

# 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.

- We want to extend propositional logic with theories.
- For satisfiability checking, SAT-solving will be extended to SAT-modulo-theories (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

- How can such an extension to SMT solving look like?

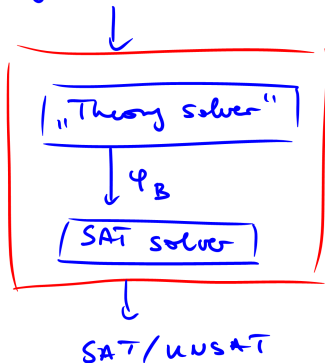
- How can such an extension to SMT solving look like?
- We will see two basically different approaches:
  - **Eager SMT solving transforms** logical formulas over some theories into satisfiability-equivalent propositional logic formulas and applies **SAT solving**. (“Eager” means theory first)
  - **Lazy 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)



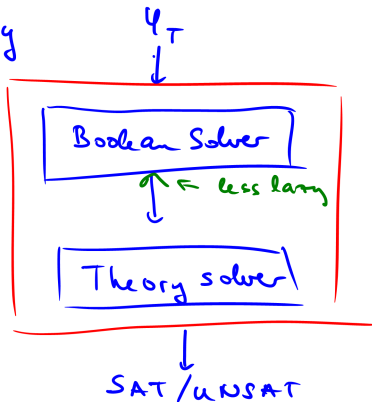
- How can such an extension to SMT solving look like?
- We will see two basically different approaches:
  - **Eager SMT solving transforms** logical formulas over some theories into satisfiability-equivalent propositional logic formulas and applies **SAT solving**. (“Eager” means theory first)
  - **Lazy 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

Eager:  $\varphi_T$



Lazy



# Theories for eager 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.

- 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  $\lambda$ -calculus (e.g., arrays)
  - ...

- 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  $\lambda$ -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).

# 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.  $= : D^2 \rightarrow \mathbb{B}$

$$\text{theory} \Rightarrow e ::= c \mid x \mid f(\underbrace{e_1, \dots, e_n}_n)$$

$$\text{boolean} \Rightarrow \varphi ::= t \mid f \mid \varphi \wedge \varphi \mid \neg \varphi \mid e = e$$

$$\begin{array}{l} f : D \rightarrow D \\ x : \text{var} \end{array}$$

$$\begin{array}{ccc} x & f(x) & f(\underbrace{f(x)}_e) \\ \downarrow & \downarrow & \downarrow \\ e & e & e \end{array}$$

# 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>	$\varphi$	$::=$	$t = t$		$(\varphi \wedge \varphi)$		$(\neg \varphi)$

# 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>	$\varphi$	$::=$	$t = t$		$(\varphi \wedge \varphi)$		$(\neg \varphi)$

Semantics: straightforward



- Equality logic and propositional logic are both **NP-complete**.
- Thus they model the same decision problems.

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

- 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:

• Formulas from propositional logic  $\varphi^B$

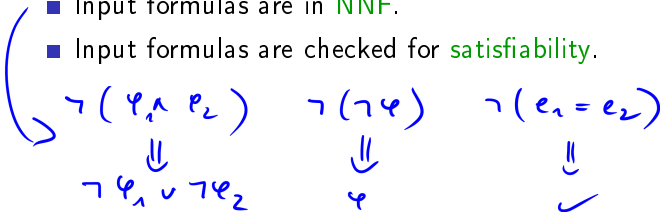
■ Formula with **equalities**:  $\varphi^E$

■ Formula with **equalities and uninterpreted functions**:  $\varphi^{UF}$

■ Same simplifications for **parentheses** as for propositional logic.

■ Input formulas are in **NNF**.

■ Input formulas are checked for **satisfiability**.



