Motivation

- more and more complex systems

Moore’s Law $\Rightarrow$ soon we will have $10^{30}$ transistors / processor
multi-million LOC / OS
$\Rightarrow$ exploding testing costs (in general not linear in system size)

- increased dependability

everything important depends on computers:
- stir by wire, banking, stock market, workflow, …
$\Rightarrow$ quality concerns

- increased functionality

- security, mobility, new business processes, …
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
  \[\Rightarrow\] 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 = mathematical

mathematical models \(\Rightarrow\) 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

- Z
- ASM
- SDL
- Synchronous Languages
- Compiler
- Formal Synthesis

Formal Verification

- Model Checking
- Theorem Proving
- Equivalence Checking
- SAT
- Formal Synthesis

Formal Synthesis
- 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

Initial Formal Spec

1st Refinement

2nd Refinement

3rd Refinement

4th Refinement (last formal step)

Compiler

C Program

Compiler
- 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 powerful 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
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 (e.g. SLAM uses BDDs, Model Checking, Theorem Proving)
- 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
## optimization of if-then-else chains

### original C code

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

### optimized C code

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

### Transformation

1. `if(!a && !b) h();` is replaced with `if(a) f();`.
2. `else if(!a) g();` is replaced with `else if(b) g();`.
3. `else f();` is replaced with `else h();`.

### Code Transformation

1. `if(!a) {
   if(!b) h();
   else g();
} else f();` is replaced with `if(a) f();` and `else {
   if(!b) h();
   else g();
} `.

### Question

How to check that these two versions are equivalent?
1. represent procedures as *independent* boolean variables

\[
\begin{align*}
\text{original} & := \\
\text{if } \neg a \land \neg b \text{ then } h \\
\text{else if } \neg a \text{ then } g \\
\text{else } f
\end{align*}
\]

\[
\begin{align*}
\text{optimized} & := \\
\text{if } a \text{ then } f \\
\text{else if } b \text{ then } g \\
\text{else } h
\end{align*}
\]

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

\[
\text{compile}(\text{if } x \text{ then } y \text{ else } z) \equiv (x \land y) \lor (\neg x \land z)
\]

3. check equivalence of boolean formulae

\[
\text{compile}(\text{original}) \iff \text{compile}(\text{optimized})
\]
\[\text{original} \equiv \text{if } \neg a \land \neg b \text{ then } h \text{ else if } \neg a \text{ then } g \text{ else } f \]
\[\equiv (\neg a \land \neg b) \land h \lor \neg (\neg a \land \neg b) \land \text{if } \neg a \text{ then } g \text{ else } f \]
\[\equiv (\neg a \land \neg b) \land h \lor \neg (\neg a \land \neg b) \land (\neg a \land g \lor a \land f)\]

\[\text{optimized} \equiv \text{if } a \text{ then } f \text{ else if } b \text{ then } g \text{ else } h \]
\[\equiv a \land f \lor \neg a \land \text{if } b \text{ then } g \text{ else } h \]
\[\equiv a \land f \lor \neg a \land (b \land g \lor \neg b \land h)\]

\[\neg a \land \neg b) \land h \lor \neg (\neg a \land \neg b) \land (\neg a \land g \lor a \land f) \Leftrightarrow a \land f \lor \neg a \land (b \land g \lor \neg b \land h)\]
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 $\text{compile}(\text{original}) \neq \text{compile}(\text{optimized})$ 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

\[ b \lor a \land c \]

\[ (a \lor b) \land (b \lor c) \]

equivalent?

\[ b \lor a \land c \iff (a \lor b) \land (b \lor c) \]
**SAT (Satisfiability)** the classical NP complete Problem:

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

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

**SAT belongs to NP**

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

*guess* the assignment $\sigma$ (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)
**Definition**

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

\[ C_1 \land C_2 \land \ldots \land C_n \]

each clause \( C \) is a disjunction of literals

\[ C = L_1 \lor \ldots \lor L_m \]

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

**Example**

\[(a \lor b \lor c) \land (\overline{a} \lor \overline{b}) \land (\overline{a} \lor \overline{c})\]

**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
**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

⇒ 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:

$$\text{NNF of } f \leftrightarrow g \text{ is NNF of } f \land g \lor \overline{f} \land \overline{g}$$

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);
    }
}
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
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;
}

Eliminate 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: e.g. BDD sweeping, Stålmarcks Method
  (we will look at both techniques in more detail later)
Example of Tseitin Transformation: Circuit to CNF

CNF

\[ o \land (x \leftrightarrow a \land c) \land (y \leftrightarrow b \lor x) \land (u \leftrightarrow a \lor b) \land (v \leftrightarrow b \lor c) \land (w \leftrightarrow u \land v) \land (o \leftrightarrow y \oplus w) \]

\[ o \land (x \rightarrow a) \land (x \rightarrow c) \land (x \leftarrow a \land c) \land \ldots \]

\[ o \land (\overline{x} \lor a) \land (\overline{x} \lor c) \land (x \lor \overline{a} \lor \overline{c}) \land \ldots \]
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 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)
Negation: \[ x \leftrightarrow \overline{y} \iff (x \rightarrow \overline{y}) \land (\overline{y} \rightarrow x) \iff (\overline{x} \lor \overline{y}) \land (y \lor x) \]

Disjunction: \[ x \leftrightarrow (y \lor z) \iff (y \rightarrow x) \land (z \rightarrow x) \land (x \rightarrow (y \lor z)) \iff (\overline{y} \lor x) \land (\overline{z} \lor x) \land (\overline{x} \lor y \lor z) \]

Conjunction: \[ x \leftrightarrow (y \land z) \iff (x \rightarrow y) \land (x \rightarrow z) \land ((y \land z) \rightarrow x) \iff (\overline{x} \lor y) \land (\overline{x} \lor z) \land (\overline{y} \lor z \lor x) \]

Equivalence: \[ x \leftrightarrow (y \leftrightarrow z) \iff (x \rightarrow (y \leftrightarrow z)) \land ((y \leftrightarrow z) \rightarrow x) \iff (x \rightarrow ((y \rightarrow z) \land (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \iff (x \rightarrow (y \rightarrow z)) \land (x \rightarrow (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \land z) \rightarrow x) \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \land z) \rightarrow x) \land ((\overline{y} \land \overline{z}) \rightarrow x) \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land (\overline{y} \lor \overline{z} \lor x) \land (y \lor z \lor x) \]
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 can 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

  [PlaistedGreenbaum’86] and then [ChambersManoliosVroon’09]

- structural circuit optimizations like in the ABC tool from Berkeley
- however might be simulated on CNF level [JärvisaloBiereHeule-TACAS’10]
- compact technology mapping based encoding [EénMishchenkoSörensson’07]
Intermediate Representations

- encoding directly into CNF is hard, so we use intermediate levels:

  1. application level
  2. bit-precise semantics world-level operations: bit-vector theory
  3. bit-level representations such as AIGs or vectors of AIGs
  4. CNF

- encoding application level formulas into word-level: as generating machine code
- word-level to bit-level: bit-blasting similar to hardware synthesis
- encoding “logical” constraints is another story
addition of 4-bit numbers $x, y$ with result $s$ also 4-bit: $s = x + y$

$$[s_3, s_2, s_1, s_0]_4 = [x_3, x_2, x_1, x_0]_4 + [y_3, y_2, y_1, y_0]_4$$

$$[s_3, \cdot]_2 = \text{FullAdder}(x_3, y_3, c_2)$$
$$[s_2, c_2]_2 = \text{FullAdder}(x_2, y_2, c_1)$$
$$[s_1, c_1]_2 = \text{FullAdder}(x_1, y_1, c_0)$$
$$[s_0, c_0]_2 = \text{FullAdder}(x_0, y_0, \text{false})$$

where

$$[s, o]_2 = \text{FullAdder}(x, y, i) \quad \text{with}$$

$$s = x \oplus y \oplus i$$
$$o = (x \land y) \lor (x \land i) \lor (y \land i) = ((x + y + i) \geq 2)$$
widely adopted bit-level intermediate representation
- see for instance our AIGER format [http://fmv.jku.at/aiger](http://fmv.jku.at/aiger)
- used in Hardware Model Checking Competition (HWMCC)
- also used in the *structural track* in SAT competitions
- many companies use similar techniques

basic logical operators: *conjunction* and *negation*

DAGs: nodes are conjunctions, negation/sign as *edge attribute*
- bit stuffing: signs are compactly stored as LSB in pointer

automatic sharing of isomorphic graphs, constant time (peep hole) simplifications

*or even* SAT sweeping, full reduction, etc ... see ABC system from Berkeley
\[ x \text{ xor } y \equiv (\overline{x} \land y) \lor (x \land \overline{y}) \equiv \overline{(\overline{x} \land y) \lor (x \land \overline{y})} \]

negation/sign are edge attributes
not part of node
typedef struct AIG AIG;

struct AIG
{
    enum Tag tag;       /* AND, VAR */
    void *data[2];
    int mark, level;    /* traversal */
    AIG *next;          /* hash collision chain */
};

#define sign_aig(aig) (1 & (unsigned) aig)
#define not_aig(aig) ((AIG*)(1 ^ (unsigned) aig))
#define strip_aig(aig) ((AIG*)(~1 & (unsigned) aig))
#define false_aig ((AIG*) 0)
#define true_aig ((AIG*) 1)

assumption for correctness:

sizeof(unsigned) == sizeof(void*)
4-bit adder

8-bit adder
bit-vector of length 16 shifted by bit-vector of length 4
Tseitin’s construction suitable for most kinds of “model constraints”

