Finite Model Finding in SMT

A. Reynolds[1]

C. Tinelli[1]

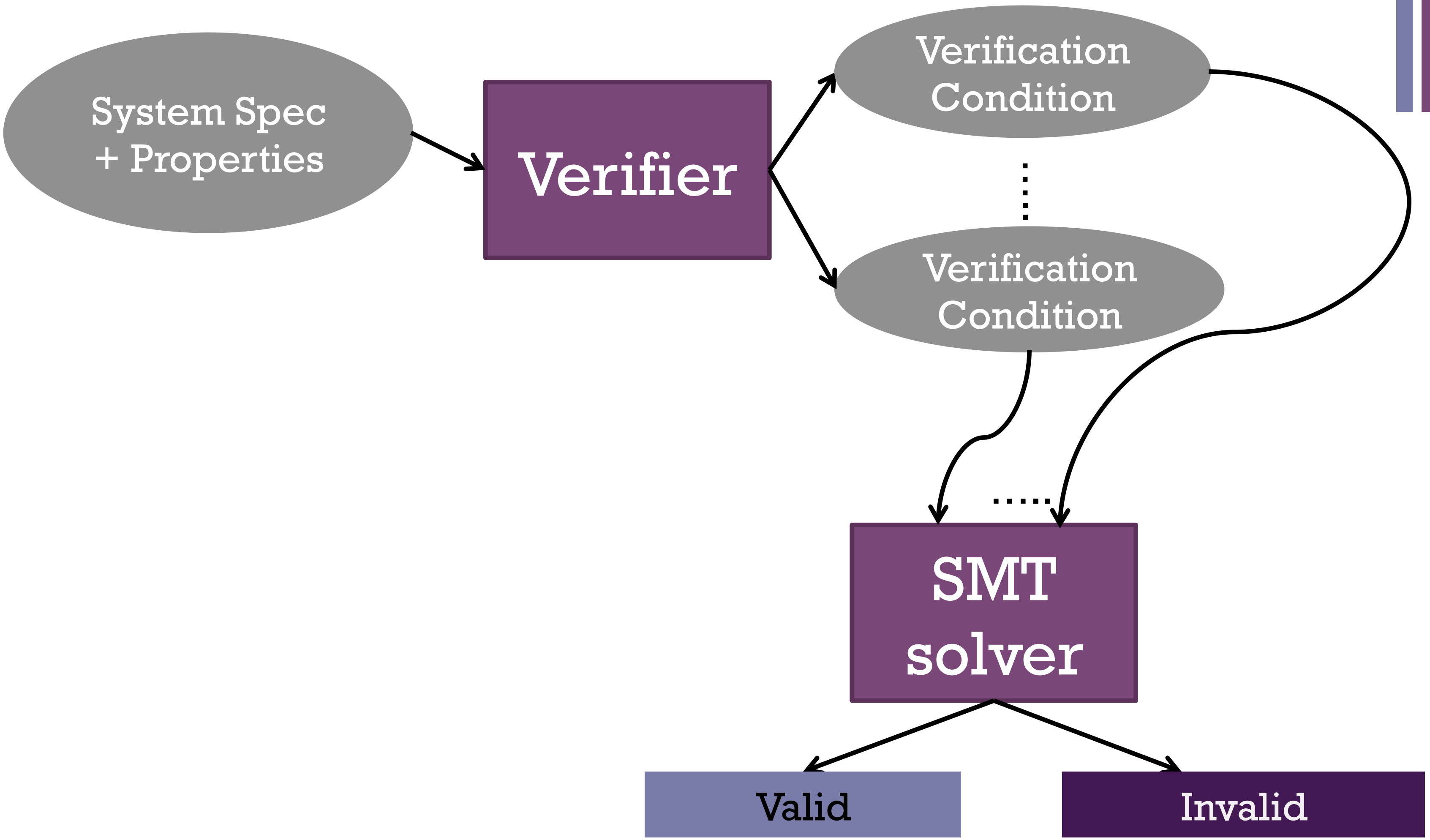
Goel[2]

S. Krstic[2]

[1] The University of Iowa

[2] Intel Corporation

SMT-Based Verification



Sample SMT Query

Definitions

S, P, R : type
null : R
valid: Array(R, Bool)
count: Array(R, Int)
ref: Array(P, R)
empty : S
mem : (S, P) -> Bool
add, remove : (S, P) -> S
…