Handwritten logical equivalences for NNF conversion:

$$\neg(\varphi_1 \wedge \varphi_2) \quad \neg(\neg\varphi) \quad \neg(e_1 = e_2)$$
$$\Downarrow \qquad \qquad \Downarrow \qquad \qquad \Downarrow$$
$$\neg\varphi_1 \vee \neg\varphi_2 \qquad \varphi \qquad \checkmark$$



## Theorem

*There is an algorithm that generates for an input formula  $\varphi^{UF}$  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

$\varphi^{UF}$   
 $\downarrow$   
 $\varphi^E$   
 $\downarrow$   
 $\varphi^B$

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

# Motivation



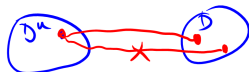
- Replacing functions by **uninterpreted functions** in a given formula is a common technique to make reasoning easier.
- It makes the formula **weaker**:  $\models \varphi^{UF} \rightarrow \varphi$
- Ignore the semantics of the function, but:

$$\dots f(\dots) \dots \rightsquigarrow \dots x_{f(\dots)} \dots$$

$$\boxed{a=b \wedge f(a) \neq f(b)} \rightsquigarrow x_{f(a)} \neq x_{f(b)} \wedge a=b$$

$\models$

Function:  $f : D^n \rightarrow D$   $f \in D^n \times D$



$\forall a, b \in D^n \quad f(a) \neq f(b) \Rightarrow a \neq b$

congruent

- 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  $\varphi^{UF}$  of equality logic with uninterpreted functions, transform the formula to a **satisfiability-equivalent** equality logic formula  $\varphi^E$  of the form

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

$$\boxed{\neg a} \vee \boxed{b}$$

where  $\varphi_{flat}$  is a flattening of  $\varphi^{UF}$ , and  $\varphi_{cong}$  is a conjunction of constraints for functional congruence.

For **validity-equivalence** check

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

# Ackermann's reduction

- **Input:**  $\varphi^{UF}$  with  $m$  instances of an uninterpreted function  $F$ .
- **Output:** Satisfiability-equivalent  $\varphi^E$  without any occurrences of  $F$ .

## Algorithm

1. For all term  $F(t)$  introduce  $f_t$  and build  $\varphi^{UF}[\cancel{f_t / F(t)}]$   
in  $\varphi^{UF}$
2. For all pairs  $F(t), F(t')$  in  $\varphi^{UF}$ ,  $t \neq t'$  add  
 $t = t' \Rightarrow F(t) = F(t')$



$$\textcircled{1} \quad F(x) = F(y) \wedge F(y) = F(z) \wedge (F(x) \neq F(z) \vee F(y) \neq F(u))$$

↓ encoding

$$f_x = f_y \wedge f_y = f_z \wedge (f_x \neq f_z \vee f_y \neq f_u) \wedge$$

$$(x = y \Rightarrow f_x = f_y) \wedge$$

$$(y = z \Rightarrow f_y = f_z) \wedge$$

$$(x = z \Rightarrow f_x = f_z) \wedge$$

$$(y = u \Rightarrow f_y = f_u) \wedge$$

all combinations of function parameters

$\textcircled{2}$

$$x = y \wedge \underbrace{G(\underbrace{F(x)}_{f_x})}_{g_{f_x}} \neq \underbrace{G(\underbrace{F(y)}_{f_y})}_{g_{f_y}}$$

↓ encoding

$$x = y \rightarrow f_x = f_y \wedge$$

$$f_x = f_y \rightarrow g_{f_x} = g_{f_y} \wedge$$

$$x = y \wedge g_{f_x} = g_{f_y}$$

# Ackermann's reduction

- **Input:**  $\varphi^{UF}$  with  $m$  instances of an uninterpreted function  $F$ .
- **Output:** Satisfiability-equivalent  $\varphi^E$  without any occurrences of  $F$ .

## Algorithm

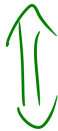
- 1 Assign indices to the  $F$ -instances.
- 2  $\varphi_{flat} := \mathcal{T}(\varphi^{UF})$  where  $\mathcal{T}$  replaces each occurrence  $F_i$  of  $F$  by a fresh theory variable  $f_i$ .
- 3  $\varphi_{cong} := \bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^m (\mathcal{T}(arg(F_i)) = \mathcal{T}(arg(F_j)))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

