

Satisfiability Checking

Eager SMT Solving (Equality Logic, Bit-blasting)

Prof. Dr. Erika Ábrahám

RWTH Aachen University
Informatik 2
LuFG Theory of Hybrid Systems

WS 16/17

- 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

- 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

- 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

- 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

- All NP-complete problems can be transformed to equivalent propositional SAT problems (with polynomial effort).
- However, in praxis, this is not always the most effective solution.
- Some well-suited theories for eager SMT solving:
 - Equalities and uninterpreted functions
 - Finite-precision bit-vector arithmetic
 - Quantifier-free linear integer arithmetic (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 optimisation, 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.]

- 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

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 $D^n \rightarrow D$, and
- **equality** as predicate symbol.

<i>Terms:</i>	t	$::=$	c		x		$F(t, \dots, t)$
<i>Formulas:</i>	φ	$::=$	$t = t$		$(\varphi \wedge \varphi)$		$(\neg \varphi)$

Semantics: straightforward

- 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**.

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.

- 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

- 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

- 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

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

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

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 theory variable f_i .
- 3 $\varphi_{cong} := \bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^m (\mathcal{T}(\text{arg}(F_i)) = \mathcal{T}(\text{arg}(F_j))) \rightarrow f_i = f_j$
- 4 Return $\varphi_{flat} \wedge \varphi_{cong}$.

Ackermann's reduction: Example

$$\varphi^{UF} := (x_1 \neq x_2) \vee (F(x_1) = F(x_2)) \vee (F(x_1) \neq F(x_3))$$

$$\varphi_{flat} := (x_1 \neq x_2) \vee (f_1 = f_2) \vee (f_1 \neq f_3)$$

$$\begin{aligned} \varphi_{cong} := & ((x_1 = x_2) \rightarrow (f_1 = f_2)) \wedge \\ & ((x_1 = x_3) \rightarrow (f_1 = f_3)) \wedge \\ & ((x_2 = x_3) \rightarrow (f_2 = f_3)) \end{aligned}$$

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

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

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

$$\varphi_{flat} := (out_0 = in \wedge out_1 = G_1 \wedge out_2 = G_2 \wedge out_b = G_4) \rightarrow (out_2 = out_b) \text{ with}$$

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

- 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

- Case expression:

$$\begin{aligned} F_i^* = \text{case} \quad & x_1 = x_i & : & f_1 \\ & x_2 = x_i & : & f_2 \\ & \dots & & \\ & x_{i-1} = x_i & : & f_{i-1} \\ & \text{true} & : & f_i \end{aligned}$$

where x_i is the argument $\text{arg}(F_i)$ 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 $F_i(\text{arg}(F_i))$ by

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

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

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

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

Bryant's reduction: Example

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

$$\varphi^E := (out_0 = in \wedge out_1 = G_1^* \wedge out_2 = G_2^* \wedge out_b = G_4^*) \rightarrow (out_2 = out_b) \text{ with}$$

$$G_1^* = g_1$$

$$G_2^* = \text{case } out_0 = out_1 \wedge in = in : g_1 \\ \text{true} : g_2$$

$$G_3^* = \text{case } out_0 = in \wedge in = in : g_1 \\ out_1 = in \wedge in = in : g_2 \\ \text{true} : g_3$$

$$G_4^* = \text{case } out_0 = G_3^* \wedge in = in : g_1 \\ out_1 = G_3^* \wedge in = in : g_2 \\ in = G_3^* \wedge in = in : g_3 \\ \text{true} : g_4$$

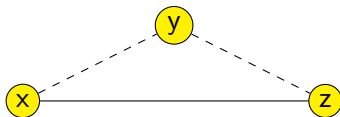
- 1 Eager SMT Solving
- 2 Eager SMT Solving for Equality Logic with Uninterpreted Functions
 - Equality Logic with Uninterpreted Functions
 - Eager SMT Solving for Uninterpreted Functions
 - Ackermann's reduction
 - Bryant's reduction
 - Eager SMT Solving for Equality Logic
 - Equality Graphs
 - The Sparse Method
- 3 Eager SMT Solving for Finite-precision Bit-vector Arithmetic

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

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

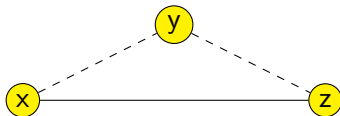


The E-graph and Boolean structure in φ^E

$$\varphi_1^E : x = y \wedge y = z \wedge z \neq x \quad \text{unsatisfiable}$$

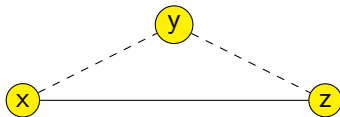
$$\varphi_2^E : (x = y \wedge y = z) \vee z \neq x \quad \text{satisfiable!}$$

Their E-graph is the same:



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

Equality and disequality paths



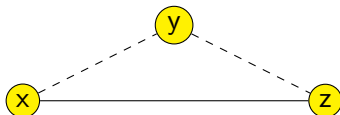
Definition (Equality Path)

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)

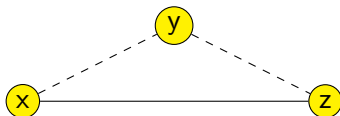
A cycle with one disequality edge is a *contradictory cycle*.