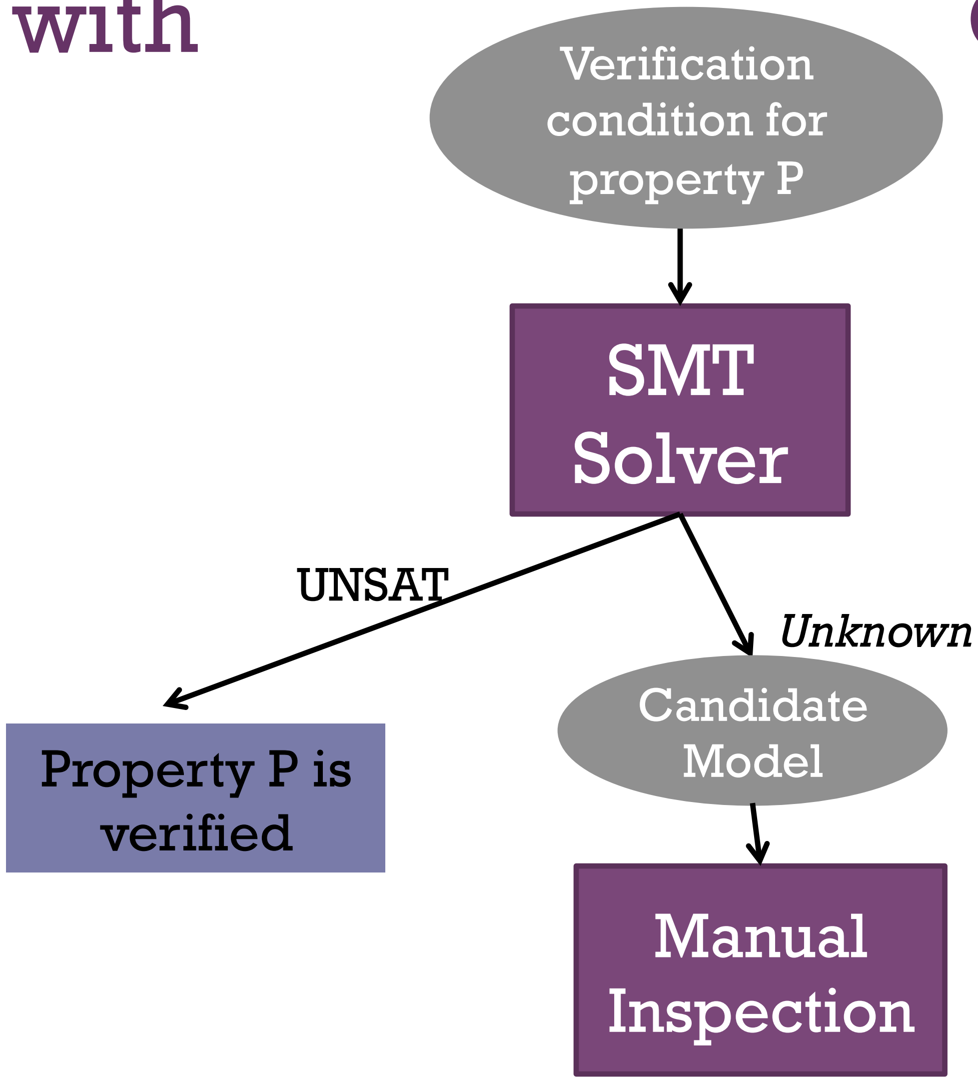
Axioms

$\forall x : R.\ count[x] > 0 \Rightarrow valid[\ x\ ]$
$\forall x : P.\ \neg\ mem(\ empty, x\ )$
$\forall x : S, y, z : P.\ mem(\ add(\ x, y\ ), z\ ) \Rightarrow (\ z = y \lor mem(\ x, z\ )\ )$
$\forall x : S, y, z : P.\ mem(\ remove(\ x, y\ ), z\ ) \Rightarrow (\ z \neq y \land mem(\ x, z\ )\ )$
…

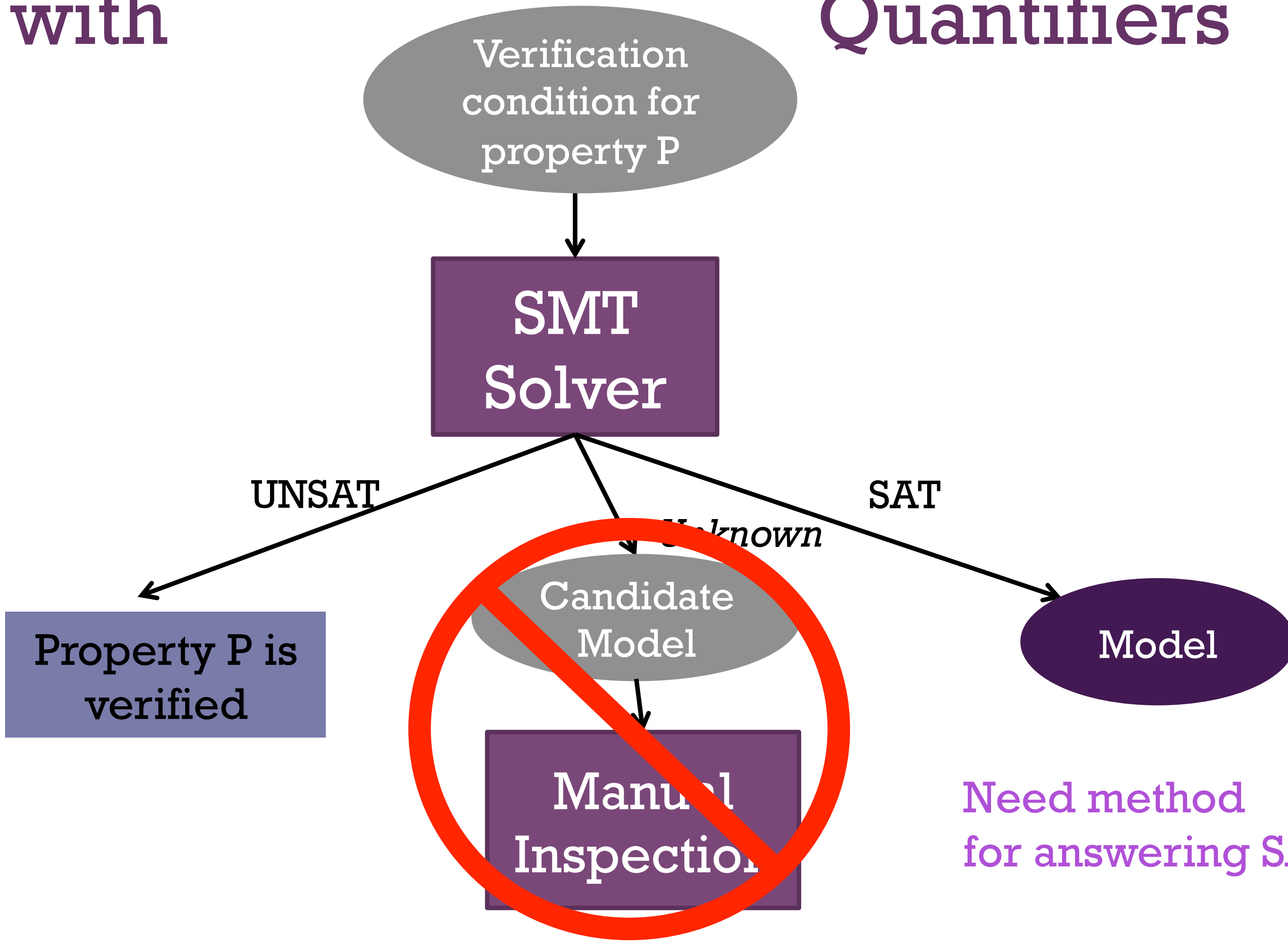
¬ (… ∀x. (ref[x] != null => valid[ref[x]]) …)

Property to verify

Handling Verification Conditions with Quantifiers



Handling Verification Conditions with Quantifiers



Need method for answering SAT

+

Quantifiers in SMT

- Quantifiers and theories do not play well together
- Current approaches: instantiation
  1. generate ground instances of quantified input formulas
  2. check their satisfiability
  3. repeat

Quantifier Instantiation

- Setting:
  - Q = {quantified formulas} ({$\forall$x. f(x) = g(x) + 4, ...})
  - G = {ground formulas} ({f(a) = b $\vee$ f(a) = c, c+1 = b})
- Main questions:
  - Which instances of Q do we add to G?
  - When can we answer SAT?