- assuming simple operational semantics: encode an interpreter
- small domains: one-hot encoding
- large domains: binary encoding

- harder to encode properties or additional constraints
  - temporal logic / fix-points
  - environment constraints

Example for fix-points / recursive equations:

\[ x = (a \lor y), \quad y = (b \lor x) \]

- has unique least fix-point \( x = y = (a \lor b) \)
- and unique largest fix-point \( x = y = \text{true} \)
- only largest fix-point can be (directly) encoded in SAT

otherwise need ASP
Example of Logical Constraints: Cardinality Constraints

- given a set of literals \( \{l_1, \ldots, l_n\} \)
  - constraint the *number* of literals assigned to true
  - \( |\{l_1, \ldots, l_n\}| \geq k \) or \( |\{l_1, \ldots, l_n\}| \leq k \) or \( |\{l_1, \ldots, l_n\}| = k \)

- multiple encodings of cardinality constraints
  - naïve encoding exponential: *at-most-two* quadratic, *at-most-three* cubic, etc.
  - quadratic \( O(k \cdot n) \) encoding goes back to Shannon
  - linear \( O(n) \) parallel counter encoding [Sinz’05]
  - for an \( O(n \cdot \log n) \) encoding see Prestwich’s chapter in our Handbook of SAT

- generalization *Pseudo-Boolean* constraints (PB), e.g. \( 2 \cdot \overline{a} + \overline{b} + c + d + 2 \cdot e \geq 3 \)
  - actually used to handle MaxSAT in SAT4J for configuration in Eclipse
\[ 2 \leq |\{l_1, \ldots, l_9\}| \leq 3 \]

\[
\begin{array}{cccccccccc}
  l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & 0 \\
  l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & 0 \\
  l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & 1 \\
  l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & 1 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

"then" edge downward, "else" edge to the right
Davis & Putnam Procedure (DP)

- dates back to the 50ies:
  - original version is *resolution based* (successful only in preprocessors)
  - improved DPLL: case analysis (try $x = 0, 1$ in turn and recurse)
  - evolved to CDCL (conflict driven clause learning): state-of-the-art

- recent ($\leq 20$ years) optimizations:
  - backjumping, learning, UIPs, dynamic splitting heuristics, fast data structures
    we will have a look at each of them

- elimination procedure of original DP is similar to
  - Gaussian elimination on linear real equalities
  - Fourier-Motzikin on linear real inequalities
  - Buchberger’s algorithm on polynomial equations
- basis for first (less successful) resolution based DP
- can be extended to first order logic
- helps to explain learning

**Resolution Rule**

\[
\begin{align*}
C \cup \{v\} & \quad D \cup \{\neg v\} \\
\hline
C \cup D
\end{align*}
\]

\[
\{v, \neg v\} \cap C = \{v, \neg v\} \cap D = \emptyset
\]

**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.
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 \((\sigma(c) = 1 \text{ or } \sigma(v) = 1)\) and \((\sigma(d) = 1 \text{ or } \sigma(\neg v) = 1)\)

if \(\sigma(c) = 1 \text{ or } \sigma(d) = 1\) conclusion follows immediately

otherwise \(\sigma(v) = \sigma(\neg v) = 1\) \(\Rightarrow\) contradiction \(\text{q.e.d.}\)
Theorem. (conclusion satisfiable $\Rightarrow$ premise satisfiable)

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

Proof.

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

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

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.

Example consider incorrect resolution $\frac{\{v\} \cup \{v\}}{v}$ violating side condition
Example for Completeness of Resolution Rule

consider the following resolution

\[
\frac{a \lor b \quad \neg b \lor c}{a \lor c}
\]

in logical notation, not set notation for a change

let \( \sigma(x) = \begin{cases} 
1 & \text{if } x = a \\
1 & \text{if } x = b \\
0 & \text{if } x = c 
\end{cases} \)

be a model of resolvent \((a \lor c)\) thus \(\sigma(a \lor c) = 1\)

note that \(\sigma(\neg b \lor c) = 0\) and thus \(\sigma\) is not a model of 2nd antecedent (2nd premisse)

however \(\sigma\) satisfies remaining literal \(a\) of 1st antecedent in resolvent

thus simply flip value of pivot \(b\) (satisfy its occurrence in 2nd antecedent)

we get \(\sigma'(x) = \begin{cases} 
1 & \text{if } x = a \\
\mathbf{0} & \text{if } x = b \\
0 & \text{if } x = c 
\end{cases} \)

satisfying both antecedents \(\sigma'(a \lor b) = \sigma'(-b \lor c) = 1.\)
**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:

\[ a \lor b \to (a \oplus b) \]

which is equivalent to unsatisfiability of the negation:

\[ (a \lor b) \land \neg(a \oplus b) \]

since negation of XOR is XNOR (equivalence):

\[ (a \lor b) \land (a \leftrightarrow b) \]

we end up checking the following CNF for satisfiability:

\[ (a \lor b) \land (\neg a \lor b) \land (a \lor \neg b) \]
Example for Resolution DP cont.

\[(a \lor b) \land (\neg a \lor b) \land (a \lor \neg b)\]

initially we can skip 1. - 4. of the algorithm and choose \(x = b\) in 5.

in 6. we resolve \((\neg a \lor b)\) with \((a \lor \neg b)\) and \((a \lor b)\) with \((a \lor \neg b)\) both on \(b\)

and add the results \((a \lor \neg a)\) and \((a \lor a)\):

\[(a \lor b) \land (\neg a \lor b) \land (a \lor \neg b) \land (a \lor \neg a) \land (a \lor a)\]

the trivial clause \((a \lor \neg a)\) and clauses with occurrences of \(b\) are removed:

\[(a \lor a)\]

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)
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 ⇒ all clauses satisfied
**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)
**CNF transformations preserve unsatisfiability:**

adding a clause does not change unsatisfiability

thus only removing clauses could potentially not preserve unsatisfiability

trivial clauses \((v \lor \neg v \lor \ldots)\) 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 \((f\ \text{unsat} \Rightarrow g\ \text{unsat})\), or equivalently \((g\ \text{sat} \Rightarrow f\ \text{sat})\)

the latter can be proven as the completeness proof for the resolution rule
(see next slide)
If we interpret $\cup$ as disjunction and clauses as formulae, then

$$(C_1 \lor x) \land \ldots \land (C_k \lor x) \land (D_1 \lor \neg x) \land \ldots \land (D_l \lor \neg x)$$

is, via distributivity law, equivalent to

$$(\underbrace{C_1 \land \ldots \land C_k}_C \lor x) \land (\underbrace{D_1 \land \ldots \land D_l}_D \lor \neg x)$$

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 $\sigma'$ derived for multiple resolutions are formally allowed to assign different values for the resolution variable.
if variables have many occurrences, then many resolutions are necessary

in the worst case, $x$ 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 might latter see how BDDs can be used to overcome some of these problems)

How to obtain the satisfying assignment efficiently (counter example)?
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
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

\[
C \cup \{\neg l\} \quad \{l\} \quad \{l, \neg l\} \cap C = \emptyset
\]

here we identify \(\neg\neg v\) with \(v\) for all variables \(v\).
check whether XNOR is weaker than AND, i.e. validity of:

\[ a \land b \rightarrow (a \leftrightarrow b) \]

which is equivalent to unsatisfiability of the CNF (exercise)

\[ a \land b \land (a \lor b) \land (\neg a \lor \neg b) \]

adding clause obtained from unit resolution on \( a \) results in

\[ a \land b \land (a \lor b) \land (\neg a \lor \neg b) \land (\neg b) \]

removing clauses containing \( a \) or \( \neg a \)

\[ b \land (\neg b) \]

unit resolution on \( b \) results in an empty clause and we conclude unsatisfiability.
- if unit resolution produces a unit, e.g. resolving \((a \lor \neg b)\) 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

\[
C \text{ subsumes } D \iff C \subseteq D
\]

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
Skip 1. - 6., and choose \( x = a \). First recursive call:

\[
(\neg a \lor b) \land (a \lor \neg b) \land (\neg a \lor \neg b)
\]

unit resolution on \( a \) and removal of subsumed clauses gives

\[
b \land (\neg b)
\]

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

\[
(\neg a \lor b) \land (a \lor \neg b) \land (\neg a \lor \neg b) \land \neg a
\]

BCP gives \( \neg b \), only positive recurrence of \( b \) left, return satisfiable

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

\[ f(x) \equiv x \land f(1) \lor \overline{x} \land f(0) \]

Proof.

Let \( \sigma \) be an arbitrary assignment to variables in \( f \) including \( x \)

case \( \sigma(x) = 0 \):

\[ \sigma(f(x)) = \sigma(f(0)) = \sigma(0 \land f(1) \lor 1 \land f(0)) = \sigma(x \land f(1) \lor \overline{x} \land f(0)) \]

case \( \sigma(x) = 1 \):

\[ \sigma(f(x)) = \sigma(f(1)) = \sigma(1 \land f(1) \lor 0 \land f(0)) = \sigma(x \land f(1) \lor \overline{x} \land f(0)) \]
first observe: \( x \land f(x) \) is satisfiable \iff \( x \land f(1) \) is satisfiable

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

then use expansion theorem of Shannon:

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

rest follows along the lines of the the correctness proof for resolution based DP
Simple Data Structures in DP Implementation

Variables

Clauses

SAT Solving #342.201 WS 2019

Armin Biere JKU Linz
- 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
- **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

#### Decision Level

- **Assignments:**
  - $X_1$: 1
  - $X_2$: 2
  - $X_3$: 3
  - $X_4$: 4
  - $X_5$: 5

- **Clauses:**
  - $X_1$: $C_1 = 1$
  - $X_2$: $C_2 = 2$
  - $X_3$: $C_3 = 3$
  - $X_4$: $C_4 = 4$
  - $X_5$: $C_5 = 5$

#### Control Trail

- **Trail:**
  - $0 \rightarrow 0$

---

**Diagram:**

- **Decision Level:**
  - Nodes: 0

- **Control Trail:**
  - Edge: 0 to 0

- **Variables Assignment:**
  - $X_1$, $X_2$, $X_3$, $X_4$, $X_5$

- **Clauses:**
  - $C_1$, $C_2$, $C_3$, $C_4$, $C_5$
Example cont.

