Formal Methods in System Design
https://doi.org/10.1007/s10703-024-00454-1

SAT solving for variants of first-order subsumption

Robin Coutelier[1] · Jakob Rath[1] · Michael Rawson[1] · Armin Biere[2] · Laura Kovács[1]

Received: 20 January 2024 / Accepted: 6 May 2024

Abstract
Automated reasoners, such as SAT/SMT solvers and first-order provers, are becoming the backbones of rigorous systems engineering, being used for example in applications of system verification, program synthesis, and cybersecurity. Automation in these domains crucially depends on the efficiency of the underlying reasoners towards finding proofs and/or counterexamples of the task to be enforced. In order to gain efficiency, automated reasoners use dedicated proof rules to keep proof search tractable. To this end, (variants of) subsumption is one of the most important proof rules used by automated reasoners, ranging from SAT solvers to first-order theorem provers and beyond. It is common that millions of subsumption checks are performed during proof search, necessitating efficient implementations. However, in contrast to propositional subsumption as used by SAT solvers and implemented using sophisticated polynomial algorithms, first-order subsumption in first-order theorem provers involves NP-complete search queries, turning the efficient use of first-order subsumption into a huge practical burden. In this paper we argue that the integration of a dedicated SAT solver opens up new venues for efficient implementations of first-order subsumption and related rules. We show that, by using a flexible learning approach to choose between various SAT encodings of subsumption variants, we greatly improve the scalability of first-order theorem proving. Our experimental results demonstrate that, by using a tailored SAT solver within first-order reasoning, we gain a large speedup in solving state-of-the-art benchmarks.

Keywords First-order theorem proving · SAT solving · Saturation · Subsumption

Robin Coutelier and Jakob Rath have contributed equally to this work.

✉ Robin Coutelier
robin.coutelier@tuwien.ac.at

Jakob Rath
jakob.rath@tuwien.ac.at

Michael Rawson
michael@rawsons.uk

Armin Biere
biere@cs.uni-freiburg.de

Laura Kovács
laura.kovacs@tuwien.ac.at

[1] TU Wien, Vienna, Austria

[2] University of Freiburg, Freiburg, Germany

Published online: 11 November 2024

1 Introduction

Most formal verification approaches use automated reasoners in their backend to, for example, discharge verification conditions [1–3], produce/block counter-examples [4–7], or enforce security and privacy properties [8–11]. All these approaches crucially depend on the efficiency of the underlying reasoning procedures, ranging from SAT/SMT solving [12–14] to first-order proving [15–18]. *In this paper, we focus on effective extensions of first-order theorem proving with SAT-based reasoning, improving the state-of-the-art in proving first-order (program) properties.*

Saturation-Based Theorem Proving The leading algorithmic approach in first-order theorem proving is saturation [16, 17]. While the concept of saturation is relatively unknown outside of the theorem proving community, similar algorithms that are used in other areas, such as Gröbner basis computation [19], can be considered examples of saturation algorithms. The key idea in saturation theorem proving is to reduce the problem of proving the validity of a first-order formula A to the problem of establishing unsatisfiability of $\neg A$ by using a sound inference system. That is, instead of proving A, we refute $\neg A$, by selecting and applying inferences rules. In this paper, *we focus on saturation algorithms using the superposition calculus*, which is the most commonly used inference system for first-order logic with equality [20].

Saturation with Redundancy During saturation, the first-order prover keeps a set of *usable clauses* $C_1, \ldots, C_k$ with $k \geq 0$. This is the set of clauses that the prover considers as possible premises for inferences. After applying an inference with one or more usable clauses as premises, the consequence C_{k+1} is added to the set of usable clauses. The number of usable clauses is an important factor for the efficiency of proof search. A naive saturation algorithm that keeps all derived clauses in the usable set would not scale in practice. One reason is that first-order formulas in general yield infinitely many consequences. For example, consider the clause

$$\neg \mathit{positive}(x) \vee \mathit{positive}(\mathit{reverse}(x)), \tag{1}$$

where x is a universally quantified variable ranging over the algebraic datatype `list`, where list elements are integers; *positive* is a unary predicate over `list` such that $\mathit{positive}(x)$ is valid iff all elements of x are nonnegative integers; and *reverse* is a unary function symbol reversing a list. As such, clause (1) asserts that the reverse of a list x of nonnegative integers is also a list of nonnegative integers (which is clearly valid). Note that, when having clause (1) as a usable clause during proof search, the clause $\neg \mathit{positive}(x) \vee \mathit{positive}(\mathit{reverse}^n(x))$ can be derived for any $n \geq 1$ from clause (1). Adding $\neg \mathit{positive}(x) \vee \mathit{positive}(\mathit{reverse}^n(x))$ to the set of usable clauses would, however, blow up the search space unnecessarily. This is because $\neg \mathit{positive}(x) \vee \mathit{positive}(\mathit{reverse}^n(x))$ is a logical consequence of clause (1), and hence, if a formula A can be proved using $\neg \mathit{positive}(x) \vee \mathit{positive}(\mathit{reverse}^n(x))$, then A is also provable using clause (1). Yet, storing $\neg \mathit{positive}(x) \vee \mathit{positive}(\mathit{reverse}^n(x))$ as usable formulas is highly inefficient as n can be arbitrarily large.

To avoid such and similar cases of unnecessarily increasing the set of usable formulas during proof search, first-order theorem provers implement the notion of *redundancy* [21], by extending the standard superposition calculus with term/clause ordering and literal selection functions. These orderings and selection functions are used to eliminate so-called redundant clauses from the search space, where redundant clauses are logical consequences of smaller clauses w.r.t. the considered ordering. In our example

above, the clause $\neg positive(x) \vee positive(reverse^n(x))$ would be a redundant clause as it is a logical consequence of clause (1), with clause (1) being smaller (i.e. using fewer symbols) than $\neg positive(x) \vee positive(reverse^n(x))$. As such, if clause (1) is already a usable clause, saturation algorithms implementing redundancy should ideally not store $\neg positive(x) \vee positive(reverse^n(x))$ as usable clauses. To detect and reason about redundant clauses, saturation algorithms with redundancy extend the superposition inference system with so-called *simplification rules*. Simplification rules do not add new formulas to the set of (usable) clauses in the search space, but instead simplify and/or delete redundant formulas from the search space, without destroying the refutational completeness of superposition: if a formula A is valid, then $\neg A$ can be refuted using the superposition calculus extended with simplification rules. In our example above, this means that if $\neg A$ can be refuted using $\neg positive(x) \vee positive(reverse^n(x))$, then $\neg A$ can be refuted in the superposition calculus extended with simplification rules, without using $\neg positive(x) \vee positive(reverse^n(x))$ but using clause (1) instead.

Ensuring that simplification rules are applied efficiently for eliminating redundant clauses is, however, not trivial. In this paper, we show that *SAT-based approaches can effectively identify the application of simplification rules during saturation*, improving thus the efficiency of saturation algorithms with redundancy.

Subsumption for Effective Saturation While redundancy is a powerful criterion for keeping the set of clauses used in proof search as small as possible, establishing whether an arbitrary first-order formula is redundant is as hard as proving whether it is valid. For example, in order to derive that $\neg positive(x) \vee positive(reverse^n(x))$ is redundant in our example above, the prover should establish (among other conditions) that it is a logical consequence of (1), which essentially requires proving based on superposition. To reduce the burden of proving redundancy, first-order provers implement sufficient conditions towards deriving redundancy, so that these conditions can be efficiently checked (ideally using only syntactic arguments and no proofs). One such condition comes with the notion of *subsumption*, yielding one of the most impactful simplification rules in superposition-based theorem proving [22] and SAT solving [23].

The intuition behind first-order subsumption is that a (potentially large) instance of a clause C does not convey any additional information over C, and thus it should be avoided to have both C and its instance in the set of usable clauses; to this end, we say that the instance of C is subsumed by C. More formally, a clause C subsumes another clause D if there is a substitution σ such that $\sigma(C)$ is a submultiset of D.[1] In such a case, subsumption removes the subsumed clause D from the clause set. To continue our example above, a unit clause $positive(reverse^m(x))$, with $m \geq 1$, would prevent us from deriving $\neg positive(x) \vee positive(reverse^n(x))$ for any $n \geq m$, and hence eliminate an infinite branch of clause derivations from the search space.

To detect possible inferences of subsumption and related rules, state-of-the-art provers use a two-step approach [24]: (i) retrieve a small set of candidate clauses, using literal filtering methods, and then (ii) check whether any of the candidate clauses represents an actual instance of the rule. Step (i) has been well researched over the years, leading to highly efficient indexing solutions [24–26]. Interestingly, step (ii) has not received much attention, even though it is known that checking subsumption relations between multi-literal clauses is an NP-complete problem [27]. Although indexing in step (i) allows the first-order prover to skip step (ii) in many cases, the application of (ii) in the remaining cases

[1] we consider a clause C as a multiset of its literals.

may remain problematic (due to NP-hardness). For example, while profiling subsumption in the world-leading theorem prover Vampire [16], we observed subsumption applications, and in particular calls to the literal-matching algorithm of step (ii), that consume more than 20 s of running time. Given that millions of such matchings are performed during a typical first-order proof attempt, we consider such cases highly inefficient, calling for improved solutions towards step (ii). In this paper we address this demand and show that a *tailored SAT-based encoding can significantly improve the literal matching, and thus subsumption*, in first-order theorem proving. We also advocate the flexibility of SAT solving for variants of subsumption, in particular when *combining subsumption with resolution.*

Our Contributions We bring the following main contributions.

(1) We propose a *generic SAT-based encoding for capturing potential applications of both subsumption and subsumption resolution* in first-order theorem proving (Sects. 4–5).
A solution to our SAT-based encoding gives a concrete application of subsumption and/or subsumption resolution, allowing the first-order prover to apply that instance of subsumption (resolution) as a simplification rule during saturation. Moreover, our encoding is complete in the sense that any instance of subsumption (resolution) is a model of our SAT problem (Theorems 4, 6 and 7).

(2) We tailor encodings of subsumption and subsumption resolution for effective SAT-based redundancy elimination (Sect. 6). Importantly, we adjust unit propagation and conflict resolution in SAT solving towards efficient handling of subsumption and subsumption resolution (Sect. 6.1). Our resulting SAT-based redundancy approaches are directly integrated in saturation (Sect. 7), without changing the underlying design of efficient saturation.

