First-Order Subsumption via SAT Solving

Jakob Rath[1] Armin Biere[2] Laura Kovács[1]

[1]TU Wien

[2]University of Freiburg

Motivation

Setting:
Verification of program properties using first-order theorem proving

Example

- Data type: lists of integers
- Predicate $p(x)$ ... "all elements of x are positive"
- Function $f(x)$ ... "reverse of x"

Motivation

Setting:
Verification of program properties using first-order theorem proving

Example

- Data type: lists of integers
- Predicate $p(x)$ ... "all elements of x are positive"
- Function $f(x)$ ... "reverse of x"

- Assuming we know: $\forall x.(p(x) \rightarrow p(f(x)))$
  ... "the reverse of a positive list is positive"
- given $p(a)$ for a concrete list a,
- Goal: show that $p(f(f(a)))$

First-Order Theorem Proving

- Goal: prove formula F in classical first-order logic

$$p(a) \wedge \forall x.(p(x) \rightarrow p(f(x))) \rightarrow p(f(f(a)))$$

First-Order Theorem Proving

- Goal: prove formula F in classical first-order logic
  $$p(a) \wedge \forall x.(p(x) \rightarrow p(f(x))) \rightarrow p(f(f(a)))$$
- Approach: derive contradiction from $\neg F$
  $$p(a) \wedge \forall x.(p(x) \rightarrow p(f(x))) \wedge \neg p(f(f(a)))$$

First-Order Theorem Proving

- Goal: prove formula F in classical first-order logic

$$p(a) \land \forall x.(p(x) \rightarrow p(f(x))) \rightarrow p(f(f(a)))$$

- Approach: derive contradiction from $\neg F$

$$p(a) \land \forall x.(p(x) \rightarrow p(f(x))) \land \neg p(f(f(a)))$$

- Solver works with a set of clauses

$$p(a)$$
$$\neg p(x) \lor p(f(x))$$
$$\neg p(f(f(a)))$$

First-Order Theorem Proving

- Goal: prove formula F in classical first-order logic
$$p(a) \wedge \forall x.(p(x) \rightarrow p(f(x))) \rightarrow p(f(f(a)))$$
- Approach: derive contradiction from $\neg F$
$$p(a) \wedge \forall x.(p(x) \rightarrow p(f(x))) \wedge \neg p(f(f(a)))$$
- Solver works with a set of clauses
$$p(a)$$
$$\neg p(x) \vee p(f(x))$$
$$\neg p(f(f(a)))$$
- Saturation:
derive new clauses using a calculus, usually superposition

First-Order Theorem Proving

- Goal: prove formula F in classical first-order logic
  $$p(a) \wedge \forall x.(p(x) \rightarrow p(f(x))) \rightarrow p(f(f(a)))$$
- Approach: derive contradiction from $\neg F$
  $$p(a) \wedge \forall x.(p(x) \rightarrow p(f(x))) \wedge \neg p(f(f(a)))$$
- Solver works with a set of clauses
  $$p(a)$$
  $$\neg p(x) \vee p(f(x))$$
  $$\neg p(f(f(a)))$$
- Saturation:
  derive new clauses using a calculus, usually superposition
- Goal: derive the empty clause

Example

Example

Input clauses: $p(a), \quad \neg p(x) \vee p(f(x)), \quad \neg p(f(f(a)))$

Example

Example

Input clauses: $p(a), \quad \neg p(x) \vee p(f(x)), \quad \neg p(f(f(a)))$

Derivation:

$$\frac{\neg p(x) \vee p(f(x)) \qquad \neg p(f(f(a)))}{\neg p(f(a))}$$

Example

Example

Input clauses: $p(a), \quad \neg p(x) \vee p(f(x)), \quad \neg p(f(f(a)))$

Derivation:

$$\dfrac{\neg p(x) \vee p(f(x)) \qquad \dfrac{\neg p(x) \vee p(f(x)) \qquad \neg p(f(f(a)))}{\neg p(f(a))}}{\neg p(a)}$$