Decide

<table>
<thead>
<tr>
<th>Variables</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$ 1</td>
<td></td>
</tr>
<tr>
<td>$X$ 2</td>
<td></td>
</tr>
<tr>
<td>$X$ 3</td>
<td></td>
</tr>
<tr>
<td>$X$ 4</td>
<td></td>
</tr>
<tr>
<td>$X$ 5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
</tr>
<tr>
<td>$2$</td>
</tr>
<tr>
<td>$3$</td>
</tr>
<tr>
<td>$4$</td>
</tr>
<tr>
<td>$5$</td>
</tr>
</tbody>
</table>
Assign

Variables

Clauses
### BCP

<table>
<thead>
<tr>
<th>Variables</th>
<th>Assignment</th>
<th>Control</th>
<th>Trail</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Decision level: 1
- Clauses: 3
- Variables: 5
- Assignment:
  - X: 4, 5
- Clauses:
  - \(-1, 2\)
  - \(-2, 3\)
  - \(-4, 5\)
Example cont.

Decide

decision level

Control

Trail

Variables

Assignment

Clauses

1 1
1 2
1 3
X 4
X 5

1 3
0 0

3
0

2
1

3
0

3
2

−4 5
−2 3
−1 2
Example cont.

Assign

decision level

Control

Trail

Variables

Clauses

Assign

1 1
1 2
1 3
1 4
X 5

−1 2
−2 3
−4 5
Example cont.

BCP

decision level

Control

Trail

Variables

Assignment

Clauses

TrailControldecision level
1
ClausesVariables
2
3
4
5 −4
3−2
−1 2
1
1
1
1
5
BCP
1
2
3
0
0 1
2
3
4
5
Assignment

\[\begin{array}{c|c|c|c}
1 & 1 & \cdot \\
1 & 2 & \cdot \\
1 & 3 & \cdot \\
1 & 4 & \cdot \\
1 & 5 & \cdot \\
\end{array}\]

\[\begin{array}{c|c|c}
5 & & \\
4 & & \\
3 & & \\
2 & & \\
1 & & \\
\end{array}\]

\[\begin{array}{c|c|c}
-1 & 2 & \\
-2 & 3 & \\
-4 & 5 & \\
\end{array}\]
**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 occurrences 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 (VSIDS in Chaff ⇒ *see learning*)
- 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

\[-2 \ 1 \ -3 \ 1 \ -1 \ 2 \ 3 \ -4 \ 3 \ 1, \ 2, \ -1, \ -2\]

assume no occurrences of 1, 2, -1, -2 on the right side
```c
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;
}
```
- 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)
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 assignment 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 (e.g. 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 (e.g. SATZ or MARCH solver)** failed literals, probing
  - 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)
  - decision variable maximizes number of propagated assignments
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.
Split on $-3$ first (bad decision).
Split on $-1$ and get first conflict.
Regularly backtrack and assign 1 to get second conflict.
Backjumping Example

Backtrack to root, assign 3 and derive same conflicts.
Assignment $-3$ does not contribute to conflict.
So just \textit{backjump} to root before assigning 1.
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
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
(edges of directed hyper graphs may have multiple source and target nodes)
- 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
- 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
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*
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)
unit clause \(-3\) is generated as learned clause and we backtrack to \(l = 0\)

since \(-3\) has a real unit clause as reason, an empty conflict clause is learned
Since FIRST clause was used to derive 2, conflict clause is \(1 \land \neg 3\).

Backtrack to \( l = 1 \) (smallest level for which conflict clause is a unit clause).

\( l = 0 \) (no unit clause originally, so no implications)

\( l = 1 \) \(-1\) (no implications on this decision level either)
$l = 0$ (no unit clause originally, so no implications)

$\vdash 0$

$\vdash 1$

decision

$\vdash 1$

$\vdash 3$

$\vdash 4\ 5\ 6$

1st conflict clause

empty clause (conflict)

learned conflict clause is the unit clause 1

backtrack to decision level $l = 0$
since the learned clause is the empty clause, conclude unsatisfiability
$l = 0$  
(no unit clause originally, so no implications)


decision

$l = 1$  
$-6$  
(no implications on this decision level either)


$l = 2$

decision


learn the unit clause $-3$ and BACKJUMP to decision level $l = 0$
finally the empty clause is derived which proves unsatisfiability
```c
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;
  }
}
```
```c
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, learnedClause);
    undo (solver, new_level);

    return 1;
}
```
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 [BayardoSchrag'97]
a simple cut always exists: set of roots (decisions) contributing to the conflict
UIP = \textit{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
- \textit{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 \textit{sentinel}
- assume a non decision UIP is found
- this UIP is part of a potential cut
- graph traversal may stop (everything \textit{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
1st UIP learned clause increases chance of backjumping
("pulls in" as few decision levels as possible)
intuitively it 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)
Antecedents / Reasons

d \land g \land s \rightarrow t \equiv (\overline{d} \lor \overline{g} \lor \overline{s} \lor t)
\( \neg(y \land z) \equiv (\neg y \lor \neg z) \)
Resolving Antecedents 1st Time

\[ (\overline{h} \lor \overline{i} \lor \overline{t} \lor y) \quad (\overline{y} \lor z) \]
Resolving Antecedents 1st Time

\[(\overline{h} \lor \overline{i} \lor \overline{j} \lor y) \land (\overline{y} \lor \overline{z}) \land (\overline{h} \lor \overline{i} \lor \overline{j} \lor \overline{z})\]
Resolvents = **Cuts** = Potential Learned Clauses

\[
\begin{align*}
\top-level & \quad \text{unit} & \quad a = 1 @ 0 & \quad \text{unit} & \quad b = 1 @ 0 \\
\text{decision} & \quad c = 1 @ 1 & \quad d = 1 @ 1 & \quad e = 1 @ 1 \\
\text{decision} & \quad f = 1 @ 2 & \quad g = 1 @ 2 & \quad h = 1 @ 2 & \quad i = 1 @ 2 \\
\text{decision} & \quad k = 1 @ 3 & \quad l = 1 @ 3 & \quad t = 1 @ 4 \\
\text{decision} & \quad r = 1 @ 4 & \quad s = 1 @ 4 & \quad y = 1 @ 4 \\
\phantom{\text{decision}} & \quad x = 1 @ 4 & \quad z = 1 @ 4 & \quad \kappa \quad \text{conflict} \\
\end{align*}
\]

\[
\begin{align*}
(h \lor \neg i \lor \neg t \lor y) & \quad (\neg y \lor \neg z) \\
(h \lor \neg i \lor \neg t \lor \neg z) & \quad _\text{conflict}
\end{align*}
\]
(\overline{h} \lor \overline{i} \lor \overline{t} \lor \overline{z})
Resolving Antecedents 2nd Time

\[
\begin{align*}
\text{top–level} & \quad \text{unit} & a &= 1 \quad 0 & \quad \text{unit} & b &= 1 \quad 0 \\
\text{decision} & & c &= 1 \quad 1 & d &= 1 \quad 1 & e &= 1 \quad 1 \\
\text{decision} & & f &= 1 \quad 2 & g &= 1 \quad 2 & h &= 1 \quad 2 & i &= 1 \quad 2 \\
\text{decision} & & k &= 1 \quad 3 & l &= 1 \quad 3 \\
\text{decision} & & r &= 1 \quad 4 & s &= 1 \quad 4 & t &= 1 \quad 4 & y &= 1 \quad 4 \\
\text{decision} & & x &= 1 \quad 4 & z &= 1 \quad 4 \\
& & & \kappa & \quad \text{conflict}
\end{align*}
\]

\[
(d \lor g \lor s \lor t) \quad (h \lor i \lor t \lor z)
\]

\[
(d \lor g \lor s \lor \overline{h} \lor \overline{i} \lor \overline{z})
\]
\[(\overline{x} \lor z) \land (d \lor \overline{g} \lor \overline{s} \lor \overline{h} \lor \overline{i} \lor \overline{z})\]
Resolving Antecedents 4th Time

\[ (\overline{s} \lor x) \quad (\overline{x} \lor \overline{d} \lor \overline{g} \lor \overline{s} \lor \overline{h} \lor \overline{i}) \]

self subsuming resolution

\[ (\overline{d} \lor \overline{g} \lor \overline{s} \lor \overline{h} \lor \overline{i}) \]
1st UIP Clause after 4 Resolutions

\[(\bar{d} \lor \bar{g} \lor \bar{s} \lor \bar{h} \lor \bar{i})\]
Resolving Antecedents 5th Time

\[(\bar{l} \lor \bar{r} \lor s) \quad (d \lor \bar{g} \lor \bar{s} \lor \bar{h} \lor \bar{i})\]

\[(\bar{l} \lor \bar{r} \lor \bar{d} \lor g \lor \bar{h} \lor \bar{i})\]
1st UIP Clause after 4 Resolutions

