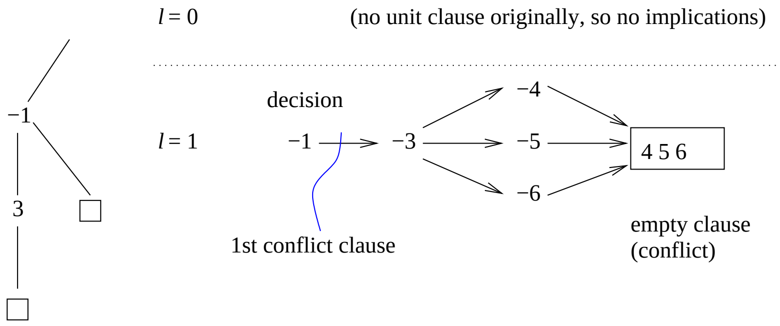
DP with Learning Run 2 Fig. 1 (-1, 3 as decision order)



since FIRST clause was used to derive 2, conflict clause is (1 −3)

backtrack to $l = 1$ (smallest level for which conflict clause is a unit clause)

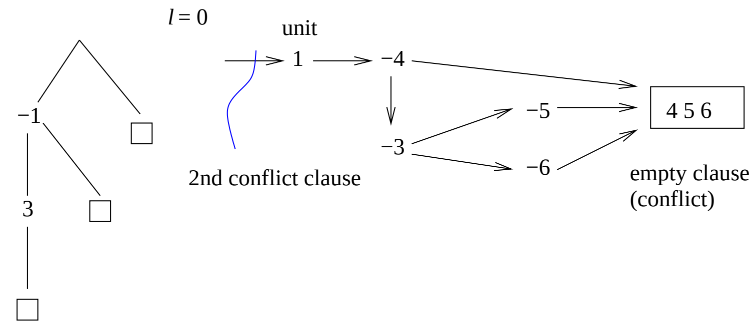
DP with Learning Run 2 Fig. 2 (-1, 3 as decision order)



learned conflict clause is the unit clause 1

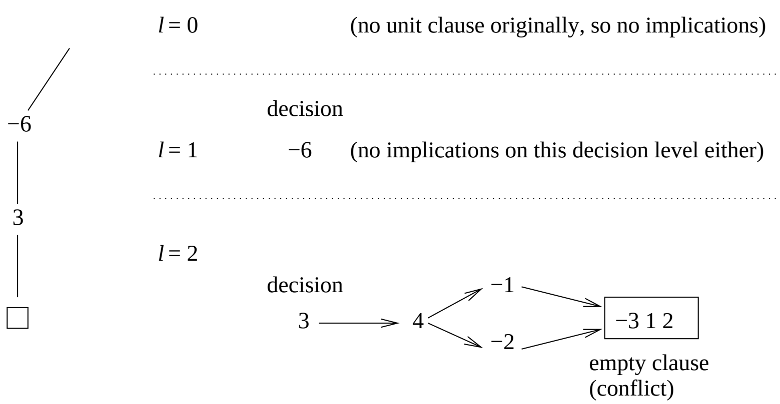
backtrack to decision level $l = 0$

DP with Learning Run 2 Fig. 3 (-1, 3 as decision order)



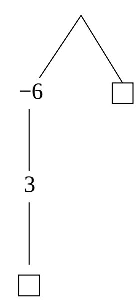
since the learned clause is the empty clause, conclude unsatisfiability

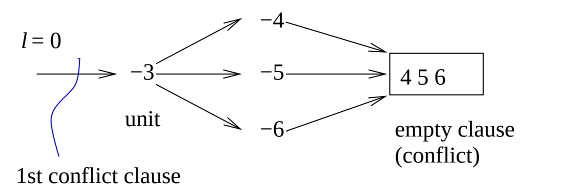
DP with Learning Run 3 Fig. 1 (-6, 3 as decision order)



learn the unit clause −3 and BACKJUMP to decision level $l = 0$

DP with Learning Run 3 Fig. 1 (-6, 3 as decision order)





finally the empty clause is derived which proves unsatisfiability

Toplevel Loop in DP with Learning


```
int
sat (Solver solver)
{
  Clause conflict;

  for (;;)
    {
      if (bcp_queue_is_empty (solver) && !decide (solver))
        return SATISFIABLE;

      conflict = deduce (solver);

      if (conflict && !backtrack (solver, conflict))
        return UNSATISFIABLE;
    }
}
```

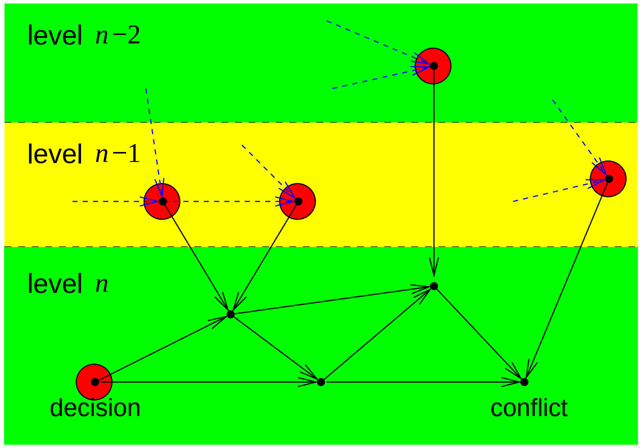

Backtracking in DP with Learning


```
int
backtrack (Solver solver, Clause conflict)
{
  Clause learned_clause; Assignment assignment; int new_level;

  if (decision_level(solver) == 0)
    return 0;

  analyze (solver, conflict);
  learned_clause = add (solver);

  assignment = drive (solver, learned_clause);
  enqueue_bcp_queue (solver, assignment);

  new_level = jump (solver, learned_clause);
  undo (solver, new_level);

  return 1;
}
```