(3) We establish a flexible learning approach to choose between encodings with different properties. We detail how to train decision trees to obtain the best complexity-efficiency trade-off in choosing encodings for subsumption resolution (Sect. 8). As part of an empirical study, we analyse the utility of solving subsumption and subsumption resolution problems for a large portion of our computation budget. We introduce a method to choose an appropriate cutoff threshold and stop the SAT search prematurely. We empirically show that solely solving simple instances of subsumption and subsumption resolution is not a good solution, even with an educated timeout strategy.

(4) We implemented our SAT-based redundancy approach as a new SAT solver in the Vampire theorem prover. We empirically evaluate our approach on the standard benchmark library TPTP (Sect. 9). Our experiments demonstrate that using SAT solving for deciding and applying subsumption and subsumption resolution brings clear improvements in the saturation process of first-order proving, for example, improving the (time) performance of the prover by a factor of 1.36 when both subsumption and subsumption resolution are enabled.

Extension of Previous Works This paper is an extended version of the conference papers "First-Order Subsumption via SAT Solving" [28] and "SAT-Based Subsumption Resolution" [29] published at FMCAD 2022 and CADE 2023, respectively.

In Sect. 5, we extend the SAT-based subsumption framework of [28] to subsumption resolution and complemented [29] with unifying support for both subsumption and subsumption resolution. In Sect. 6 we extend the SAT solving algorithms of [28, 29] to solve both subsumption and subsumption resolution. As such, Sects. 4–6 unify the approaches of [28, 29] into a flexible technique for SAT-based redundancy checking in saturation. Our

paper therefore adjusts the texts of [28, 29] and extends their results with formal theoretical arguments and proofs.

In addition, this paper brings the following new contributions when compared to our papers [28, 29]. First, we introduce a symbolic approach to combine SAT-based encodings with learning heuristics to dynamically select the most promising encoding during runtime (Sect. 8.2). Second, we expand preprocessing via pruning techniques and use our SAT solver only on hard(er) problems (Sect. 6.2). Here, we provide a faster multilayered filter to detect unsatisfiable instances before they even reach the SAT engine. Third, we bring in an empirically motivated approach to trade completeness of SAT-based subsumption (resolution) for computation time, by cutting off early the harder instances of subsumption and subsumption resolution (Sect. 8.1). We show that subsuming simple clauses is not enough in practice, strengthening our argument for more scalable algorithms in the context of redundancy elimination.

2 Preliminaries

We consider standard multi-sorted first-order logic, where we support all standard Boolean connectives $\wedge$, $\vee$, $\rightarrow$, $\neg$ and quantifiers $\forall$ and $\exists$. We assume that the language contains the logical constants $\top$ and $\bot$ for always true and always false formulas, respectively. Let $\mathcal{V}$ denote the set of *first-order* variables. For the purpose of our work, we also use $\mathcal{B}$ to denote a set of *Boolean* variables, where Boolean variables (constants, atoms) are written as b. Throughout the paper, we write x, y, z for *first-order* variables; c, d for constants; f, g for function symbols; and p, q, r for predicates. The set of first-order terms $\mathcal{T}$ consists of variables, constants, and function symbols applied to other terms; we denote terms by t. First-order *atoms*, or simply atoms, are predicates applied to terms. Atoms and negated atoms are also called first-order *literals*, and denoted by ℓ, s, m. First-order *clauses*, or simply clauses, are disjunctions of literals, denoted by C, D, S, M. For convenience, the literals of a clause will often be written with subscripted lower case letters, e.g., $S = s_1 \vee s_2 \vee \cdots \vee s_k$. For simplicity, the notation used throughout this paper may possibly use indices.

A clause that consists of a single literal is called a *unit clause*. Clauses are often viewed as multisets of literals; that is, a clause $S = s_1 \vee s_2 \vee \ldots \vee s_k$ is considered to be the multiset $\{s_1, s_2, \ldots, s_k\}$.

An expression E is a term, literal, or clause. We denote the set of variables occurring in the expression E by $\mathcal{V}(E)$. A *substitution* is a partial function $\sigma : \mathcal{V} \rightarrow \mathcal{T}$; we occasionally write it as a set of mappings $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. The function σ is extended to arbitrary expressions E by simultaneously replacing each variable x in E by $\sigma(x)$, for all variables x on which σ is defined. We say an expression E_1 can be matched to an expression E_2 if there exists a substitution σ such that $\sigma(E_1) = E_2$. Additionally, we make the distinction between positive and negative polarity matches. A positive polarity match σ matches two literals s and m such that $\sigma(s) = m$, whereas a negative (or opposite) polarity match would complement one of the literals (i.e., $\sigma(s) = \neg m$).

Saturation and Subsumption Most first-order theorem provers, see e.g. [15–17], implement saturation with redundancy, using the superposition calculus [22]. A clause S subsumes a clause M iff there exists a substitution σ such that $\sigma(S) \sqsubseteq M$, where S and M are treated as multisets of literals and $\sqsubseteq$ is the multiset inclusion operator. *Subsumption* is a simplification rule that deletes subsumed clauses from the search space during saturation. Subsumption gives a powerful basis for other simplification rules. For example,

subsumption resolution [16, 17], also known as contextual literal cutting or self-subsuming resolution, is the combination of subsumption with binary resolution. On the other hand, subsumption demodulation [30] results from combining subsumption with demodulation/rewriting.

SAT Solving Modern SAT solvers, see e.g. [31–33], are based on conflict-driven clause learning (CDCL) [34], with the core procedures to *decide*, *unit-propagate*, and *resolve-conflict*. The SAT solver maintains a partial assignment of truth values to the Boolean variables. Unit propagation (also called Boolean constraint propagation), that is *unit-propagate* in a SAT solver, propagates clauses w.r.t. the partial assignment. If exactly one literal l in a clause remains unassigned in the current assignment while all other literals are false, the solver sets l to true to avoid a conflict. The two-watched-literal scheme [35] is the standard approach for efficient implementation of unit propagation.

If no propagation is possible, the solver may choose a currently unassigned variable b and set it to true or false; hence, *decide* in SAT solving. The number of variables in the current assignment that have been assigned by decision is called the *decision level*.

If all literals in a clause are false in the current assignment, the solver enters conflict resolution, via the *resolve-conflict* block of SAT solving. If the current decision level is zero, the conflict follows unconditionally from the input clauses and the solver returns "unsatisfiable" (UNSAT). Otherwise, by analysing how the literals in the conflicting clause have been assigned, the SAT solver may derive and learn a conflict lemma, undo some decisions, and continue solving.

3 Subsumption and subsumption resolution

In this section we formally define subsumption and subsumption resolution. These concepts yield important deletion/simplification rules during saturation.

Definition 1 (*Subsumption*) A clause S *subsumes* a clause M iff there exists a substitution σ such that

$$\sigma(S) \sqsubseteq M, \tag{2}$$

where $\sqsubseteq$ denotes multiset inclusion.

We call S the *side premise* of subsumption, and M the *main premise* of subsumption.

Subsumed clauses are redundant [22] and can thus be deleted from the search space without compromising the completeness of the saturation algorithm. Removing subsumed clauses M from the search space F is implemented through a simplifying rule, checking condition (2) over pairs of clauses (S, M) from F. To check condition (2) for a clause pair (S, M), every literal in S is matched to some literal in M; if a compatible set of matches is found and no literal in M is matched more than once, then M can be removed from F.

Example 1 Consider the clause $M := p(g(c, d)) \vee \neg p(f(d)) \vee \neg q(y_1)$.

- $S_1 := p(g(x_1, x_2)) \vee \neg q(x_3)$ subsumes M, as witnessed by the substitution $\sigma = \{x_1 \mapsto c, x_2 \mapsto d, x_3 \mapsto y_1\}$.
- $S_2 := p(g(x_1, x_2)) \vee \neg q(x_1)$, does *not* subsume M. This is because the first literal of S_2 imposes $x_1 \mapsto c$, while the second literal requires $x_1 \mapsto y_1$ in order to have M sub-

sumed. Note that the substitution is applied only to the side premise; we do *not* unify the clauses.
- $S_3 := p(g(x_1, d)) \vee p(g(c, x_2)) \vee \neg q(x_3)$ does not subsume M, because only set inclusion can be satisfied, rather than multi-set inclusion.

When subsumption (2) for a clause pair (S, M) fails, it might still be possible to simplify the clause M by deleting one of its literals. Subsumption resolution, referred to as SR in the sequel, aims exactly to remove one redundant literal from a clause and is defined below.

Definition 2 (*Subsumption Resolution*) Clauses S and M are the *side premise* and *main premise* of *subsumption resolution* SR , respectively, iff there is a substitution σ, a set of literals $S' \subseteq S$, and a literal $m' \in M$ such that

$$\sigma(S') = \{\neg m'\} \quad \text{and} \quad \sigma(S \setminus S') \subseteq M \setminus \{m'\}, \tag{3}$$

implying that M can be replaced by $M \setminus \{m'\}$. Subsumption resolution SR is hence the rule

$$\frac{S \qquad \cancel{M}}{M \setminus \{m'\}} \ (\mathtt{SR})$$

We indicate the deletion of a clause M by drawing a line through it, that is ($\cancel{M}$). We refer to the literal m' of M as the *resolution literal* of SR. Intuitively, subsumption resolution is binary resolution followed by subsumption of one of its premises by the conclusion. However, by combining two inferences into one it can be treated as a simplifying inference, which is advantageous from the perspective of efficient proof search.

Example 2 Consider clause $M := p(g(c, d)) \vee \neg p(f(d)) \vee \neg q(y_1)$ from Example 1.

- $S_4 := \neg p(g(x_1, x_2)) \vee \neg q(x_3)$ allows subsumption resolution with main premise M using the substitution $\sigma = \{x_1 \mapsto c, x_2 \mapsto d, x_3 \mapsto y_1\}$. Under this substitution, we have $\sigma(S_4) = \neg p(g(c, d)) \vee \neg q(y_1)$. We resolve $\sigma(S_4)$ and M to obtain the conclusion $\neg p(f(d)) \vee \neg q(y_1)$, which subsumes M. We thus have

$$\frac{\neg p(g(x_1, x_2)) \vee \neg q(x_3) \qquad \underline{p(g(c, d)) \vee \cancel{\neg p(f(d)) \vee \neg q(y_1)}}}{\neg p(f(d)) \vee \neg q(y_1)} \ (\mathtt{SR})$$

- $S_5 := \neg p(g(x_1, d)) \vee \neg p(g(c, x_2)) \vee \neg q(x_3)$ allows subsumption resolution with M with the same substitution σ and conclusion as used for S_4. In contrast to S_4, two literals of S_5 are mapped to the resolution literal.
- $S_6 := p(f(x_1)) \vee q(x_2)$ does not allow subsumption resolution with M, because at most one opposite polarity match is permitted.
- $S_7 := p(g(c, x_1)) \vee p(f(x_1)) \vee \neg p(f(x_2))$ does not allow subsumption resolution with M. While we can find a candidate resolution literal by matching $p(f(x_1))$ to $\neg p(f(d))$,

there is no possible match for $\neg p(f(x_2))$ since same-polarity matches to the resolution literal are not permitted.