\((\bar{d} \lor \bar{g} \lor \bar{s} \lor \bar{h} \lor \bar{i})\)
Locally Minimizing 1st UIP Clause

\[
\begin{align*}
\text{top–level} & \quad \text{unit} \quad a = 1 @ 0 \quad \text{unit} \quad b = 1 @ 0 \\
\text{decision} & \quad c = 1 @ 1 \quad d = 1 @ 1 \quad e = 1 @ 1 \\
\text{decision} & \quad f = 1 @ 2 \quad g = 1 @ 2 \quad h = 1 @ 2 \quad i = 1 @ 2 \\
\text{decision} & \quad k = 1 @ 3 \quad l = 1 @ 3 \\
\text{decision} & \quad r = 1 @ 4 \quad s = 1 @ 4 \quad t = 1 @ 4 \quad y = 1 @ 4 \\
& \quad x = 1 @ 4 \quad z = 1 @ 4 \quad \kappa \quad \text{conflict}
\end{align*}
\]

\[
\begin{align*}
(h \lor i) & \quad (d \lor \bar{g} \lor \bar{s} \lor h \lor \bar{i}) \\
& \quad (d \lor \bar{g} \lor \bar{s} \lor \bar{h})
\end{align*}
\]

self subsuming resolution
Locally Minimized Learned Clause

\[(\overline{d} \lor g \lor \overline{s} \lor \overline{h})\]

Diagram:
- **top-level**
  - unit \(a = 1 @ 0\)
  - unit \(b = 1 @ 0\)

- **decision** \(c = 1 @ 1\)
  - unit \(d = 1 @ 1\)
  - unit \(e = 1 @ 1\)

- **decision** \(f = 1 @ 2\)
  - unit \(g = 1 @ 2\)
  - unit \(h = 1 @ 2\)
  - unit \(i = 1 @ 2\)

- **decision** \(k = 1 @ 3\)
  - unit \(l = 1 @ 3\)

- **decision** \(r = 1 @ 4\)
  - unit \(s = 1 @ 4\)
  - unit \(t = 1 @ 4\)
  - unit \(y = 1 @ 4\)
  - \(x = 1 @ 4\)
  - \(z = 1 @ 4\)
  - \(\kappa\) conflict
Local Minimization Algorithm

Two step algorithm:

1. mark all variables in 1st UIP clause

2. remove literals with all antecedent literals also marked

Correctness:
- removal of literals in step 2 are self subsuming resolution steps.
- implication graph is acyclic.

Confluence: produces a unique result.
Minimizing Locally Minimized Learned Clause Further?

\[ (\overline{d} \lor \overline{g} \lor \overline{s} \lor \overline{h}) \]

\( a = 1 \oplus 0 \)

\( b = 1 \oplus 0 \)

\( c = 1 \oplus 1 \)

\( d = 1 \oplus 1 \)

\( e = 1 \oplus 1 \)

\( f = 1 \oplus 2 \)

\( g = 1 \oplus 2 \)

\( h = 1 \oplus 2 \)

\( i = 1 \oplus 2 \)

\( k = 1 \oplus 3 \)

\( l = 1 \oplus 3 \)

\( m = 1 \oplus 3 \)

\( n = 1 \oplus 4 \)

\( o = 1 \oplus 4 \)

\( p = 1 \oplus 4 \)

\( q = 1 \oplus 4 \)

\( r = 1 \oplus 4 \)

\( s = 1 \oplus 4 \)

\( t = 1 \oplus 4 \)

\( y = 1 \oplus 4 \)

\( x = 1 \oplus 4 \)

\( z = 1 \oplus 4 \)

\( \kappa = \text{conflict} \)
Recursively Minimizing Learned Clause

\[
\begin{array}{ccc}
\text{top–level} & \text{unit} & a = 1 @ 0 & \text{unit} \\
\text{decision} & c = 1 @ 1 & d = 1 @ 1 & \text{decision} & e = 1 @ 1 \\
\text{decision} & f = 1 @ 2 & g = 1 @ 2 & \text{decision} & h = 1 @ 2 \\
\text{decision} & k = 1 @ 3 & l = 1 @ 3 & \text{decision} & r = 1 @ 4 & s = 1 @ 4 \\
\text{decision} & r = 1 @ 4 & s = 1 @ 4 & \text{decision} & t = 1 @ 4 & y = 1 @ 4 \\
\end{array}
\]

\[
(b) \quad (\bar{b} \lor \bar{d} \lor \bar{g} \lor \bar{s})
\]

\[
\begin{align*}
\frac{\bar{e} \lor \bar{g} \lor h}{(\bar{d} \lor \bar{g} \lor \bar{s} \lor \bar{h})} & \quad \frac{(\bar{d} \lor \bar{g} \lor \bar{s})}{(\bar{e} \lor \bar{d} \lor \bar{g} \lor \bar{s})} \\
\frac{(\bar{d} \lor \bar{b} \lor e)}{(b)} & \quad \frac{(\bar{b} \lor \bar{d} \lor \bar{g} \lor \bar{s})}{(\bar{d} \lor \bar{g} \lor \bar{s})}
\end{align*}
\]
Recursively Minimized Learned Clause

\[(\overline{d} \lor \overline{g} \lor \overline{s})\]
Recursive Minimization Algorithm

[MiniSAT 1.13]

Four step algorithm:

1. mark all variables in 1st UIP clause

2. for each candidate literal: search implication graph

3. start at antecedents of candidate literals

4. if search always terminates at marked literals remove candidate

Correctness and Confluence as in local version!!!

Optimization: terminate early with failure if new decision level is “pulled in”
<table>
<thead>
<tr>
<th></th>
<th>solved instances</th>
<th>time in hours</th>
<th>space in GB</th>
<th>out of memory</th>
<th>deleted literals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MINISat</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with preprocessing</td>
<td>recur</td>
<td>788 (9%)</td>
<td>170 (11%)</td>
<td>198 (49%)</td>
<td>11 (89%)</td>
</tr>
<tr>
<td></td>
<td>local</td>
<td>774 (7%)</td>
<td>177 (8%)</td>
<td>298 (24%)</td>
<td>72 (30%)</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>726</td>
<td>192</td>
<td>392</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>without preprocessing</td>
<td>recur</td>
<td>705 (13%)</td>
<td>222 (8%)</td>
<td>232 (59%)</td>
<td>11 (94%)</td>
</tr>
<tr>
<td></td>
<td>local</td>
<td>642 (3%)</td>
<td>237 (2%)</td>
<td>429 (24%)</td>
<td>145 (26%)</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>623</td>
<td>242</td>
<td>565</td>
<td>196</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PicoSat</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with preprocessing</td>
<td>recur</td>
<td>767 (10%)</td>
<td>182 (13%)</td>
<td>144 (45%)</td>
<td>10 (60%)</td>
</tr>
<tr>
<td></td>
<td>local</td>
<td>745 (6%)</td>
<td>190 (9%)</td>
<td>188 (29%)</td>
<td>10 (60%)</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>700</td>
<td>209</td>
<td>263</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>without preprocessing</td>
<td>recur</td>
<td>690 (6%)</td>
<td>221 (8%)</td>
<td>105 (63%)</td>
<td>10 (68%)</td>
</tr>
<tr>
<td></td>
<td>local</td>
<td>679 (5%)</td>
<td>230 (5%)</td>
<td>194 (31%)</td>
<td>10 (68%)</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>649</td>
<td>241</td>
<td>281</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>altogether</td>
<td>recur</td>
<td>2950 (9%)</td>
<td>795 (10%)</td>
<td>679 (55%)</td>
<td>42 (88%)</td>
</tr>
<tr>
<td></td>
<td>local</td>
<td>2840 (5%)</td>
<td>834 (6%)</td>
<td>1109 (26%)</td>
<td>237 (33%)</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>2698</td>
<td>884</td>
<td>1501</td>
<td>355</td>
</tr>
</tbody>
</table>

10 runs for each configuration with 10 seeds for random number generator
## Large Variance for Different Seeds

### MINISAT with preprocessing

<table>
<thead>
<tr>
<th>Seed</th>
<th>Solved</th>
<th>Time</th>
<th>Space</th>
<th>Mo</th>
<th>Del</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. recur</td>
<td>8</td>
<td>82</td>
<td>16</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>2. recur</td>
<td>6</td>
<td>81</td>
<td>17</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>3. local</td>
<td>0</td>
<td>81</td>
<td>16</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>4. local</td>
<td>7</td>
<td>80</td>
<td>17</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>5. recur</td>
<td>4</td>
<td>80</td>
<td>17</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>6. recur</td>
<td>1</td>
<td>79</td>
<td>17</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>7. recur</td>
<td>9</td>
<td>79</td>
<td>17</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>8. local</td>
<td>5</td>
<td>78</td>
<td>18</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>9. local</td>
<td>1</td>
<td>78</td>
<td>17</td>
<td>29</td>
<td>6</td>
</tr>
<tr>
<td>10. recur</td>
<td>0</td>
<td>78</td>
<td>17</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>11. recur</td>
<td>5</td>
<td>78</td>
<td>17</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>12. local</td>
<td>3</td>
<td>77</td>
<td>18</td>
<td>31</td>
<td>7</td>
</tr>
<tr>
<td>13. local</td>
<td>8</td>
<td>77</td>
<td>18</td>
<td>30</td>
<td>8</td>
</tr>
<tr>
<td>14. recur</td>
<td>7</td>
<td>77</td>
<td>17</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>15. recur</td>
<td>3</td>
<td>77</td>
<td>17</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>16. recur</td>
<td>2</td>
<td>77</td>
<td>17</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>17. none</td>
<td>7</td>
<td>76</td>
<td>19</td>
<td>39</td>
<td>9</td>
</tr>
</tbody>
</table>
- often it is a good strategy to abandon what you do and restart
  - for satisfiable instances the solver may get stuck in the unsatisfiable part
  - for unsatisfiable instances focusing on one part might miss short proofs
  - restart after the number of conflicts 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 during restart