Learning as Resolution

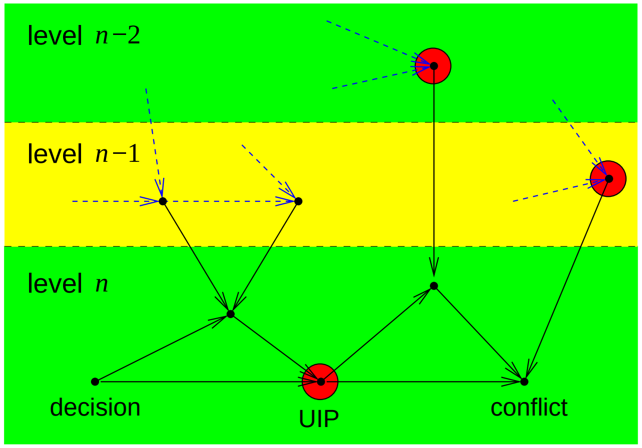
- conflict clause: obtained by backward resolving empty clause with reasons
  - start at clause which has all its literals assigned to false
  - resolve one of the false literals with its reason
  - invariant: result still has all its literals assigned to false
  - continue until user defined size is reached
- gives a nice correspondence between resolution and learning in DP
  - allows to generate a resolution proof from a DP run
  - implemented in RELSAT solver

Conflict Clauses as Cuts in the Implication Graph



a simple cut always exists: set of roots (decisions) contributing to the conflict

Unique Implication Points (UIP)



UIP = *articulation point* in graph decomposition into biconnected components
(simply a node which, if removed, would disconnect two parts of the graph)

Detection of UIPs

- can be found by graph traversal in the order of made assignments
- *trail* respects this order
- traverse reasons of variables on trail starting with conflict
- count "open paths"
  (initially size of clause with only false literals)
- if all paths converged at one node, then UIP is found
- decision of current decision level is a UIP and thus a *sentinel*

Further Options in Using UIPs

- assume a non decision UIP is found
- this UIP is part of a potential cut
- graph traversal may stop (everything *behind* the UIP is ignored)
- negation of the UIP literal constitutes the conflict driven assignment
- may start new clause generation (UIP replaces conflict)
  - each conflict may generate multiple learned clauses
  - however, using only the first UIP encountered seems to work best

More Heuristics for Conflict Clauses Generation

- intuitively is is important to localize the search (cf cutwidth heuristics)
- cuts for learned clauses may only include UIPs of current decision level
- on lower decision levels an arbitrary cut can be chosen
- multiple alternatives
  - include all the roots contributing to the conflict
  - find minimal cut (heuristically)
  - **cut off at first literal of lower decision level** (works best)

Second Order Dynamic Decision Heuristics: VSIDS

- “second order” because it involves statistics about the search
- Variable State Independent Decaying Sum (VSIDS) decision heuristic (implemented in CHAFF and LIMMAT solver)
- VSIDS just counts the occurrences of a literals in conflict clauses
- literal with maximal count (score) is chosen
- score is multiple by a factor $f < 1$ after a certain number of conflicts occurred (this is the “decaying” part and also called *rescoring*)
- emphasizes (negation of) literals contributing recently to conflicts (**localization**)

BERKMIN’s Dynamic Second Order Heuristics

- observation:
  - recently added conflict clauses contain all the good variables of VSIDS
  - the order of those clauses is not used in VSIDS
- basic idea:
  - simply try to satisfy recently learned clauses first
  - use VSIDS to chose the decision variable for one clause
  - if all learned clauses are satisfied use other heuristics
  - intuitively obtains another order of localization (no proofs yet)
- results are mixed, but in general sligthly more robust than just VSIDS

Restarts

- for satisfiable instances the solver may get stuck in the unsatisfiable part
  - even if the search space contains a large satisfiable part
- often it is a good strategy to abandon the current search and restart
  - restart after the number of decisions reached a *restart limit*
- avoid to run into the same dead end
  - by randomization (either on the decision variable or its phase)
  - and/or just keep all the learned clauses
- for completeness dynamically increase restart limit

Other Types of Learning

- similar to look-ahead heuristics: polynomially bounded search
  - may be recursively applied (however, is often too expensive)
- Stålmarck’s Method
  - works on triplets (intermediate form of the Tseitin transformation):
    $x = (a \wedge b)$, $y = (c \vee d)$, $z = (e \oplus f)$ etc.
  - generalization of BCP to (in)equalities between variables
  - **test rule** splits on the two values of a variable
- Recursive Learning (Kunz & Pradhan)
  - works on circuit structure (derives implications)
  - splits on different ways to *justify* a certain variable value

Stålmarck's Method

1. BCP over (in)equalities: $\dfrac{x = y \quad z = (x \oplus y)}{z = 0}$ $\quad \dfrac{x = 0 \quad z = (x \vee y)}{z = y}$ etc.

2. structural rules: $\dfrac{x = (a \vee b) \quad y = (a \vee b)}{x = y}$ etc.

3. test rule: $$\dfrac{\begin{array}{cc} \{x = 0\} \cup E & \{x = 1\} \cup E \\ \Downarrow & \Downarrow \\ E_0 \cup E & E_1 \cup E \end{array}}{(E_0 \cap E_1) \cup E}$$

Assume $x = 0$, BCP and derive (in)equalities E_0, then assume $x = 1$, BCP and derive (in)equalities E_1. The intersection of E_0 and E_1 contains the (in)equalities valid in *any* case.

Stålmarck's Method Recursively

$$\dfrac{\begin{array}{cc} \begin{array}{c} x = 0 \\ \Downarrow \\ \dfrac{\begin{array}{cc} y = 0 & y = 1 \\ \Downarrow & \Downarrow \\ E_{00} & E_{01} \end{array}}{E_0} \end{array} & \begin{array}{c} x = 1 \\ \Downarrow \\ \dfrac{\begin{array}{cc} y = 0 & y = 1 \\ \Downarrow & \Downarrow \\ E_{10} & E_{11} \end{array}}{E_1} \end{array} \end{array}}{E}$$

(we do not show the (in)equalities that do not change)

Stålmarck's Method Summary

- recursive application
  - depth of recursion bounded by number of variables
  - complete procedures (determines satisfiability or unsatisfiability)
  - for a fixed (constant) recursion depth k polynomial!
- k-saturation:
  - apply split rule on recursively up to depth k on all variables
  - 0-saturation: apply all rules accept test rule (just BCP: linear)
  - 1-saturation: apply test rule (not recursively) for all variables (until no new (in)equalities can be derived)

