Satisfiability Checking

Eager SMT Solving (Equality Logic, Bit-blasting)

Prof. Dr. Erika Ábrahám

RWTH Aachen University
Informatik 2
LuFG Theory of Hybrid Systems

WS 14/15

Outline

Outline

SMT solving

- We want to extend propositional logic with theories.
- For satisfiability checking, SAT-solving will be extended to SAT-modulo-theories (SMT) solving.
- SMT-LIB: language, benchmarks, tutorials, ...
- SMT-COMP: performance and capabilities of tools
- SMT Workshop: held annually

Eager SMT solving

- How can such an extension to SMT solving look like?
- We will see two basically different approaches:
  - Eager SMT solving transforms logical formulas over some theories into satisfiability-equivalent propositional logic formulas and applies SAT solving. ("Eager" means theory first)
  - Lazy SMT solving uses a SAT solver to find solutions for the Boolean skeleton of the formula, and a theory solver to check satisfiability in the underlying theory. ("Lazy" means theory later)
- Today we will have a closer look at the eager approach.

Eager vs. Lazy SMT Solving

Theories for Eager SMT Solving

- All NP-complete problems can be transformed to equivalent propositional SAT problems (with polynomial effort).
- However, this is not always effective in praxis (the transformation would sometimes solve the hardest part of the problem).
- Some well-suited theories for eager SMT solving:
  - Equalities and uninterpreted functions
  - Finite-precision bit-vector arithmetic
  - Quantifier-free linear integer arithemtic (QF_LIA)
  - Restricted λ-calculus (e.g., arrays)
  - ...
  - Combinations of the above theories

Some Eager SMT Solver Implementations

- UCLID: Proof-based abstraction-refinement [Bryant et al., TACAS'07]
- STP: Solver for linear modular arithmetic to simplify the formula [Ganesh&Dill, CAV'07]
- Spear: Automatic parameter tuning for SAT [Hutter et al., FMCAD'07]
- Boolector: Rewrites, underapproximation, efficient SAT engine [Brummayer&Biere, TACAS'09]
- Beaver: Equality/constant propagation, logic optimization, special rules for non-linear operations [Jha et al., CAV'09]
- SONOLAR: Non-linear arithmetic [Brummayer et al., SMT'08]
- SWORD: Fixed-size bit-vectors [Jung et al, SMTCOMP'09]
- Layered eager approaches embedded in the lazy DPLL(T) framework: CVC3 [Barrett et al.], MathSAT [Bruttomesso et al.], Z3 [de Moura et al.]

Outline

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.

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

Semantics: straightforward

Motivation

- Equality logic and propositional logic are both NP-complete.
- Thus they model the same decision problems.
- Why to study both?
  - Convenience of modeling
  - Efficiency

Equality logic with uninterpreted functions

Notation and assumptions:

- Formula with equalities: φ^{E}
- Formula with equalities and uninterpreted functions: φ^{UF}
- Same simplifications for parentheses as for propositional logic.
- Input formulas are in NNF.
- Input formulas are checked for satisfiability.

Removing constants

Theorem

There is an algorithm that generates for an input formula φ^{UF} an equisatisfiable output formula $\varphi^{UF'}$ without constants, in polynomial time.

Algorithm: Exercise

In the following we assume that the formulas do not contain constants.

Outline

Motivation

- Replacing functions by uninterpreted functions in a given formula is a common technique to make reasoning easier.
- It makes the formula weaker: $\models \varphi^{UF} \rightarrow \varphi$
- Ignore the semantics of the function, but:
- Functional congruence: Instances of the same function return the same value for equal arguments.

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.

Two possible reductions:

- Ackermann's reduction
- Bryant's reduction

Outline

Ackermann's reduction

Given an input formula φ^{UF} of equality logic with uninterpreted functions, transform the formula to a satisfiability-equivalent equality logic formula φ^{E} of the form

$$\varphi^{E} := \varphi_{flat} \land \varphi_{cong},$$

where φ_{flat} is a flattening of φ^{UF}, and φ_{cong} is a conjunction of constraints for functional congruence.