- for completeness dynamically increase restart limit
  - arithmetically, geometrically, Luby, Inner/Outer

- Glucose restarts [AudemardSimon-CP’12]
  - short vs. large window *exponential moving average* (EMA) over LBD
  - if recent LBD values are larger than long time average then restart

- interleave “stabilizing” (no restarts) and “non-stabilizing” phases [Chanseok Oh]
Inner/Outer Restart Intervals

378 restarts in 104408 conflicts
int inner = 100, outer = 100;
int restarts = 0, conflicts = 0;

for (;;)
{
    ... // run SAT core loop for 'inner' conflicts

    restarts++;
    conflicts += inner;

    if (inner >= outer)
    {
        outer *= 1.1;
        inner = 100;
    }
    else
    {
        inner *= 1.1;
    }
}
Luby’s Restart Intervals

70 restarts in 104448 conflicts
unsigned
luby (unsigned i)
{
    unsigned k;

    for (k = 1; k < 32; k++)
        if (i == (1 << k) - 1)
            return 1 << (k - 1);

    for (k = 1;; k++)
        if ((1 << (k - 1)) <= i && i < (1 << k) - 1)
            return luby (i - (1 << (k-1)) + 1);
}

limit = 512 * luby (++restarts);
... // run SAT core loop for 'limit' conflicts
(\(u_1, v_1\)) := (1, 1)

(\(u_{n+1}, v_{n+1}\)) := \(u_n \& -u_n = v_n \ ? (u_n + 1, 1) : (u_n, 2v_n)\)

\((1, 1), (2, 1), (2, 2), (3, 1), (4, 1), (4, 2), (4, 4), (5, 1), \ldots\)
Restart Scheduling with Exponential Moving Averages

[BIereFröhlich-POS'15]

- LBD
  - fast EMA of LBD with $\alpha = 2^{-5}$
  - restart
    - slow EMA of LBD with $\alpha = 2^{-14}$ (ema-14)
  - inprocessing
    - CMA of LBD (average)
double fast, slow;
...

bool analyze () {
    int lbd;
    ...
    slow += (lbd - slow)/(double)(1<<14);
    fast += (lbd - fast)/(double)(1<<5);
    ...
}

bool restarting () {
    return conflicts > limit && fast > 1.25 * slow;
}
fast 64-bit fixed point implementation avoiding floating point
inspired by Donald Knuth’s implementation of our agility metric

long fast, slow; // assume (sizeof (long) == 8);
... // initialization code skipped ...

bool analyze () {
    int lbd; // assume (sizeof (int) == 4);
    ...
    fast -= fast >> 5;
    fast += lbd << (32 - 5);
    slow -= slow >> 14;
    slow += lbd << (32 - 14);
    ...
}

bool restarting () {
    return conflicts > limit && fast / 125 > slow / 100;
}
Phase Saving and Rapid Restarts

- **phase assignment:**
  - assign decision variable to 0 or 1?
  - “Only thing that matters in *satisfiable* instances” [Hans van Maaren]

- “phase saving” as in RSat [PipatsrisawatDarwiche’07]
  - pick phase of last assignment (if not forced to, do not toggle assignment)
  - initially use statically computed phase (typically LIS)
  - so can be seen to maintain a *global full assignment*

- **rapid restarts**
  - varying restart interval with bursts of restarts
  - not only theoretically avoids local minima
  - works nicely together with phase saving
  - reusing the trail can reduce the cost of restarts [RamosVanDerTakHeule-JSAT’11]
  - target phases of largest conflict free trail / assignment [Biere-SAT-Race-2019]
Second Order Dynamic Decision Heuristics: VSIDS

[MoskewiczMadiganZhaoZhangMalik-DAC’01: CHAFF]

- “second order” because it involves statistics about the search
- Variable State Independent Decaying Sum (VSIDS) decision heuristic (implemented in Chaff, Limmat, MiniSAT, PicoSAT, and many more)
- VSIDS just counts the occurrences of literals in conflict clauses
- literal/variable with maximal count (score) is chosen (from a priority queue ordered by score)
- 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)
Normalized VSIDS: NVSIDS

[Biere-SAT’08]

- VSIDS score can be normalized to the interval [0,1] as follows:
  - pick a decay factor $f$ per conflict: typically $f = 0.95$
  - each variable is punished by this decay factor at every conflict
  - if a variable is involved in conflict, add $1 - f$ to score

\[
s, f \leq 1, \text{ then } s' \leq s \cdot f + 1 - f \leq f + 1 - f = 1\]

  with $s$ old score before conflict, $s'$ new score after conflict

- recomputing score of all variables at each conflict is costly
  - linear in the number of variables, e.g. millions
  - particularly, because number of involved variables $\ll$ number of variables
Exponential VSIDS: EVSIDS

- Chaff: precision of score traded for faster decay
  - increment score of involved variables by 1
  - decay score of all variables every 256 conflicts by halving the score
  - sort priority queue after decay and not at every conflict

- MiniSAT uses Exponential VSIDS
  - also just update score of involved variables
  - dynamically adjust increment: \( \delta' = \delta \cdot \frac{1}{j} \) (typically increment \( \delta \) by 5%)
  - use floating point representation of score
  - “rescore” to avoid overflow in regular intervals
  - EVSIDS linearly related to NVSIDS
Relating EVSIDS and NVSIDS

consider again only one variable with score sequence $s_n$ resp. $S_n$

$$\delta_k = \begin{cases} 
1 & \text{if involved in } k\text{-th conflict} \\
0 & \text{otherwise}
\end{cases}$$

$$i_k = (1 - f) \cdot \delta_k$$

$$s_n = (\ldots (i_1 \cdot f + i_2) \cdot f + i_3) \cdot f \ldots ) \cdot f + i_n = \sum_{k=1}^{n} i_k \cdot f^{n-k} = (1 - f) \cdot \sum_{k=1}^{n} \delta_k \cdot f^{n-k} \quad \text{(NVSIDS)}$$

$$S_n = \frac{f^{-n}}{(1 - f)} \cdot s_n = \frac{f^{-n}}{(1 - f)} \cdot (1 - f) \cdot \sum_{k=1}^{n} \delta_k \cdot f^{n-k} = \sum_{k=1}^{n} \delta_k \cdot f^{-k} \quad \text{(EVSIDS)}$$
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 choose 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 (by some authors considered to be more robust than just VSIDS)
Other Variable Scoring Variants

- **variable move to front strategy (VMTF)**
  - Siege SAT Solver [Ryan’04]
  - easy and cheap to implement with doubly linked list
    - need pointer to last picked variable in queue
    - reset during back-tracking
  - rather aggressive

- **clause move to front strategy (CMTF)**
  - HaifaSAT [GershansStrichman’08] variant keeps clauses in a queue
  - queue can also be used to find less important clauses to throw away
  - refined version in PrecoSAT [Biere’09] (multiple queues per glucose level)
How to Compute the Score?

- SAT solver picks unassigned variable with largest score as next decision
  - consider only change of the score $s_i$ of one variable $v$ during $i$-th conflict
  - let $\beta_i = 1$ if $v$ is bumped in the $i$-th conflict otherwise 0

- some possible variable score update functions:
  - static $s_{i+1} = s_i$ initialize score statically and do not change it
  - inc $s_{i+1} = s_i + \beta_i$ this is in essence DLIS from Grasp
  - vmtf $s_{i+1} = i$
  - sum $s_{i+1} = s_i + i \cdot \beta_i$ emphasis on recent conflicts
  - vsids $s_{i+1} = d \cdot s_i + \beta_i$ decay $d \in [0, 1)$ e.g. $d = 0.95$
  - evsids $s_{i+1} = s_i + g_i \cdot \beta_i$, $g_{i+1} = e \cdot g_i$ factor $e \in [1, 2)$ e.g. $e = 1.05$
  - avg $s_{i+1} = s_i + \beta_i \cdot (i - s_i)/2$ another filter function

- last four share the idea of “low-pass filtering” of the involvement of variables
  - for this interpretation see our SAT’08 paper and the video
  - important practical issue: number of bumped variables is usually small
Run-Time Distribution (Time Limit 1000 seconds)

SAT Competition 2013 Application Track Benchmarks Solved by Lingeling

- static
- inc
- sum
- vmtf
- vsids256
- evsids
- avg
- sc13
should not keep all learned clauses forever
  - some of them become useless
  - for instance subsumed or satisfied under learned units
  - were but are not anymore relevant to current search focus
  - memory consumption / BCP speed

throw *unimportant* learned clauses away (reduce)
  - in regular intervals (controlled by geometric, Luby, arithmetic scheme)
  - *size* heuristics: discard long clauses
  - *least recently used* (LRU): as in HW cache (see also CMTF)
  - clause scores with bumping scheme as for VSDIS (BerkMin)
  - glucose level: number decision levels in learned clause
called also LBD in original paper [AudemardLaurentSimon’09]
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):
    \[ x = (a \land b), \quad y = (c \lor d), \quad z = (e \oplus f) \text{ etc.} \]
  - generalization of BCP to (in)equalities between variables
  - **test rule** splits on the two values of a variable

Recursive Learning (Kunz & Pradhan)
  - (originally) works on circuit structure (derives implications)
  - splits on different ways to *justify* a certain variable value
1. BCP over (in)equalities: 
\[
\begin{align*}
    x &= y & z &= (x \oplus y) \\
    z &= 0
\end{align*}
\]

\[
\begin{align*}
    x &= 0 & z &= (x \lor y) \\
    z &= y
\end{align*}
\]

etc.