Example

Example

Input clauses: $p(a), \quad \neg p(x) \vee p(f(x)), \quad \neg p(f(f(a)))$

Derivation:

$$\dfrac{p(a) \qquad \dfrac{\neg p(x) \vee p(f(x)) \qquad \dfrac{\neg p(x) \vee p(f(x)) \qquad \neg p(f(f(a)))}{\neg p(f(a))}}{\neg p(a)}}{\square}$$

Example (2)

Example

Input clauses: $p(a), \quad \neg p(x) \vee p(f(x)), \quad \neg p(f(f(a)))$

Alternative derivation:

$$\frac{\neg p(x) \vee p(f(x)) \qquad \neg p(x) \vee p(f(x))}{\neg p(x) \vee p(f(f(x)))}$$

Example (2)

Example

Input clauses: $p(a), \quad \neg p(x) \vee p(f(x)), \quad \neg p(f(f(a)))$

Alternative derivation:

$$\dfrac{\neg p(x) \vee p(f(x)) \qquad \dfrac{\neg p(x) \vee p(f(x)) \qquad \neg p(x) \vee p(f(x))}{\neg p(x) \vee p(f(f(x)))}}{\neg p(x) \vee p(f(f(f(x))))}$$

$$\vdots$$

Example (2)

Example

Input clauses: $p(a)$, $\neg p(x) \vee p(f(x))$, $\neg p(f(f(a)))$

Alternative derivation:

$$\dfrac{\neg p(x) \vee p(f(x)) \qquad \dfrac{\neg p(x) \vee p(f(x)) \qquad \neg p(x) \vee p(f(x))}{\neg p(x) \vee p(f(f(x)))}}{\neg p(x) \vee p(f(f(f(x))))}$$

$$\vdots$$

$\rightsquigarrow$ introduce ordering and notion of redundancy!

Redundancy

- Naive saturation does not scale!
- Solution: delete redundant clauses
- Roughly: a clause is redundant, if it is the logical consequence of smaller clauses

Redundancy

- Naive saturation does not scale!
- Solution: delete redundant clauses
- Roughly: a clause is redundant, if it is the logical consequence of smaller clauses

Redundancy is undecidable!

In practice: rely on sufficient conditions

- Simplifying rules: premise becomes redundant
- Deletion rules: detect redundant clause according to sufficient conditions

Subsumption

C subsumes D
$\iff C\sigma \subseteq D$ for some substitution σ (multiset inclusion)

Subsumption

C subsumes D
$\iff$ $C\sigma \subseteq D$ for some substitution σ (multiset inclusion)

Deletion rule: If C subsumes D, then D is redundant!

Subsumption

C subsumes D
$\iff C\sigma \subseteq D$ for some substitution σ (multiset inclusion)

Deletion rule: If C subsumes D, then D is redundant!

Example

- $p(f(x))$ subsumes $p(f(f(a))) \vee q(y, b)$

Subsumption

C subsumes D
$\iff C\sigma \subseteq D$ for some substitution σ (multiset inclusion)

Deletion rule: If C subsumes D, then D is redundant!

Example

- $p(f(x))$ subsumes $p(f(f(a))) \vee q(y, b)$
- $p(x) \vee p(y)$ does not subsume $p(a)$

Subsumption

C subsumes D
$\iff C\sigma \subseteq D$ for some substitution σ (multiset inclusion)

Deletion rule: If C subsumes D, then D is redundant!