For validity-equivalence check

$$\varphi^{E} := \varphi_{cong} \rightarrow \varphi_{flat}.$$

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 Boolean 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 $\varphi_{flat} \wedge \varphi_{cong}$.

Ackermann's reduction: Example

$$
\begin{array}{lcl}
\varphi^{UF} & := & (x_1 \neq x_2) \vee (F(x_1) = F(x_2)) \vee (F(x_1) \neq F(x_3)) \\
\varphi_{flat} & := & (x_1 \neq x_2) \vee (f_1 = f_2) \vee (f_1 \neq f_3) \\
\varphi_{cong} & := & ((x_1 = x_2) \quad \rightarrow \quad (f_1 = f_2)) \wedge \\
 & & ((x_1 = x_3) \quad \rightarrow \quad (f_1 = f_3)) \wedge \\
 & & ((x_2 = x_3) \quad \rightarrow \quad (f_2 = f_3)) \\
\varphi^{E} & := & \varphi_{flat} \wedge \varphi_{cong}
\end{array}
$$

Ackermann's reduction: Example

- ```
  int power3 (int in){
    int out = in;
    for (int i=0; i<2; i++)
      out = out * in;
    return out;
  }
  ```
- ```
  int power3_b (int in){
    return ((in * in) * in);
  }
  ```

- $\varphi_1 \;:=\; out_0 = in \wedge out_1 = out_0 * in \wedge out_2 = out_1 * in$
- $\varphi_2 \;:=\; out_b = (in * in) * in$
- $\varphi_3 \;:=\; (\varphi_1 \wedge \varphi_2) \rightarrow (out_2 = out_b)$

Ackermann's reduction: Example

$$\begin{aligned}\varphi_3 \quad := \quad & (out_0 = in \wedge out_1 = out_0 * in \wedge \\ & out_2 = out_1 * in \wedge out_b = (in * in) * in) \rightarrow \\ & (out_2 = out_b)\end{aligned}$$

$$\begin{aligned}\varphi^{UF} \quad := \quad & (out_0 = in \wedge out_1 = G(out_0, in) \wedge \\ & out_2 = G(out_1, in) \wedge out_b = G(G(in, in), in)) \rightarrow \\ & (out_2 = out_b)\end{aligned}$$

Ackermann's reduction: Example

$$\begin{aligned}
\varphi^{UF} \quad &:= \quad (out_0 = in \wedge out_1 = G(out_0, in) \wedge out_2 = G(out_1, in) \wedge \\
&\qquad out_b = G(G(in, in), in)) \rightarrow (out_2 = out_b) \\
\varphi_{flat} \quad &:= \quad (out_0 = in \wedge out_1 = G_1 \wedge out_2 = G_2 \wedge \\
&\qquad out_b = G_4) \rightarrow (out_2 = out_b) \text{ with}
\end{aligned}$$

$$\begin{array}{lllll}
\varphi_{cong} := & ((out_0 = out_1 \wedge in = in) & \rightarrow & G_1 = G_2)\wedge \\
& ((out_0 = in \wedge in = in) & \rightarrow & G_1 = G_3)\wedge \\
& ((out_0 = G_3 \wedge in = in) & \rightarrow & G_1 = G_4)\wedge \\
& ((out_1 = in \wedge in = in & \rightarrow & G_2 = G_3)\wedge \\
& ((out_1 = G_3 \wedge in = in) & \rightarrow & G_2 = G_4)\wedge \\
& ((in = G_3 \wedge in = in) & \rightarrow & G_3 = G_4)
\end{array}$$

Outline

1 Eager SMT Solving

2 Eager SMT Solving for Equality Logic with Uninterpreted Functions
- Equality Logic with Uninterpreted Functions
- Eager SMT Solving for Uninterpreted Functions
  - Ackermann's reduction
  - Bryant's reduction
- Eager SMT Solving for Equality Logic
  - Equality Graphs
  - The Sparse Method

3 Eager SMT Solving for Finite-precision Bit-vector Arithmetic

Bryant's reduction

- Case expression:

$$F_i^* = \begin{array}{lll} case & x_1 = x_i & : & f_1 \\ & x_2 = x_i & : & f_2 \\ & \ldots & & \\ & x_{i-1} = x_i & : & f_{i-1} \\ & true & : & f_i \end{array}$$

  where x_i is the argument $arg(F_i)$ of F_i for all i.