2. structural rules:
\[
\begin{align*}
    x &= (a \lor b) & y &= (a \lor b) \\
    x &= y
\end{align*}
\]

etc.

3. test rule:
\[
\begin{align*}
\{x = 0\} \cup E & \quad \{x = 1\} \cup E \\
\Downarrow & \quad \Downarrow \\
E_0 \cup E & \quad E_1 \cup E \\
(E_0 \cap E_1) \cup E
\end{align*}
\]

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

\[
\begin{align*}
x &= 0 \\
\downarrow & \\
y &= 0 & y &= 1 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
E_{00} & E_{01} & E_{10} & E_{11} \\
\hline
E_0 & \hline
E_1 & \hline
\end{align*}
\]

(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 except test rule (just BCP: linear)
  - 1-saturation: apply test rule (not recursively) for all variables
    (until no new (in)equalities can be derived)
circuits

output 0 implies middle input 0 indirectly

CNF
- for each clause $c$ in the CNF
  - for each literal $l$ in the clause $c$
    - assume $l$ and propagate
    - collect set of all implied literals (direct/indirect “implications” of $l$)
  - intersect these sets of implied literals over all $l$ in $c$
  - literals in the intersection are implied without any assumption
use DP to existentially quantify out variables as in [DavisPutnam60]

- only remove a variable if this does not add (too many) clauses
  - do not count tautological resolvents
  - detect units on-the-fly

- schedule removal attempts with a priority queue [Biere SAT’04] [EénBiere SAT’05]
  - variables ordered by the number of occurrences

- strengthen and remove subsumed clauses (on-the-fly)
  (SATeLite [EénBiere SAT’05] and Quantor [Biere SAT’04])
Fast (Self) Subsumption

- for each (new or strengthened) clause
  - traverse list of clauses of the least occurring literal in the clause
  - check whether traversed clauses are subsumed or
  - strengthen traversed clauses by self-subsumption [EénBiere SAT’05]
  - use Bloom Filters (as in “bit-state hashing”), aka signatures

- checking new clauses against existing clauses: **backward (self) subsumption**
  - new clause (self) subsumes existing clause
  - new clause smaller or equal in size

- check clause being subsumed by existing clauses **forward (self) subsumption**
  - can be made more efficient by one-watcher scheme [Zhang-SAT’05]
for all iterals $l$
  - for all clauses $c$ in which $l$ occurs (with this particular phase)
    - assume the negation of all the other literals in $c$, assume $l$
    - if BCP does not lead to a conflict continue with next literal in outer loop
  - if all clauses produced a conflict permanently assign $\neg l$

**Correctness:** Let $c = l \lor d$, assume $\neg d \land l$.

If this leads to a conflict $d \lor \neg l$ could be learned (but is not added to the CNF).

Self subsuming resolution with $c$ results in $d$ and $c$ is removed.

If all such cases lead to a conflict, $\neg l$ becomes a pure literal.
Generalization of pure literals.

Given a partial assignment \( \sigma \).

A clause of a CNF is “touched” by \( \sigma \) if it contains a literal assigned by \( \sigma \).

A clause of a CNF is “satisfied” by \( \sigma \) if it contains a literal assigned to true by \( \sigma \).

If all touched clauses are satisfied then \( \sigma \) is an “autarky”.

All clauses touched by an autarky can be removed.

Example: \((-1\ 2)(-1\ 3)(1\ -2\ -3)(2\ 5)\cdots\) (more clauses without 1 and 3).

Then \( \sigma = \{-1, -3\} \) is an autarky.
[Kullman’99]

fixed clause \( C \in F \) all clauses in \( F \) with \( \overline{l} \)

fix a CNF \( F \)

\[(\overline{l} \lor \overline{a} \lor c)\]

\[(a \lor b \lor l)\]

\[(\overline{l} \lor \overline{b} \lor d)\]

since all resolvents of \( C \) on \( l \) are tautological \( C \) can be removed

Proof

assignment \( \sigma \) satisfying \( F \setminus C \) but not \( C \)

can be extended to a satisfying assignment of \( F \) by flipping value of \( l \)
Blocked Clauses and Encoding / Preprocessing Techniques

[JärvisaloBiereHeule-TACAS’10]

**COI**  Cone-of-Influence reduction

**MIR**  Monontone-Input-Reduction

**NSI**  Non-Shared Inputs reduction

**PG**  Plaisted-Greenbaum polarity based encoding

**TST**  standard Tseitin encoding

**VE**  Variable-Elimination as in DP / Quantor / SATeLite

**BCE**  Blocked-Clause-Elimination
PrecoSAT [Biere’09], Lingeling [Biere’10], also in CryptoMiniSAT (Mate Soos)

- preprocessing can be extremely beneficial
  - most SAT competition solvers use variable elimination (VE) [EénBiere SAT’05]
  - equivalence / XOR reasoning
  - probing / failed literal preprocessing / hyper binary resolution
  - however, even though polynomial, can not be run until completion

- simple idea to benefit from full preprocessing without penalty
  - “preempt” preprocessors after some time
  - resume preprocessing between restarts
  - limit preprocessing time in relation to search time
Benefits of Inprocessing

- special case *incremental preprocessing*:
  - preprocessing during incremental SAT solving
- allows to use *costly* preprocessors
  - without increasing run-time “much” in the worst-case
  - still useful for benchmarks where these costly techniques help
  - good examples: probing and distillation
    - even VE can be costly
- additional benefit:
  - makes units / equivalences learned in search available to preprocessing
    - particularly interesting if preprocessing simulates encoding optimizations
- danger of hiding “bad” implementation though …
- … and hard(er) to debug and get right
  - [JävisaloHeuleBiere’12]
ZChaff Occurrence Stacks

SAT Solving #342.201 WS 2019

Armin Biere JKU Linz
Average Number Clauses Visited Per Propagation

SAT Solving #342.201 WS 2019 Armin Biere JKU Linz
Average Learned Clause Length

SAT Solving #342.201 WS 2019 Armin Biere JKU Linz

Version 2019.3
Watcher of A

Watcher of B
invariant: first two literals are watched
invariant: first two literals are watched
Additional Binary Clause Watcher Stack

-2
start

-2 −3

1

-3

end

-3 −2

Additional Binary Clause Watcher Stack
observation: often the other watched literal satisfies the clause
so cache this literals in watch list to avoid pointer dereference
for binary clause no need to store clause at all

can easily be adjusted for ternary clauses (with full occurrence lists)

LINGELING uses more compact pointer-less variant
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
- 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
the two traffic lights should never show a green light at the same time
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 infinite state spaces as well

TLC example:
- single assignment $\sigma: \{\text{southnorth, eastwest}\} \rightarrow \{\text{green, yellow, red}\}$
- set of assignments is isomorphic to $\{\text{green, yellow, red}\}^2$
- eg state space is isomorphic to the crossproduct of variable ranges

not all states are reachable: $(\text{green, green})$
- 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
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;
      TRUE : yellow;
    esac;

  next (back) :=
    case
      light = red & enable : FALSE;
      light = green : TRUE;
      TRUE : back;
    esac;

MODULE main

VAR
  southnorth : trafficlight (TRUE);
  eastwest : trafficlight (TRUE);

SPEC
  AG !(southnorth.light = green & eastwest.light = green)
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

1 red
0 red
0 red
1 red
1 red
1 red
0 red
1 red
1 red
0 red
1 yellow
0 yellow
0 yellow
1 green
0 green
0 green
1 yellow
0 yellow
0 yellow
enable light back

Version 2019.3
7 reachable states out of 144 states (or 72 if only one 'enable')

unfortunately buggy state is among them
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
traffic lights showing red should eventually show green
traffic lights showing red should eventually show green
traffic lights showing red should eventually show green
- 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

<table>
<thead>
<tr>
<th>light</th>
<th>light@1</th>
<th>light@0</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>yellow</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>red</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
• initial state predicate \( I \) represented as boolean formula

\[
!\text{eastwest.light}@0 \ & \ \text{eastwest.light}@1
\]

(equivalent to \( \text{init(eastwest.light)} := \text{red} \))

• transition relation \( T \) represented as boolean formula

• encoding of atomic predicates \( p \) as boolean formulae

\[
!\text{eastwest.light}@1 \ & \ !\text{eastwest.light}@0
\]

(equivalent to \( \text{eastwest.light} \neq \text{green} \))
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
checking safety property $G_p$ for a bound $k$ as SAT problem:

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

check occurrence of $\neg p$ in the first $k$ states
generic counter example trace of length $k$ for liveness $\mathbb{F}p$

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

(However we recently showed that liveness can always be reformulated as safety [BiereArthoSchuppan02])
Time Frame Expansion in HW

sequential feedback loop

sequential circuit

combinational logic

inputs

states

outputs
Time Frame Expansion in HW

inputs

states

states

outputs

break sequential loop
Time Frame Expansion in HW

inputs

states

outputs

inputs

states

outputs

added 1st copy
Time Frame Expansion in HW

![Diagram showing expanded time frame with inputs, states, and outputs. There is a note indicating "added 2nd copy." ]
Time Frame Expansion in HW

added 3rd copy
Time Frame Expansion in HW

inputs
states
outputs
inputs
states
outputs
inputs
states
outputs
inputs
states
outputs
inputs
states
outputs

added 4th copy
Time Frame Expansion in HW
find inputs for which failed becomes true
find inputs for which \texttt{failed} becomes true
find bounds on the maximal length of counter examples
  - also called \textit{completeness threshold}
  - exact bounds are hard to find $\Rightarrow$ approximations

induction
  - use inductive invariants as we have seen before
  - generalization of inductive invariants: \textit{pseudo induction}

use SAT for quantifier elimination as with BDDs
  - then model checking becomes fixpoint calculation
**Distance**: length of shortest path between two states