Example

- $p(f(x))$ subsumes $p(f(f(a))) \vee q(y, b)$
- $p(x) \vee p(y)$ does not subsume $p(a)$

Intuition:

- Non-ground clause is abbreviation for set of ground instances
  $p(x) \mathrel{\widehat{=}} \{p(a), p(f(a)), \ldots\}$

Subsumption Properties

- In practice:
  subsumption is one of the most important redundancy criteria
- Subsumption check is NP-complete
- Large number of subsumption checks
- Large portion of runtime used for subsumption checks

Subsumption during Saturation

Two-step approach:

1. Use indexing to retrieve a set of candidates
2. Perform subsumption check for each candidate C

Our focus: step 2

Backtracking-Based Subsumption Check

Backtracking-Based Subsumption Check

Example (Subsumption)

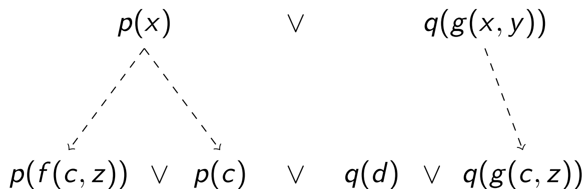
$$p(x) \qquad \vee \qquad q(g(x,y))$$

$$p(f(c,z)) \;\vee\; p(c) \quad \vee \quad q(d) \;\vee\; q(g(c,z))$$

Backtracking-Based Subsumption Check

2.a. Setup phase: find possible unit matchings

Example (Subsumption)



Backtracking-Based Subsumption Check

2.a. Setup phase: find possible unit matchings

2.b. Solving phase: backtracking search

Example (Subsumption)

$$p(x) \quad \vee \quad q(g(x,y))$$

$$p(f(c,z)) \vee p(c) \quad \vee \quad q(d) \vee q(g(c,z))$$

$$\sigma = \{x \mapsto f(c,z)\}$$

Backtracking-Based Subsumption Check

2.a. Setup phase: find possible unit matchings

2.b. Solving phase: backtracking search

Example (Subsumption)

$$p(x) \quad \lor \quad q(g(x, y))$$

$$p(f(c, z)) \lor p(c) \lor q(d) \lor q(g(c, z))$$

$$\sigma = \{x \mapsto f(c, z)\}$$

Backtracking-Based Subsumption Check

2.a. Setup phase: find possible unit matchings

2.b. Solving phase: backtracking search

Example (Subsumption)

$$p(x) \quad \vee \quad q(g(x,y))$$

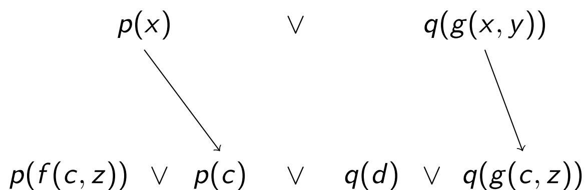
$$p(f(c,z)) \vee p(c) \vee q(d) \vee q(g(c,z))$$

$$\sigma = \{x \mapsto c\}$$

Backtracking-Based Subsumption Check

2.a. Setup phase: find possible unit matchings
2.b. Solving phase: backtracking search

Example (Subsumption)



$\sigma = \{x \mapsto c, y \mapsto z\}$ Subsumed!

Substitution Constraints

Observations:

- Matching literal L to literal M imposes constraint on substitution
- Constraints from different literal matchings must be compatible

Substitution Constraints

Observations:

- Matching literal L to literal M imposes constraint on substitution
- Constraints from different literal matchings must be compatible

Formalize matching using substitution constraints:

$$\Gamma(L, M) := (x_1, \ldots, x_n) \triangleright (t_1, \ldots, t_n)$$

Substitution Constraints

Observations:

- Matching literal L to literal M imposes constraint on substitution
- Constraints from different literal matchings must be compatible

Formalize matching using substitution constraints:

$$\Gamma(L, M) := (x_1, \ldots, x_n) \triangleright (t_1, \ldots, t_n)$$

Contraints are compatible if shared variables are mapped to the same terms.

