Satisfiability Checking

Summary I

Prof. Dr. Erika Ábrahám

RWTH Aachen University
Informatik 2
LuFG Theory of Hybrid Systems

WS 19/20

Outline

1. Propositional logic, theories, normal forms

2. Propositional SAT solving

3. Eager SMT-solving
   - Equality logic with uninterpreted functions
     - From UF to EQ I: Ackermann's reduction
     - From UF to EQ II: Bryant's reduction
     - From EQ to SAT: The Sparse method
   - Finite-precision bit-vector arithmetic

Outline

1. Propositional logic, theories, normal forms
2. Propositional SAT solving
3. Eager SMT-solving
   - Equality logic with uninterpreted functions
     - From UF to EQ I: Ackermann's reduction
     - From UF to EQ II: Bryant's reduction
     - From EQ to SAT: The Sparse method
   - Finite-precision bit-vector arithmetic

Propositional logic: Syntax

- Abstract grammar:

$$\varphi \ := \ AP \ | \ (\neg\varphi) \ | \ (\varphi \wedge \varphi)$$

  with $AP \in AP$.

- Syntactic sugar:

$$\begin{array}{lcccl} & & \bot & & := (a \wedge \neg a) \\ & & \top & & := (a \vee \neg a) \\ (& \varphi_1 & \vee & \varphi_2 &) := \neg((\neg\varphi_1) \wedge (\neg\varphi_2)) \\ (& \varphi_1 & \rightarrow & \varphi_2 &) := ((\neg\varphi_1) \vee \varphi_2) \\ (& \varphi_1 & \leftrightarrow & \varphi_2 &) := ((\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)) \\ (& \varphi_1 & \bigoplus & \varphi_2 &) := (\varphi_1 \leftrightarrow (\neg\varphi_2)) \end{array}$$

Propositional logic: Semantics

- Structures for predicate logic:
  - Domain: $\mathbb{B} = \{0, 1\}$
  - Interpretation: assignment $\alpha : AP \to \{0, 1\}$
    Assign: set of all assignments
    Equivalently: $\alpha \in 2^{AP}$ or $\alpha \in \{0, 1\}^{AP}$
- Semantics: $\models \subseteq$ $(Assign \times \text{Formula})$ is defined recursively:

$\alpha \models p$ iff $\alpha(p) = \text{true}$
$\alpha \models \neg\varphi$ iff $\alpha \not\models \varphi$
$\alpha \models \varphi_1 \land \varphi_2$ iff $\alpha \models \varphi_1$ and $\alpha \models \varphi_2$

$\alpha \models \varphi_1 \lor \varphi_2$ iff $\alpha \models \varphi_1$ or $\alpha \models \varphi_2$
$\alpha \models \varphi_1 \to \varphi_2$ iff $\alpha \models \varphi_1$ implies $\alpha \models \varphi_2$
$\alpha \models \varphi_1 \leftrightarrow \varphi_2$ iff $\alpha \models \varphi_2$ iff $\alpha \models \varphi_2$
$\alpha \models \varphi_1 \bigoplus \varphi_2$ iff $\alpha \models \varphi_2$ iff $\alpha \not\models \varphi_2$

Logic extensions: Theories

Propositional logic	$(x \vee y) \wedge (\neg x \vee y)$
Equality	$(x = y \wedge y \neq z) \rightarrow (x \neq z)$
Uninterpreted functions	$(F(x) = F(y) \wedge y = z) \rightarrow F(x) = F(z)$
Linear real/integer arithmetic	$2x + y > 0 \wedge x + y \leq 0$ $2x = 1$
Real algebra	$x^2 + 2xy + y^2 < 0$

Normal forms

Input for solvers:

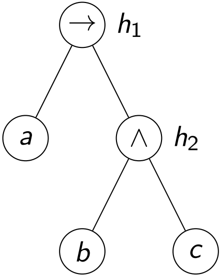
- Negation Normal Form (NNF)
- Conjunctive Normal Form (CNF)

Converting to CNF: Tseitin's encoding

- Consider the formula

  $\phi = (a \rightarrow (b \wedge c))$

The parse tree:



- Associate a new auxiliary variable with each gate.
- Add constraints that define these new variables.
- Finally, enforce the root node.

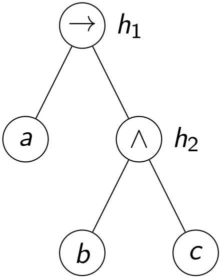
Converting to CNF: Tseitin's encoding

- Need to satisfy:

  $(h_1 \leftrightarrow (a \ \rightarrow \ h_2)) \wedge$

  $(h_2 \leftrightarrow (b \wedge c)) \wedge$

  (h_1)



- Each gate encoding has a CNF representation with 3 or 4 clauses.