- $S_8 := p(g(c, x_1)) \vee p(f(x_1)) \vee r(x_2)$ does not allow subsumption resolution with M, because there is no possible match for $r(x_2)$.

We note that subsumption and subsumption resolution are NP-complete problems [24, 27]. In this paper, we advocate the use of state-of-the-art SAT solving and provide tailored SAT encodings for subsumption and subsumption resolution, as follows. In Sect. 4, we express subsumption and subsumption resolution through constraints, allowing us to encode subsumption (resolution) as a SAT problem in Sect. 5.

4 Subsumption constraints

Throughout the remainder of the paper, we assume that clauses do not have duplicate literals and do not contain both a literal and its negation, as expressed by Assumption 1 below. Only substitutions may collapse several literals into one, as illustrated in Example 1.

Assumption 1 (No Duplicates) We assume that a clause $C = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_k$ does not have duplicate atoms. That is, C does not contain duplicate literals, nor a literal and its negation.

$$\textbf{no duplicates} \quad \text{for any } C = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_k : \ \forall i\, i'.\ \big(i \neq i' \Rightarrow \ell_i \neq \ell_{i'} \wedge \ell_i \neq \neg \ell_{i'}\big) \tag{4}$$

We first show that the application of subsumption (Theorem 1) and subsumption resolution (Theorem 2) can precisely be captured by substitution constraints, as follows.

Theorem 1 (Subsumption Constraints) *Consider two clauses $S = s_1 \vee s_2 \vee \cdots \vee s_k$ and $M = m_1 \vee m_2 \vee \cdots \vee m_n$, where M does not contain duplicate literals.*

S subsumes M iff there exists a substitution σ that satisfies the following two properties:

$$\textbf{partial completeness} \qquad \forall i.\exists j.\sigma(s_i) = m_j \tag{5}$$

$$\textbf{multiplicity conservation} \quad \forall i\, i'\, j.\big(i \neq i' \wedge \sigma(s_i) = m_j \Rightarrow \sigma(s_{i'}) \neq m_j\big) \tag{6}$$

Proof Because M does not contain duplicate literals, the subsumption condition $\sigma(S) \sqsubseteq M$ amounts to the two statements (i) each element of $\sigma(S)$ is an element of M and (ii) the multiplicity of elements in $\sigma(S)$ is at most one, i.e., there are no duplicates in $\sigma(S)$.

Statement (i) is equivalent to **partial completeness** (5).

Given (5), **multiplicity conservation** (6) can be rewritten into

$$\forall i\, i'.\big(i \neq i' \Rightarrow \sigma(s_i) \neq \sigma(s_{i'})\big),$$

which is equivalent to statement (ii). □

Note that the **partial completeness** property (5) ensures that all literals $\sigma(s_i)$ have a literal m_j to which they match. **Partial completeness** (5) alone would, however, encode a

simple subset inclusion. The **multiplicity conservation** constraint (6) ensures the preservation of the cardinality of the multi-set. In fact, due to Assumption 1, M is a simple set, and **multiplicity conservation** (6) prevents the substition σ from collapsing several literals into one. As a result of Theorem 1, only one literal in S can be matched to any literal of M.

Similarly to Theorem 1, we show that subsumption resolution can be formalised through four constraints, as follows.

Theorem 2 (Subsumption Resolution Constraints) *The clauses $M = m_1 \vee m_2 \vee \cdots \vee m_n$ and $S = s_1 \vee s_2 \vee \cdots \vee s_k$ are respectively the main and side premises of an instance of the subsumption resolution rule* SR *iff there exists a substitution σ that satisfies the following four properties*:

$$\textbf{existence} \qquad \exists i\, j.\sigma(s_i) = \neg m_j \tag{7}$$

$$\textbf{uniqueness} \qquad \exists j'.\forall i\, j.\big(\sigma(s_i) = \neg m_j \Rightarrow j = j'\big) \tag{8}$$

$$\textbf{completeness} \qquad \forall i.\exists j.\big(\sigma(s_i) = \neg m_j \vee \sigma(s_i) = m_j\big) \tag{9}$$

$$\textbf{coherence} \qquad \forall j.\big(\exists i.\sigma(s_i) = m_j \Rightarrow \forall i.\sigma(s_i) \neq \neg m_j\big) \tag{10}$$

Proof It is easy to see that the constraints (7)–(10) hold whenever subsumption resolution applies. For the other direction, we assume that the four constraints (7)–(10) hold, and prove that subsumption resolution applies on (S, M). Let S, M and σ such that the four constraints hold. **Existence** (7) implies that there exists at least one literal $m' \in M$ and a non-empty set $S' \subseteq S$ such that $\neg m' \in \sigma(S')$. **Uniqueness** (8) asserts that m' is unique, and therefore $\{\neg m'\} = \sigma(S')$. We can now divide the literals of S into two groups: S' and S^* such that $\sigma(S') = \{\neg m'\}$ and $S^* = S \backslash S'$. **Coherence** (10) ensures that $m' \notin \sigma(S^*)$.

From **completeness** (9), we derive $\sigma(S^*) \subseteq M$. Furthermore, m' is unique and $m' \notin \sigma(S^*)$. Therefore, $\sigma(S^*) \subseteq M \backslash \{m'\}$. Putting everything together, we obtain $\sigma(S') = \{\neg m'\} \wedge \sigma(S^*) \subseteq M \backslash \{m'\}$; hence SR over (S, M) applies. □

5 SAT formalization of subsumption constraints

Based on the subsumption constraints of Theorems 1 and 2, we provide tailored SAT encodings for subsumption and subsumption resolution, allowing us to devise custom SAT solving algorithms in Sect. 6.1 and integrate them in saturation Sect. 7. In what follows, we fix two arbitrary clauses $S = s_1 \vee s_2 \vee \cdots \vee s_k$ and $M = m_1 \vee m_2 \vee \cdots \vee m_n$, and give all definitions relative to (S, M). Intuitively, the constraints defined in this section encode the existence of a substitution σ which witnesses subsumption or subsumption resolution.

Variables and substitutions Given the side premise S and main premise M of subsumption or subsumption resolution, we introduce two Boolean variables $b^+_{i,j}$ and $b^-_{i,j}$ for each literal pair (s_i, m_j), as follows:

$$b^+_{i,j} \Leftrightarrow \sigma(s_i) = m_j \tag{11}$$

$$b^-_{i,j} \Leftrightarrow \sigma(s_i) = \neg m_j \tag{12}$$

We also define a set of substitutions $\Sigma^+_{i,j}$ and $\Sigma^-_{i,j}$, called *substitution constraints*, such that $\Sigma^+_{i,j}(s_i) = m_j$, and $\Sigma^-_{i,j}(s_i) = \neg m_j$. In the following, we write $\Sigma^\pm_{i,j}$ to refer to the substitution constraints of $\Sigma^+_{i,j}$ or $\Sigma^-_{i,j}$; when no such substitution exists, we write $\tilde{\Sigma}$. For example, let $s_1 = p(x, y)$ and $m_1 = \neg p(f(c), d)$. The two variables $b^+_{1,1}$, $b^-_{1,1}$ are associated to the pair (s_1, m_1) and the substitutions $\Sigma^+_{1,1} = \tilde{\Sigma}$, $\Sigma^-_{1,1} = \{x \mapsto f(c), y \mapsto d\}$.

Definition 3 (*Match set*) We define a *match set* $\Pi(S, M)$ associated to clauses S and M to contain a set of Boolean variables and positive/negative polarity matches for each literal pair (s_i, m_j) of (S, M). That is,

$$\Pi(S, M) = \left\{ \left(b^+_{i,j}, \Sigma^+_{i,j}\right), \left(b^-_{i,j}, \Sigma^-_{i,j}\right) \;\middle|\; s_i \in S \wedge m_j \in M \wedge \Sigma^+_{i,j}(s_i) = m_j \wedge \Sigma^-_{i,j}(s_i) = \neg m_j \right\} \tag{13}$$

Compatibility constraints Detecting the application of subsumption and/or subsumption resolution requires finding a substitution σ such that the subsumption constraints of Theorems 1–2 are satisfied. We achieve this by imposing that the substitution constraints $\Sigma^\pm_{i,j} \subseteq \sigma$ are true iff $\Sigma^\pm_{i,j}$ are *compatible* with a global substitution σ, in the following sense.

Definition 4 (*Substitution Compatibility*) A substitution Σ is *compatible* with another substitution Σ' if they do not map the same variable to different terms. Formally, Σ is compatible with Σ' iff

$$\forall x.(\Sigma(x) = t \wedge \Sigma'(x) = t' \wedge t \neq x \wedge t' \neq x) \implies t = t'. \tag{14}$$

The compatibility of the substitution constraints $\Sigma^\pm_{i,j} \subseteq \sigma$ with σ is encoded using the Boolean variables $b^\pm_{i,j}$, as follows:

$$\textbf{positive compatibility} \quad \bigwedge_i \bigwedge_j \left(b^+_{i,j} \Rightarrow \Sigma^+_{i,j} \subseteq \sigma\right) \tag{15}$$

$$\textbf{negative compatibility} \quad \bigwedge_i \bigwedge_j \left(b^-_{i,j} \Rightarrow \Sigma^-_{i,j} \subseteq \sigma\right) \tag{16}$$

Note that $\Sigma^+_{i,j}$ is a substitution constraint between s_i and m_j. Further, $\Sigma^+_{i,j} \subseteq \sigma \Leftrightarrow \sigma(s_i) = m_j$. Using $\Sigma^+_{i,j}$ together with (15), we derive $b^+_{i,j} \Rightarrow \sigma(s_i) = m_j$. A similar result is obtained for compatibility of $\Sigma^-_{i,j}$.