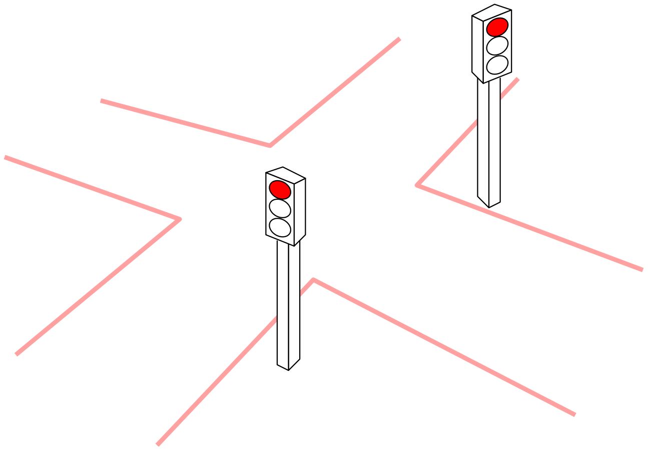
Model Checking

- check **algorithmically** temporal / sequential properties
  - systems are originally **finite state**
  - simple model: finite state automaton
- **comparison** of automata can be seen as model checking
  - check that the output streams of two finite state systems "match"
  - process algebra: simulation and bisimulation checking
- **temporal logics** as specification mechanism
  - safety, liveness and more general temporal operators, fairness

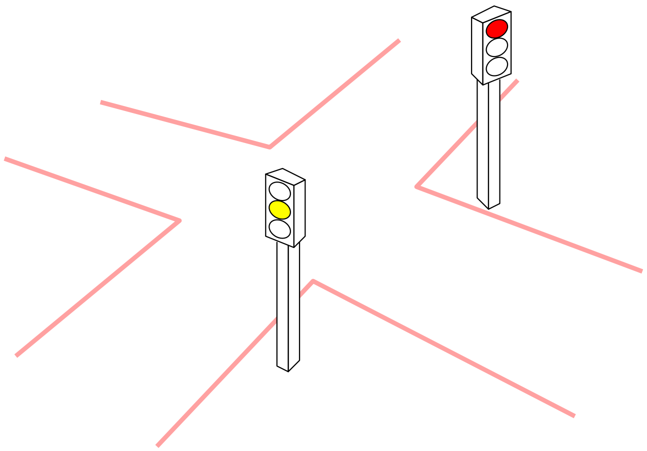
Model Checking for Infinite Systems

- fixpoint algorithms with symbolic representations:
  - timed automata (clocks)
  - hybrid automata (differential equations)
  - termination guaranteed if finite quotient structure exists
- simply run model checker *for some time*, e.g. Java Pathfinder
- run time verification
  1. example: add checker synthesized from temporal spec
  2. example: run all schedules for one test case
- check programs (incl. loops and recursion) over finite domains, e.g. SLAM

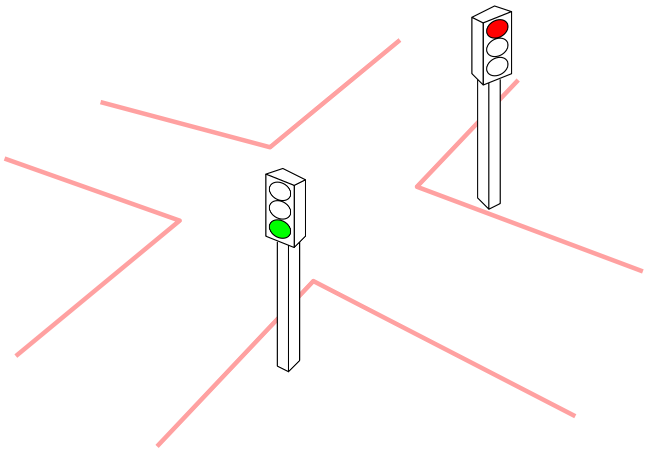
Traffic Light Controller (TLC)

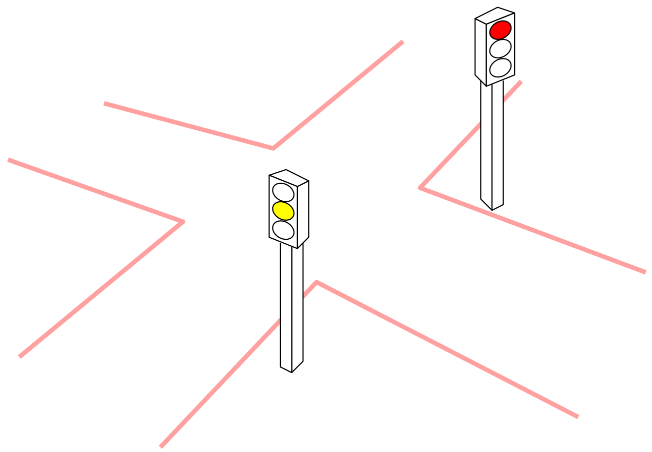


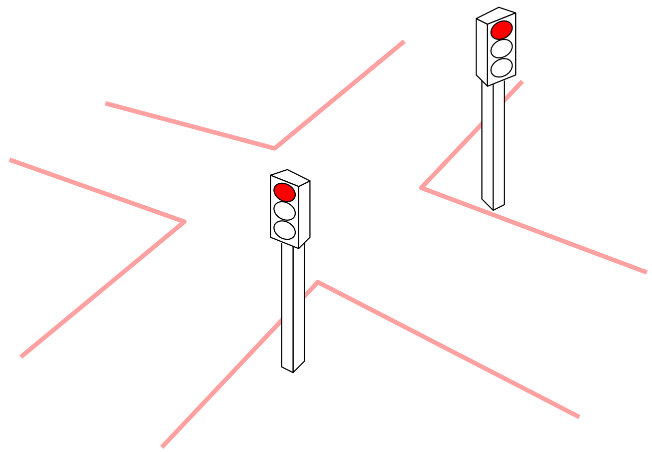
Traffic Light Controller (TLC)