Main Instantiation Approaches

- Pattern-Based
  - Determine instantiations heuristically
    - Based on matching terms in Q with (ground) terms in G
  - Usually unable to answer SAT
- Model-Based
  - Construct from a model of G a candidate model M for Q
  - Look for instances of Q that are falsified by M
  - Can answer SAT by determining absence of such instances

This Work: Finite Model Finding

- Main Idea
  - Generate finite candidate model:
    - model that treats the uninterpreted sorts as finite domains
  - Instantiate exhaustively over domain elements
  - Answer SAT if exhaustive instantiation admits same model

This Work: Finite Model Finding

- Applicable when universal quantifiers range only over
  - uninterpreted sorts
  - finite built-in sorts (finite datatypes, bit vectors, …)
- Practical when
  - relatively small models exist
  - *redundant* instances are avoided

Contribution

- A finite model finding method fully integrated into the DPLL(T) architecture
- An efficient candidate model representation [CADE'13]
- A simple but powerful notion of instance redundancy [CADE'13]

Contribution

- A finite model finding method fully integrated into the DPLL(T) architecture
- An efficient candidate model representation [CADE'13]
- A simple but powerful notion of instance redundancy [CADE'13]

Implementation

- Fully functional implementation in CVC4
- A number of alternative configurations:
  - **cvc4** (no finite model finding)
  - **cvc4+f** (finite model finding with regions)
  - **cvc4+f-r** (finite model finding without regions)

Experimental Evaluation

Benchmarks

- Derived from real verification examples from Intel
- Both SAT and UNSAT
  - SAT benchmarks generated by removing necessary assumptions
- Many theories:
  - EUF, arithmetic, arrays, algebraic data types
- Quantifiers only over uninterpreted sorts

Experimental Results

Sat	german (45)		refcount (6)		agree (42)		apg (19)		bmk (37)	
	solved	time	solved	time	solved	time	solved	time	solved	time
cvc3	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0
yices	2	0.02	0	0.0	0	0.0	0	0.0	0	0.0
z3	45	1.1	1	7.0	0	0.0	0	0.0	0	0.0
cvc4	2	0.00	0	0.00	0	0.0	0	0.0	0	0.0
cvc4+f	**45**	0.3	**6**	0.1	**42**	15.5	**18**	200.0	**36**	1201.5
cvc4+f-r	**45**	0.3	**6**	0.1	42	18.6	15	364.3	34	720.4

Unsat	german (145)		refcount (40)		agree (488)		apg (304)		bmk (244)	
	solved	time	solved	time	solved	time	solved	time	solved	time
cvc3	145	0.4	**40**	0.2	457	6.8	267	77.0	229	76.2
yices	145	1.8	40	7.0	488	1475.4	304	35.8	244	25.3
z3	145	1.9	40	0.9	**488**	10.6	304	12.2	244	5.3
cvc4	**145**	0.1	**40**	0.2	484	6.8	**304**	11.2	**244**	2.9
cvc4+f	145	0.8	40	0.4	476	3782.1	298	2252.5	242	1507.0
cvc4+f-r	145	0.4	**40**	0.2	475	1574.3	294	3836.0	240	1930.5

Times in seconds   timeout = 600s

Our Method: Overview

- Wish to find reasonably small models
  - Impose cardinality constraints on uninterpreted sorts
  - Try models with domains of size 1, 2, 3, …
- What this requires:
  - Control to DPLL(T) search for postulating cardinalities
  - Solver for EUF + cardinality constraints
  - Instantiation strategy for avoiding redundant instances

EUF + Cardinality Constraints

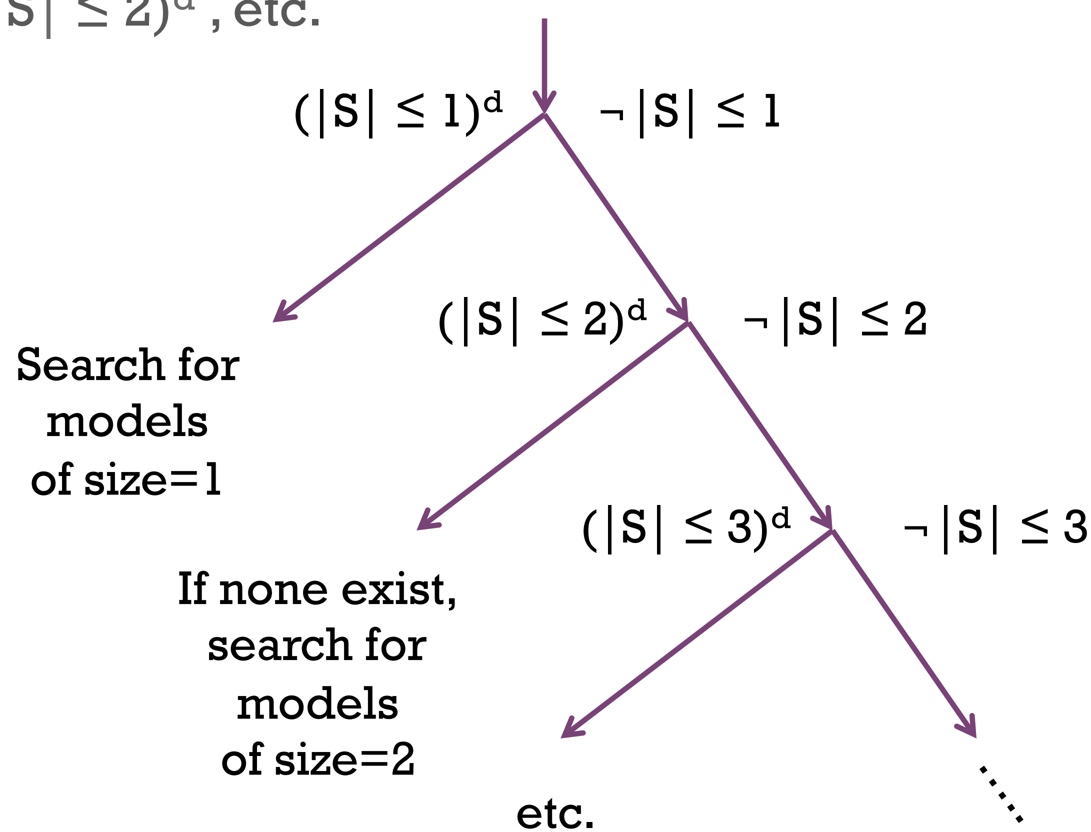
- Extend EUF solver to handle (propositional) atoms of the form:

$$|S| \leq k$$

- Meaning: cardinality of sort S is at most k
- Consider wlog only term-generated models
  - ie, domain of S is an equivalence relation over ground terms

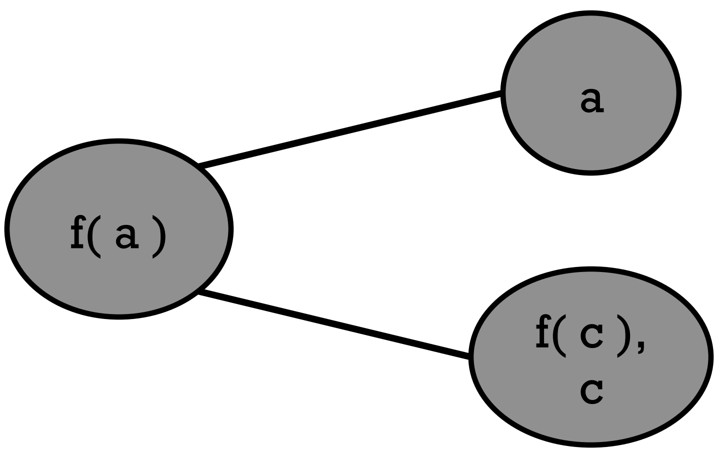
DPLL(T) for EUF + CC

- Idea: try to find models of size 1, 2, 3, …
  - Choose $(|S| \leq 1)^d$ as first decision literal
  - If fail, then try $(|S| \leq 2)^d$, etc.