$$\begin{aligned}\mathcal{T}(x) &= x \\ \mathcal{T}(F(x)) &= f_x\end{aligned}$$

$$\mathcal{T}(F(G(x))) = f_{g_x}$$

$$\varphi^{UF} \longrightarrow \text{SAT / UNSAT}$$

Alg.  
from prev.  
slide



$$\varphi^{EQ} \longrightarrow \text{SAT / UNSAT}$$

$$\boxed{\neg(\underline{x}) = \neg(\underline{y}) \wedge \neg(\underline{y}) \neq \underline{z}} \quad \varphi_{\text{long}}$$

$$\begin{array}{c} \Downarrow \\ \boxed{\neg x = \neg y \wedge \neg y \neq z} \quad \wedge \quad \boxed{x = y \Rightarrow \neg x = \neg y} \\ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ \boxed{2 \quad 2 \quad 3 \quad 1 \quad 1} \end{array}$$

$$\underline{\underline{\varphi^{EQ}}}$$

$\rightarrow \text{SAT}$

$$\boxed{x = 1 \quad y = 1 \quad z = 3 \quad \neg(x) = 2 \quad \neg(y) = 2}$$

# Ackermann's reduction: Example

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

$$\varphi_{\text{flat}} = (x_1 \neq x_2) \vee (f_{x_1} = f_{x_2}) \vee (f_{x_1} \neq f_{x_3})$$

$$\begin{aligned} \varphi_{\text{cong}} = & x_1 = x_2 \rightarrow f_{x_1} = f_{x_2} \wedge \\ & x_1 = x_3 \rightarrow f_{x_1} = f_{x_3} \wedge \\ & x_2 = x_3 \rightarrow f_{x_2} = f_{x_3} \end{aligned}$$

$$x = y \wedge F(x) \neq F(y)$$

$$\begin{array}{c} \Downarrow \\ x = y \wedge f_x \neq f_y \wedge (x = y \rightarrow f_x = f_y) \\ \begin{array}{ccccccc} \downarrow & \downarrow & & \downarrow & & \downarrow & \\ 1 & 1 & & 2 & & 3 & \end{array} \end{array} \quad \left| \quad \begin{array}{c} \Downarrow \\ \text{UNSAT} \end{array} \right.$$

# Ackermann's reduction: Example

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

$$\varphi_{flat} :=$$

# Ackermann's reduction: Example

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

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

# Ackermann's reduction: Example

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

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

$$\varphi_{cong} :=$$

# Ackermann's reduction: Example

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

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

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



# Ackermann's reduction: Example

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

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

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

$$\varphi^E :=$$

# 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;  
}
```

$out_0 = in \wedge$   
 $out_1 = out_0 * in \wedge$   
 $out_2 = out_1 * in$      (  $\varphi_1$  )

```
■ int power3_b (int in){  
    return ((in * in) * in);  
}
```

$\Rightarrow$   $out_b = in * in * in$      (  $\varphi_2$  )

$out_2 = out_b$

$\Downarrow$

$\varphi_3$

# 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

$$\begin{aligned} \varphi_3 := & (out_0 = in \wedge out_1 = \overset{\textcircled{1}}{out_0} * \overset{\textcircled{2}}{in} \wedge \\ & out_2 = \underline{\overset{\textcircled{2}}{out_1} * \overset{\textcircled{2}}{in}} \wedge out_b = (\underline{\overset{\textcircled{3}}{in} * \overset{\textcircled{4}}{in}}) * \overset{\textcircled{4}}{in} \rightarrow \\ & (out_2 = out_b) \end{aligned}$$

$$\varphi_{feat} = (out_0 = in \wedge out_a = f_1 \wedge out_2 = f_2 \wedge out_b = f_4) \rightarrow out_2 = out_b$$

$$\begin{aligned} \varphi_{cong} = & (out_0 = out_a \wedge in = in) \rightarrow f_1 = f_2 & (1, 2) \\ & (out_0 = in \wedge in = in) \rightarrow f_1 = f_3 & (1, 3) \\ & (out_0 = f_3 \wedge in = in) \rightarrow f_1 = f_4 & (1, 4) \\ & (out_a = in \wedge in = in) \rightarrow f_2 = f_3 & (2, 3) \\ & (out_a = f_3 \wedge in = in) \rightarrow f_2 = f_4 & (2, 4) \\ & (in = f_3 \wedge in = in) \rightarrow f_3 = f_4 & (3, 4) \end{aligned}$$

# 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

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

# 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:**  $\varphi^{UF}$  with  $m$  instances of an uninterpreted function  $F$ .
- **Output:** Satisfiability-equivalent  $\varphi^E$  without any occurrences of  $F$ .

## Algorithm

# Bryant's reduction

- **Input:**  $\varphi^{UF}$  with  $m$  instances of an uninterpreted function  $F$ .
- **Output:** Satisfiability-equivalent  $\varphi^E$  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);  
}
```