Outline

The basic SAT algorithm

```
if (!BCP()) return UNSAT;
while (true)
{
        if (!decide()) return SAT;
        while (!BCP())
                if (!resolve_conflict()) return UNSAT;
}
```

The basic SAT algorithm

```
if (!BCP()) return UNSAT;
while (true)
{
    if (!decide()) return SAT;
    while (!BCP())
        if (!resolve_conflict()) return UNSAT;
}
```

Choose the next variable and value.
Return false if all variables are assigned.

The basic SAT algorithm

```
if (!BCP()) return UNSAT;
while (true)
{
        if (!decide()) return SAT;
        while (!BCP())
                if (!resolve_conflict()) return UNSAT;
}
```

Choose the next variable and value.
Return false if all variables are assigned.

Boolean constraint propagation.
Return false if reached a conflict.

The basic SAT algorithm

```
if (!BCP()) return UNSAT;
while (true)
{
      if (!decide()) return SAT;
      while (!BCP())
            if (!resolve_conflict()) return UNSAT;
}
```

Choose the next variable and value. Return false if all variables are assigned.

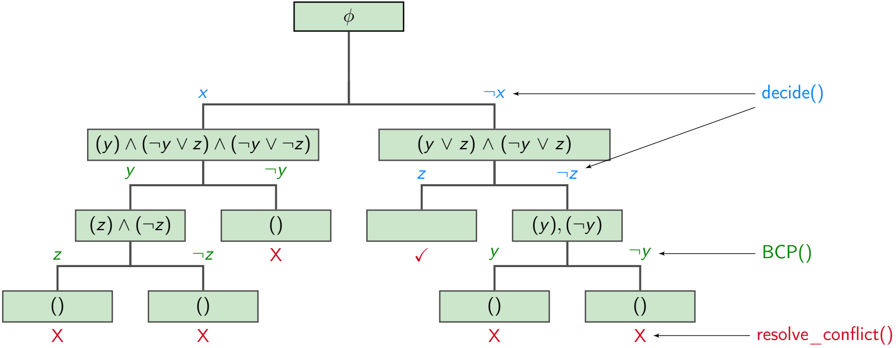
Boolean constraint propagation. Return false if reached a conflict.

Conflict resolution and backtracking. Return false if impossible.

A basic SAT algorithm

Assume the CNF formula

$\phi \;:\; (x \vee y \vee z) \wedge (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$



SAT solving: Components

- Decision
- Boolean constraint propagation (BCP)
- Conflict resolution
- Backtracking

Boolean constraint propagation

- A clause can be

Satisfied:	at least one literal is true
Unsatisfied:	all literals are false
	$\rightarrow$ conflict
Unit:	one literal is unassigned, the remaining literals are false
	$\rightarrow$ propagation
Unresolved:	all other cases

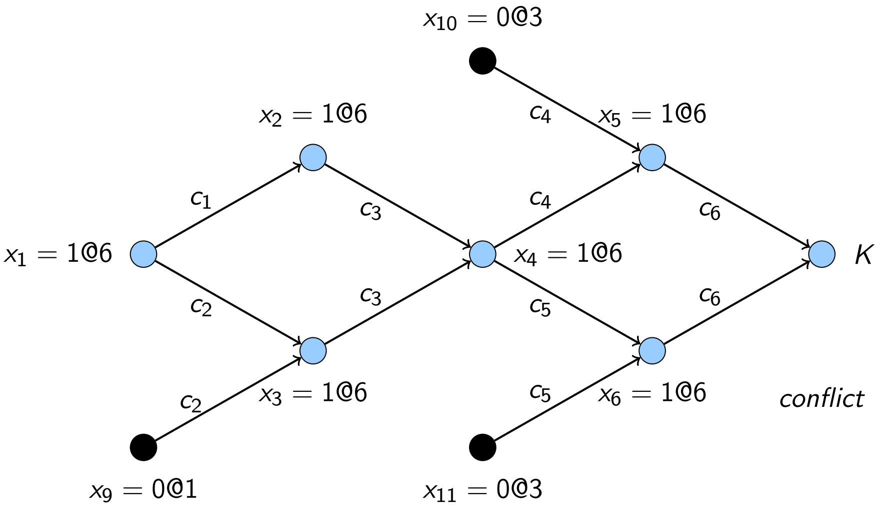
- Example: $C = (x_1 \vee x_2 \vee x_3)$

x_1	x_2	x_3	C
1	0		satisfied
0	0	0	unsatisfied
0	0		unit
	0		unresolved

Boolean constraint propagation

- Organize the search in the form decision trees
  - Each root corresponds to a decision
  - Definition: decision level (DL) gives an index to each decision and its implications in chronological order
  - Notation: x =v @ d
    proposition x is assigned to value $v \in \{0, 1\}$ at decision level d