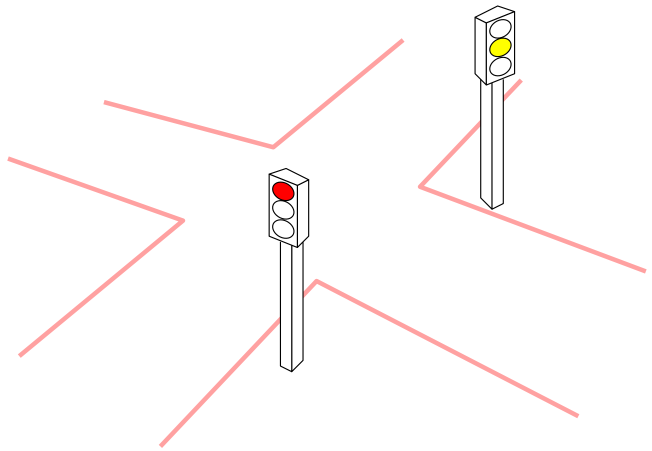


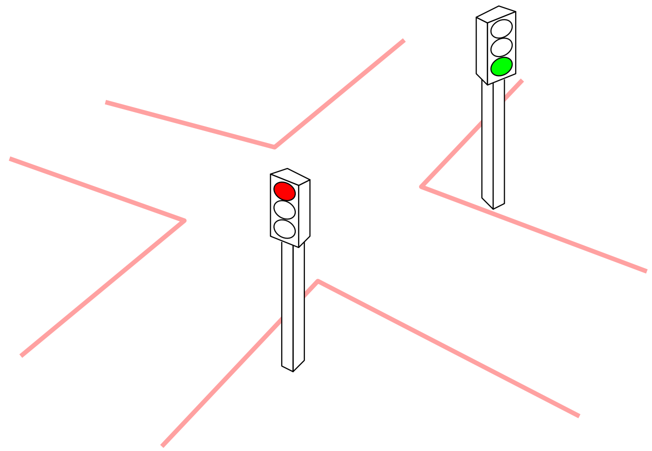
Traffic Light Controller (TLC)



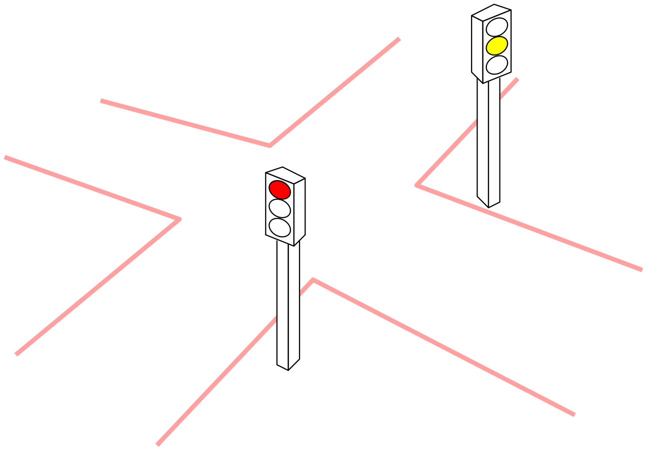




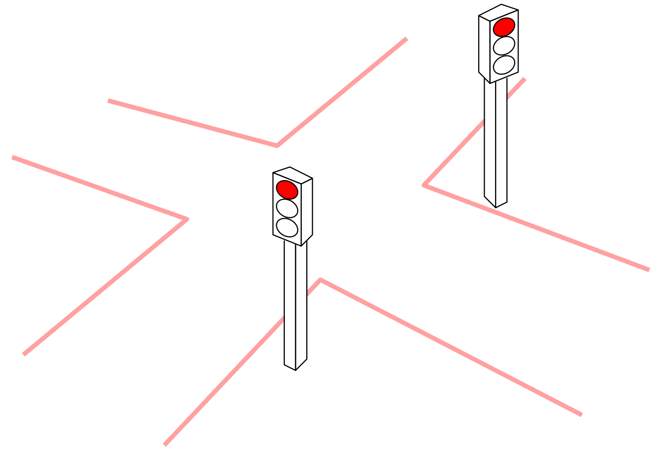




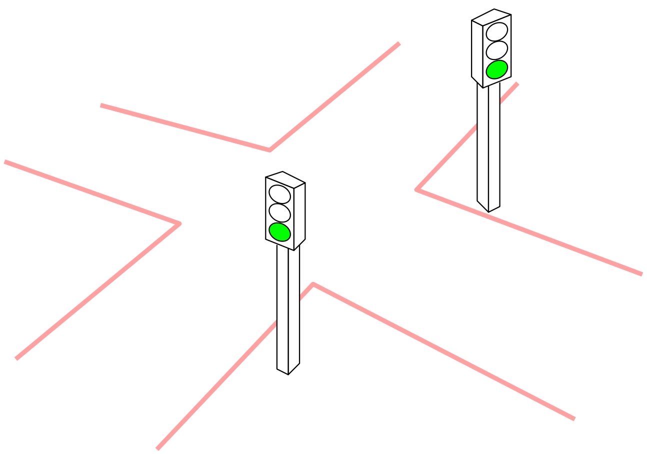
Traffic Light Controller (TLC)



Traffic Light Controller (TLC)



Safety



the two traffic lights should never show a green light at the same time

State Space

- state space is the set of assignments to variables of the system
  - state space is finite if the **range** of variables is finite
  - this notion works for inifinite state spaces as well
- TLC example:
  - single assignment $\sigma : \{southnorth, eastwest\} \to \{green, yellow, red\}$
  - set of assignments is isomorphic to $\{green, yellow, red\}^2$
  - eg state space is isomorphic to the crossproduct of variable ranges
- not all states are reachable: $(green, green)$

Safety

- safety properties specify **invariants** of the system
- simple generic algorithm for checking safety properties:
  1. iteratively generate all reachable states
  2. check for violation of invariant for newly reached states
  3. terminate if all newly reached states can be found
- compare with **assertions**
  - used in run time checking: `assert` in C and VHDL
  - contract checking: `require`, `ensure`, etc. in Eiffel

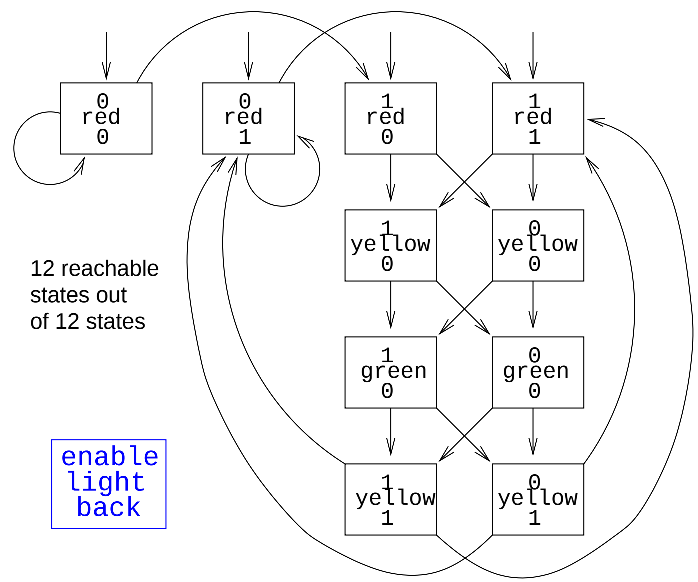
Unsafe TLC in SMV


```
MODULE trafficlight (enable)
VAR
  light : { green, yellow, red };
  back : boolean;
ASSIGN
  init (light) := red;
  next (light) :=
    case
      light = red & !enable : red;
      light = red & enable : yellow;
      light = yellow & back : red;
      light = yellow & !back : green;
      1 : yellow;
    esac;
  next (back) :=
    case
      light = red & enable : 0;
      light = green : 1;
      1 : back;
    esac;
MODULE main
VAR
  southnorth : trafficlight (1);
  eastwest : trafficlight (1);
SPEC
  AG !(southnorth.light = green & eastwest.light = green)
```