Substitution Constraints (example)

Example

Literals:

$$L_1 = p(x_1, x_2, x_3) \qquad L_2 = p(f(x_2), x_4, x_4)$$
$$M_1 = p(f(c), d, y_1) \qquad M_2 = p(f(d), c, c)$$

Substitution Constraints (example)

Example

Literals:

$$L_1 = p(x_1, x_2, x_3) \qquad L_2 = p(f(x_2), x_4, x_4)$$
$$M_1 = p(f(c), d, y_1) \qquad M_2 = p(f(d), c, c)$$

Substitution constraints:

$$\Gamma(L_1, M_1) = (x_1, x_2, x_3) \rhd (f(c), d, y_1)$$

Substitution Constraints (example)

Example

Literals:

$$L_1 = p(x_1, x_2, x_3) \qquad L_2 = p(f(x_2), x_4, x_4)$$
$$M_1 = p(f(c), d, y_1) \qquad M_2 = p(f(d), c, c)$$

Substitution constraints:

$$\Gamma(L_1, M_1) = (x_1, x_2, x_3) \triangleright (f(c), d, y_1)$$
$$\Gamma(L_1, M_2) = (x_1, x_2, x_3) \triangleright (f(d), c, c)$$

Substitution Constraints (example)

Example

Literals:

$$L_1 = p(x_1, x_2, x_3) \qquad L_2 = p(f(x_2), x_4, x_4)$$
$$M_1 = p(f(c), d, y_1) \qquad M_2 = p(f(d), c, c)$$

Substitution constraints:

$$\Gamma(L_1, M_1) = (x_1, x_2, x_3) \triangleright (f(c), d, y_1)$$
$$\Gamma(L_1, M_2) = (x_1, x_2, x_3) \triangleright (f(d), c, c)$$
$$\Gamma(L_2, M_1) = \bot$$

Substitution Constraints (example)

Example

Literals:

$$L_1 = p(x_1, x_2, x_3) \qquad L_2 = p(f(x_2), x_4, x_4)$$
$$M_1 = p(f(c), d, y_1) \qquad M_2 = p(f(d), c, c)$$

Substitution constraints:

$$\Gamma(L_1, M_1) = (x_1, x_2, x_3) \triangleright (f(c), d, y_1)$$
$$\Gamma(L_1, M_2) = (x_1, x_2, x_3) \triangleright (f(d), c, c)$$
$$\Gamma(L_2, M_1) = \perp$$
$$\Gamma(L_2, M_2) = (x_2, x_4) \triangleright (d, c)$$

Substitution Constraints (example)

Example

Literals:

$$L_1 = p(x_1, x_2, x_3) \qquad L_2 = p(f(x_2), x_4, x_4)$$
$$M_1 = p(f(c), d, y_1) \qquad M_2 = p(f(d), c, c)$$

Substitution constraints:

$$\Gamma(L_1, M_1) = (x_1, x_2, x_3) \triangleright (f(c), d, y_1)$$
$$\Gamma(L_1, M_2) = (x_1, x_2, x_3) \triangleright (f(d), c, c)$$
$$\Gamma(L_2, M_1) = \bot$$
$$\Gamma(L_2, M_2) = (x_2, x_4) \triangleright (d, c)$$

Only $\Gamma(L_1, M_1)$ and $\Gamma(L_2, M_2)$ are compatible.

SAT-Encoding of Clausal Subsumption

Input:

- Clause $C = L_1 \vee L_2 \vee \cdots \vee L_n$
- Clause $D = M_1 \vee M_2 \vee \cdots \vee M_m$

Goal: satisfiable iff C subsumes D

SAT-Encoding of Clausal Subsumption

Input:

- Clause $C = L_1 \vee L_2 \vee \cdots \vee L_n$
- Clause $D = M_1 \vee M_2 \vee \cdots \vee M_m$

Goal: satisfiable iff C subsumes D

Components:

- Boolean variables b_{ij}
  (b_{ij} is true $\iff$ literal L_i is matched to M_j)

SAT-Encoding of Clausal Subsumption

Input:

- Clause $C = L_1 \vee L_2 \vee \cdots \vee L_n$
- Clause $D = M_1 \vee M_2 \vee \cdots \vee M_m$

Goal: satisfiable iff C subsumes D

Components:

- Boolean variables b_{ij}
  (b_{ij} is true $\iff$ literal L_i is matched to M_j)
- Substitution constraints $\Gamma(L_i, M_j)$

SAT-Encoding of Clausal Subsumption

- Each L_i of C must be matched to at least one M_j of D:

$$\bigwedge_{1 \leq i \leq n} b_{i1} \vee b_{i2} \vee \cdots \vee b_{im} \qquad (1)$$

SAT-Encoding of Clausal Subsumption

- Each L_i of C must be matched to at least one M_j of D:

$$\bigwedge_{1 \leq i \leq n} b_{i1} \vee b_{i2} \vee \cdots \vee b_{im} \tag{1}$$

- Connect b_{ij} to the substitution constraints $\Gamma(L_i, M_j)$:

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m} b_{ij} \rightarrow \Gamma(L_i, M_j) \tag{2}$$

SAT-Encoding of Clausal Subsumption

- Each L_i of C must be matched to at least one M_j of D:

$$\bigwedge_{1 \leq i \leq n} b_{i1} \vee b_{i2} \vee \cdots \vee b_{im} \tag{1}$$

- Connect b_{ij} to the substitution constraints $\Gamma(L_i, M_j)$:

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m} b_{ij} \rightarrow \Gamma(L_i, M_j) \tag{2}$$

- Each M_j of D may only be matched at most once:

$$\bigwedge_{1 \leq j \leq m} \mathrm{AtMostOne}(b_{1j}, \ldots, b_{nj}) \tag{3}$$

SAT-Encoding of Clausal Subsumption

- Each L_i of C must be matched to at least one M_j of D:

$$\bigwedge_{1 \leq i \leq n} b_{i1} \vee b_{i2} \vee \cdots \vee b_{im} \tag{1}$$

- Connect b_{ij} to the substitution constraints $\Gamma(L_i, M_j)$:

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m} b_{ij} \to \Gamma(L_i, M_j) \tag{2}$$

- Each M_j of D may only be matched at most once:

$$\bigwedge_{1 \leq j \leq m} \text{AtMostOne}(b_{1j}, \ldots, b_{nj}) \tag{3}$$

- C subsumes $D \iff (1) \wedge (2) \wedge (3)$ is satisfiable.

Implementation

Developed a lean, dedicated SAT solver

Implementation

Developed a lean, dedicated SAT solver

- Dedicated: optimized for subsumption instances

Implementation

Developed a lean, dedicated SAT solver

- Dedicated: optimized for subsumption instances
- Lean: low overhead for instance setup

Implementation

Developed a lean, dedicated SAT solver

- Dedicated: optimized for subsumption instances
- Lean: low overhead for instance setup
- Implemented in the theorem prover Vampire

Implementation

Developed a lean, dedicated SAT solver

- Dedicated: optimized for subsumption instances
- Lean: low overhead for instance setup
- Implemented in the theorem prover Vampire

Our implementation:

- Direct support for substitution constraints and AtMostOne constraints via propagators

Implementation

Developed a lean, dedicated SAT solver

- Dedicated: optimized for subsumption instances
- Lean: low overhead for instance setup
- Implemented in the theorem prover Vampire

Our implementation:

- Direct support for substitution constraints and AtMostOne constraints via propagators
- Re-use solver objects

Implementation

Developed a lean, dedicated SAT solver

- Dedicated: optimized for subsumption instances
- Lean: low overhead for instance setup
- Implemented in the theorem prover Vampire

Our implementation:

- Direct support for substitution constraints and AtMostOne constraints via propagators
- Re-use solver objects
- Defer construction of data structures (e.g., watch lists)

Implementation

Developed a lean, dedicated SAT solver

- Dedicated: optimized for subsumption instances
- Lean: low overhead for instance setup
- Implemented in the theorem prover Vampire