5.1 SAT encoding of subsumption

Note that Definition 1 and Theorem 1 imply that subsumption is restricted to only positive matches between literals of S, M. As such, $b^-_{i,j}$ need not to be considered for subsumption.

Using (11)–(12), we rewrite the subsumption constraints of Theorem 1 by replacing substitution constraints with the Boolean variable $b^+_{i,j}$, yielding:

SAT-based partial completeness $$\bigwedge_i \bigvee_j b^+_{i,j} \tag{17}$$

SAT-based multiplicity conservation $$\bigwedge_j AMO(\{b^+_{i,j} \mid i = 1, ..., k\}) \tag{18}$$

where $AMO(\{b^+_{i,j} \mid i = 1, ..., k\})$ is an at-most-one constraint ensuring that at most one variable $b^+_{i,j}$ is true at the same time.

Theorem 3 *Assume that clause M does not have duplicate atoms, as in (4). Let $\Pi(S, M) = \left\{ \left(b^{\pm}_{i,j}, \Sigma^{\pm}_{i,j} \right) \right\}$ be the match set of S and M.* **Positive compatibility** (15) *and* **SAT-based partial completeness** (17) *imply $\Sigma^+_{i,j} \subseteq \sigma \Rightarrow b^+_{i,j}$.*

Proof Towards a contradiction, assume there exist i, j such that $\Sigma^+_{i,j} \subseteq \sigma$ and $b^+_{i,j} = \bot$. Condition (17) implies that there exists j' such that $b^+_{i,j'} = \top$; then, by constraint (15) we have $\Sigma^+_{i,j'} \subseteq \sigma$, that is, $\Sigma^+_{i,j'}(s_i) = m_{j'}$. Since both $\Sigma^+_{i,j}$ and $\Sigma^+_{i,j'}$ impose a mapping on the same literal s_i, the mappings are on the same variables. Therefore, for $\Sigma^+_{i,j}$ and $\Sigma^+_{i,j'}$ to be compatible with σ simultaneously, they must be identical. Hence, $m_j = \Sigma^+_{i,j}(s_i) = \Sigma^+_{i,j'}(s_i) = m_{j'}$, which contradicts Assumption 1. □

We have now all the ingredients to introduce our SAT-based encoding of subsumption.

Definition 5 (*SAT-Based Subsumption Encoding*) The *SAT-based subsumption encoding* $\mathcal{E}_{\mathrm{S}}(S, M)$ of the clauses S and M is the conjunction of **positive compatibility** (15), **SAT-based partial completeness** (17), and **SAT-based multiplicity conservation** (18).

As a consequence of Theorem 3, we obtain the following corollary.

Corollary 1 *A model of the subsumption encoding $\mathcal{E}_{\mathrm{S}}(S, M)$ satisfies $\forall i, j.\ b^+_{i,j} \Leftrightarrow \sigma(s_i) = m_j$.*

Corollary 1 ensures that $\forall i, j.\ b^+_{i,j} \Leftrightarrow \Sigma^+_{i,j} \subseteq \sigma$, based on which soundness of our SAT-based subsumption encoding is derived.

Theorem 4 (Soundness) *Assume M does not contain duplicate literals. Clause S subsumes M iff the subsumption encoding $\mathcal{E}_{\mathrm{S}}(S, M)$ is satisfiable.*

Proof Corollary 1 implies that, if $\mathcal{E}_{\mathrm{S}}(S, M)$ is satisfied, all propositional variables $b^+_{i,j}$ can be replaced by $\sigma(s_i) = m_j$. **SAT-based partial completeness** (17) yields $\bigwedge_i \bigvee_j \sigma(s_i) = m_j$, that is $\forall i \exists j.\ \sigma(s_i) = m_j$. Using the at-most-one constraint of (18), if $b^+_{i,j} \neq b^+_{i,j'}$ then $b^+_{i,j} \implies \neg b^+_{i,j'}$. Based on (11), we hence obtain that if $\mathcal{E}_{\mathrm{S}}(S, M)$ is satisfiable, then $\sigma(S) \sqsubseteq M$ by Theorem 1.

For the other direction, assume S subsumes M; that is, $S \sqsubseteq M$. Based on Assumption 1, M has no duplicate literals. For $\sigma(S)$ to be a sub-multiset of M, it should not contain duplicates either. Therefore, there exists a total bijective function $j(i)$ such that $\sigma(s_i) = m_{j(i)}$. From this function, one can build a model such that $b^+_{i,j(i)} = \top$ for all i, and all other variables are false. This model satisfies **positive compatibility** (15). Indeed, since $\sigma(s_i) = m_{j(i)}$, we have $\Sigma^+_{i,j(i)} \subseteq \sigma$. **SAT-based partial completeness** (17) is also satisfied since $j(i)$ is a

total function. **SAT-based multiplicity conservation** (18) is ensured by the bijectivity of $j(i)$. In summary, if $S \sqsubseteq M$, then $\mathcal{E}_{\mathrm{S}}$ is satisfiable. □

Example 3 (Subsumption with $\mathcal{E}_{\mathrm{S}}(S, M)$) Consider the following clause pair $S = s_1 \vee s_2 \vee s_3$ and $M = m_1 \vee m_2 \vee m_3$, with

$$\begin{array}{ll} s_1 = q(x_1) & m_1 = q(c) \\ s_2 = p(x_1, x_2) & m_2 = p(c, d) \\ s_3 = p(x_2, x_1) & m_3 = p(d, c) \end{array}$$

We first construct the substitution constraints matching the different literal pairs (s_i, m_j):

$$\Sigma^{+}_{i,j} = \begin{pmatrix} \{x_1 \mapsto c\} & \tilde{\Sigma} & \tilde{\Sigma} \\ \tilde{\Sigma} & \{x_1 \mapsto c, x_2 \mapsto d\} & \{x_1 \mapsto d, x_2 \mapsto c\} \\ \tilde{\Sigma} & \{x_1 \mapsto d, x_2 \mapsto c\} & \{x_1 \mapsto c, x_2 \mapsto d\} \end{pmatrix}$$

The SAT encoding $\mathcal{E}_{\mathrm{S}}(S, M)$ of subsumption is given by:

$b^{+}_{1,1} \Rightarrow \{x_1 \mapsto c\} \subseteq \sigma$	**positive compatibility**
$b^{+}_{2,2} \Rightarrow \{x_1 \mapsto c, x_2 \mapsto d\} \subseteq \sigma$	**positive compatibility**
$b^{+}_{2,3} \Rightarrow \{x_1 \mapsto d, x_2 \mapsto c\} \subseteq \sigma$	**positive compatibility**
$b^{+}_{3,2} \Rightarrow \{x_1 \mapsto d, x_2 \mapsto c\} \subseteq \sigma$	**positive compatibility**
$b^{+}_{3,3} \Rightarrow \{x_1 \mapsto c, x_2 \mapsto d\} \subseteq \sigma$	**positive compatibility**
$b^{+}_{1,1}$	**SAT-based partial completeness**
$b^{+}_{2,2} \vee b^{+}_{2,3}$	**SAT-based partial completeness**
$b^{+}_{3,2} \vee b^{+}_{3,3}$	**SAT-based partial completeness**
$AMO(\{b^{+}_{1,1}\})$	**SAT-based multiplicity conservation**
$AMO(\{b^{+}_{2,2}, b^{+}_{3,2}\})$	**SAT-based multiplicity conservation**
$AMO(\{b^{+}_{2,3}, b^{+}_{3,3}\})$	**SAT-based multiplicity conservation**

Our tailored SAT solver from Sect. 6 returns a model $\{b^{+}_{1,1}, b^{+}_{2,2}, \neg b^{+}_{2,3}, \neg b^{+}_{3,2}, b^{+}_{3,3}\}$ that satisfies $\mathcal{E}_{\mathrm{S}}(S, M)$. We build the final substitution σ witnessing that $\sigma(S) \sqsubseteq M$, and hence S subsumes M, as the union of all the substitutions bound to variables assigned to true. This gives $\sigma = \{x_1 \mapsto c, x_2 \mapsto d\}$.

5.2 Direct SAT encoding of subsumption resolution

Similarly to subsumption, we translate the constraints of Theorem 2 into SAT, while also considering both **positive compatibility** (15) and **negative compatibility** (16). The following SAT constraints are derived from Theorem 2:

SAT-based existence $$\bigvee_i \bigvee_j b^-_{i,j} \tag{19}$$

SAT-based uniqueness $$\bigwedge_j \bigwedge_i \bigwedge_{i' \geq i} \bigwedge_{j' > j} \neg b^-_{i,j} \vee \neg b^-_{i',j'} \tag{20}$$

SAT-based completeness $$\bigwedge_i \bigvee_j b^+_{i,j} \vee b^-_{i,j} \tag{21}$$

SAT-based coherence $$\bigwedge_j \bigwedge_i \bigwedge_{i'} \neg b^+_{i,j} \vee \neg b^-_{i',j} \tag{22}$$

Theorem 5 *Assume that clause M does not have duplicate atoms, as in (4). Let $\Pi(S, M) = \left\{ \left(b^{\pm}_{i,j}, \Sigma^{\pm}_{i,j} \right) \right\}$ be the match set of S and M.* **Positive compatibility** (15), **negative compatibility** (16), *and* **completeness** (21) *ensures that $\Sigma^+_{i,j} \subseteq \sigma \Rightarrow b^+_{i,j}$ and $\Sigma^-_{i,j} \subseteq \sigma \Rightarrow b^-_{i,j}$.*

Proof We use a similar argumentation as in proving Theorem 3. We only prove the claim for s_i, m_j such that $\Sigma^+_{i,j} \subseteq \sigma$ and $b^+_{i,j} = \bot$; the other case is similar. **SAT-based completeness** (21) ensures that there exists j' such that $b^+_{i,j'} \vee b^-_{i,j'}$. Using compatibility (15)–(16), we have $b^+_{i,j'} \vee b^-_{i,j'} \Rightarrow (\sigma(s_i) = m_{j'} \vee \sigma(s_i) = \neg m_{j'})$. Similarly as in Theorem 3, we obtain $\Sigma^+_{i,j}(s_i) = \Sigma^+_{i,j'}(s_i) \vee \Sigma^+_{i,j}(s_i) = \neg \Sigma^-_{i,j'}(s_i)$, which is equivalent to $m_j = m_{j'} \vee m_j = \neg m_{j'}$. Since $\Sigma^+_{i,j} \subseteq \sigma$, we have $\Sigma^+_{i,j} \neq \tilde{\Sigma}$. Therefore, $\Sigma^-_{i,j}$ is the incompatible substitution $\tilde{\Sigma}$; that is, $\Sigma^-_{i,j} = \tilde{\Sigma}$. From $\Sigma^-_{i,j} = \tilde{\Sigma}$, we infer $b^-_{i,j} = \bot$. In short, we have $\neg b^+_{i,j} \wedge \neg b^-_{i,j} \wedge (b^+_{i,j'} \vee b^-_{i,j'})$, therefore $j \neq j'$ and (4) of Assumption 1 is violated. In conclusion, (4) $\wedge$ (15) $\wedge$ (16) $\wedge$ (21) implies (11) $\wedge$ (12). □

Following upon Theorem 5, the (direct) SAT formalization of subsumption resolution is given below.