- Semantics:

$$\bigvee_{j=1}^{i} \left(\left(\bigwedge_{k=1}^{j-1} (x_k \neq x_i) \right) \wedge (x_j = x_i) \wedge (F_i^* = f_j) \right)$$

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

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

Bryant's reduction: Example

- int power3 (int in){
```
int out = in;
for (int i=0; i<2; i++)
   out = out * in;
return out;
```
}
- int power3_b (int in){
```
return ((in * in) * in);
```
}

- $\varphi_1 \;:=\; out_0 = in \wedge out_1 = out_0 * in \wedge out_2 = out_1 * in$
- $\varphi_2 \;:=\; out_b = (in * in) * in$
- $\varphi_3 \;:=\; (\varphi_1 \wedge \varphi_2) \rightarrow (out_2 = out_b)$

Bryant's reduction: Example

$$\begin{aligned}\varphi_3 \quad := \quad & (out_0 = in \land out_1 = out_0 * in \land \\ & out_2 = out_1 * in \land out_b = (in * in) * in) \rightarrow \\ & (out_2 = out_b)\end{aligned}$$

$$\begin{aligned}\varphi^{UF} \quad := \quad & (out_0 = in \land out_1 = G(out_0, in) \land \\ & out_2 = G(out_1, in) \land out_b = G(G(in, in), in)) \rightarrow \\ & (out_2 = out_b)\end{aligned}$$

Bryant's reduction: Example

$$
\begin{aligned}
\varphi^{UF} \; &:= \; (out_0 = in \wedge out_1 = G(out_0, in) \wedge out_2 = G(out_1, in) \wedge \\
&\qquad out_b = G(G(in, in), in)) \to (out_2 = out_b) \\
\varphi^{E} \; &:= \; (out_0 = in \wedge out_1 = G_1^* \wedge out_2 = G_2^* \wedge \\
&\qquad out_b = G_4^*) \to (out_2 = out_b) \text{ with}
\end{aligned}
$$

$$
\begin{array}{lllll}
G_1^* & = & & & g_1 \\
G_2^* & = & case & out_0 = out_1 \wedge in = in & : \; g_1 \\
 & & & true & : \; g_2 \\
G_3^* & = & case & out_0 = in \wedge in = in & : \; g_1 \\
 & & & out_1 = in \wedge in = in & : \; g_2 \\
 & & & true & : \; g_3 \\
G_4^* & = & case & out_0 = G_3^* \wedge in = in & : \; g_1 \\
 & & & out_1 = G_3^* \wedge in = in & : \; g_2 \\
 & & & in = G_3^* \wedge in = in & : \; g_3 \\
 & & & true & : \; g_4
\end{array}
$$

Outline

1. Eager SMT Solving

2. Eager SMT Solving for Equality Logic with Uninterpreted Functions
   - Equality Logic with Uninterpreted Functions
   - Eager SMT Solving for Uninterpreted Functions
     - Ackermann's reduction
     - Bryant's reduction
   - Eager SMT Solving for Equality Logic
     - Equality Graphs
     - The Sparse Method

3. Eager SMT Solving for Finite-precision Bit-vector Arithmetic

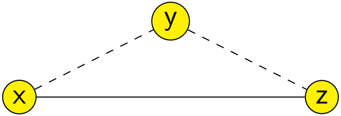
E-graphs

$$\varphi^E : x = y \land y = z \land z \neq x$$

- The equality predicates: $\{x = y, y = z, z \neq x\}$
- Break into two sets:

$$E_= = \{x = y, y = z\}, \quad E_{\neq} = \{z \neq x\}$$

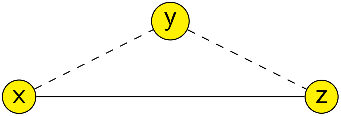
- The equality graph (E-graph) $G^E(\varphi^E) = \langle V, E_=, E_{\neq} \rangle$



The E-graph and Boolean structure in φ^E

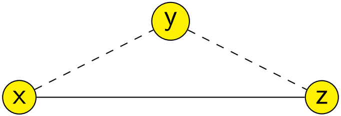
$$\begin{array}{lll} \varphi_1^E: & x = y \wedge y = z \wedge z \neq x & \text{unsatisfiable} \\ \varphi_2^E: & (x = y \wedge y = z) \vee z \neq x & \text{satisfiable!} \end{array}$$

Their E-graph is the same:



$\Longrightarrow$ The graph $G^E(\varphi^E)$ represents an abstraction of φ^E.
It ignores the Boolean structure of φ^E.