Our implementation:

- Direct support for substitution constraints and AtMostOne constraints via propagators
- Re-use solver objects
- Defer construction of data structures (e.g., watch lists)
- Custom variable selection heuristic

Implementation: substitution constraints

In our encoding:

$$b_{ij} \to \Gamma(L_i, M_j) \qquad \text{for } 1 \leq i \leq n, 1 \leq j \leq m$$

Implementation: substitution constraints

In our encoding:

$$b_{ij} \to \Gamma(L_i, M_j) \qquad \text{for } 1 \leq i \leq n, 1 \leq j \leq m$$

Equivalent to binary clauses:

$$\neg b_{ij} \vee \neg b_{i'j'} \qquad \text{for } \Gamma(L_i, M_j),\ \Gamma(L_{i'}, M_{j'}) \text{ incompatible}$$

Implementation: substitution constraints

In our encoding:

$$b_{ij} \to \Gamma(L_i, M_j) \qquad \text{for } 1 \leq i \leq n, 1 \leq j \leq m$$

Equivalent to binary clauses:

$$\neg b_{ij} \vee \neg b_{i'j'} \qquad \text{for } \Gamma(L_i, M_j),\ \Gamma(L_{i'}, M_{j'}) \text{ incompatible}$$

- But do not create binary clauses explicitly

Implementation: substitution constraints

In our encoding:

$$b_{ij} \to \Gamma(L_i, M_j) \qquad \text{for } 1 \leq i \leq n, 1 \leq j \leq m$$

Equivalent to binary clauses:

$$\neg b_{ij} \vee \neg b_{i'j'} \qquad \text{for } \Gamma(L_i, M_j),\ \Gamma(L_{i'}, M_{j'}) \text{ incompatible}$$

- But do not create binary clauses explicitly
- Propagation whenever b_{ij} is set to true:
  - set $b_{i'j'}$ to false if $\Gamma(L_{i'}, M_{j'})$ is incompatible
  - *before* standard unit propagation
  - conflict cannot occur at this stage

Implementation: substitution constraints

In our encoding:

$$b_{ij} \to \Gamma(L_i, M_j) \qquad \text{for } 1 \leq i \leq n, 1 \leq j \leq m$$

Equivalent to binary clauses:

$$\neg b_{ij} \vee \neg b_{i'j'} \qquad \text{for } \Gamma(L_i, M_j), \Gamma(L_{i'}, M_{j'}) \text{ incompatible}$$

- But do not create binary clauses explicitly
- Propagation whenever b_{ij} is set to true:
  - set $b_{i'j'}$ to false if $\Gamma(L_{i'}, M_{j'})$ is incompatible
  - *before* standard unit propagation
  - conflict cannot occur at this stage
- Conflict resolution:
  - pretend $\neg b_{ij} \vee \neg b_{i'j'}$ is in the clause set

Implementation: AtMostOne constraints

In our encoding:

$$\text{AtMostOne}(b_{1j}, \ldots, b_{nj})$$

Implementation: AtMostOne constraints

In our encoding:

$$\text{AtMostOne}(b_{1j}, \ldots, b_{nj})$$

As binary clauses:

$$\bigwedge_{1 \leq i_1 < i_2 \leq n} \neg b_{i_1 j} \vee \neg b_{i_2 j}$$

- better translations exist, but require additional variables

Implementation: AtMostOne constraints

In our encoding:

$$\text{AtMostOne}(b_{1j}, \ldots, b_{nj})$$

As binary clauses:

$$\bigwedge_{1 \leq i_1 < i_2 \leq n} \neg b_{i_1 j} \vee \neg b_{i_2 j}$$

- better translations exist, but require additional variables
- For unit propagation and conflict resolution:
  pretend binary clauses are in the clause set.