# 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

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

# 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

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

# Bryant's reduction: Example

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

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

$$G_1^* = g_1$$

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

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

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

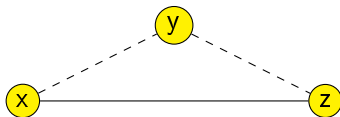
- 1 Eager SMT Solving
- 2 Eager SMT Solving for Equality Logic with Uninterpreted Functions
  - Equality Logic with Uninterpreted Functions
  - Eager SMT Solving for Uninterpreted Functions
    - Ackermann's reduction
    - Bryant's reduction
  - Eager SMT Solving for Equality Logic
    - Equality Graphs
    - 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)  $G^E(\varphi^E) = \langle V, E_ =, E_{\neq} \rangle$

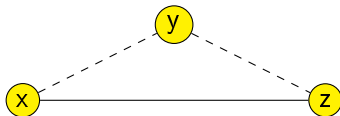


# The E-graph and Boolean structure in $\varphi^E$

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

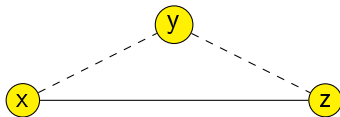
$$\varphi_2^E : (x = y \wedge y = z) \vee z \neq x \quad \text{satisfiable!}$$

Their E-graph is the same:



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

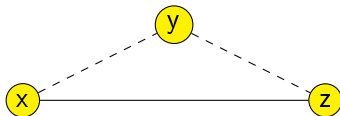
# Equality and disequality paths



## Definition (Equality Path)

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

# Equality and disequality paths



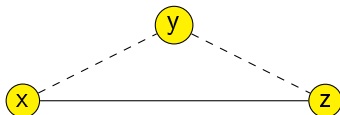
## Definition (Equality Path)

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*.

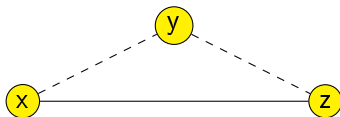
## Theorem

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

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



# Contradictory cycles



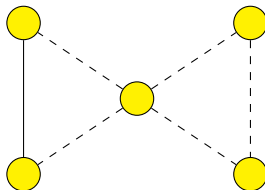
## Definition

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.*

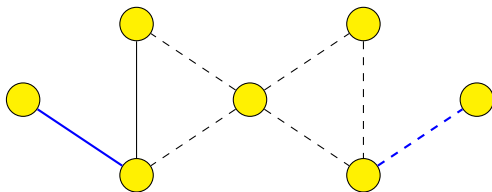
**Question:** What is a simple cycle?



## Theorem

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

# Simplifying the E-graph of $\varphi^E$



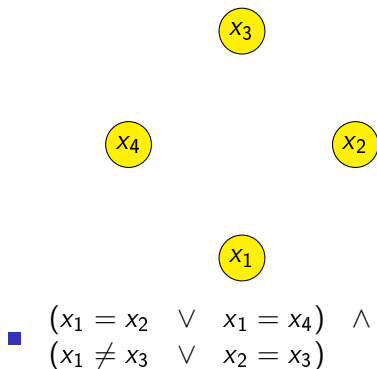
Let  $S$  be the set of edges that are **not part of any contradictory cycle**.