Definition 6 (*Direct SAT Encoding of Subsumption Resolution*) The *direct SAT encoding of subsumption resolution* $\mathcal{E}^d_{\text{SR}}(S, M)$ for the side and main premises S and M is the conjunction of **positive compatibility** (15), **negative compatibility** (16), **existence** (19), **uniqueness** (20), **completeness** (9) and **coherence** (22).

Corollary 2 *A model of the direct subsumption resolution encoding $\mathcal{E}^d_{\text{SR}}(S, M)$ satisfies $\forall i, j.\ b^+_{i,j} \Leftrightarrow \sigma(s_i) = m_j$ and $\forall i, j.\ b^-_{i,j} \Leftrightarrow \sigma(s_i) = \neg m_j$.*

Towards finding an effective SAT-solving approach, Theorem 6 yields a direct algorithmic solution to subsumption resolution.

Theorem 6 (Soundness) *Assume M does not contain duplicate atoms. Clauses S and M are respectively the side and main premises of subsumption resolution iff $\mathcal{E}^d_{\text{SR}}(S, M)$ is satisfiable.*

Proof Similarly to Theorem 4, we use Corollary 2 together with the definition of $b^{\pm}_{i,j}$ to obtain the SAT constraints of $\mathcal{E}^d_{\text{SR}}(S, M)$ from Theorem 2.

Assume S and M are the side and main premises of subsumption resolution. There exists a substitution σ, a literal $m' \in M$ and a set of literals $S' \subseteq S$ such that $\sigma(S') = \{\neg m'\} \wedge \sigma(S \backslash S') \subseteq M \backslash \{m'\}$. We can build a model for $b^{\pm}_{i,j}$ that satisfies each constraint of $\mathcal{E}^{d}_{\mathtt{SR}}(S, M)$. Without loss of generality, let $m' = m_1$. For each literal in $s_{i'} \in S'$, we set $b^{-}_{i',1} = \top$. All other variables $b^{\pm}_{i',j}$ are set to false. Let $S^* = S \backslash S'$ and $M^* = M \backslash \{m'\}$. If $\sigma(S^*) \subseteq M^*$, then there exists a function $j^*(i^*)$ such that for each literal s_{i^*}, we have $\sigma(s_{i^*}) = m_{j^*(i^*)}$. For each literal s_{i^*}, we set $b^{+}_{i^*,j^*(i^*)} = \top$ and all other variables are false. This assignment is indeed a model of $\mathcal{E}^{d}_{\mathtt{SR}}(S, M)$. □

Example 4 (Subsumption Resolution with $\mathcal{E}^{d}_{\mathtt{SR}}(S, M)$) Consider the following clause pair $S = s_1 \vee s_2 \vee s_3$ and $M = m_1 \vee m_2 \vee m_3$, with

$$\begin{array}{ll} s_1 = p(f(x_1), x_2) & m_1 = \neg p(f(c), d) \\ s_2 = \neg p(x_2, x_1) & m_2 = \neg p(d, c) \\ s_3 = p(f(x_3), x_1) & m_3 = p(f(y_1), c) \end{array}$$

We build the following match sets:

$$\Sigma^{+}_{i,j} = \begin{pmatrix} \tilde{\Sigma} & \tilde{\Sigma} & \{x_1 \mapsto y_1, x_2 \mapsto c\} \\ \{x_1 \mapsto d, x_2 \mapsto f(c)\} & \{x_1 \mapsto c, x_2 \mapsto d\} & \tilde{\Sigma} \\ \tilde{\Sigma} & \tilde{\Sigma} & \{x_1 \mapsto c, x_3 \mapsto y_1\} \end{pmatrix}$$

$$\Sigma^{-}_{i,j} = \begin{pmatrix} \{x_1 \mapsto c, x_2 \mapsto d\} & \tilde{\Sigma} & \tilde{\Sigma} \\ \tilde{\Sigma} & \tilde{\Sigma} & \{x_1 \mapsto c, x_2 \mapsto f(y_1)\} \\ \{x_1 \mapsto d, x_3 \mapsto c\} & \tilde{\Sigma} & \tilde{\Sigma} \end{pmatrix}$$

We can express the direct subsumption resolution encoding $\mathcal{E}^{d}_{\mathtt{SR}}(S, M)$ as

$$b^+_{1,3} \Rightarrow \{x_1 \mapsto y_1, x_2 \mapsto c\} \subseteq \sigma \qquad \textbf{positive compatibility}$$
$$b^+_{2,1} \Rightarrow \{x_1 \mapsto d, x_2 \mapsto f(c)\} \subseteq \sigma \qquad \textbf{positive compatibility}$$
$$b^+_{2,2} \Rightarrow \{x_1 \mapsto c, x_2 \mapsto d\} \subseteq \sigma \qquad \textbf{positive compatibility}$$
$$b^+_{3,3} \Rightarrow \{x_1 \mapsto c, x_3 \mapsto y_1\} \subseteq \sigma \qquad \textbf{positive compatibility}$$
$$b^-_{1,1} \Rightarrow \{x_1 \mapsto c, x_2 \mapsto d\} \subseteq \sigma \qquad \textbf{negative compatibility}$$
$$b^-_{2,3} \Rightarrow \{x_1 \mapsto c, x_2 \mapsto f(y_1)\} \subseteq \sigma \qquad \textbf{negative compatibility}$$
$$b^-_{3,1} \Rightarrow \{x_1 \mapsto d, x_3 \mapsto c\} \subseteq \sigma \qquad \textbf{negative compatibility}$$
$$b^-_{1,1} \vee b^-_{2,3} \vee b^-_{3,1} \qquad \textbf{SAT-based existence}$$
$$\neg b^-_{1,1} \vee \neg b^-_{2,3} \qquad \textbf{SAT-based uniqueness}$$
$$\neg b^-_{2,3} \vee \neg b^-_{3,1} \qquad \textbf{SAT-based uniqueness}$$
$$b^-_{1,1} \vee b^+_{1,3} \qquad \textbf{SAT-based completeness}$$
$$b^+_{2,1} \vee b^+_{2,2} \vee b^-_{2,3} \qquad \textbf{SAT-based completeness}$$
$$b^-_{3,1} \vee b^+_{3,3} \qquad \textbf{SAT-based completeness}$$
$$\neg b^-_{1,1} \vee \neg b^+_{2,1} \qquad \textbf{SAT-based coherence}$$
$$\neg b^-_{3,1} \vee \neg b^+_{2,1} \qquad \textbf{SAT-based coherence}$$
$$\neg b^-_{2,3} \vee \neg b^+_{1,3} \qquad \textbf{SAT-based coherence}$$
$$\neg b^-_{2,3} \vee \neg b^+_{3,3} \qquad \textbf{SAT-based coherence}$$

Our SAT solver from Sect. 6 derives the model $\{\neg b^+_{1,3}, \neg b^+_{2,1}, b^+_{2,2}, b^+_{3,3}, b^-_{1,1}, \neg b^-_{2,3}, \neg b^-_{3,1}\}$ of $\mathcal{E}^d_{\mathrm{SR}}(S, M)$, as detailed in Example 6. The substitution σ is correct and is composed of the union of all the substitutions bound to variables assigned true:

$$\sigma = \bigcup \left\{ \Sigma^+_{i,j} \mid b^+_{i,j} = \top \right\} \cup \bigcup \left\{ \Sigma^-_{i,j} \mid b^-_{i,j} = \top \right\}$$

that is,

$$\begin{aligned} \sigma &= \{x_1 \mapsto c, x_2 \mapsto d\} \cup \{x_1 \mapsto c, x_3 \mapsto y_1\} \cup \{x_1 \mapsto c, x_2 \mapsto d\} \\ &= \{x_1 \mapsto c, x_2 \mapsto d, x_3 \mapsto y_1\} \end{aligned}$$

The conclusion clause of `SR` is built from the model by removing m_1 from the main premise because $b^-_{1,1} = \top$. This gives us the resolution clause $M \backslash \{m_1\} = \neg p(d, c) \vee p(f(y_1), c)$, which subsumes M.

5.3 Indirect SAT encoding of subsumption resolution

The direct SAT encoding $\mathcal{E}^d_{\mathrm{SR}}(S, M)$ of subsumption resolution has a potential inefficiency due to the fact that the **uniqueness constraint** (20) may create a quartic number of clauses in the worst case. We circumvent this issue by trading off constraints for variables, resulting in an indirect SAT encoding $\mathcal{E}^i_{\mathrm{SR}}(S, M)$ of subsumption resolution. Doing so, we introduce a

new set of propositional variables c_j such that c_j is true iff m_j is the resolution literal of SR. In other words, $c_j \Leftrightarrow \exists i.\ \sigma(s_i) = \neg m_j$.

We encode the role of c_j with constraint (23) given below:

$$\textbf{SAT-based structurality} \quad \bigwedge_j \left[\neg c_j \vee \bigvee_i b^-_{i,j}\right] \wedge \bigwedge_j \bigwedge_i \left(c_j \vee \neg b^-_{i,j}\right) \tag{23}$$

Using variables c_j, the constraints of Theorem 2 are turned into the following SAT formulas:

$$\textbf{SAT-based revised existence} \quad \bigvee_j c_j \tag{24}$$

$$\textbf{SAT-based revised uniqueness} \quad AMO(\{c_j, j = 1, ..., |M|\}) \tag{25}$$

$$\textbf{SAT-based completeness} \quad \bigwedge_i \bigvee_j b^+_{i,j} \vee b^-_{i,j} \tag{26}$$

$$\textbf{SAT-based revised coherence} \quad \bigwedge_j \bigwedge_i \left(\neg c_j \vee \neg b^+_{i,j}\right) \tag{27}$$

Definition 7 (*Indirect SAT Encoding of Subsumption Resolution*) The *indirect SAT encoding for subsumption resolution* $\mathcal{E}^i_{\text{SR}}(S, M)$ for clauses S and M is the conjunction of **positive compatibility** (15), **negative compatibility** (16), **structurality** (23), **revised existence** (24), **revised uniqueness** (25), **completeness** (26), and **revised coherence** (27).