Equality and disequality paths



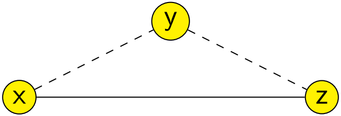
Definition (Equality Path)

A path that uses $E_=$ edges is an *equality path*. We write $x =^* z$.

Definition (Disequality Path)

A path that uses edges from $E_=$ and exactly one edge from $E_{\neq}$ is a *disequality path*. We write $x \neq^* z$.

Contradictory cycles



Definition (Contradictory Cycle)

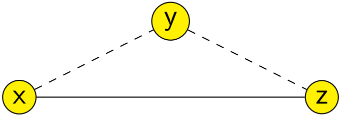
A cycle with one disequality edge is a *contradictory cycle*.

Theorem

For every two nodes x, y on a contradictory cycle the following holds:

- $x =^* y$
- $x \neq^* y$

Contradictory cycles



Definition

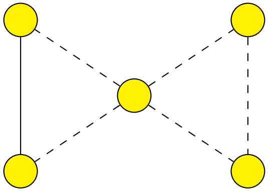
A subgraph of E is called *satisfiable* iff the conjunction of the predicates represented by its edges is satisfiable.

Theorem

A subgraph is unsatisfiable iff it contains a contradictory cycle.

Simple cycles

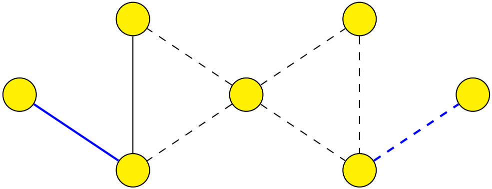
Question: What is a simple cycle?



Theorem

Every contradictory cycle is either simple, or contains a simple contradictory cycle.

Simplifying the E-graph of φ^E



Let S be the set of edges that are not part of any contradictory cycle.

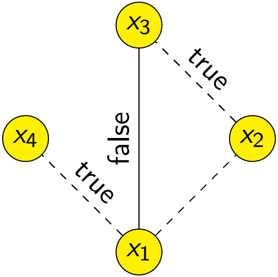
Theorem

Replacing

- *all equations in* φ^E *that correspond to solid edges in* S *with false, and*
- *all equations in* φ^E *that correspond to dashed edges in* S *with true*

preserves satisfiability.

Simplifying the E-graph: Example



- $(x_1 = x_2 \quad \vee \quad x_1 = x_4) \quad \wedge$
  $(x_1 \neq x_3 \quad \vee \quad x_2 = x_3)$
- $(x_1 = x_2 \quad \vee \quad \text{true} \quad) \quad \wedge$
  $(x_1 \neq x_3 \quad \vee \quad x_2 = x_3)$
- $(x_1 \neq x_3 \quad \vee \quad x_2 = x_3)$
- $\neg$false $\vee$ true
- $\rightarrow$ Satisfiable!

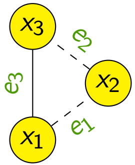
Outline

Bryant & Velev 2000: The *Sparse* method

Goal: Transform equality logic to propositional logic

Step 1: Replace all equalities in the formula by Boolean variables

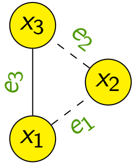
$$\begin{array}{lcl} \varphi^E & \leftrightarrow & x_1 = x_2 \wedge x_2 = x_3 \wedge x_1 \neq x_3 \\ \varphi_{sk} & \leftrightarrow & \quad e_1 \quad \wedge \quad e_2 \quad \wedge \quad \neg e_3 \end{array}$$



- This is called the propositional skeleton
- This is an over-approximation
- Transitivity of equality is lost!
- $\rightarrow$ must add transitivity constraints!