Implementation: AtMostOne constraints

In our encoding:

$$\text{AtMostOne}(b_{1j}, \ldots, b_{nj})$$

As binary clauses:

$$\bigwedge_{1 \leq i_1 < i_2 \leq n} \neg b_{i_1 j} \vee \neg b_{i_2 j}$$

- better translations exist, but require additional variables
- For unit propagation and conflict resolution:
  pretend binary clauses are in the clause set.
- Requires watch lists for AtMostOne constraints
  - watch every variable of the constraint
  - no need to update watches

Experiments

Preparation:

- Basis: 16,312 problems from TPTP v7.5.0
- Run original Vampire on each problem
- Record subsumption log during the run
- Output: 114 billion subsumption checks
  that are representative for real Vampire runs

Experiments

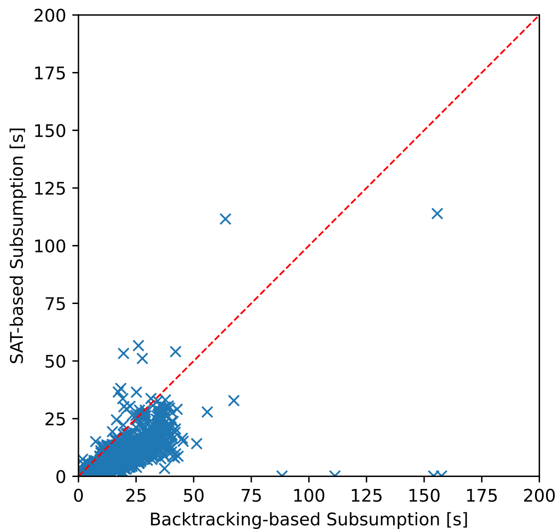
Preparation:

- Basis: 16,312 problems from TPTP v7.5.0
- Run original Vampire on each problem
- Record subsumption log during the run
- Output: 114 billion subsumption checks
  that are representative for real Vampire runs

Benchmarking:

- Load subsumption log
- Execute only the subsumption checks
- Backtracking-based vs. SAT-based algorithm
- Measure running times

Experiments



Experiments

Subsumption log for problem	Backtracking-based Subsumption Setup	Solve	Total	SAT-based Subsumption Setup	Solve	Total	Δ_{abs}	Δ_{rel}
GRP134-1.005	42.87 s	2.21 s	45.08 s	13.87 s	2.61 s	16.48 s	28.60 s	2.74 x
GRP396+1	67.05 s	88.65 s	155.70 s	15.90 s	98.01 s	113.91 s	41.79 s	1.37 x
HAL007+1	33.25 s	30.54 s	63.79 s	17.05 s	94.51 s	111.56 s	-47.78 s	0.57 x
HWV056+1	26.72 s	1.01 s	27.73 s	48.73 s	2.37 s	51.10 s	-23.37 s	0.54 x
HWV058-1	17.32 s	1.05 s	18.37 s	37.57 s	0.53 s	38.10 s	-19.73 s	0.48 x
HWV059-1	24.21 s	0.95 s	25.16 s	35.79 s	0.68 s	36.48 s	-11.31 s	0.69 x
HWV060+1	16.61 s	0.66 s	17.26 s	35.82 s	0.73 s	36.55 s	-19.28 s	0.47 x
HWV086+1	17.76 s	1.80 s	19.57 s	50.12 s	3.15 s	53.27 s	-33.71 s	0.37 x
LCL662+1.020	43.78 s	1.64 s	45.42 s	14.33 s	0.86 s	15.19 s	30.23 s	2.99 x
MGT038-1	13.15 s	12.88 s	26.04 s	15.35 s	41.33 s	56.67 s	-30.64 s	0.46 x
MGT066+1	3.45 s	63.99 s	67.44 s	1.95 s	30.87 s	32.82 s	34.63 s	2.06 x
NLP023+1	0.08 s	154.05 s	154.13 s	0.04 s	0.10 s	0.14 s	153.99 s	1082.84 x
NLP023-1	0.09 s	157.46 s	157.55 s	0.05 s	0.10 s	0.14 s	157.40 s	1087.59 x
NLP024+1	0.08 s	88.26 s	88.34 s	0.04 s	0.09 s	0.14 s	88.20 s	642.68 x
NLP024-1	0.09 s	111.20 s	111.28 s	0.05 s	0.10 s	0.15 s	111.13 s	748.52 x
PUZ073+1	24.69 s	26.60 s	51.29 s	14.02 s	0.14 s	14.17 s	37.12 s	3.62 x
SYN307-1	2.09 s	53.81 s	55.90 s	1.17 s	26.73 s	27.90 s	28.01 s	2.00 x
TOP003-2	41.71 s	0.43 s	42.13 s	48.92 s	5.13 s	54.05 s	-11.92 s	0.78 x
... (+16,294)	...	...	...	...	...	...	...	...
Total	16.31 h	2.39 h	18.70 h	7.21 h	1.23 h	8.44 h	10.27 h	**2.22** x
Total (no reuse)	-	-	-	8.08 h	2.05 h	10.12 h	-	-
Total (VMTF)	-	-	-	7.62 h	1.40 h	9.02 h	-	-