SMV

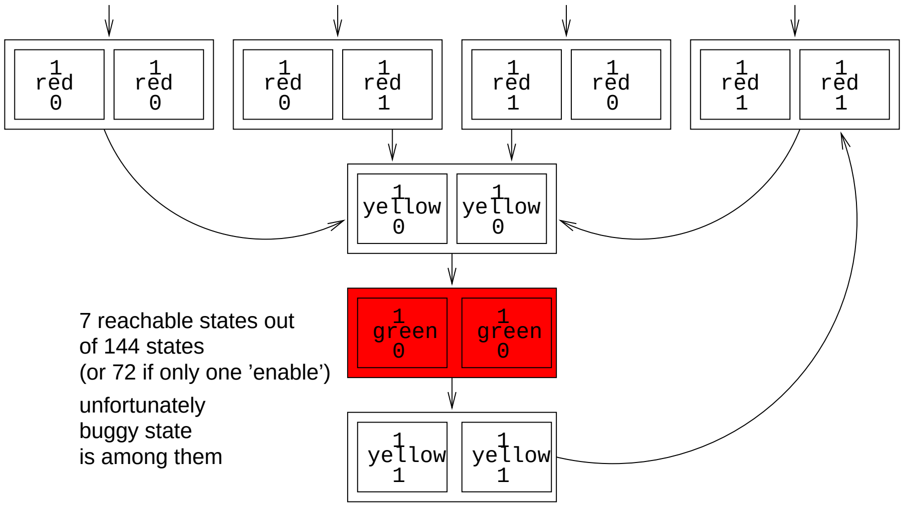
- symbolic model checker implemented by Ken McMillan at CMU (early 90'ies)
- input language: finite models + temporal specification
- hierarchical description, similar to hardware description language (HDL)
- integer and enumeration types, arithmetic operations
- heavily relies on the data structure Binary Decision Diagrams (BDDs)

Reachable States of One Traffic Light



12 reachable states out of 12 states

Reachable States of Unsafe TLC



7 reachable states out of 144 states (or 72 if only one 'enable')

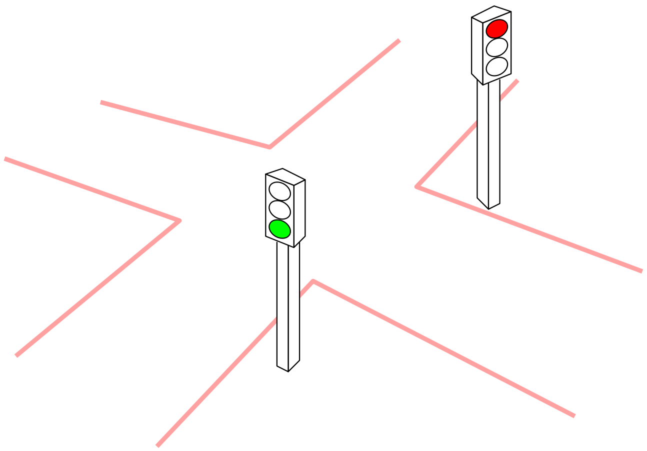
unfortunately buggy state is among them

Safe TLC in SMV


```
MODULE main
VAR
  turn : { ew, sn };
  southnorth : trafficlight (enablesouthnorth);
  eastwest : trafficlight (enableeastwest);
DEFINE
  enableeastwest := southnorth.light = red & turn = ew;
  enablesouthnorth := eastwest.light = red & turn = sn;
SPEC
  AG !(southnorth.light = green & eastwest.light = green)
```

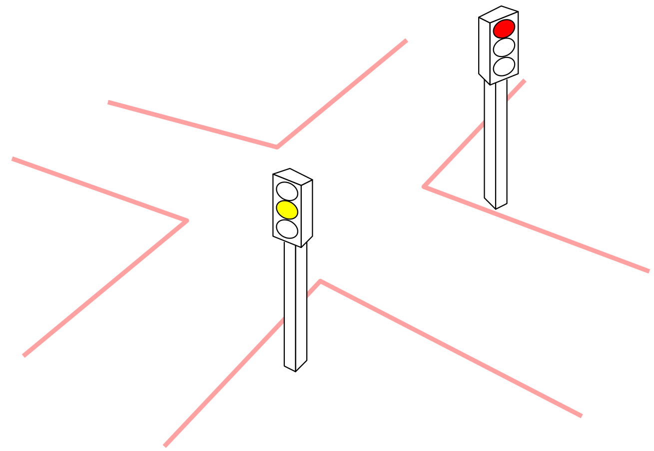
idea: disable traffic light as long the other is not red and its not the others turn

Liveness



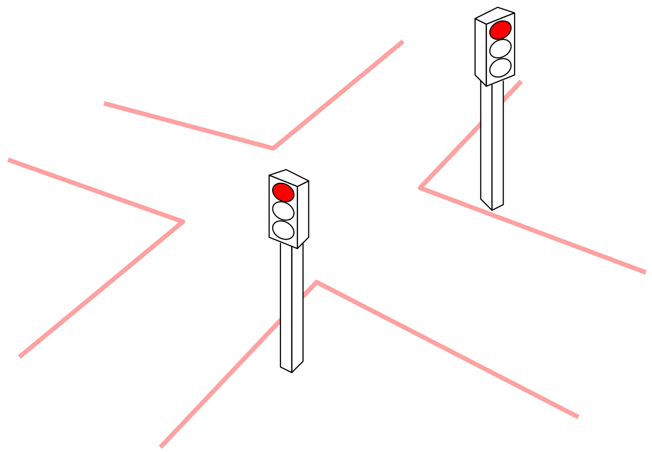
traffic lights showing red should eventually show green