Adding transitivity constraints

$$\begin{array}{lcl} \varphi^E & \leftrightarrow & x_1 = x_2 \wedge x_2 = x_3 \wedge x_1 \neq x_3 \\ \varphi_{sk} & \leftrightarrow & e_1 \quad \wedge \quad e_2 \quad \wedge \quad \neg e_3 \end{array}$$



Step 2: For each cycle in the equality graph: add a transitivity constraint

$$\begin{array}{rl} \varphi_{trans} = & (e_1 \wedge e_2 \rightarrow e_3) \wedge \\ & (e_1 \wedge e_3 \rightarrow e_2) \wedge \\ & (e_3 \wedge e_2 \rightarrow e_1) \end{array}$$

Step 3: Check $\varphi_{sk} \wedge \varphi_{trans}$

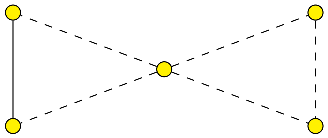
Question: Complexity?

Optimizations

There can be an *exponential number of cycles*, so let's try to improve this idea.

Theorem

It is sufficient to constrain simple cycles only.



Only two simple cycles here.

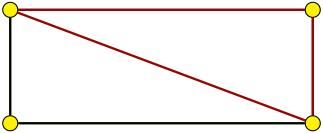
Question: Complexity?

Optimizations

Still, there may be an exponential number of simple cycles.

Theorem

It is sufficient to constrain chord-free simple cycles.



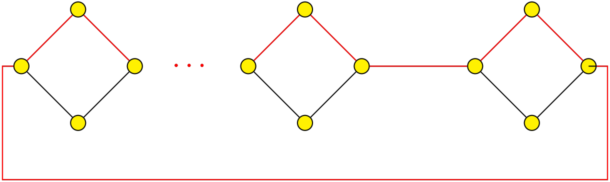
Question: How many simple cycles?

Question: How many chord-free simple cycles?

Question: Complexity?

Optimizations

Still, there may be an exponential number of chord-free simple cycles...



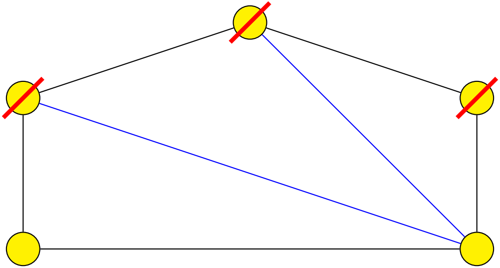
Solution: make graph 'chordal' by adding edges!

Making the E-graph chordal

Definition (Chordal graph)

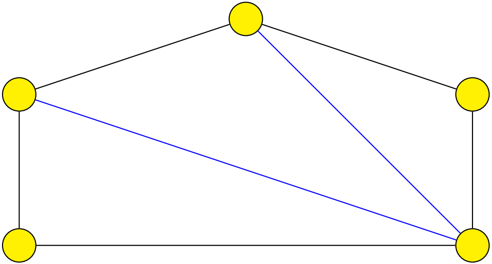
A graph is *chordal* iff every cycle of length 4 or more has a chord.

Question: How to make a graph chordal?
A: Iteratively connect the neighbors of the vertices.



Making the E-graph chordal

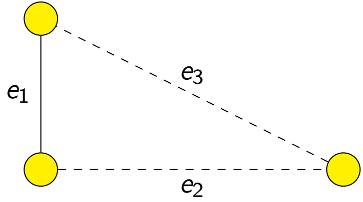
- Once the graph is chordal, we only need to constrain the triangles.



- Note that this procedure adds not more than a polynomial number of edges, and results in a polynomial number of constraints.

Exploiting the polarity

- So far we did not consider the polarity of the edges.
- Claim: in the following graph, $\varphi_{trans} = e_2 \wedge e_3 \rightarrow e_1$ is sufficient.



- This works because of the monotonicity of NNF.

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
- 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]

Slides...

...from the Decision Procedures website.