Systemtheory 2
Formal Systems 2
#342201

SS 2006
Johannes Kepler Universität
Linz, Österreich

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

http://fmv.jku.at/fs2
Motivation

• more and more complex systems

Moore’s Law $\Rightarrow$ 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
• not unusual to have more than 50% of resources allocated to testing

• testing and verification are (becoming) the bottleneck of development

• quality dilemma (drop quality for more features)

• more efficient methods for test and verification needed
  ⇒ formal verification is the most promising approach

• experts in new testing and verification methods are lacking

• long term: more formal development process not just formal verification
• formal = 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)
• 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
• 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
  \[ \Rightarrow \] more powerful verification algorithms allow more automation

• good example: B-Method
### Layered System Design

<table>
<thead>
<tr>
<th>HW</th>
<th>SW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Requirements</td>
</tr>
<tr>
<td>RTL</td>
<td>High-Level Design</td>
</tr>
<tr>
<td>Gate</td>
<td>Low-Level Design</td>
</tr>
<tr>
<td>Transistor</td>
<td>Implementation</td>
</tr>
</tbody>
</table>

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

---

**Synthesis**

**Verification**
• assumptions: specification and system are given

• formal verification checks formally that system fulfills specification

• least change in development process

• full blown verification is really difficult: “post mortem verification”

• simplifications: focus on simple partial specifications
  (type safety, functional equivalence of two systems, ...)

• methods (implemented in tools):
  simple algorithms for deducing properties directly
  complex algorithms for hard or even undecidable problems
• boolean methods:
  
 SAT, BDDs, ATPG, Combinational Equivalence Checking

• finite state methods:
  
  Bisimulation and Equivalence Checking of Automata, Model Checking

• term based methods:
  
  Term Rewriting, Resolution, Tableaux, Theorem Proving

• Abstraction (eg SLAM uses BDDs, Model Checking, Theorem Proving)
• 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

<table>
<thead>
<tr>
<th>original C code</th>
<th>optimized C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(!a &amp;&amp; !b) h(); else if(!a) g(); else f();</td>
<td>if(a) f(); else if(b) g(); else h();</td>
</tr>
<tr>
<td>if(!a) { if(a) f(); if(!b) h(); else g(); } else f();</td>
<td>if(a) f(); else { if(!b) h(); else g(); }</td>
</tr>
</tbody>
</table>

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

\[
\text{original} :=
\]

\[
\text{if } \neg a \land \neg b \text{ then } h
\]
\[
\text{else if } \neg a \text{ then } g
\]
\[
\text{else } f
\]

\[
\text{optimized} :=
\]

\[
\text{if } a \text{ then } f
\]
\[
\text{else if } b \text{ then } g
\]
\[
\text{else } h
\]

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

\[
\text{compile}([\text{original}]) \iff \text{compile}([\text{optimized}])
\]
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)

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)
How to Check (In)Equivalence?

Reformulate it as a satisfiability (SAT) problem:

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

or equivalently:

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

\[ b \lor a \land c \]

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

Are these circuits equivalent?
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)
Conjunctive Normal Form

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

CNF

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

\( o \land (\bar{x} \lor a) \land (\bar{x} \lor c) \land (x \lor \bar{a} \lor \bar{c}) \land \ldots \)
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 and only the original formula is satisfiable

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

extract satisfying assignment for original formula, from one of the result
(just project satisfying assignment onto the original variables)
Tseitin Transformation: Input / Output Constraints

Negation: \[ x \leftrightarrow \neg y \quad \iff \quad (x \rightarrow \neg y) \land (\neg y \rightarrow x) \]
\[ \quad \iff \quad (\neg x \lor \neg y) \land (y \lor x) \]

Disjunction: \[ x \leftrightarrow (y \lor z) \quad \iff \quad (y \rightarrow x) \land (z \rightarrow x) \land (x \rightarrow (y \lor z)) \]
\[ \quad \iff \quad (\neg y \lor x) \land (\neg z \lor x) \land (\neg x \lor y \lor z) \]

Conjunction: \[ x \leftrightarrow (y \land z) \quad \iff \quad (x \rightarrow y) \land (x \rightarrow z) \land ((y \land z) \rightarrow x) \]
\[ \quad \iff \quad (\neg x \lor y) \land (\neg x \lor z) \land (\neg (y \land z) \lor x) \]
\[ \quad \iff \quad (\neg x \lor y) \land (\neg x \lor z) \land (\neg y \lor \neg z \lor x) \]