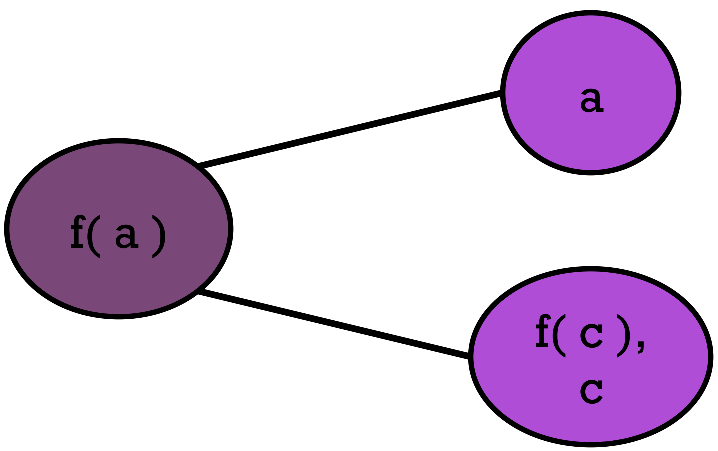
EUF + Cardinality Constraints

- For each sort S, maintain disequality graph $G_S = (V, E)$
  - V are equivalence classes of ground terms of sort S
  - E represent disequalities between terms in those classes
- Example. $f(a) \neq a, f(a) \neq c, f(c) = c$ becomes:



EUF + Cardinality Constraints

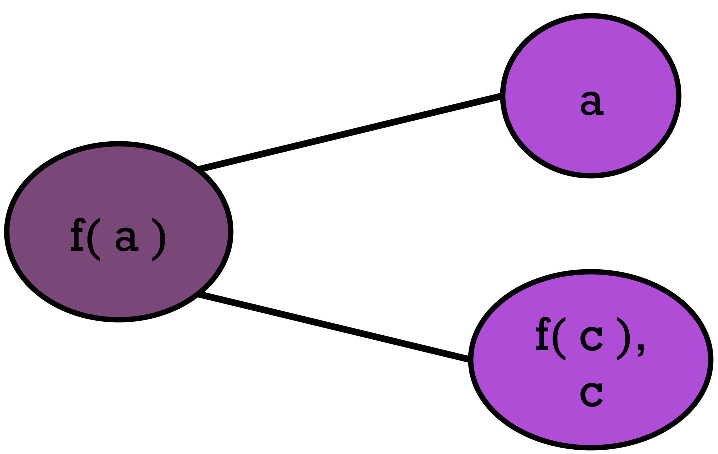
- Consider sort S with cardinality constraint $|S| \leq k$
- Check if G_S is k-colorable
  - If *not*, then we have a conflict ($C \Rightarrow \neg |S| \leq k$)
    - C explanation of sub-graph of G_S that is not k-colorable
  - Otherwise, then we *cannot* be sure a model of size k exists:
    - merging eq classes may have consequences for the theory



$|S| \leq 2$

EUF + Cardinality Constraints

- Solution: explicitly shrink model
- Use splitting on demand:
  - Add lemma (a = c ∨ a ≠ c) and explore the branch a = c first
    - If successful, # of equivalence classes is reduced by one
    - If unsuccessful,
      - a theory conflict/backtrack will occur
        - may or may not involve cardinality constraints



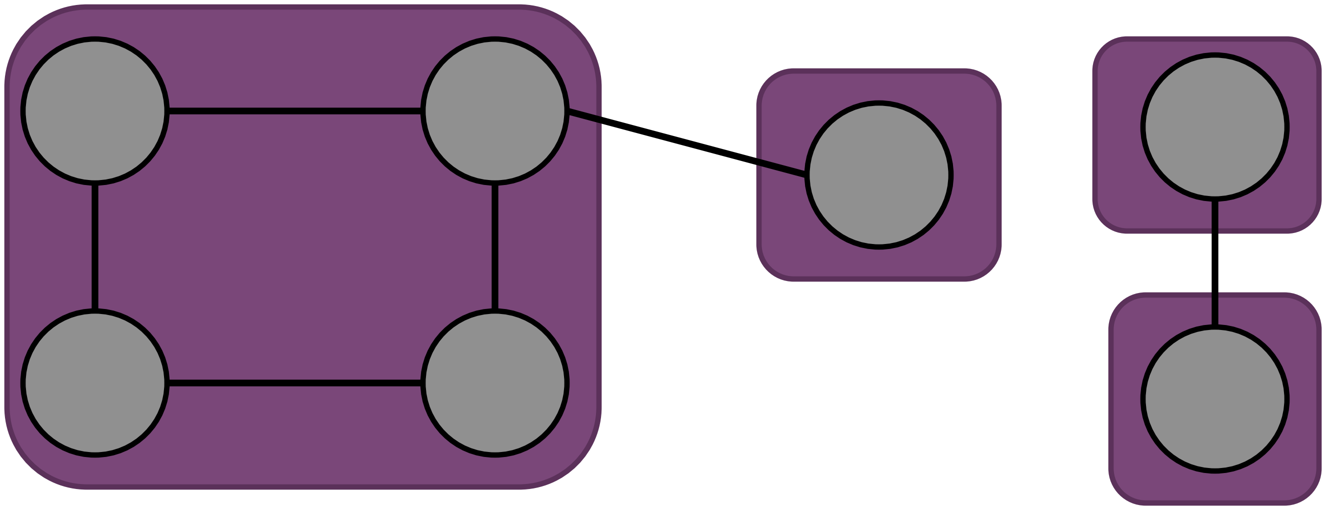
$|S| \leq 2$

EUF + Cardinality Constraints

- Good heuristics for EUF+CC solver must be:
  - able to recognize efficiently when G_S is not k-colorable
  - good at suggesting merges
- Solution: use a region-based approach
  - Partition G_S into *regions* with high edge density
  - Advantages:
    - Likely to find (k+1)-cliques
    - Can suggest relevant merges