With this new indirect encoding $\mathcal{E}^i_{\text{SR}}(S, M)$, the number of clauses is only quadratic with respect to the length of the clauses.

Theorem 7 (Soundness) *Assume M does not contain duplicate literals, as in* (4). *Clauses S and M are the side and main premise of subsumption resolution iff $\mathcal{E}^i_{\text{SR}}(S, M)$ is satisfiable.*

Proof From Theorem 5, if (15) $\wedge$ (16) $\wedge$ (26) is satisfiable, then $\forall i, j.\ b^+_{i,j} \Leftrightarrow \sigma(s_i) = m_j$ and $\forall i, j.\ b^-_{i,j} \Leftrightarrow \sigma(s_i) = \neg m_j$. Using (23), we obtain $\forall j.\ c_j \Leftrightarrow \exists i.\ \sigma(s_i) = \neg m_j$. Based on (24)-(27), we obtain the subsumption resolution constraints of Theorem 2. Therefore, if $\mathcal{E}^i_{\text{SR}}(S, M)$ is satisfiable, then subsumption resolution can be applied over (S, M).

For the other direction, assume subsumption resolution can be applied over (S, M). Then, we can build a model that satisfies $\mathcal{E}^i_{\text{SR}}(S, M)$, as follows. There exists a substitution σ, a literal $m' \in M$ and a set of literals $S' \subseteq S$ such that $\sigma(S') = \{\neg m'\} \wedge \sigma(S \backslash S') \subseteq M \backslash \{m'\}$. Without loss of generality, let $m_1 = m'$ be the resolution literal of SR. We set $c_1 = \top$ and all the other c_j to false. For each literal in $s_{i'} \in S'$, we set $b^-_{i',1} = \top$ and $b^-_{i',j} = \bot$, for $j \neq 1$; further, $b^+_{i',j} = \bot$, for all j. Let $S^* = S \setminus S'$. For each literal $s_{i^*} \in S^*$, there exists a literal $m_{j^*} \in M \setminus \{m_1\}$ such that $\sigma(s_{i^*}) = m_{j^*}$. We set $b^+_{i^*,j^*} = \top$; $b^+_{i^*,j} = \bot$, for $j \neq j^*$; and $b^-_{i^*,j} = \bot$, for all j. This is indeed a model of $\mathcal{E}^i_{\text{SR}}(S, M)$.

□

We note that, in practice, the number of clauses of the indirect SAT encoding can be greater than the direct SAT encoding, even for large clauses. Indeed, it is not necessary to define

variables for literal pairs that we know in advance cannot be matched. If $\Sigma^+_{i,j} = \tilde{\Sigma}$, we do not define $b^+_{i,j}$ because the constraint $b^+_{i,j} \Rightarrow \Sigma^+_{i,j} \subseteq \sigma$ will be reduced to $b^+_{i,j} \Rightarrow \bot$ and $b^+_{i,j}$ is always false. We do not need to add the clauses containing $\neg b^+_{i,j}$, and we remove the literals $b^+_{i,j}$ where it appears. In practice, most instances of subsumption and subsumption resolution have a sparse Boolean variable set, and behave quite well even with the direct SAT encoding. Choosing which encoding to use is discussed in Sect. 8.2.

Example 5 (Subsumption Resolution with $\mathcal{E}^i_{\text{SR}}(S, M)$) Consider clauses from Example 4. Namely, $S = s_1 \vee s_2 \vee s_3$ and $M = m_1 \vee m_2 \vee m_3$, with

$$\begin{array}{ll} s_1 = p(f(x_1), x_2) & m_1 = \neg p(f(c), d) \\ s_2 = \neg p(x_2, x_1) & m_2 = \neg p(d, c) \\ s_3 = p(f(x_3), x_1) & m_3 = p(f(y_1), c) \end{array}$$

In the indirect SAT encoding $\mathcal{E}^i_{\text{SR}}(S, M)$, we introduce two extra variables c_1 and c_3 such that c_1 is true iff $\exists i.\ b^-_{i,1}$, and c_3 is true iff $\exists i.\ b^-_{i,3}$. It is not necessary to define c_2 since no negative polarity matches exist towards m_2, and c_2 is set to false. The SAT constraints identical to the direct SAT encoding $\mathcal{E}^d_{\text{SR}}(S, M)$ are written below in light gray to better highlight the difference between $\mathcal{E}^d_{\text{SR}}(S, M)$ and $\mathcal{E}^i_{\text{SR}}(S, M)$.

$b^+_{1,3} \Rightarrow \{x_1 \mapsto y_1, x_2 \mapsto c\} \subseteq \sigma$	positive compatibility
$b^+_{2,1} \Rightarrow \{x_1 \mapsto d, x_2 \mapsto f(c)\} \subseteq \sigma$	positive compatibility
$b^+_{2,2} \Rightarrow \{x_1 \mapsto c, x_2 \mapsto d\} \subseteq \sigma$	positive compatibility
$b^+_{3,3} \Rightarrow \{x_1 \mapsto c, x_3 \mapsto y_1\} \subseteq \sigma$	positive compatibility
$b^-_{1,1} \Rightarrow \{x_1 \mapsto c, x_2 \mapsto d\} \subseteq \sigma$	negative compatibility
$b^-_{2,3} \Rightarrow \{x_1 \mapsto c, x_2 \mapsto f(y_1)\} \subseteq \sigma$	negative compatibility
$b^-_{3,1} \Rightarrow \{x_1 \mapsto d, x_3 \mapsto c\} \subseteq \sigma$	negative compatibility
$\neg c_1 \vee b^-_{1,1} \vee b^-_{3,1}$	**SAT-based structurality**
$c_1 \vee \neg b^-_{1,1}$	**SAT-based structurality**
$c_1 \vee \neg b^-_{3,1}$	**SAT-based structurality**
$\neg c_3 \vee b^-_{2,3}$	**SAT-based structurality**
$c_3 \vee \neg b^-_{2,3}$	**SAT-based structurality**
$c_1 \vee c_3$	**SAT-based revised existence**
$AMO(\{c_1, c_3\})$	**SAT-based revised uniqueness**
$b^-_{1,1} \vee b^+_{1,3}$	SAT-based completeness
$b^+_{2,1} \vee b^+_{2,2} \vee b^-_{2,3}$	SAT-based completeness
$b^-_{3,1} \vee b^+_{3,3}$	SAT-based completeness
$\neg c_1 \vee \neg b^+_{2,1}$	**SAT-based revised coherence**
$\neg c_3 \vee \neg b^+_{1,3}$	**SAT-based revised coherence**
$\neg c_3 \vee \neg b^+_{3,3}$	**SAT-based revised coherence**

Using the above indirect encoding $\mathcal{E}^i_{\text{SR}}(S, M)$, our SAT solver in Sect. 6 finds the same model (substitution) of subsumption resolution as in Example 4, with c_1, c_2 being assigned true and false respectively.

We remark that the indirect encoding $\mathcal{E}^i_{\text{SR}}(S, M)$ does not seem to have much of an advantage on small examples similar to Example 5. Indeed, structurality (23) adds a few clauses that are not necessary with the direct encoding $\mathcal{E}^d_{\text{SR}}(S, M)$. In Sect. 8.2 we empirically show that the indirect encoding $\mathcal{E}^i_{\text{SR}}(S, M)$ of subsumption resolution performs better on larger clauses.

6 SAT solving for subsumption variants

We now describe our approach for solving the SAT-based encodings $\mathcal{E}_{\text{S}}(S, M)$, $\mathcal{E}^d_{\text{SR}}(S, M)$ and $\mathcal{E}^i_{\text{SR}}(S, M)$ of Sect. 5 for subsumption and subsumption resolution. We first introduce our SAT solver adjusted for the efficient handling of subsumption (resolution) constraints, important for reasoning about substitution constraints $\Sigma^{\pm}_{i,j} \subseteq \sigma$ and at-most-one constraints (Sect. 6.1). We then describe pruning-based preprocessing steps of subsumption (resolution) instances (Sect. 6.2), with the purpose of improving SAT-based solving of subsumption and subsumption resolution.

Lightweight SAT Solving We use the term *lightweight SAT Solving* to highlight an important engineering aspect when designing a SAT solver for subsumption and subsumption resolution. A typical run of a first-order theorem prover involves a large number of simple subsumption (resolution) tests and a small number of hard tests. Even after pruning, most instances that make it to the SAT solver are solved quickly (see also Sect. 8.1 and Fig. 1). As a result, some care must be taken to ensure that setup of the SAT instances is efficient, because a large overhead may easily outweigh gains in solving efficiency.

6.1 SAT solver for subsumption encodings

Recall that the SAT-based encodings $\mathcal{E}_{\text{S}}(S, M)$, $\mathcal{E}^d_{\text{SR}}(S, M)$ and $\mathcal{E}^i_{\text{SR}}(S, M)$ of subsumption and subsumption resolution use substitution constraints $\Sigma^{\pm}_{i,j} \subseteq \sigma$ and at-most-one constraints (*AMO*), which are out of scope for standard SAT solvers [31, 32]. A naïve SAT approach of handling such constraints would be translating $\Sigma^{\pm}_{i,j} \subseteq \sigma$ and *AMO* formulas into purely propositional clauses. However, such a translation would either require additional propositional variables to encode *AMO* constraints or would come with a quadratic[2] number of propositional clauses [36]; a similar situation also occurs for substitution constraints $\Sigma^{\pm}_{i,j} \subseteq \sigma$. To ensure efficient solving of subsumption (resolution), solving our SAT encodings needs to be *lightweight* in order to be practically feasible during redundancy checking in a first-order theorem prover.

As a remedy to overcome the increase in propositional variables/clauses in a naïve SAT translation approach, we support substitution constraints as in (15) and (16), as well as *AMO* constraints as in SAT-based **multiplicity conservation** (18) and **revised uniqueness** (25), natively in SAT solving. In particular, we adjust unit propagation and conflict

[2] Quadratic in the size of the *AMO* constraint.

resolution in CDCL-based SAT solving for handling propositional formulas with substitution constraints and *AMO* constraints.