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

(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

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

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

Initial states

Unreachable states

States with distance 1 from initial states

Single state with distance 2 from initial states

Diameter 4, radius 2

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

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

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

\[ \forall s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_{r-1} \rightarrow s_r \rightarrow s_{r+1} \]

\[ (t_r = s_{r+1}) \]

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

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

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 $n$
for $k = 0 \ldots \infty$, check

1. $k$-induction base case:

$$I(s_0) \land T(s_0, s_1) \land \ldots \land T(s_{k-1}, s_k) \land B(s_k) \land \bigwedge_{0 \leq i < k} \neg B(s_i) \quad \text{satisfiable?}$$

2. $k$-induction induction step:

$$T(s_0, s_1) \land \ldots \land T(s_{k-1}, s_k) \land B(s_k) \land \bigwedge_{0 \leq i < k} \neg B(s_i) \quad \text{unsatisfiable?}$$

if base case satisfiable (= BMC), then bad state reachable

if inductive step unsatisfiable, then bad state unreachable

incomplete without simple path constraints
Incremental SAT Solving for BMC and $k$-Induction

[EénSörensson’03]

$k = 0$ base case
\[ B \]

\[ k = 0 \text{ inductive step} \]
\[ k = 1 \quad \text{base case} \]
\[ k = 1 \quad \text{inductive step} \]
$k = 2 \quad \text{base case}$
$\mathbf{T}$ $\mathbf{B}$

$\mathbf{T}$

$k = 2$ inductive step
$k = 3 \quad \text{base case}$
$k = 3$ inductive step
k = 4  base case
\[ k = 4 \quad \text{inductive step} \]
$k = 5$ base case
Incremental SAT Solving for BMC and $k$-Induction


$T$  $T$  $T$

$T$  $T$  $B$

$k = 5$  inductive step
$k = 6$  base case
$k = 6$ inductive step
Simple Path Constraints

- bounded model checking: \[\text{[BiereCimattiClarkeZhu'99]}\]
  \[I(s_1) \land T(s_1, s_2) \land \ldots \land T(s_{k-1}, s_k) \land \bigvee_{0 \leq i \leq k} B(s_i)\] satisfiable?

- reoccurrence diameter checking: \[\text{[BiereCimattiClarkeZhu'99]}\]
  \[T(s_1, s_2) \land \ldots \land T(s_{k-1}, s_k) \land \bigwedge_{1 \leq i < j \leq k} s_i \neq s_j\] unsatisfiable?

- \(k\)-induction base case: \[\text{[SheeranSinghStålmarck'00]}\]
  \[I(s_1) \land T(s_1, s_2) \land \ldots \land T(s_{k-1}, s_k) \land B(s_k) \land \bigwedge_{0 \leq i < k} \neg B(s_i)\] satisfiable?

- \(k\)-induction induction step: \[\text{[SheeranSinghStålmarck'00]}\]
  \[T(s_1, s_2) \land \ldots \land T(s_{k-1}, s_k) \land B(s_k) \land \bigwedge_{0 \leq i < k} \neg B(s_i) \land \bigwedge_{1 \leq i < j \leq k} s_i \neq s_j\] unsatisfiable?

- automatic abstraction refinement = lemmas on demand of simple path constraints \[\text{[EénSörensson'03]}\]
let $G = \neg B$ denote the “good states”:

- 0-induction base case: $I(s_0) \land B(s_0)$ satisfiable iff initial bad state exists

- 0-induction inductive step: $B(s_0)$ unsatisfiable iff $\neg B$ propositional tautology

- 1-induction base: $I(s_0) \land T(s_0, s_1) \land B(s_1)$ satisfiable iff bad state reachable in one step

- 1-induction inductive step: $\neg B(s_0) \land T(s_0, s_1) \land B(s_1)$ unsatisfiable iff $G$ inductive

assuming 0-induction base case was unsatisfiable and thus $I \models G$

where $G = \neg B$ is called inductive iff 1. $I \models G$ and 2. $G \land T \models G'$
task is to prove that $p$ is an invariant

- guess a formula $G$ stronger than $p$: $G \models p$  

- show $G$ inductive: $I \models G$, $G \land T \models G'$  

- all three checks can be formulated as UNSAT checks  

- if one check fails refine $G$ based on satisfying assignment

manual process and thus **complete** on finite state systems

there are also automatic abstraction/refinement versions of this approach  

CEGAR [ClarkeGrumbergJhaLuVeith’00]
**Definition**  

$I$ interpolant of $A$ and $B$ iff $A \Rightarrow I$, $V(I) \subseteq V(A) \cap V(B)$ and $I \land B$ unsat.

**Note:** $A \land B$ unsatisfiable as a consequence.

**Intuition:** $I$ is an abstraction of $A$ over the common (interface) variables of $A$ and $B$ which still is inconsistent with $B$.

Let $A$ and $B$ formulas in CNF.

From a resolution proof in a refutation of $A \land B$ generate interpolant $I$ (next slide)

This is used in many applications, generalizations exists, also gives one of the fastest model checking algorithms.
Interpolation Algorithm

[McMillan’03, McMillan’05] + [Biere’09] (BMC chapter in Handbook)

**Definition**

interpolating quadruple \( (A, B) \ c \ [f] \) is **well-formed** iff

\[
\begin{align*}
(W1) \quad & V(c) \subseteq V(A) \cup V(B) \\
(W2) \quad & V(f) \subseteq G \cup (V(c) \cap V(A)) \subseteq V(A)
\end{align*}
\]

**Definition**

well-formed interpolating quadruple \( (A, B) \ c \ [f] \) is **valid** iff

\[
\begin{align*}
(V1) \quad & A \Rightarrow f \\
(V2) \quad & B \land f \Rightarrow c
\end{align*}
\]

**Definition**

proof rules for interpolating quadrupels

\[
\begin{align*}
(R1) \quad & c \in A \quad \quad (A, B) \ c \ [c] \\
(R2) \quad & c \in B \quad \quad (A, B) \ c \ [\top]
\end{align*}
\]

\[
\begin{align*}
& (A, B) \ c \ \lor \ l \ [f] \quad (A, B) \ d \ \lor \ \bar{l} \ [g] \\
\Rightarrow & (A, B) \ c \land d \ [f \land g]
\end{align*}
\]

\[
\begin{align*}
& (A, B) \ c \ \lor \ l \ [f] \quad (A, B) \ d \ \lor \ \bar{l} \ [g] \\
\Rightarrow & (A, B) \ c \land d \ [f \lor g \lor l]
\end{align*}
\]

**Theorem**

proof rules produce well-formed and valid interpolating quadruples
Interpolation-based Model Checking

\[ I(s_{-1}) \land T(s_{-1}, s_0) \land T(s_0, s_1) \land T(s_1, s_2) \land T(s_2, s_3) \land \bigvee_{i=0}^{3} \neg G(s_i) \]

interpolant \( P_1(s_0) \) let \( R_1 \equiv I \lor P_1 \)

\[ R_1(s_{-1}) \land T(s_{-1}, s_0) \land T(s_0, s_1) \land T(s_1, s_2) \land T(s_2, s_3) \land \bigvee_{i=0}^{3} \neg G(s_i) \]

interpolant \( P_2(s_0) \leftarrow R_1(s_{-1}) \land T(s_{-1}, s_0) \) let \( R_2 \equiv R_1 \lor P_2 \)

\[ R_2(s_{-1}) \land T(s_{-1}, s_0) \land T(s_0, s_1) \land T(s_1, s_2) \land T(s_2, s_3) \land \bigvee_{i=0}^{3} \neg G(s_i) \]

\[ \vdots \]

\[ R_{n-1}(s_{-1}) \land T(s_{-1}, s_0) \land T(s_0, s_1) \land T(s_1, s_2) \land T(s_2, s_3) \land \bigvee_{i=0}^{3} \neg G(s_i) \]

interpolant \( P_n(s_0) \)

until \( R_n \equiv R_{n-1} \) fix-point guaranteed for \( k = \text{backward radius of } \neg G \)
(E)LTL formula in NNF

let the path $\pi$ be a $(k, l)$ lasso

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

$$\pi \models^i_k \neg p \quad \text{iff} \quad p \not\in L(\pi(i))$$

$$\pi \models^i_k f \land g \quad \text{iff} \quad \pi \models^i_k f \text{ and } \pi \models^i_k g$$

$$\pi \models^i_k Xf \quad \text{iff} \quad \begin{cases} 
\pi \models^l_k f & \text{if } i = k \\
\pi \models^{i+1}_k f & \text{else}
\end{cases}$$

$$\pi \models^i_k Gf \quad \text{iff} \quad \bigwedge_{j = \min(i, l)}^k \pi \models^j_k f$$

$$\pi \models^i_k Ff \quad \text{iff} \quad \bigvee_{j = \min(i, l)}^k \pi \models^j_k f$$
there is no \( l \) for which path \( \pi \) is a \((k, l)\) lasso