Region-Based Approach

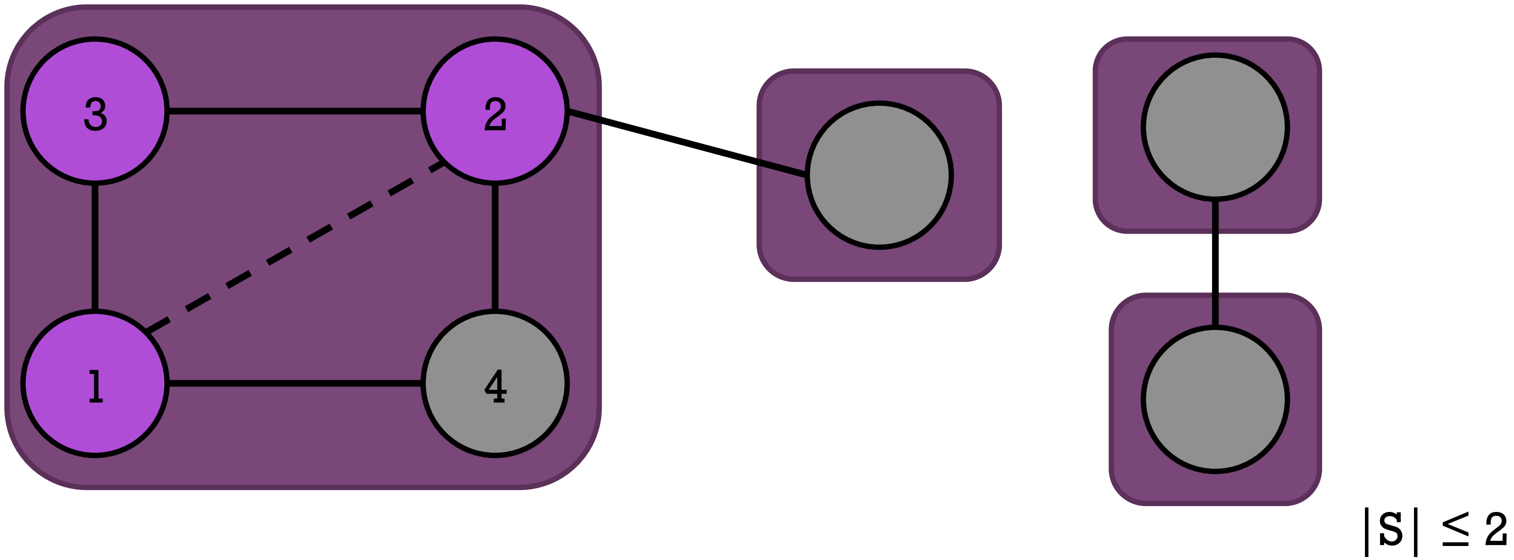
- Partition the graph G_S into regions



$|S| \leq 2$

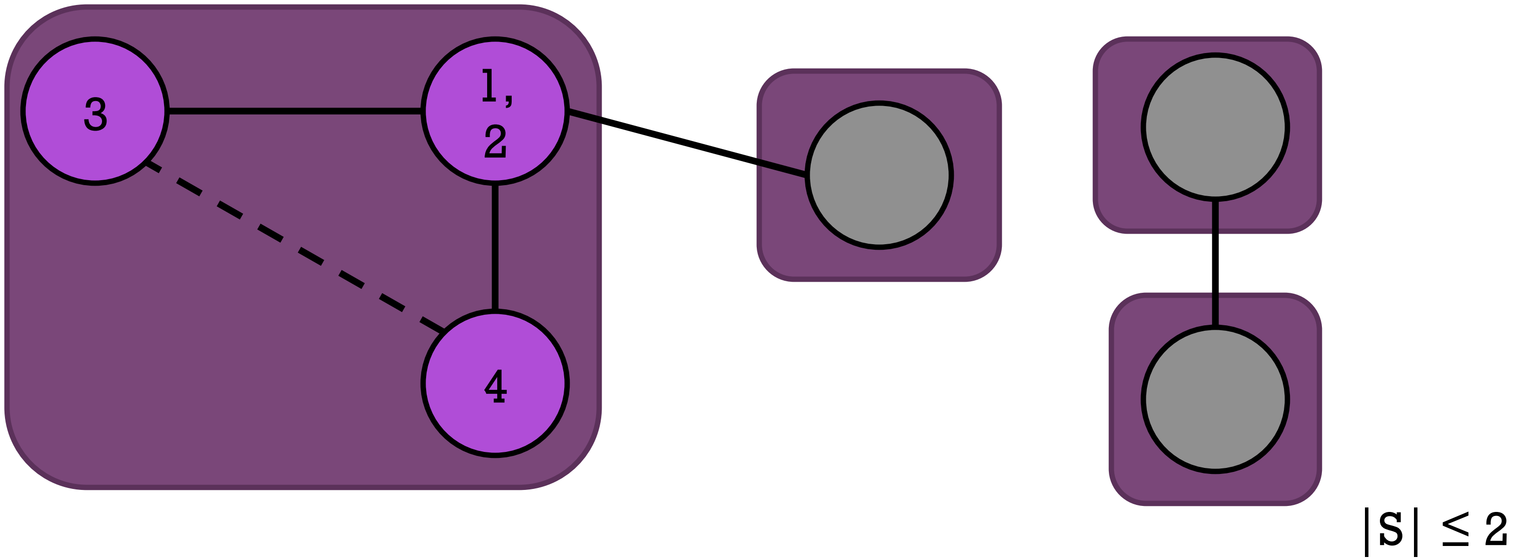
- Maintain the invariant:
  - Any (k+1)-clique is completely contained in a region
- Thus, we only need to search for cliques locally to regions
  - Regions with $\leq$ k nodes can be ignored

Region-Based Approach



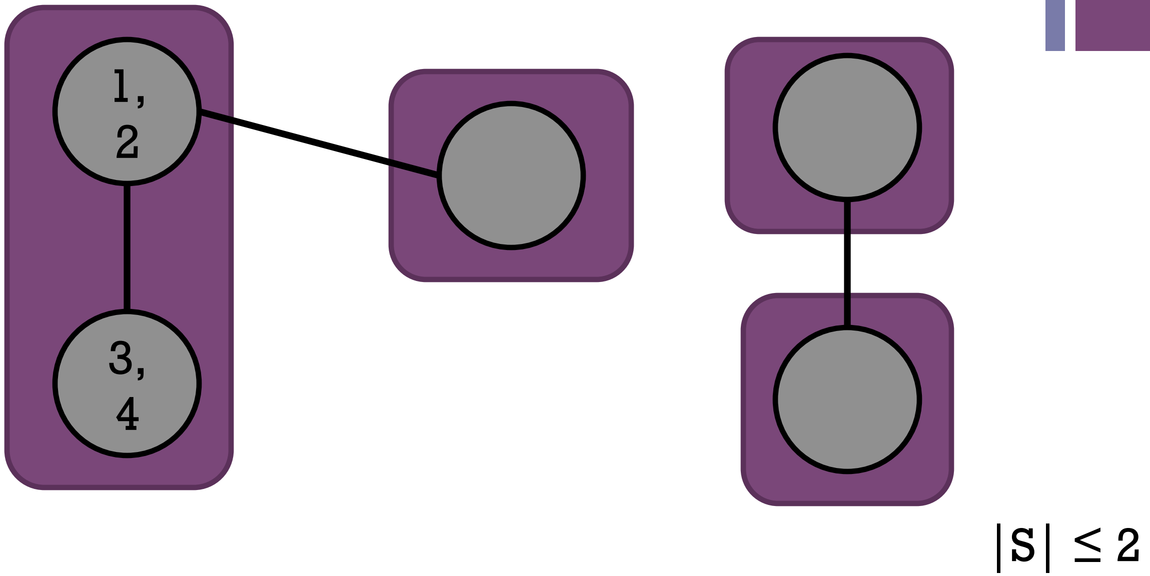
- Within each region with size > k :
  - Maintain a watched set of k+1 nodes
    - If these nodes form a clique, report a conflict
    - Otherwise, split on equalities over unlinked nodes