Liveness



traffic lights showing red should eventually show green

Liveness



traffic lights showing red should eventually show green

Boolean Encoding

- compilation of finite model into pure propositional domain
- first step is to flatten the hierarchy
  - recursive instantiation of all submodules
  - name and parameter substitution
  - may increase program size exponentially
- second step is to encode variables with boolean variables

light		light@1	light@0
green	$\mapsto$	0	0
yellow	$\mapsto$	0	1
red	$\mapsto$	1	0

Result of Boolean Encoding

- initial state predicate I represented as boolean formula

  `!eastwest.light@0 & eastwest.light@1`

  (equivalent to `init(eastwest.light) := red`)

- transition relation T represented as boolean formula
- encoding of atomic predicates p as boolean formulae

  `!eastwest.light@1 & !eastwest.light@0`

  (equivalent to `eastwest.light != green`)

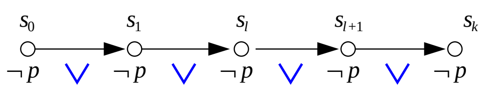
Bounded Model Checking

[BiereCimattiClarkeZhu99]

- uses SAT for model checking
  - historically not the first symbolic model checking approach
  - scales better than original BDD based techniques
- mostly incomplete in practice
  - validity of a formula can often not be proven
  - focus on counter example generation
  - only counter example up to certain length (the bound k) are searched

Bounded Model Checking Safety

checking safety property $\mathbf{G}p$ for a bound k as SAT problem:

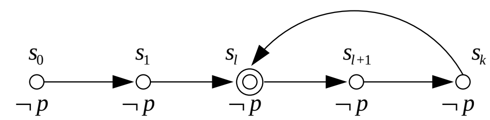


$$I(s_0) \wedge T(s_0,s_1) \wedge \cdots \wedge T(s_{k-1},s_k) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

check occurrence of $\neg p$ in the first k states

Bounded Model Checking Liveness

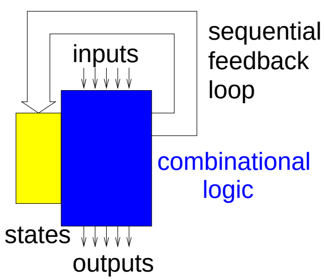
generic counter example trace of length k for liveness $\mathbf{F}p$



$$I(s_0) \wedge T(s_0,s_1) \wedge \cdots \wedge T(s_k,s_{k+1}) \wedge \bigvee_{l=0}^{k} s_l = s_{k+1} \wedge \bigwedge_{i=0}^{k} \neg p(s_i)$$

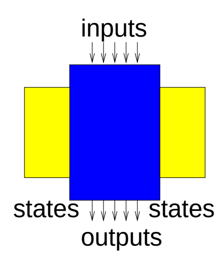
(however we recently showed that liveness can always be reformulated as safety [BiereArthoSchuppan02])