Equivalence: \[ x \leftrightarrow (y \leftrightarrow z) \quad \iff \quad (x \rightarrow (y \leftrightarrow z)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \quad \iff \quad (x \rightarrow ((y \rightarrow z) \land (z \rightarrow y))) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \quad \iff \quad (x \rightarrow (y \rightarrow z)) \land (x \rightarrow (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \quad \iff \quad (\neg x \lor \neg y \lor z) \land (\neg x \lor \neg z \lor y) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \quad \iff \quad (\neg x \lor \neg y \lor z) \land (\neg x \lor \neg z \lor y) \land ((y \land z) \lor (\neg y \land \neg z) \lor x) \]
\[ \quad \iff \quad (\neg x \lor \neg y \lor z) \land (\neg x \lor \neg z \lor y) \land ((y \land z) \lor (\neg y \land \neg z) \lor x) \]
\[ \quad \iff \quad (\neg x \lor \neg y \lor z) \land (\neg x \lor \neg z \lor y) \land (\neg \neg y \lor \neg z \lor x) \land (y \lor z \lor x) \]
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 may be removed for *unnegated* nodes
  
a node occurs negated if it has an ancestor which is a negation

  half of the constraints determine parent assignment from child assignment

  those are unnecessary if node is not used negated

- those have to be carefully applied to DAG structure
  
  (compare with the implementation of the BMC tool from CMU)
• dates back to the 50ies:
  
  original version is *resolution based* (less successful)

• **idea:** case analysis (try \(x = 0, 1\) in turn and recurse)

• most successful SAT solvers (autumn 2003)
  
  works for very large instances

• recent (\(\leq 10\) years) optimizations:
  
  backjumping, learning, UIPs, dynamic splitting heuristics, fast data structures
  
  (we will have a look at each of them)
Resolution

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

Resolution Rule

\[
C \cup \{v\} \quad D \cup \{\neg v\}
\]

\[
\frac{\{v, \neg v\} \cap C = \{v, \neg v\} \cap D = 0}{C \cup D}
\]

**Read:** resolving the clause \(C \cup \{v\}\) with the clause \(D \cup \{\neg v\}\), both above the line, on the variable \(v\), results in the clause \(D \cup C\) below the line.
Correctness of Resolution Rule

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

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

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

**Proof.**

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

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

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

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

q.e.d.
Resolution Based DP

Idea: use resolution to existentially quantify out variables

1. if empty clause found then terminate with result unsatisfiable

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

3. remove all clauses in which these variables occur

4. if no clause left then terminate with result satisfiable

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

6. add results of all possible resolutions on this variable

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

8. continue with 1.
check whether XOR is weaker than OR, i.e. validity of:

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

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:

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

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 \((f \text{ unsat } \Rightarrow g \text{ unsat}), \text{ or equivalently } (g \text{ sat } \Rightarrow f \text{ sat})\)

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

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

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.
Problems with Resolution Based DP

- 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 will later see how BDDs can be used to overcome some of these problems)

- How to obtain the satisfying assignment efficiently (counter example)?
Second version of DP

- resolution based version often called DP, second version DPLL
  (DP after [DavisPutnam60] and DPLL after [DavisLogemannLoveland62])

- it eliminates variables through case analysis: time vs space

- only unit resolution used (also called boolean constraint propagation)

- case analysis is on-the-fly:
  cases are not elaborated in a predefined fixed order, but . . .
  . . . only remaining crucial cases have to be considered

- allows sophisticated optimizations
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**

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

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
DPLL Example

\((\neg a \vee b) \land (a \vee \neg b) \land (\neg a \vee \neg b)\)

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

\((\neg a \vee b) \land (a \vee \neg b) \land (\neg a \vee \neg b) \land a\)

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

\(b \land (\neg b)\)

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

\((\neg a \vee b) \land (a \vee \neg b) \land (\neg a \vee \neg b) \land \neg a\)

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:

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

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

Decision level

Control

Trail

Variables

Assignment

Clauses

X 1
X 2
X 3
X 4
X 5

−1 2
−2 3
−4 5

Assign

decision level  

Clause Variables

Assign

Variables

Clauses

1 1
X 2
X 3
X 4
X 5

-1 2
-2 3
-4 5
Example cont.