Region-Based Approach



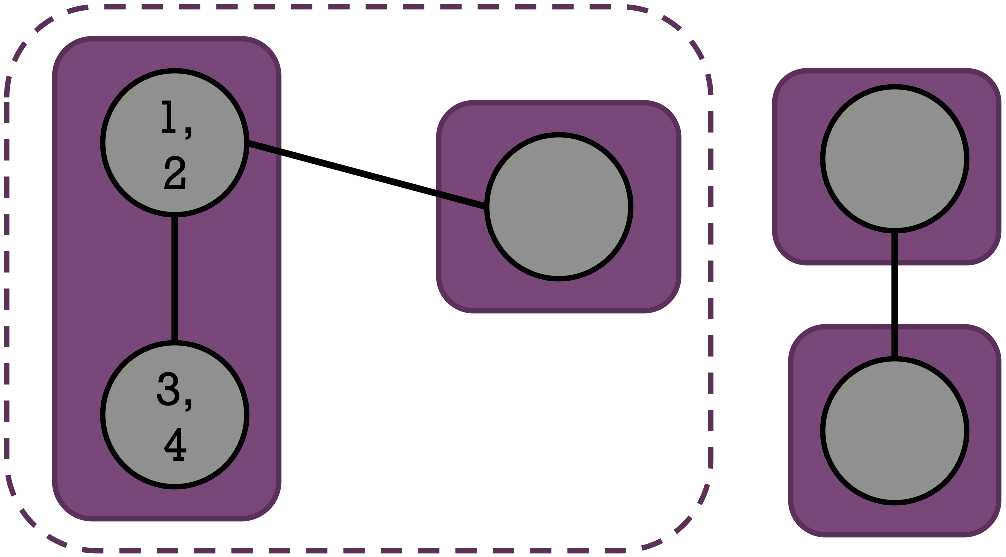
- Continue merging nodes until all regions have ≤ k nodes

Region-Based Approach



- All regions have ≤ k terms
  - k-colorability is guaranteed
  - However, still unsure a model of size k exists
    - again, due to theory consequences

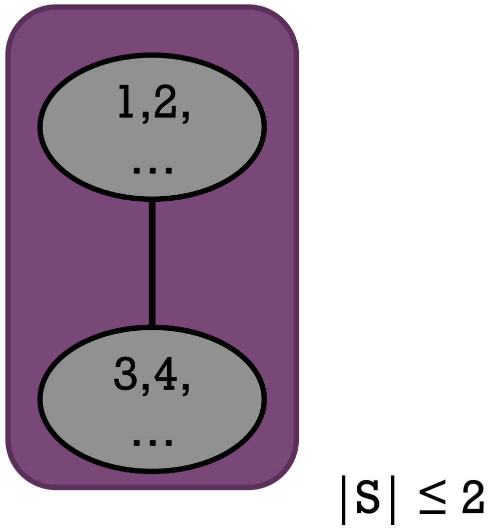
Region-Based Approach



$|S| \leq 2$

- Must shrink the model explicitly
  - Combine regions based on heuristics
    - For example, # links between regions

Region-Based Approach



- Continue merging regions and nodes until we have until ≤ k nodes overall
  - Then we have minimal model for sort S

EUF + CC Summary

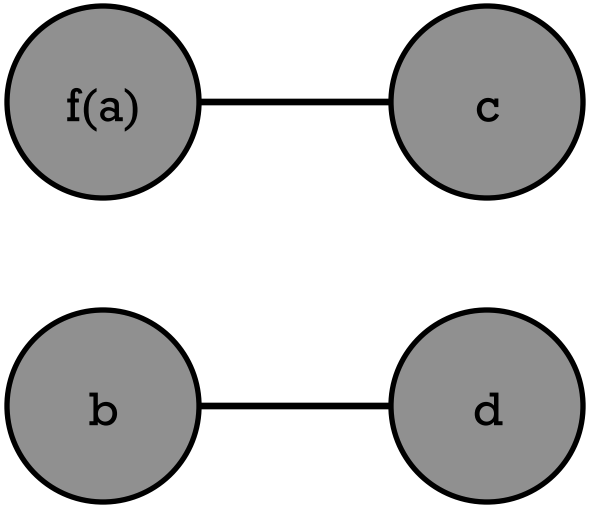
- For $|S| \leq k$, maintain a node partition into regions
  - At *weak* effort check,
    - if any (k+1)- cliques exist, report them as conflicts clauses
  - At *strong* effort check,
    - if # representatives for sort $S \leq k$
      - return SAT
    - else if there is any region R, $|R| > k$
      - split on an equality between nodes in R
    - else
      - combine regions, repeat strong effort check
- Both checks are constant time

Finite Model Finding

- Use DPLL(T) to guide search to small models
- Why small models?
  - Easier to test against quantifiers
  - Assuming model is small,
    - Instantiate quantifiers exhaustively over domain
    - If model does not *change*,
      - it satisfies quantified formulas, can answer SAT