Conflict resolution



Conflict resolution

The resolution inference rule for CNF:

$$\frac{(l \vee l_1 \vee l_2 \vee \ldots \vee l_n) \quad (\neg l \vee l'_1 \vee \ldots \vee l'_m)}{(l_1 \vee \ldots \vee l_n \vee l'_1 \vee \ldots \vee l'_m)} \text{ Resolution}$$

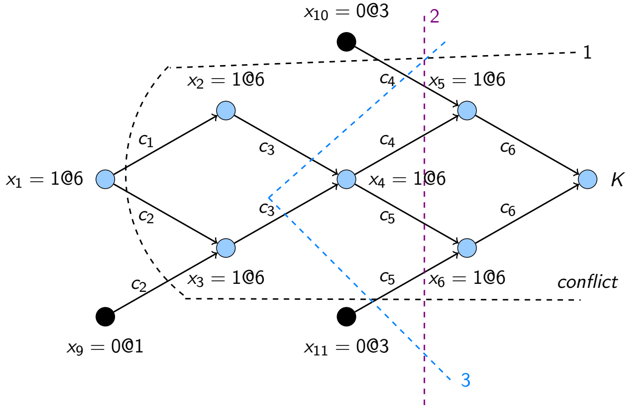
Example:

$$\frac{(a \vee b) \quad (\neg a \vee c)}{(b \vee c)}$$

- Resolution is a sound and complete inference system for CNF.
- The input formula is unsatisfiable iff there exists a proof of the empty clause.

Conflict resolution

Apply resolution up in the implication tree until a UIP (Unique Implication Point) has been reached:



1.$(x_{10} \vee \neg x_1 \vee x_9 \vee x_{11})$

2.$(x_{10} \vee \neg x_4 \vee x_{11})$

3.$(x_{10} \vee \neg x_2 \vee \neg x_3 \vee x_{11})$

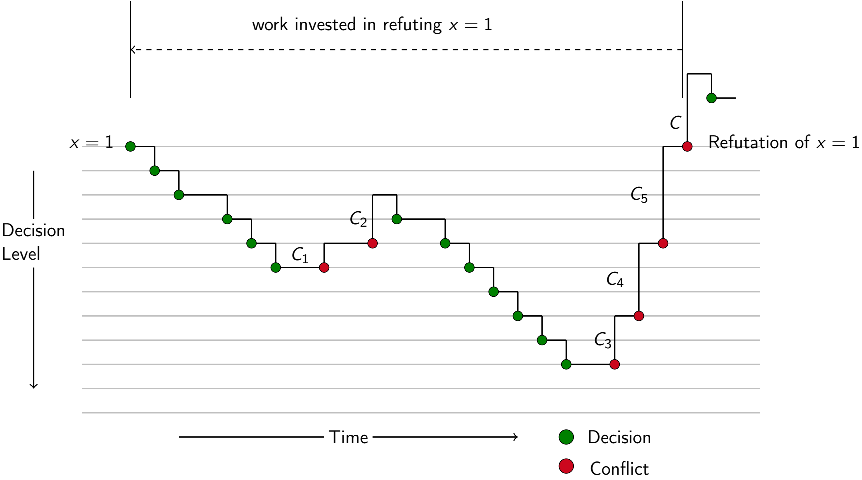
⋮

⋮

Non-chronological backtracking

- Backtrack to the second largest decision level in the conflict clause.
- This resolves the conflict and triggers an implication by the new conflict clause.

Progress of a SAT solver



Decision heuristics - VSIDS

VSIDS(Variable State Independent Decaying Sum)

1. Each variable (in each polarity) has an activity initialized to 0.
2. When resolution gets applied to a clause, the activities of its literals are increased.
3. Decision: The unassigned variable with the highest activity is chosen.
4. Periodically, all the activities are divided by a constant.