At-most-one constraints Consider the constraint $AMO(\{b_1, b_2, \dots, b_n\})$, which is equivalent to the following purely propositional formula:

$$\bigwedge_i \bigwedge_{j>i} \neg b_i \vee \neg b_j \tag{28}$$

To keep our encoding of *AMO* constraints lightweight, we combine SAT solving with *AMO* constraints in a way similar to SMT solving, as follows.

When the constraint $AMO(\{b_1, b_2, \dots, b_n\})$ is added to the SAT solver, each of the variables $b_1, b_2, \dots, b_n$ watches the constraint. Whenever one of the variables b_i is assigned true, all b_j with $j \neq i$ must be false in order not to violate $AMO(\{b_1, b_2, \dots, b_n\})$; hence b_j are propagated to false. The reasons of these propagations are exactly the clauses $\neg b_i \vee \neg b_j$ of (28); however, these clauses do not need to be explicitly constructed. Conflict analysis in SAT solving then behaves as usual, without special considerations for *AMO* constraints.

Compatibility constraints Similar to *AMO* constraints, a compatibility constraint is equivalent to a set of binary clauses, as given in (15)-(16). Let $\Sigma_1 \Cap \Sigma_2$ denote that the substitutions Σ_1 and Σ_2 are incompatible; based on Definition 4, there exists thus a variable x such that $\Sigma_1(x) \neq \Sigma_2(x)$. Let F be the set of constraints under consideration. The purely propositional semantics of the compatibility constraints (15)-(16) is the clause set:

$$\left\{\neg b \vee \neg b' \mid (b \Rightarrow \Sigma \subseteq \sigma) \in F \wedge (b' \Rightarrow \Sigma' \subseteq \sigma) \in F \wedge \Sigma \Cap \Sigma'\right\} \tag{29}$$

We remark that it is not necessary to generate the clauses (29) explicitly. Conceptually, our SAT solver updates a global substitution σ_τ whenever a Boolean variable b with associated substitution constraint $\Sigma \subseteq \sigma$ is assigned true. Our SAT solver then ensures that the following invariant holds:

$$\sigma_\tau = \bigcup \left\{\Sigma \mid b \Rightarrow (\Sigma \subseteq \sigma) \in F \wedge b \in \tau\right\}, \tag{30}$$

where τ is the current set of assigned literals of the SAT solver (i.e., the trail). Our SAT solver uses σ_τ to propagate any Boolean variables bound to incompatible substitutions to false.

We note that, in practice, it is not necessary to keep σ_τ explicitly; instead it suffices to maintain a lookup table that allows propagating such incompatible substitutions. Concretely, each first-order variable x watches the set $Bindings(x)$ of Boolean variables b that impose a binding on x along with the bound term t:

$$Bindings(x) = \{(b, t) \mid b \rightarrow (\Sigma \subseteq \sigma) \in F \wedge t = \Sigma(x)\}$$

When the global substitution σ_τ is updated with a variable x newly mapped to a term t, our SAT solver uses $Bindings(x)$ to retrieve all the Boolean variables b' with an associated substitution constraint $\Sigma' \subseteq \sigma$ such that $\Sigma'(x) \neq t$. The solver then propagates b' to false, and the propagation reason is the binary clause $\neg b \vee \neg b'$, where b is the Boolean variable that caused $\sigma_\tau(x) = t$.

As a result, our SAT solver ensures that $\Sigma' \Cap \sigma$ implies $\neg b'$. We perform this propagation of incompatible substitution constraints immediately when a Boolean variable is assigned true. This way, we enforce the invariant (30) and guarantee there can be no conflict due to substitution constraints. Indeed, if $b \Rightarrow \Sigma \in \sigma$ and $\Sigma \Cap \sigma_\tau$, then b would have been assigned false before.

Example 6 (SAT Solving of the SAT-Based Direct Encoding of Subsumption Resolution) We illustrate the main steps of our SAT solver using the direct encoding $\mathcal{E}^d_{\text{SR}}(S, M)$ of subsumption resolution from Example 4.

A potential execution of our SAT solver on $\mathcal{E}^d_{\text{SR}}(S, M)$ decides $b^+_{1,3} = \top$. This imposes, among others, the mapping $x_1 \mapsto y_1$, and due to the **compatibility** constraints all other Boolean variables are immediately propagated to false. This leads to conflicts with the **existence** and some **completeness** constraints. Assume the solver discovers the conflict with $b^-_{3,1} \vee b^+_{3,3}$. As explained above, the reasons for propagating these literals are the implicit binary clauses $\neg b^+_{1,3} \vee \neg b^-_{3,1}$ and $\neg b^+_{1,3} \vee \neg b^+_{3,3}$, and after resolution, the solver will backtrack, learn the asserting clause $\neg b^+_{1,3}$, and propagate $b^+_{1,3} = \bot$. With **completeness** $b^-_{1,1} \vee b^+_{1,3}$, the solver propagates $b^-_{1,1} = \top$, which imposes the mappings $x_1 \mapsto c$ and $x_2 \mapsto d$ on σ_τ. By **compatibility**, the solver now propagates $b^+_{2,1} = \bot$, $b^-_{2,3} = \bot$, and $b^-_{3,1} = \bot$. With the remaining **completeness** constraints, the solver now propagates $b^+_{2,2} = \top$ and $b^+_{3,3} = \top$. At this point, all Boolean variables are assigned and all constraints are satisfied, yielding the model $\{\neg b^+_{1,3}, \neg b^+_{2,1}, b^+_{2,2}, b^+_{3,3}, b^-_{1,1}, \neg b^-_{2,3}, \neg b^-_{3,1}\}$ of $\mathcal{E}^d_{\text{SR}}(S, M)$ from Example 4.

6.2 Pruning subsumption variants for SAT solving

Reducing the number of (trivially unsat) instances of subsumption and subsumption resolution is an important preprocessing step for increasing the effectiveness of our SAT solving engine from Sect. 6.1.

Pruning subsumption We prune unsat subsumption instances between (S, M) by checking multiset inclusion between the predicateof atoms of S, M, together with their polarities. Intuitively, this pruning step allows to easily determine that there exists no bijective function $j(i)$ such that $\sigma(s_i) = m_{j(i)}$ if the atom cardinalities do not match.

More formally, let $\mathcal{P}(\ell)$ compute the predicate corresponding to literal ℓ and $\mathcal{Q}(\ell)$ denote the polarity of ℓ. Our pruning criterion for subsumption is:

$$\left\{\left(\mathcal{P}(s_i), \mathcal{Q}(s_i)\right) \mid s_i \in S\right\} \sqsubseteq \left\{\left(\mathcal{P}(m_j), \mathcal{Q}(m_j)\right) \mid m_j \in M\right\} \tag{31}$$

Theorem 8 (Pruning Subsumption) *If the pruning criterion* (31) *is unsat, then S does not subsume M.*

Proof The multisets $\{(\mathcal{P}(s_i), \mathcal{Q}(s_i)) \mid s_i \in S\}$ and $\{(\mathcal{P}(m_j), \mathcal{Q}(m_j)) \mid m_j \in M\}$ are projections π of the multisets of literals of S and M respectively. This projection π has the property to make its argument substitution agnostic. That is, if there exists σ such that $\sigma(s_i) = m_j$, then s_i and m_j are projected on the same location; that is, $(\sigma(s_i) = m_j) \Rightarrow (\pi(s_i) = \pi(m_j))$. Therefore, if $\pi(s_i) \neq \pi(m_j)$, then there exist no matching substitution between s_i and m_j. If formula (31) is unsat, then $\pi(S) \sqsubseteq \pi(M)$, implying that there exists no substitution σ such that $\sigma(S) \sqsubseteq M$; as such, subsumption cannot be applied between (S, M). □

Pruning subsumption resolution We similarly prune unsat instances of subsumption resolution, by using a weaker version of (31). Namely, for pruning unsat subsumption resolution instances, we only check set inclusion between the predicate sets of S and M:

$$\left\{\mathcal{P}(s_i) \mid s_i \in S\right\} \subseteq \left\{\mathcal{P}(m_j) \mid m_j \in M\right\} \tag{32}$$

Theorem 9 *Validity of the subsumption pruning criterion* (31) *implies validity of the subsumption resolution pruning criterion* (32).

Proof The sets $\{\mathcal{P}(s_i) \mid s_i \in S\}$ and $\{\mathcal{P}(m_j) \mid m_j \in M\}$ are obtained by a projection π from $\{(\mathcal{P}(s_i), \mathcal{Q}(s_i)) \mid s_i \in S\}$ and $\{(\mathcal{P}(m_j), \mathcal{Q}(m_j)) \mid m_j \in M\}$, respectively. Therefore, for each pair of elements $(e, e') \in \{(\mathcal{P}(s_i), \mathcal{Q}(s_i)) \mid s_i \in S\} \times \{(\mathcal{P}(m_j), \mathcal{Q}(m_j)) \mid m_j \in M\}$, if $e = e'$, then $\pi(e) = \pi(e')$, and the multiset inclusion is preserved.

As $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$ (multiset inclusion) implies $\mathcal{S}_1 \subseteq \mathcal{S}_2$ (set inclusion), we obtain that (31) implies (32). □

The following is an immediate consequence of Theorems 8-9.

Corollary 3 *If the pruning criterion* (32) *is not satisfied, then S does not subsume M.*

Similarly to Theorem 8, we use the pruning criterion (32) to detect (and delete) unsat subsumption resolution instances between (S, M).

Theorem 10 (Pruning Subsumption Resolution) *If the pruning criterion* (32) *is unsat, then S and M are not side and main premises of subsumption resolution.*

Proof Similarly to Theorem 8, if criterion (32) is not satisfied, then there exists a literal $s_i \in S$ that cannot be matched with any literal in M; as such, the **completeness** constraint (9) of subsumption resolution is violated. □

Fast implementations of pruning To represent the predicate sets used in our pruning criterion, we use an array $\mathcal{A}$ of unsigned integers whose index is the index of the predicate. We first build the multiset with the predicates of the main premise M. When a predicate is hit, the value stored in $\mathcal{A}$ is incremented, we check that S contains a sub-multiset of predicates, and decrement the previously stored value within A.