Theorem

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

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

Contradictory cycles



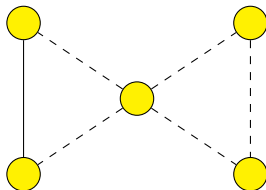
Definition

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.

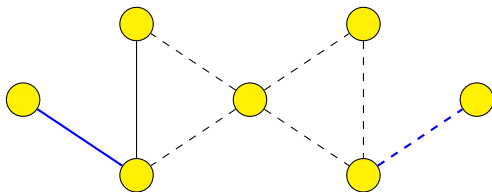
Question: What is a simple cycle?



Theorem

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

Simplifying the E-graph of φ^E



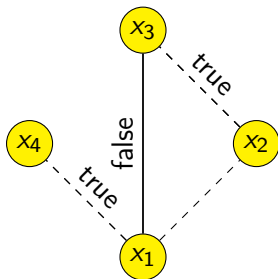
Let S be the set of edges that are **not part of any contradictory cycle**.

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



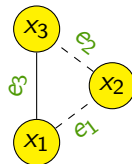
- $(x_1 = x_2 \vee x_1 = x_4) \wedge$
- $(x_1 \neq x_3 \vee x_2 = x_3)$
- $(x_1 = x_2 \vee \text{true}) \wedge$
- $(x_1 \neq x_3 \vee x_2 = x_3)$
- $(x_1 \neq x_3 \vee x_2 = x_3)$
- $\neg \text{false} \vee \text{true}$
- $\rightarrow \text{Satisfiable!}$

- 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

Goal: Transform equality logic to propositional logic

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

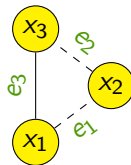
$$\begin{aligned}\varphi^E &\leftrightarrow x_1 = x_2 \wedge x_2 = x_3 \wedge x_1 \neq x_3 \\ \varphi_{sk} &\leftrightarrow e_1 \wedge e_2 \wedge \neg e_3\end{aligned}$$



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

Adding transitivity constraints

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



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

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

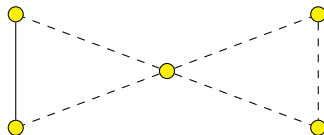
Step 3: Check $\varphi_{sk} \wedge \varphi_{trans}$

Question: Complexity?

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.*



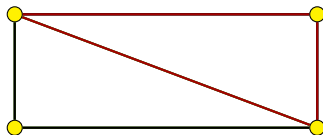
Only two simple cycles here.

Question: Complexity?

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

Theorem

*It is sufficient to constrain **chord-free simple cycles**.*

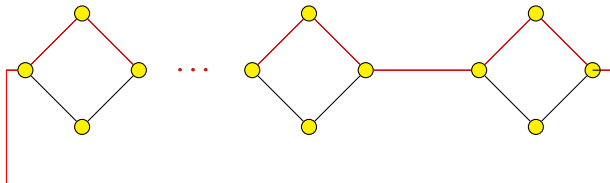


Question: How many simple cycles?

Question: How many chord-free simple cycles?

Question: Complexity?

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



Solution: make graph 'chordal' by adding edges!

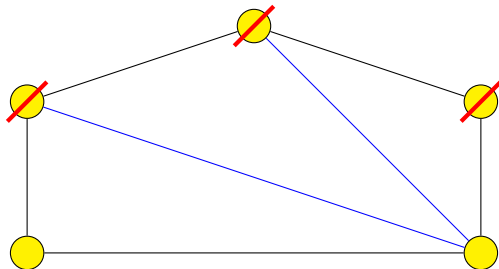
Making the E-graph chordal

Definition (Chordal graph)

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

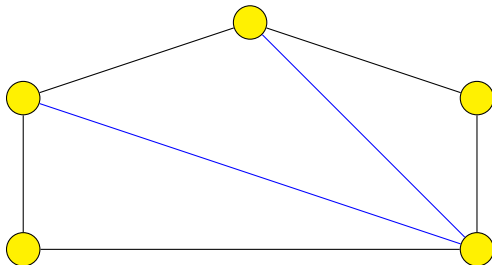
Question: How to **make a graph chordal**?

A: Iteratively connect the neighbors of the vertices.



Making the E-graph chordal

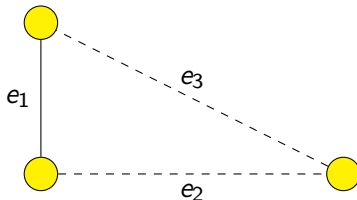
- Once the graph is chordal, we only need to **constrain the triangles**.



- 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 $\varphi_{trans} = \text{true}$.
- 5 For each triangle $(e_{i,j}, e_{j,k}, e_{k,i})$ in $G^E(\varphi^E)$:
$$\begin{aligned}\varphi_{trans} &:= \varphi_{trans} \quad \wedge (e_{i,j} \wedge e_{j,k}) \rightarrow e_{k,i} \\ &\quad \wedge (e_{i,j} \wedge e_{i,k}) \rightarrow e_{j,k} \\ &\quad \wedge (e_{i,k} \wedge e_{j,k}) \rightarrow e_{i,j}\end{aligned}$$
- 6 Return $\varphi_{sk} \wedge \varphi_{trans}$.

- 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

“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]

...from the Decision Procedures website.