Conclusion

First-Order Subsumption via SAT Solving

- Improve efficiency of subsumption check
- Lean SAT solver is competitive even for small instances

Future Work:

- Subsumption Resolution
- Subsumption Demodulation

Conclusion

First-Order Subsumption via SAT Solving

- Improve efficiency of subsumption check
- Lean SAT solver is competitive even for small instances

Future Work:
- Subsumption Resolution
- Subsumption Demodulation

Thank you!

Implementation: variable selection heuristic

- Subsumption can be seen as constraint satisfaction problem: for each L_i, choose one of the M_j
- Boolean variables b_{ij} correspond to direct encoding of CSP into SAT
- Well-known heuristic in CSP solving: assign variable with fewest remaining choices
- Adapted for our SAT solver

Future Work

- Subsumption Resolution:

$$\frac{L \lor C \qquad \cancel{\neg M \lor D}}{D} \qquad\qquad \frac{\neg L \lor C \qquad \cancel{M \lor D}}{D}$$

  where $L\sigma = M$ and $C\sigma \subseteq D$ for some substitution σ.

- Subsumption Demodulation:

$$\frac{s \simeq t \lor C \qquad \cancel{L[s\sigma] \lor D}}{L[t\sigma] \lor D}$$

  where $s\sigma \succ t\sigma$, $C\sigma \subseteq D$, and $L[s\sigma] \lor D \succ s\sigma \simeq t\sigma \lor C\sigma$.

Ordering

Simplification Ordering

A partial ordering $\succ$ is a simplification ordering if

1. Monotonicity: $s \succ t \implies f(\dots, s, \dots) \succ f(\dots, t, \dots)$
2. Substitution: $s \succ t \implies s\theta \succ t\theta$
3. Subterm property: $f(\dots, s, \dots) \succ s$

Clause Ordering

The clause ordering $\succ$ is the multiset extension of some literal ordering $\succ_L$.

Multiset Extension

The (finite) multiset extension $\succ_M$ of a partial ordering $\succ$ is the smallest partial ordering such that

$$X \cup \{x\} \succ_M X \cup \{x_1, \ldots, x_n\}$$

where $x \succ x_i$ for all $i \in \{1, \ldots, n\}$.
Intuition: one element dominates finitely many smaller elements.

Redundancy

Ground Redundancy

A ground clause C is redundant in a set of ground clauses S if there are $C_1, \ldots, C_n \in S$ such that

- $C_1, \ldots, C_n \models C$, and
- $C_i \prec C$ for all $i \in \{1, \ldots, n\}$.

Lifted Redundancy

A clause C is redundant in a set of clauses S if all ground instances of C are redundant w.r.t. ground instances of clauses in S.