## Theorem

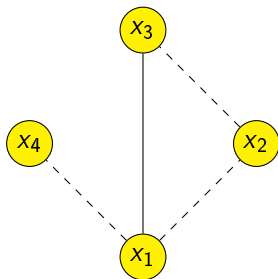
### Replacing

- all equations in  $\varphi^E$  that correspond to solid edges in  $S$  with false, and
  - all equations in  $\varphi^E$  that correspond to dashed edges in  $S$  with true
- preserves satisfiability.

# Simplifying the E-graph: Example

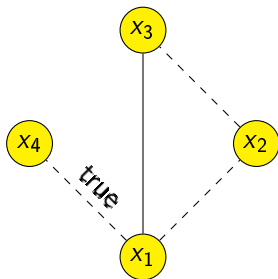


# Simplifying the E-graph: Example



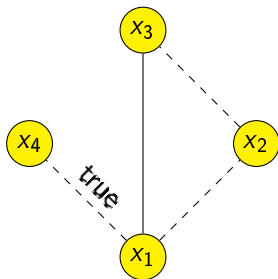
■ 
$$\begin{aligned} & (x_1 = x_2 \vee x_1 = x_4) \wedge \\ & (x_1 \neq x_3 \vee x_2 = x_3) \end{aligned}$$

# Simplifying the E-graph: Example



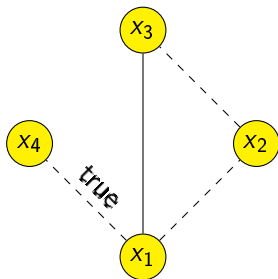
■ 
$$\begin{aligned} & (x_1 = x_2 \vee x_1 = x_4) \wedge \\ & (x_1 \neq x_3 \vee x_2 = x_3) \end{aligned}$$

# Simplifying the E-graph: Example



- $(x_1 = x_2 \vee x_1 = x_4) \wedge$   
 $(x_1 \neq x_3 \vee x_2 = x_3)$
- $(x_1 = x_2 \vee \text{true}) \wedge$   
 $(x_1 \neq x_3 \vee x_2 = x_3)$

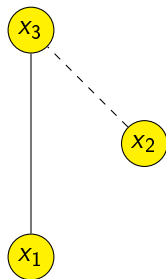
# Simplifying the E-graph: Example



- $(x_1 = x_2 \vee x_1 = x_4) \wedge$   
 $(x_1 \neq x_3 \vee x_2 = x_3)$
- $(x_1 = x_2 \vee \text{true}) \wedge$   
 $(x_1 \neq x_3 \vee x_2 = x_3)$
- $(x_1 \neq x_3 \vee x_2 = x_3)$

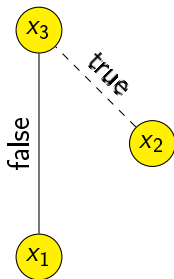


# Simplifying the E-graph: Example



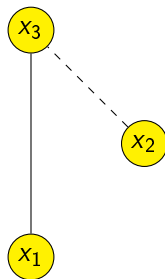
- $(x_1 = x_2 \vee x_1 = x_4) \wedge$   
 $(x_1 \neq x_3 \vee x_2 = x_3)$
- $(x_1 = x_2 \vee \text{true}) \wedge$   
 $(x_1 \neq x_3 \vee x_2 = x_3)$
- $(x_1 \neq x_3 \vee x_2 = x_3)$

# Simplifying the E-graph: Example



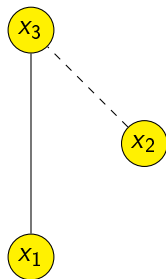
- $(x_1 = x_2 \vee x_1 = x_4) \wedge$   
 $(x_1 \neq x_3 \vee x_2 = x_3)$
- $(x_1 = x_2 \vee \text{true}) \wedge$   
 $(x_1 \neq x_3 \vee x_2 = x_3)$
- $(x_1 \neq x_3 \vee x_2 = x_3)$

# Simplifying the E-graph: Example



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

# 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

# Bryant & Velev 2000: The *Sparse* method

Goal: Transform equality logic to propositional logic

# Bryant & Velev 2000: The *Sparse* method

Goal: Transform equality logic to propositional logic

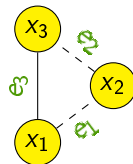
Step 1: Replace all equalities in the formula by Boolean variables