BCP

decision level

Control

Trail

Variables

Assignment

Clauses

1 | 1
1 | 2
1 | 3
X | 4
X | 5

-1 | 2
-2 | 3
-4 | 5
Assign

<table>
<thead>
<tr>
<th>Decision Level</th>
<th>Control</th>
<th>Trail</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Variables

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

Assignment

Example cont.
Example cont.

BCP

Decision level

Control

Trail

Assignment

Variables

Clauses

• **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 (Chaff)
Cut Width Heuristics

- view CNF as a graph:
  - clauses as nodes, edges between clauses with same variable

- a cut is a set of variables that splits the graph in two parts

- recursively find short cuts that cut off parts of the graph

- static or dynamically order variables according to the cuts

\[ \begin{array}{cccccc}
\cdots & -1 & 2 & 3 & -2 & 1 & -3 & 1 & 3 & -4 & \cdots \\
\end{array} \]

assume no occurrences of 1, 2, -1, -2 on the right side

short cut
```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.
• 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.
Other popular Decision Heuristics

- Dynamic Largest Individual Sum (DLIS)
  - fastest dynamic first order heuristic (eg GRASP solver)
  - choose literal (variable + phase) which occurs most often
  - ignore satisfied clauses
  - requires explicit traversal of CNF (or more expensive BCP)

- look-forward heuristics (eg SATZ solver)
  - do trial assignments and BCP for all unassigned variables (both phases)
  - if BCP leads to conflict, force toggled assignment of current trial decision
  - skip trial assignments implied by previous trial assignments
    (removes a factor of $|V|$ from the runtime of one decision search)
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 *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)
Optimized Implication Graph for Unit Resolution in DP

- graph becomes an ordinary (non hyper) directed graph

- simplifies implementation:
  - store a pointer to the reason clause with each assigned variable
  - decision variables just have a null pointer as reason
  - decisions are the roots of the graph
• 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)
$l = 0$ (no unit clause originally, so no implications)

$\vdash \bot$ (conflict)

$\bot$ (empty clause)

unit clause $-3$ is generated as learned clause and we backtrack to $l = 0$

$\vdash \bot$ (conflict)

since $-3$ has a real unit clause as reason, an empty conflict clause is learned
\[ l = 0 \] (no unit clause originally, so no implications)

\[ l = 1 \] (no implications on this decision level either)

\[ l = 2 \] (using the FIRST clause)

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

backtrack to \( l = 1 \) (smallest level for which conflict clause is a unit clause)
$l = 0$  
(no unit clause originally, so no implications)

$\vdash \neg 4 \land \neg 5 \land \neg 6$  
(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)

$\begin{array}{c}
\text{decision} \\
3 \rightarrow 4 \\
\end{array}$

$\begin{array}{c}
\text{empty clause} \\
(-3, 1, 2) \\
\text{(conflict)}
\end{array}$

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

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

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

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

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

  return 1;
}
• 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
a simple cut always exists: set of roots (decisions) contributing to the conflict
Unique Implication Points (UIP)

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

- can be found by graph traversal in the order of made assignments

- trail respects this order

- traverse reasons of variables on trail starting with conflict

- count “open paths”
  (initially size of clause with only false literals)

- if all paths converged at one node, then UIP is found

- decision of current decision level is a UIP and thus a sentinel
Further Options in Using UIPs

- assume a non decision UIP is found

- this UIP is part of a potential cut

- graph traversal may stop (everything *behind* the UIP is ignored)

- negation of the UIP literal constitutes the conflict driven assignment

- may start new clause generation (UIP replaces conflict)
  
  - each conflict may generate multiple learned clauses
  
  - however, using only the first UIP encountered seems to work best
• intuitively is is important to localize the search (cf cutwidth heuristics)

• cuts for learned clauses may only include UIPs of current decision level

• on lower decision levels an arbitrary cut can be chosen

• multiple alternatives
  – include all the roots contributing to the conflict
  – find minimal cut (heuristically)
  – cut off at first literal of lower decision level (works best)
• “second order” because it involves statistics about the search

• Variable State Independent Decaying Sum (VSIDS) decision heuristic (implemented in CHAFF and LIMMAT solver)

• VSIDS just counts the occurrences of a literals in conflict clauses