Time Frame Expansion in HW



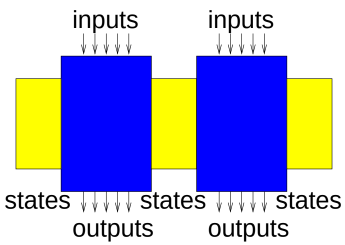
sequential circuit

Time Frame Expansion in HW



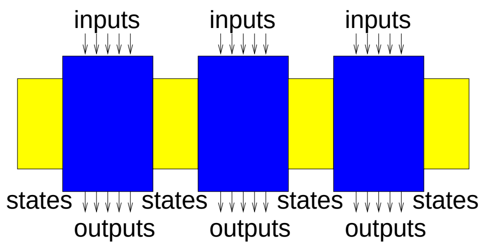
break sequential loop

Time Frame Expansion in HW



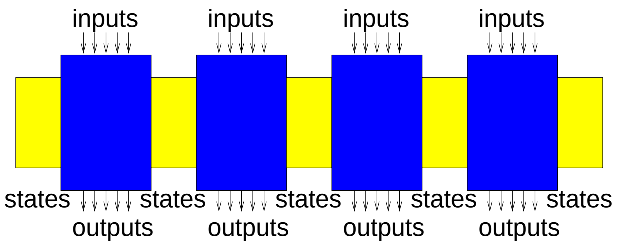
added 1st copy

Time Frame Expansion in HW



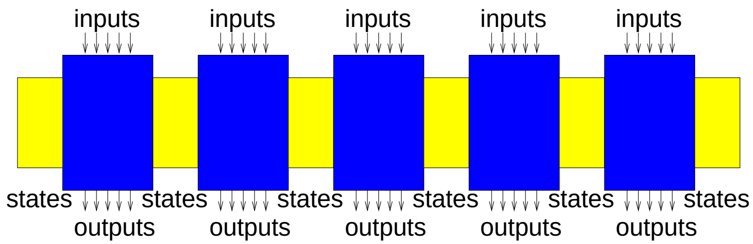
added 2nd copy

Time Frame Expansion in HW



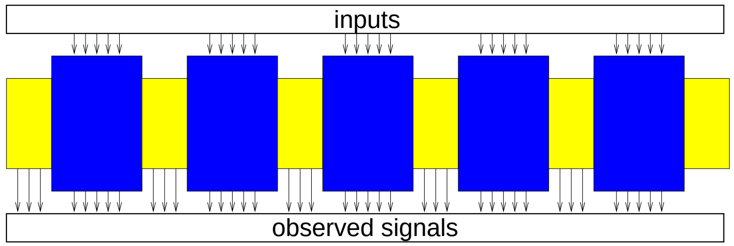
added 3rd copy

Time Frame Expansion in HW

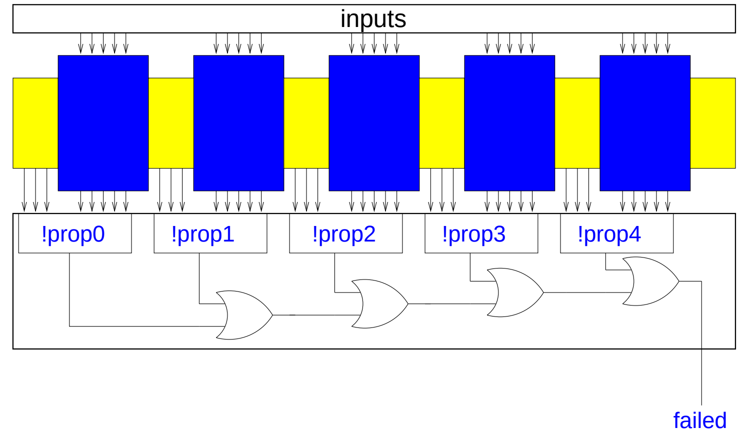


added 4th copy

Time Frame Expansion in HW

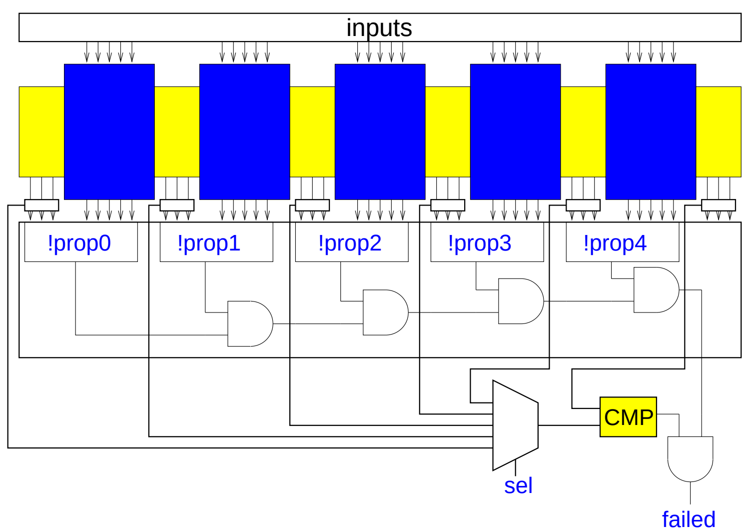


Bounded Model Checking Safety in HW



find inputs for which failed becomes true

Bounded Model Checking Liveness in HW



find inputs for which failed becomes true

Completeness in Bounded Model Checking

- find bounds on the maximal length of counter examples
  - also called **completeness threshold**
  - exact bounds are hard to find $\Rightarrow$ approximations
- induction
  - use inductive invariants as we have seen before
  - generalization of inductive invariants: **pseudo induction**
- use SAT for quantifier elimination as with BDDs (later)
  - then model checking becomes fixpoint calculation

Measuring Distances

Distance: length of shortest path between two states

$$\delta(s,t) \equiv \min\{n \mid \exists s_0,\ldots,s_n[s = s_0, t = s_n \text{ and } T(s_i,s_{i+1}) \text{ for } 0 \le i < n]\}$$

(distance can be infinite if s and t are not connected)

Diameter: maximal distance between two connected states

$$d(T) \equiv \max\{\delta(s,t) \mid T^*(s,t)\}$$

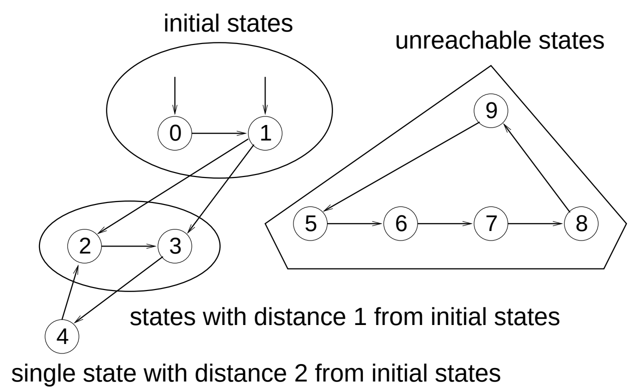
with T^* defined as the transitive reflexive hull of T.

Radius: maximal distance of a reachable state from the initial states

$$r(T,I) \equiv \max\{\delta(s,t) \mid T^*(s,t) \text{ and } I(s) \text{ and } \delta(s,t) \le \delta(s',t) \text{ for all } s' \text{ with } I(s')\}$$

(minimal number of steps to reach an arbitrary state in BFS)

Diameter Example



diameter 4, radius 2

(*reachable diameter* 3, distance from 0 to 4 or max. distance between 2,3,4)

Completeness Threshold for Safety

- a bad state is reached in at most $r(T,I)$ steps from the initial states
  - a bad state is a state violating the invariant to be proven
- thus, the radius is a completeness threshold for safety properties
- for safety properties the max. k for doing bounded model checking is $r(T,I)$
- if no counter example of this length can be found the safety property holds

How to determine the radius?

reformulation:

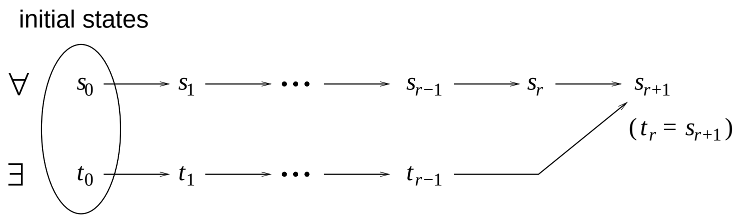
the radius is the max. length r of a path leading from an initial state to a state t, such there is no other path from an initial state to t with length less than r.

Thus radius r is the minimal number which makes the following formula valid:

$$\forall s_0,\ldots,s_{r+1}[\,(I(s_0) \wedge \bigwedge_{i=0}^{r} T(s_i,s_{i+1})) \rightarrow$$
$$\exists\, n \le r\,[\,\exists t_0,\ldots,t_n[\,I(t_0) \wedge \bigwedge_{i=0}^{n-1} T(t_i,t_{i+1}) \wedge t_n = s_{r+1}\,]\,]\,]$$

after replacing $\exists\, n \le r \cdots$ by $\bigvee_{n=0}^{r} \cdots$ we get a **Quantified Boolean Formula** (QBF), which is much harder to prove un/satisfiable (PSPACE complete).