# Bryant & Velev 2000: The *Sparse* method

Goal: Transform equality logic to propositional logic

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

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



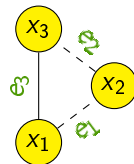


# Bryant & Velev 2000: The *Sparse* method

Goal: Transform equality logic to propositional logic

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

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



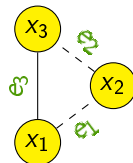
- This is called the **propositional skeleton**
- This is an over-approximation

# Bryant & Velev 2000: The *Sparse* method

Goal: Transform equality logic to propositional logic

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

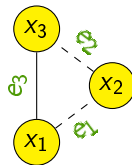
$$\begin{aligned}\varphi^E &\leftrightarrow x_1 = x_2 \wedge x_2 = x_3 \wedge x_1 \neq x_3 \\ \varphi_{sk} &\leftrightarrow e_1 \wedge e_2 \wedge \neg e_3\end{aligned}$$



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

# Adding transitivity constraints

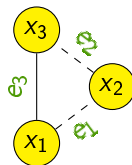
$$\begin{aligned}\varphi^E &\leftrightarrow x_1 = x_2 \wedge x_2 = x_3 \wedge x_1 \neq x_3 \\ \varphi_{sk} &\leftrightarrow e_1 \wedge e_2 \wedge \neg e_3\end{aligned}$$



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

# Adding transitivity constraints

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



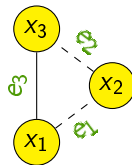
Step 2: For each cycle in the equality graph: add a transitivity constraint

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

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

# 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

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

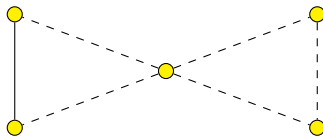
Step 3: Check  $\varphi_{sk} \wedge \varphi_{trans}$

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.*

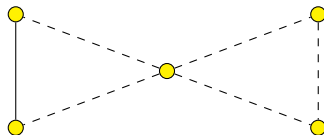


Only two simple cycles here.

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.*



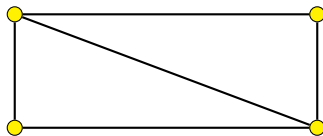
Only two simple cycles here.

**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?

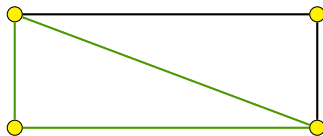
**Question:** How many chord-free simple cycles?



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?

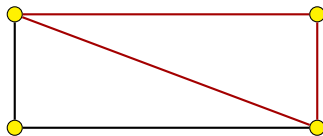
**Question:** How many chord-free simple cycles?

**Question:** Complexity?

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

## Theorem

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

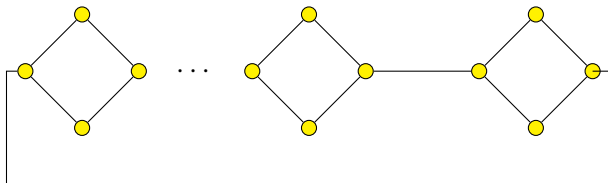


**Question:** How many simple cycles?

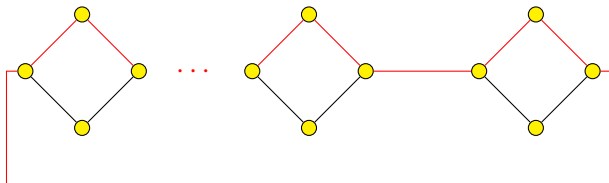
**Question:** How many chord-free simple cycles?

**Question:** Complexity?

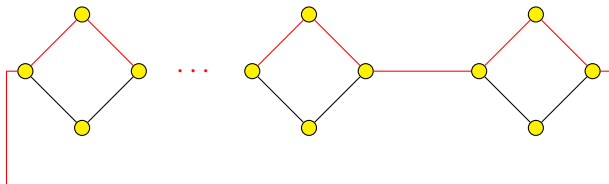
Still, there may be an exponential number of chord-free simple cycles...



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



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?