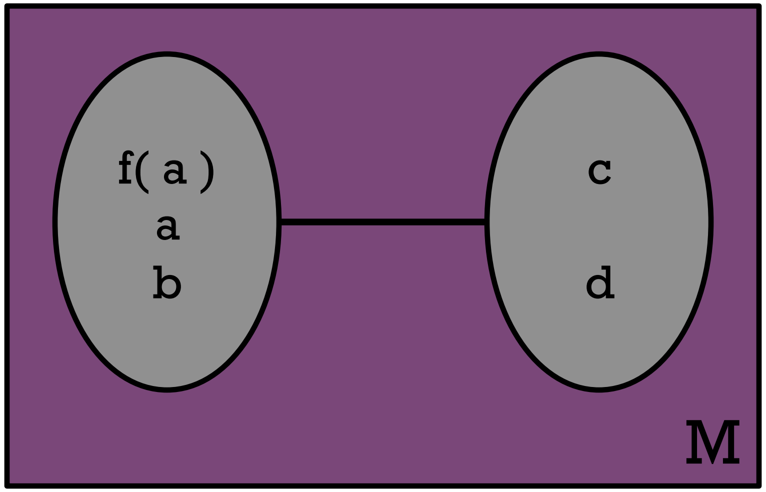
Instantiation: Example

- Current assertions: $f(a) \neq c$, $b \neq d$, $\forall xy.\, f(x) \neq g(y)$



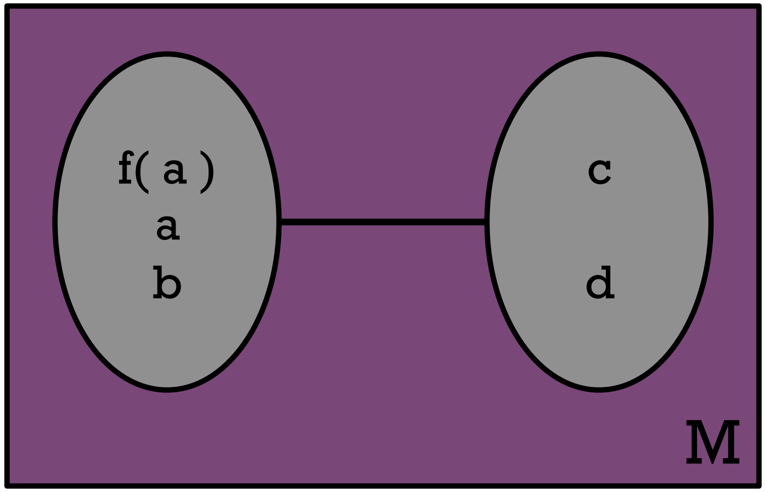
Instantiation: Example

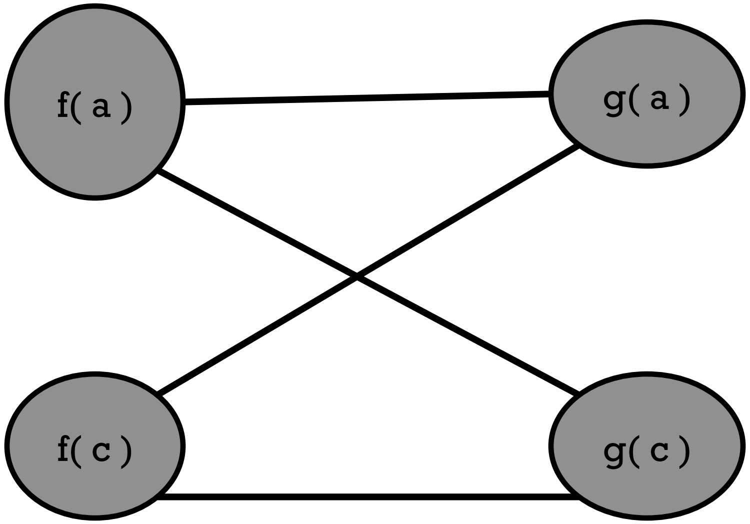
- Current assertions: $f(a) \neq c$, $b \neq d$, $\forall xy.\, f(x) \neq g(y)$
- Find minimal model M of ground part:



Instantiation: Example

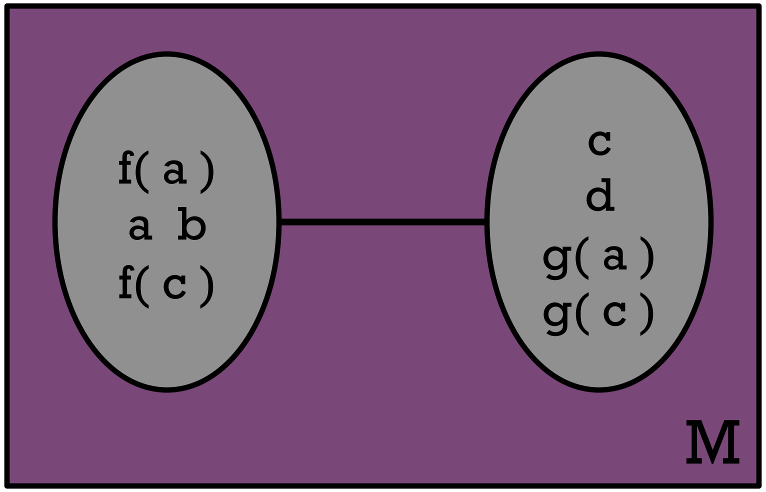
- Current assertions: $f(a) \neq c$, $b \neq d$, $\forall xy.\ f(x) \neq g(y)$
- Instantiate quantifiers with representatives a, c:





Instantiation: Example

- Current assertions: $f(a) \neq c$, $b \neq d$, $\forall xy.\ f(x) \neq g(y)$
- Try to incorporate new nodes into M



Success:

M satisfies $\forall xy.\ f(x) \neq g(y)$

Answer SAT

Conclusion

- Finite model finding with DPLL(T)
  - Uses solver for EUF + cardinality constraints
  - Finds minimal models for ground constraints
  - Uses exhaustive instantiation to test quantifiers
- Practical approach for some classes of verification problems
  - Can answer SAT quickly in many cases
  - Competitive with state of the art in SMT
  - Orthogonal to other approaches to quantifiers