Visualization of Reformulation



(we allow t_{i+1} to be identical to t_i in the lower path)

Reoccurrence Radius/Diameter

- we can not find the real radius / diameter with SAT efficiently
- over approximation idea:
  - drop requirement that there is no shorter path
  - enforce *different* (no reoccurring) states on single path instead

reoccurrence diameter:

length of the longest path without reoccurring states

reoccurrence radius:

length of the longest initialized path without reoccurring states

Determination of Reoccurrence Diameter

reformulation:

the reoccurrence radius is the length of the longest path from initial states without reoccurring states
(one may further assume that only the first state is an initial state)

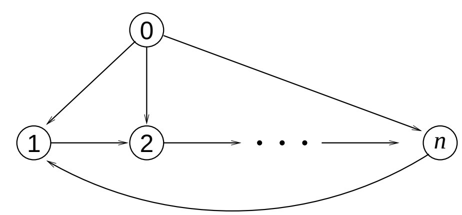
The reoccurring radius is the minimal r which makes the following formula valid:

$$\forall s_0,\ldots,s_{r+1}[\,(I(s_0)\wedge\bigwedge_{i=0}^{r} T(s_i,s_{i+1}))\;\rightarrow \bigvee_{0\le i<j\le r+1} s_i = s_j]$$

this is a propositional formula and can be checked by SAT

(exercise: reoccurrence radius/diameter is an upper bound on real radius/diameter)

Bad Example for Reoccurrence Radius



radius 1, reoccurrence radius n

Bounded Semantics with Loop

(E)LTL formula in NNF

let the path π be a (k,l) lasso

$$\pi \models_k^i p \quad \text{iff} \quad p \in L(\pi(i))$$

$$\pi \models_k^i \neg p \quad \text{iff} \quad p \notin L(\pi(i))$$

$$\pi \models_k^i f \wedge g \quad \text{iff} \quad \pi \models_k^i f \text{ and } \pi \models_k^i g$$

$$\pi \models_k^i \mathbf{X} f \quad \text{iff} \quad \begin{cases} \pi \models_k^l f & \text{if } i = k \\ \pi \models_k^{i+1} f & \text{else} \end{cases}$$

$$\pi \models_k^i \mathbf{G} f \quad \text{iff} \quad \bigwedge_{j=\min(i,l)}^k \pi \models_k^j f$$

$$\pi \models_k^i \mathbf{F} f \quad \text{iff} \quad \bigvee_{j=\min(i,l)}^k \pi \models_k^j f$$

Bounded Semantics without Loop

ELTL formula in NNF

there is no l for which path π is a (k,l) lasso

$$\pi \models_k^i p \quad \text{iff} \quad p \in L(\pi(i))$$

$$\pi \models_k^i \neg p \quad \text{iff} \quad p \notin L(\pi(i))$$

$$\pi \models_k^i f \wedge g \quad \text{iff} \quad \pi \models_k^i f \text{ and } \pi \models_k^i g$$

$$\pi \models_k^i \mathbf{X} f \quad \text{iff} \quad \begin{cases} \text{false} & \text{if } i = k \\ \pi \models_k^{i+1} f & \text{else} \end{cases}$$

$$\pi \models_k^i \mathbf{G} f \quad \text{iff} \quad \text{false}$$

$$\pi \models_k^i \mathbf{F} f \quad \text{iff} \quad \bigvee_{j=i}^k \pi \models_k^j f$$

Bounded Semantics

- definition:

$$\pi \models_k f \quad :\Leftrightarrow \quad \pi \models_k^0 f$$

- bounded semantics aproximates real semantics:

$$\pi_k \models f \quad \Rightarrow \quad \pi \models f \quad \text{for all } k$$

- (theoretical) completeness:

$$\text{if} \quad \pi \models f \quad \text{then there exists } k \text{ with} \quad \pi_k \models f$$

- **note:** negate original property first (e.g. $\mathbf{AG}p \mapsto \mathbf{EF}\neg p$)
  - ALTL $\longrightarrow$ ELTL
  - counter example $\longrightarrow$ witness
  - *bounded* witness is also a non-bounded witness

Translation of Bounded Semantics to SAT

- two recursive translations from (E)LTL in NNF for fixed k:
  - ${}_l[\cdot]_k^i$ assumes (k,l)-loop
  - $[\cdot]_k^i$ assumes that no (k,l)-loop exists for all l
- add time frame expansion of transition relation:

$$I(s_0) \wedge T(s_0,s_1) \wedge \cdots \wedge T(s_{k-1},s_k)$$

- add $loop_k(l)$ constraint for looping translation: $\quad loop_k(l) := T(s_k,s_l)$
- add $noloop_k$ constraint for non-looping translation:

$$noloop_k := \neg \bigvee_{l=0}^{k} loop_k(l)$$

Looping Translation

$$
\begin{aligned}
{}_l[p]_k^i &:= p(s_i) \\
{}_l[\neg p]_k^i &:= \neg p(s_i) \\
{}_l[f \wedge g]_k^i &:= {}_l[f]_k^i \wedge {}_l[g]_k^i \\
{}_l[\mathbf{X}\, f]_k^i &:= {}_l[f]_k^{next(i)} \\
{}_l[\mathbf{G}\, f]_k^i &:= \bigwedge_{j=min(l,i)}^{k} {}_l[f]_k^j \\
{}_l[\mathbf{F}\, f]_k^i &:= \bigvee_{j=min(l,i)}^{k} {}_l[f]_k^j
\end{aligned}
$$

with

$$
next(i) := \begin{cases} i+1 & \text{if } i < k \\ l & \text{else} \end{cases}
$$

Non-Looping Translation

$$
\begin{aligned}
[p]_k^i &:= p(s_i) \\
[\neg p]_k^i &:= \neg p(s_i) \\
[f \wedge g]_k^i &:= [f]_k^i \wedge [g]_k^i \\
[\mathbf{X}\, f]_k^i &:= \begin{cases} [f]_k^{i+1} & \text{if } i < k \\ \mathit{false} & \text{else} \end{cases} \\
[\mathbf{G}\, f]_k^i &:= \mathit{false} \\
[\mathbf{F}\, f]_k^i &:= \bigvee_{j=i}^{k} [f]_k^j
\end{aligned}
$$

Translation

$$
[K, f]_k := noloop_k \wedge [f]_k^0 \vee \bigvee_{l=0}^{k} loop_k(l) \wedge {}_l[f]_k^0
$$

- **Theorem:** $K \models \mathbf{E} f \quad \Leftrightarrow \quad \exists k\, [K, f]_k$ satisfiable

- ${}_l[\cdot]_k^i$ and $[\cdot]_k^i$ are **linear** in k if subformulae are shared
  - unique table for automatic sharing syntactically equivalent formulae
  - implemented as hash table (keys are pairs of formulae ids)

- more complex and quadratic translations for $\mathbf{R}$ and $\mathbf{U}$