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

\[
\pi \models_k^i \neg p \quad \text{iff} \quad p \not\in L(\pi(i))
\]

\[
\pi \models_k^i f \land g \quad \text{iff} \quad \pi \models_k^i f \quad \text{and} \quad \pi \models_k^i g
\]

\[
\pi \models_k^i Xf \quad \text{iff} \quad \begin{cases} 
\text{false} & \text{if } i = k \\
\pi \models_k^{i+1} f & \text{else}
\end{cases}
\]

\[
\pi \models_k^i Gf \quad \text{iff} \quad \text{false}
\]

\[
\pi \models_k^i Ff \quad \text{iff} \quad \bigvee_{j=i}^{k} \pi \models_k^j f
\]
Bounded Semantics

- **definition:**
  \[
  \pi \models_k f \iff \pi \models^0_k f
  \]

- bounded semantics approximates real semantics:
  \[
  \pi \models_k f \implies \pi \models f \quad \text{for all } k
  \]

- (theoretical) completeness:
  \[
  \text{if } \pi \models f \text{ then there exists } k \text{ with } \pi_k \models f
  \]

- **note:** negate original property first (e.g. \(\text{AG} p \leftrightarrow \text{EF} \neg p\))
  - ALTL \(\rightarrow\) ELTL
  - counter example \(\rightarrow\) witness
  - *bounded* witness is also a non-bounded witness
two recursive translations from (E)LTL in NNF for fixed \( k \):
- \( l[\cdot]^i_k \) assumes \((k, l)\)-loop
- \([\cdot]^i_k \) assumes that no \((k, l)\)-loop exists for all \( l \)

add time frame expansion of transition relation:

\[
I(s_0) \land T(s_0, s_1) \land \cdots \land T(s_{k-1}, s_k)
\]

add \( \text{loop}_k(l) \) constraint for looping translation:

\[
\text{loop}_k(l) := T(s_k, s_l)
\]

add \( \text{noloop}_k \) constraint for non-looping translation:

\[
\text{noloop}_k := \neg \bigvee_{l=0}^{k} \text{loop}_k(l)
\]
\[
\begin{align*}
  l[p]^i_k & := p(s_i) \\
  l[\neg p]^i_k & := \neg p(s_i) \\
  l[f \land g]^i_k & := l[f]^i_k \land l[g]^i_k \\
  l[\textbf{X} f]^i_k & := l[f]^i_{\text{next}(i)} \\
  l[\textbf{G} f]^i_k & := \bigwedge_{j=\min(l,i)}^k l[f]^i_k \\
  l[\textbf{F} f]^i_k & := \bigvee_{j=\min(l,i)}^k l[f]^i_k \\
\end{align*}
\]
with
\[
\text{next}(i) := \begin{cases} 
  i + 1 & \text{if } i < k \\
  l & \text{else}
\end{cases}
\]
Non-Looping Translation

\[
[p]_k^i := p(s_i) \\
[\neg p]_k^i := \neg p(s_i) \\
[f \land g]_k^i := [f]_k^i \land [g]_k^i \\
[X f]_k^i := \begin{cases} [f]_{k+1}^i & \text{if } i < k \\ false & \text{else} \end{cases} \\
[G f]_k^i := false \\
[F f]_k^i := \bigvee_{j=i}^{k} [f]_k^j
\]
Theorem: \( K \models Ef \iff \exists k \left[ K, f \right]_k \) satisfiable

- \( i^j \) and \( [\cdot]^j_k \) 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 \( R \) and \( U \)

\[
\left[ K, f \right]_k := \text{noloop}_k \land \left[ f \right]_k^0 \lor \bigvee_{l=0}^k \text{loop}_k(l) \land l \left[ f \right]_k^0
\]
original translation of $FGp$ after applying associativity and sharing

with $L_i = \text{loop}_k(i)$ and $k = 3$

(could be simplified further)
Simple and Linear Translation for LTL

[LatvalaBiereHeljankoJunttila FMCAD’04]

**evaluate semantics on loop in two iterations**

\[
\langle \rangle = \text{1st iteration} \quad [ ] = \text{2nd iteration}
\]

<table>
<thead>
<tr>
<th></th>
<th>(i &lt; k)</th>
<th>(i = k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [p]_i )</td>
<td>(p(s_i))</td>
<td>(p(s_k))</td>
</tr>
<tr>
<td>( [\neg p]_i )</td>
<td>(\neg p(s_i))</td>
<td>(\neg p(s_k))</td>
</tr>
<tr>
<td>( [Xf]_i )</td>
<td>( [f]_{i+1} )</td>
<td>( \bigvee_{i=0}^{k} (T(s_k,s_l) \land [f]_l) )</td>
</tr>
<tr>
<td>( [Gf]_i )</td>
<td>( [f]<em>i \land [Gf]</em>{i+1} )</td>
<td>( \bigvee_{i=0}^{k} (T(s_k,s_l) \land \langle Gf \rangle_l) )</td>
</tr>
<tr>
<td>( [Ff]_i )</td>
<td>( [f]<em>i \lor [Ff]</em>{i+1} )</td>
<td>( \bigvee_{i=0}^{k} (T(s_k,s_l) \land \langle Ff \rangle_l) )</td>
</tr>
<tr>
<td>( \langle Gf \rangle_i )</td>
<td>( [f]<em>i \land \langle Gf \rangle</em>{i+1} )</td>
<td>( [f]_k )</td>
</tr>
<tr>
<td>( \langle Ff \rangle_i )</td>
<td>( [f]<em>i \lor \langle Ff \rangle</em>{i+1} )</td>
<td>( [f]_k )</td>
</tr>
</tbody>
</table>
● semantic of LTL on *single path* is the same as CTL semantic
  ● symbolically implement fixpoint calculation for (A)CTL
  ● fixpoint computation terminates after 2 iterations (not \( k \))
  ● boolean fixpoint equations \( \Rightarrow \) boolean graphs

● easy to implement and optimize, fast
  ● generalized to past time \([\text{LatvalaBiereHeljankoJunttila VMCAI’05}]\)
  ● minimal counter examples for past time \([\text{SchuppanBiere TACAS’05}]\)
  ● incremental (and complete) \([\text{LatvalaHeljankoJunttila CAV’05}]\)
Why Not Just Try to Satisfy Boolean Equations directly?

recursive expansion $\neg p \equiv p \lor XFp$

checking $G\neg p$ implemented as search for witness for $Fp$

Kripke structure: single state with self loop in which $p$ does not hold

incorrect translation of $Fp$:

\[
\begin{align*}
I(s_0) \land T(s_0, s_0) \land \left( [Fp] \leftrightarrow p(s_0) \lor [Fp] \right) \land Fp
\end{align*}
\]

since it is satisfiable by setting $x = 1$ though $p(s_0) = 0$

($x$ fresh boolean variable introduced for $[Fp]$)
key concept in IC3 [Bradley’11]:

| clause $c$ relative inductive w.r.t. $F$ | iff $c \land F \land T \Rightarrow c'$ iff $c \land F \land T \land \overline{c'}$ unsatisfiable |

\[
\begin{array}{|c|c|c|}
\hline
F_0 & F_1 & F_2 \\
\hline
I & G & B \\
\hline
\end{array}
\]

$I$ initial states
$G$ good states
$B$ bad states

(1) $s$ is reachable from $F_0$ then bad is reachable transitively

(2) otherwise exists $c \subseteq \overline{s}$ rel. ind. w.r.t. $F_0$ and can be added to $F_1$ and maybe to $F_2$
as soon the last set is good, i.e. $F_k \Rightarrow G$ increase $k$

propagate all relative inductive clauses of last set to new set

if all can been propagated $F_k$ is an inductive invariant stronger than $G$
More Precisely …

Let $F_0, \ldots, F_k$ be a sequence of sets of clauses.

monotonic iff $F_i \supseteq F_{i+1}$ for $i = 0 \ldots k - 1$

(relative) inductive iff $F_i T \Rightarrow F'_{i+1}$ for $i = 0 \ldots k - 1$

initialized iff $I \equiv F_0$

good iff $F_i \Rightarrow G$ for $i = 0 \ldots k - 1$ last set might be bad if $F_k \land B$ satisfiable

$F$ is $k$-adequat iff all states $s$ satisfying $F$ are at least $k$ steps away from $B$

[McMillan’03]

sequence monotonic and inductive $\Rightarrow F_{k-j}$ $j$-adequat
Sketch of the Algorithm

CHECK \((s, i)\) \{ 
    actually should be DFS prioritized on \(i\)
    while \(\bar{s} \land F_{i-1} \land T \land s'\) satisfiable \{
        if \(i = 1\) throw SATISFIABLE
        choose cube \(t\) with \(t \models \bar{s} \land F_{i-1} \land T \land s'\)
        CHECK \((t, i - 1)\)
    \}
    choose clause \(c \subseteq \bar{s}\) with \(c \land F_{i-1} \land T \land \bar{c}'\) unsatisfiable
    \(F_j := F_j \cup \{c\}\) for all \(j = 1 \ldots i\)
    \}

MAIN () \{
    do not forget to check base cases first
    \(F_0 = I, \quad F_1 = \top, \quad k = 1\)
    forever \{
        CHECK \((B, k)\)
        \(k := k + 1, \quad F_i := \text{all rel. ind. clauses of } F_{i-1} \text{ w.r.t. } F_{i-1} \text{ for } i = 1 \ldots k\)
        if \(F_k \subseteq F_{k-1}\) throw UNSATISFIABLE
    \}
\}
implemented in IC3 by Aaron Bradley
- as single engine model checker extremely successful in HWMCC’10 Hardware Model Checking Competition 2010
- based on rather out-dated SAT solver (ZChaff from 2004)

- independent implementations such as [EénMishchenkoBrayton IWLS’11]
  - seem to be faster than BDDs, $k$-induction, interpolation
  - might be much easier to lift to SMT-based model checking than interpolation
  - opportunities for improvement: structural SAT/SMT solving