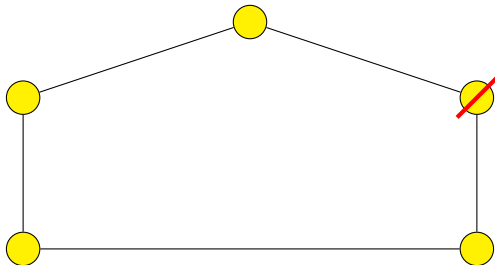
# Making the E-graph chordal

## 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.



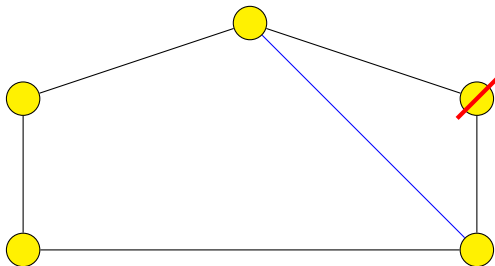
# Making the E-graph chordal

## 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.





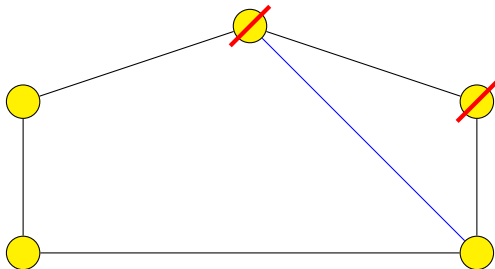
# Making the E-graph chordal

## 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.



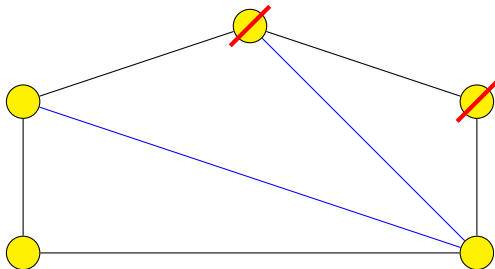
# Making the E-graph chordal

## 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.



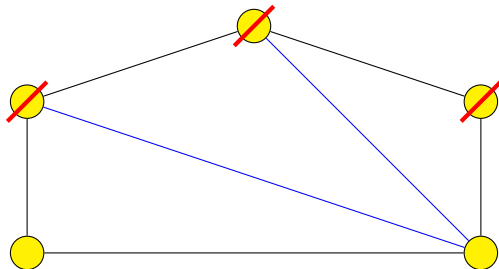
# Making the E-graph chordal

## Definition (Chordal graph)

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

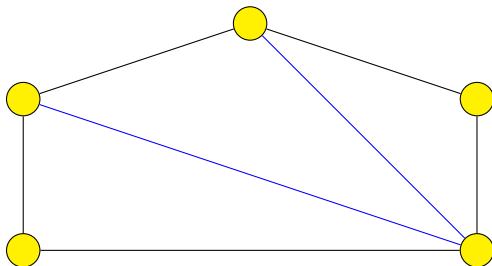
**Question:** How to **make a graph chordal**?

**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**.

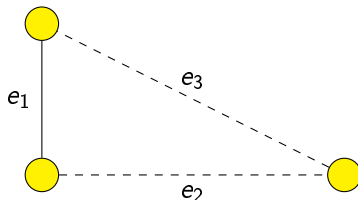


- Note that this procedure adds not more than a **polynomial number of edges**, and results in a **polynomial number of constraints**.

- So far we did not consider the **polarity** of the edges.

# 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  $\varphi^E$
- **Output:** Satisfiability-equivalent propositional logic formula  $\varphi^E$

## Algorithm

# Equality logic to propositional logic

- **Input:** Equality logic formula  $\varphi^E$
- **Output:** Satisfiability-equivalent propositional logic formula  $\varphi^E$

## Algorithm

- 1 Construct  $\varphi_{sk}$  by replacing each equality  $t_i = t_j$  in  $\varphi^E$  by a fresh Boolean variable  $e_{i,j}$ .
- 2 Construct the E-graph  $G^E(\varphi^E)$  for  $\varphi^E$ .
- 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.