Outline

1. Propositional logic, theories, normal forms
2. Propositional SAT solving
3. Eager SMT-solving
   - Equality logic with uninterpreted functions
     - From UF to EQ I: Ackermann's reduction
     - From UF to EQ II: Bryant's reduction
     - From EQ to SAT: The Sparse method
   - Finite-precision bit-vector arithmetic

Outline

1 Propositional logic, theories, normal forms

2 Propositional SAT solving

3 Eager SMT-solving
- Equality logic with uninterpreted functions
  - From UF to EQ I: Ackermann's reduction
  - From UF to EQ II: Bryant's reduction
  - From EQ to SAT: The Sparse method
- Finite-precision bit-vector arithmetic

Equality logic with uninterpreted functions

We extend propositional logic with

- equalities and
- uninterpreted functions (UFs).

Syntax:

- variables x over an arbitrary domain D,
- constants c from the same domain D,
- function symbols F for functions of the type $D^n \to D$, and
- equality as predicate symbol.

$$\begin{array}{lllllll} \textit{Terms:} & t & := & c & \mid & x & \mid & F(t,\ldots,t) \\ \textit{Formulas:} & \varphi & := & t = t & \mid & (\varphi \land \varphi) & \mid & (\neg\varphi) \end{array}$$

Semantics: straightforward

From uninterpreted functions to equality logic

We lead back the problems of equality logic with uninterpreted functions to those of equality logic without uninterpreted functions.

Basic idea: Encode functional congruence

Two possible reductions:

- Ackermann's reduction
- Bryant's reduction

Ackermann's reduction

- Input: φ^{UF} with m instances of an uninterpreted function F.
- Output: satisfiability-equivalent φ^{E} without any occurrences of F.

Algorithm

Ackermann's reduction

- Input: φ^{UF} with m instances of an uninterpreted function F.
- Output: satisfiability-equivalent φ^{E} without any occurrences of F.

Algorithm

1. Assign indices to the F-instances.
2. $\varphi_{flat} := \mathcal{T}(\varphi^{UF})$ where $\mathcal{T}$ replaces each occurrence F_i of F by a fresh variable f_i.
3. $\varphi_{cong} := \bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} (\mathcal{T}(arg(F_i)) = \mathcal{T}(arg(F_j))) \rightarrow f_i = f_j$
4. Return $\varphi_{flat} \wedge \varphi_{cong}$.

Bryant's reduction

- Input: φ^{UF} with m instances of an uninterpreted function F.
- Output: satisfiability-equivalent φ^{E} without any occurrences of F.

Algorithm

Bryant's reduction

- Input: φ^{UF} with m instances of an uninterpreted function F.
- Output: satisfiability-equivalent φ^{E} without any occurrences of F.

Algorithm

1. Assign indices to the F-instances.
2. Return $\mathcal{T}^*(\varphi^{UF})$ where $\mathcal{T}^*$ replaces each $F_i(arg(F_i))$ by

$$
\begin{array}{lll}
case & \mathcal{T}^*(arg(F_1)) = \mathcal{T}^*(arg(F_i)) & : \quad f_1 \\
 & \ldots & \\
 & \mathcal{T}^*(arg(F_{i-1})) = \mathcal{T}^*(arg(F_i)) & : \quad f_{i-1} \\
 & true & : \quad f_i
\end{array}
$$

Equality logic to propositional logic

- Input: Equality logic formula φ^E
- Output: Satisfiability-equivalent propositional logic formula φ^E

Algorithm

Equality logic to propositional logic

- Input: Equality logic formula φ^E
- Output: Satisfiability-equivalent propositional logic formula φ^E

Algorithm

1. Construct φ_{sk} by replacing each equality $t_i = t_j$ in φ^E by a fresh Boolean variable $e_{i,j}$.
2. Construct the E-graph $G^E(\varphi^E)$ for φ^E.
3. Make $G^E(\varphi^E)$ chordal.
4. $\varphi_{trans} = true$.
5. For each triangle $(e_{i,j}, e_{j,k}, e_{k,i})$ in $G^E(\varphi^E)$:

$$\begin{aligned}\varphi_{trans} := \varphi_{trans} \quad & \wedge (e_{i,j} \wedge e_{j,k}) \rightarrow e_{k,i} \\ & \wedge (e_{i,j} \wedge e_{i,k}) \rightarrow e_{j,k} \\ & \wedge (e_{i,k} \wedge e_{j,k}) \rightarrow e_{i,j}\end{aligned}$$

6. Return $\varphi_{sk} \wedge \varphi_{trans}$.

Outline

Finite-precision bit-vector arithmetic

"Bit blasting":

- Model bit-level operations (functions and predicates) by Boolean circuits
- Use Tseitin's encoding to generate propositional SAT encoding in CNF
- Use a SAT solver to check satisfiability
- Convert back the propositional solution to the theory

Effective solution for many applications.

- Example: Bounded model checking for C programs (CBMC) [Clarke, Kroening, Lerda, TACAS'04]