Further Work

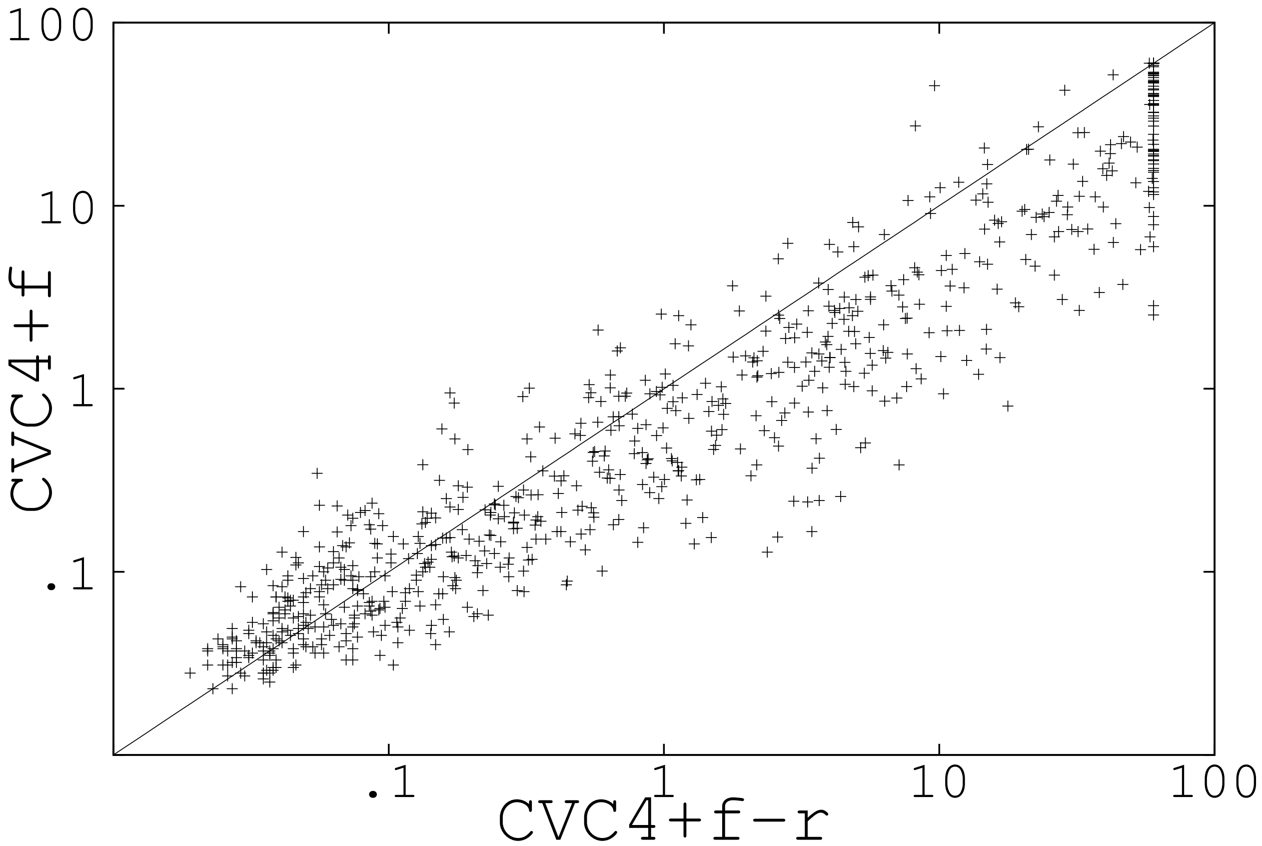
- Bounded quantification over the integers

$$\forall x.\ 0 \le x \le c \ \Rightarrow\ F[x]$$

- Incremental bounds on size of solutions over built-in structured types:
  - string length
  - list length
  - tree height
  - …

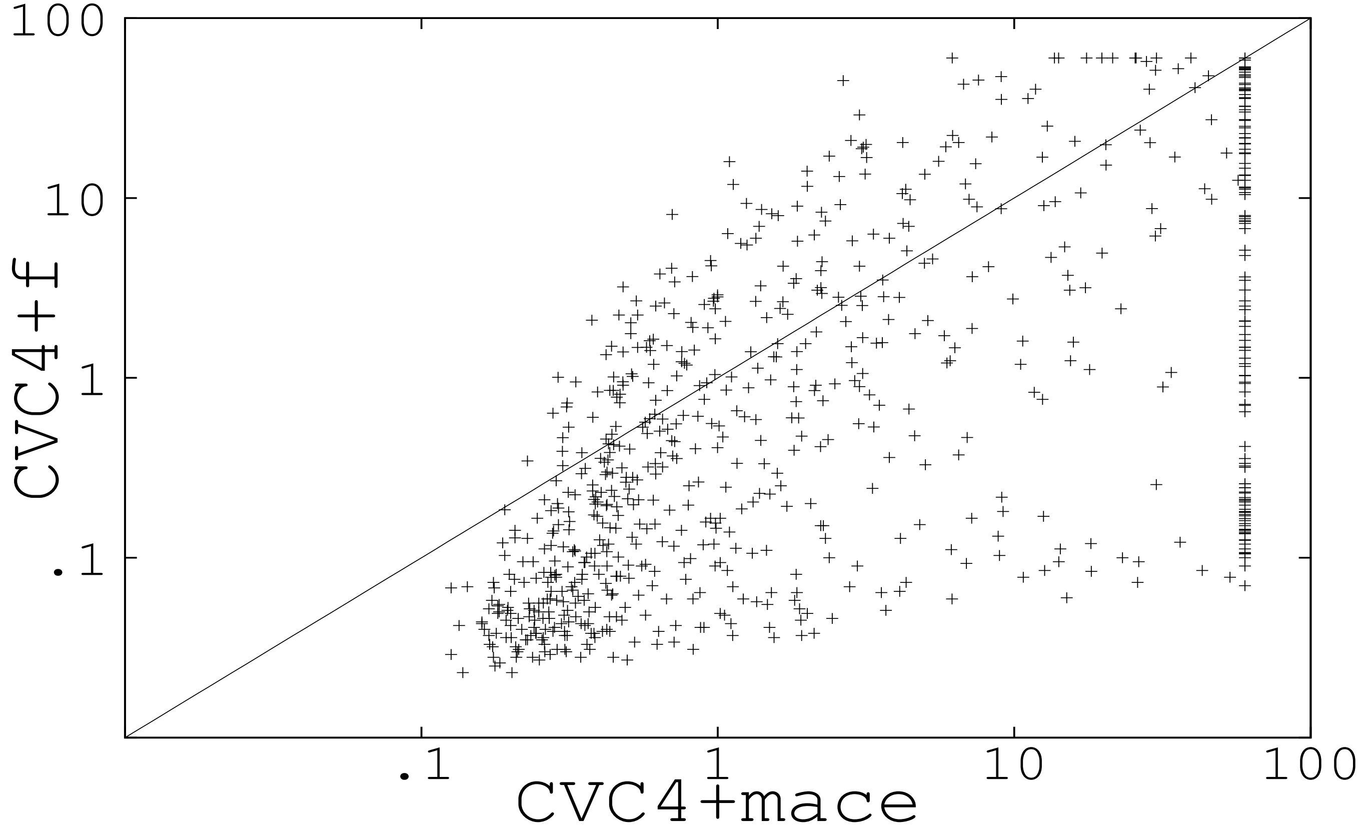
Thanks

Results: regions vs no regions



- ~800 randomly generated graph coloring problems
- 60s timeout

Results: ours vs Mace approach



- ~800 randomly generated graph coloring problems
- 60s timeout

+ Example Model from CVC4

Information regarding sorts

```
; cardinality of R is 2
(declare-sort R 0)
; cardinality of P is 1
(declare-sort P 0)
; cardinality of S is 2
(declare-sort S 0)
```

Definitions of functions and predicates in model

```
(define-fun null () R r2)
(define-fun empty () S s1)
(define-fun mem ((x1 P) (x2 S)) BOOL
        (ite (= x1 p1) (ite (= x2 s2) true false) false))
(define-fun add ((x1 P) (x2 S)) S s2)
(define-fun remove ((x1 P) (x2 S)) S s1)
(define-fun cardinality ((x1 S)) Int (ite (= x1 s1) 0 1))
(define-fun count () (Array R Int) (store count r1 0))
(define-fun ref () (Array P R) (store ref p1 r1))
(define-fun valid () (Array R BOOL) (store valid r1 true))
(define-fun destroyr () R r1)
(define-fun valid1 () (Array R BOOL) (store valid r1 true))
```