• literal with maximal count (score) is chosen

• score is multiple by a factor $f < 1$ after a certain number of conflicts occurred (this is the “decaying” part and also called rescoring)

• emphasizes (negation of) literals contributing recently to conflicts (localization)
BERKMIN’s Dynamic Second Order Heuristics

• observation:
  – recently added conflict clauses contain all the good variables of VSIDS
  – the order of those clauses is not used in VSIDS

• basic idea:
  – simply try to satisfy recently learned clauses first
  – use VSIDS to chose the decision variable for one clause
  – if all learned clauses are satisfied use other heuristics
  – intuitively obtains another order of localization (no proofs yet)

• results are mixed, but in general slightly more robust than just VSIDS
• for satisfiable instances the solver may get stuck in the unsatisfiable part
  – even if the search space contains a large satisfiable part

• often it is a good strategy to abandon the current search and restart
  – restart after the number of decisions reached a restart limit

• avoid to run into the same dead end
  – by randomization (either on the decision variable or its phase)
  – and/or just keep all the learned clauses

• for completeness dynamically increase restart limit
Other Types of Learning

• similar to look-ahead heuristics: polynomially bounded search
  – may be recursively applied (however, is often too expensive)

• Stålmarck’s Method
  – works on triplets (intermediate form of the Tseitin transformation):
    \[ x = (a \land b), \ y = (c \lor d), \ z = (e \oplus f) \] etc.
  – generalization of BCP to (in)equalities between variables
  – test rule splits on the two values of a variable

• Recursive Learning (Kunz & Pradhan)
  – works on circuit structure (derives implications)
  – splits on different ways to justify a certain variable value
Stålmarck’s Method

1. BCP over (in)equalities:
   \[
   \begin{align*}
   x &= y & \Rightarrow & & z = (x \oplus y) \\
   z &= 0 \quad \Rightarrow & & x = 0 & \Rightarrow & & z = (x \lor y) \\
   & & & \Rightarrow & & z = y
   \end{align*}
   \]
   etc.

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

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

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

\[ x = 0 \]
\[ \downarrow \]
\[ y = 0 \quad y = 1 \]
\[ \downarrow \quad \downarrow \]
\[ E_{00} \quad E_{01} \]

\[ E_0 \]

\[ x = 1 \]
\[ \downarrow \]
\[ y = 0 \quad y = 1 \]
\[ \downarrow \quad \downarrow \]
\[ E_{10} \quad E_{11} \]

\[ E_1 \]

\[ E \]

(we do not show the (in)equality (in)equalities that do not change)
• recursive application
  – depth of recursion bounded by number of variables
  – complete procedures (determines satisfiability or unsatisfiability)
  – for a fixed (constant) recursion depth $k$ polynomial!

• $k$-saturation:
  – apply split rule on recursively up to depth $k$ on all variables
  – 0-saturation: apply all rules accept test rule (just BCP: linear)
  – 1-saturation: apply test rule (not recursively) for all variables (until no new (in)equalities can be derived)
• 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
Traffic Light Controller (TLC)
Traffic Light Controller (TLC)
Traffic Light Controller (TLC)
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;
        esac;
    next (back) :=
        case
            light = red & enable : 0;
            light = green : 1;
        esac;

MODULE main

VAR
    southnorth : trafficlight (1);
    eastwest : trafficlight (1);

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

enable light back

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

$$!eastwest\.light@0 \& eastwest\.light@1$$

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

• transition relation $T$ represented as boolean formula

• encoding of atomic predicates $p$ as boolean formulae

$$!eastwest\.light@1 \& !eastwest\.light@0$$

(equivalent to $\text{eastwest\.light} \neq \text{green}$)
Bounded Model Checking

[BiereCimattiClarkeZhu99]

- uses SAT for model checking
  - historically not the first symbolic model checking approach
  - scales better than original BDD based techniques

- mostly incomplete in practice
  - validity of a formula can often not be proven
  - focus on counter example generation
  - only counter example up to certain length (the bound \( k \)) are searched
checking safety property $Gp$ 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 $\mathbf{F}p$

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

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

inputs
feedback
loop
states
outputs
sequential circuit

combinational logic
Time Frame Expansion in HW

inputs

<table>
<thead>
<tr>
<th>states</th>
<th>states</th>
</tr>
</thead>
<tbody>
<tr>
<td>outputs</td>
<td></td>
</tr>
</tbody>
</table>

break sequential loop
Time Frame Expansion in HW

`inputs` `inputs`

`states` `states` `states`
`outputs` `outputs` `outputs`

added 1st copy
Time Frame Expansion in HW

```
inputs
states
outputs
inputs
states
outputs
inputs
states
outputs

added 2nd copy
```
Time Frame Expansion in HW

inputs

states

outputs

inputs

states

outputs

inputs

states

outputs

inputs

states

outputs

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

inputs

observed signals
find inputs for which \textit{failed} becomes true
find inputs for which `failed` becomes true
Completeness in Bounded Model Checking

- find bounds on the maximal length of counter examples
  - also called completeness threshold
  - exact bounds are hard to find $\Rightarrow$ approximations

- induction
  - use inductive invariants as we have seen before
  - generalization of inductive invariants: pseudo induction

- use SAT for quantifier elimination as with BDDs (later)
  - then model checking becomes fixpoint calculation
Measuring Distances

**Distance:** length of shortest path between two states

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

(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
0 \rightarrow 1

unreachable states
5 \rightarrow 6 \rightarrow 7 \rightarrow 8
9

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
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:

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

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

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

∀

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

∃

\[
t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_{r-1}
\]

(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:

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

this is a propositional formula and can be checked by SAT

(exercise: reoccurrence radius/diameter is an upper bound on real radius/diameter)
radius 1, reoccurrence radius $n$
(E)LTL formula in NNF

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

$\pi \models^i_k p$ iff $p \in L(\pi(i))$

$\pi \models^i_k \neg p$ iff $p \not\in L(\pi(i))$

$\pi \models^i_k f \land g$ iff $\pi \models^i_k f$ and $\pi \models^i_k g$

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

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

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

\[
\begin{align*}
\pi \models^i_k p & \quad \text{iff} \quad p \in L(\pi(i)) \\
\pi \models^i_k \neg p & \quad \text{iff} \quad p \not\in L(\pi(i)) \\
\pi \models^i_k f \land g & \quad \text{iff} \quad \pi \models^i_k f \text{ and } \pi \models^i_k g \\
\pi \models^i_k Xf & \quad \text{iff} \quad \begin{cases} 
\text{false} & \text{if } i = k \\
\pi \models^i_{k+1} f & \text{else}
\end{cases} \\
\pi \models^i_k Gf & \quad \text{iff} \quad \text{false} \\
\pi \models^i_k Ff & \quad \text{iff} \quad \bigwedge_{j=i}^k \pi \models^j_k f
\end{align*}
\]
• definition:
  \[ \pi \models_k f \iff \pi \models_k^0 f \]

• bounded semantics approximates real semantics:
  \[ \pi_k \models f \Rightarrow \pi \models f \text{ for all } k \]

• (theoretical) completeness:
  if \( \pi \models f \) then there exists \( k \) with \( \pi_k \models f \)

• **note:** negate original property first (e.g. \( AGp \leftrightarrow EF\neg p \))
  
  – 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 $k$:

- $l[\cdot]_{k}^{i}$ assumes $(k, l)$-loop

- $[\cdot]_{k}^{i}$ assumes that no $(k, l)$-loop exists for all $l$

add time frame expansion of transition relation:

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

add $loop_k(l)$ constraint for looping translation: $loop_k(l) := T(s_k, s_l)$

add $noloop_k$ constraint for non-looping translation:

$$ noloop_k := \bigvee_{l=0}^{k} loop_k(l) $$
\[ i[p]^i_k := p(s_i) \]
\[ i[\neg p]^i_k := \neg p(s_i) \]
\[ i[f \land g]^i_k := l[f]^i_k \land l[g]^i_k \]
\[ i[X \ f]^i_k := l[f]^\text{next}(i) \]
\[ i[G \ f]^i_k := \bigwedge_{j=\min(l,i)}^k l[f]^j_k \]
\[ i[F \ f]^i_k := \bigvee_{j=\min(l,i)}^k l[f]^j_k \]

with

\[ \text{next}(i) := \begin{cases} 
  i + 1 & \text{if } i < k \\
  l & \text{else} 
\end{cases} \]
\[
[p]_k^i := p(s_i) \\
[\neg p]_k^i := \neg p(s_i) \\
[f \land g]_k^i := [f]_k^i \land [g]_k^i \\
[X f]_k^i := \begin{cases} [f]_{k+1}^i & \text{if } i < k \\ false & \text{else} \end{cases} \\
[G f]_k^i := false \\
[F f]_k^i := \bigvee_{j=i}^k [f]_k^j
\]
\[ [K, f]_k := \text{noloop}_k \land [f]_k^0 \lor \bigvee_{l=0}^{k} \text{loop}_k(l) \land l[f]_k^0 \]

- **Theorem:** \( K \models \mathbf{E}f \iff \exists k \ [K, f]_k \) satisfiable

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

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