Storing $\mathcal{A}$ entries while applying pruning checks may be memory-expensive. Resetting the memory before each pruning is also an expensive operation. We therefore use a time stamp t such that $\forall i.\ \mathcal{A}[i] < t + |M|$ holds. Intuitively, before pruning is applied, $\forall i.\ \mathcal{A}[i] < t$ holds. Algorithm 1 summarizes our pruning procedure using time stamps.

Algorithm 1 Pruning algorithm for subsumption and subsumption resolution

```
N ← number of predicate symbols
t ← 0                                              ▷ time stamp
𝒜 ← zeros(2 · N)                                   ▷ array to hold the (multi-)set
function HEADERINDEX(l)
    if 𝒬(l) then                                   ▷ Positive polarity atom
        return 𝒫(l)
    return 𝒫(l) + N                                ▷ Negative polarity atom
procedure PRUNESUBSUMPTION(S, M)
    if t + |M| > UINT_MAX then                     ▷ Reset before arithmetic overflow
        𝒜 ← zeros(2 · N)
        t ← 0
    for m ∈ M do
        idx ← HEADERINDEX(m)
        𝒜[idx] ← max(t, 𝒜[idx]) + 1
    for s ∈ S do
        idx ← HEADERINDEX(s)
        if 𝒜[idx] ≤ t then
            t ← t + |M|
            return ⊤
        𝒜[idx] ← 𝒜[idx] − 1
    t ← t + |M|
    return ⊥
procedure PRUNESUBSUMPTIONRESOLUTION(S, M)
    if t + 1 > UINT_MAX then                       ▷ Reset before arithmetic overflow
        𝒜 ← zeros(2 · N)
        t ← 0
    t ← t + 1
    for m ∈ M do
        𝒜[𝒫(m)] ← t
    for s ∈ S do
        if 𝒜[𝒫(s)] ≠ t then
            return ⊤
    return ⊥
```

Pruning after building match sets While our pruning criteria (31)-(32) are fast to compute, they do not reason about substitutions needed for subsumption (resolution). However, while building the match sets $\Pi(S, M)$, we may also detect unsat instances of subsumption and subsumption resolutions. For example, let $s_i = p(f(x))$ be a literal of S. If M does not contain any literal of the form $p(f(\cdot))$, there is no unifying substitution between (S, M). The non-existence of such substitutions would not necessarily be detected by (31)-(32), but could be recorded while building the substitution sets.

We therefore use the following additional pruning criteria for subsumption:

$$\forall i \exists j.\ \Sigma^{+}_{i,j} \neq \tilde{\Sigma} \tag{33}$$

Theorem 11 [*Substitution Sets for Pruning Subsumption*] *Let* $\Pi(S,M) = \left\{ \left(b^{\pm}_{i,j}, \Sigma^{\pm}_{i,j} \right) \right\}$ *be the match set of* S *and* M*. If* (33) *is unsat, then* S *does not subsume* M*.*

Proof Theorem 4 implies that, if S subsumes M, then $\forall i \exists j.\ b^{+}_{i,j}$ and $\forall i,j.\ b^{+}_{i,j} \Rightarrow \Sigma^{+}_{i,j} \subseteq \sigma$. Hence, $\forall i \exists j.\ \Sigma^{+}_{i,j} \subseteq \sigma$, which is equivalent to (33) since $\tilde{\Sigma} \in \sigma \Rightarrow \bot$. Therefore, if S subsumes M, then (33) is valid. □

A pruning criterion similar to (33) can be applied to subsumption resolution:

$$\forall i \exists j.\ \Sigma^{+}_{i,j} \neq \tilde{\Sigma} \vee \Sigma^{-}_{i,j} \neq \tilde{\Sigma} \tag{34}$$

Theorem 12 [*Substitution Sets for Pruning Subsumption Resolution*] *Let* $\Pi(S,M) = \left\{ \left(b^{\pm}_{i,j}, \Sigma^{\pm}_{i,j} \right) \right\}$ *be the match set of* S, M*. If* (34) *is unsat, then* S *and* M *are not side and main premises of subsumption resolution.*

Proof Similarly to the proof of Theorem 11, the **compatibility** and **completeness** constraints of $\mathcal{E}^{d}_{\mathrm{SR}}(S,M)$ imply (34). Based on Theorem 6, if S, M are side and main premises of subsumption resolution, then (34) is valid. □

We remark that the pruning criterion (32) is a special case of Theorem 12. Therefore, if (32) is unsat, then (33) is also unsat and no subsumption resolution is possible. Furthermore, if there are no negative polarity substitutions, then the **existence** constraint of $\mathcal{E}^{d}_{\mathrm{SR}}(S,M)$ does not hold. As such, a further pruning criterion for (unsat) subsumption resolution instances is:

$$\exists i,j.\ \Sigma^{-}_{i,j} \neq \tilde{\Sigma} \tag{35}$$

Theorem 13 [*Polarities for Pruning Subsumption Resolution*] *Let* $\Pi(S,M) = \left\{ \left(b^{\pm}_{i,j}, \Sigma^{\pm}_{i,j} \right) \right\}$ *be the match set of* S, M*. If* (35) *is unsat, then* S *and* M *are not premises of subsumption resolution.*

Proof Based on Theorem 4, if $\mathcal{E}^{d}_{\mathrm{SR}}(S,M)$ is satisfiable, then the **existence** property $\exists i,j.\ b^{-}_{i,j}$ is satisfiable and the **compatibility** constraint is satisfied by the same assignment. Therefore, if $\mathcal{E}^{d}_{\mathrm{SR}}(S,M)$ is satisfiable, then $\exists i,j.\ \Sigma^{-}_{i,j} \neq \tilde{\Sigma}$. □

Finally, if there exist two literals in S such that they do not have positive matches to literals in M and the respective predicates of the literals are different, then subsumption resolution is not possible. This yields our final pruning criterion:

$$\forall i,i'.\ (i \neq i') \Rightarrow (\mathcal{P}(s_i) = \mathcal{P}(s_{i'}) \vee \exists j\ \Sigma^{+}_{i,j} \neq \tilde{\Sigma} \vee \exists j\ \Sigma^{+}_{i',j} \neq \tilde{\Sigma}) \tag{36}$$

Theorem 14 (Predicate Matches for Pruning Subsumption Resolution) *Let* $\Pi(S, M) = \left\{ \left(b^{\pm}_{i,j}, \Sigma^{\pm}_{i,j} \right) \right\}$ *be the match set of* S, M*. If* (36) *is unsat, then* S *and* M *are not side and main premises of subsumption resolution.*

Proof By contradiction, assume that subsumption resolution could be applied to (S, M). Then, there exists a unique m' such that $\sigma(S') = \{m'\}$. However, if (36) is unsat, there exist two different literals in S that can only be mapped negatively to m' (or not at all). These literals have a different predicate, therefore they cannot be both matched to the same literals $(\forall \sigma, l, l' .\ \mathcal{P}(l) \neq \mathcal{P}(l') \Rightarrow \sigma(l) \neq \sigma(l'))$. If one of these literals cannot be matched to the resolution literal of SR, and has no positive match, then it cannot be matched to any literal in M; hence and subsumption resolution cannot be applied. □

Remark 1 It is easy to see that Algorithm 1 is a very cheap procedure. During our experiments (Sect. 9, we observed that more than 95% of instances of subsumption are filtered out by the pruning criterion (31) alone, and more than 50% are also pruned by (32). When it comes to subsumption resolution, in our experiments 90% of subsumption resolution instances are pruned by (32). The more restrictive nature of (34) and (35) prunes an additional 5 % of subsumption resolution instances. As a result, our experiments show that pruning is indeed an important and cheap preprocessing step. Thanks to pruning, in our experiments only 5% of subsumption (resolution) instances need to use more expensive SAT-based computation steps, using our SAT solver from Sect. 6.1.

7 SAT-based subsumption variants in saturation

In this section, we discuss the direct integration of the SAT solving engine of Sect. 6 within the saturation loop of first-order theorem proving. Such an integration greatly improves redundancy checking in theorem proving, without making significant changes to the underlining saturation algorithms of the prover.

Algorithm 2 Forward simplification with SAT-based subsumption resolution

```
procedure ForwardSimplify(M, F)
    M* ← NoSubsumptionResolution
    for S ∈ F \ {M} do                          ▷ Get candidates from generalisation index.
        if subsumption(S, M) is Subsumed then   ▷ using Algorithm 3
            F ← F \ {M}
            return ⊤                            ▷ M is subsumed and removed
        if M* = NoSubsumptionResolution then
            M* ← subsumptionResolution(S, M)    ▷ using Algorithm 4
    if M* ≠ NoSubsumptionResolution then
        F ← F \ {M} ∪ {M*}                      ▷ M* is the conclusion of subsumption resolution between S and M
        return ⊤
    return ⊥
```

To design a saturation algorithm, one important aspect is to understand how to organise redundancy elimination during proof search. One common design principle in this respect comes with so-called *given clause algorithms* [37], where inference selection is implemented using clause selection. At each iteration of the algorithm, a clause from the proof search is selected and inferences are performed between this clause and previously selected clauses. When a new clause is generated, this clause should only be kept if it is not redundant or it cannot be simplified by another existing clause; we refer to such redundancy checks over a new clause as *forward* redundancy, implementing *forward simplification*. On the other hand, a newly generated clause could make existing clauses in the search space redundant; we call such redundancy checks with a new clause as *backward* redundancy, implementing *backward simplification*.

Using the SAT solver of Sect. 6 for detecting subsumption (resolution) in saturation needs therefore to (i) address both forward and backward variants of subsumption and subsumption resolution, and (ii) organize proof search with these subsumption variants solved via SAT. In the rest of this section, we mainly focus on forward simplification via subsumption and subsumption resolution, and briefly discuss differences with respect to backward simplification.

Forward simplification Intuitively, as subsumption is a stronger inference than subsumption resolution, subsumption should be performed first. As such, a standard forward simplification loop for subsumption (resolution) would be:

1. From a selected clause M, search some subsumption candidate clauses $\{S_k \mid k = 1, \dots\}$ using a generalisation term index [24];
2. For each clause in $\{S_k \mid k = 1, \dots\}$, check if S_k subsumes M. If this is the case, then stop and remove M from the clause set.
3. For each clause in $\{S_k \mid k = 1, \dots\}$, check if S_k can delete a literal from M using subsumption resolution. If it is the case, then replace M by the conclusion of subsumption resolution `SR` and stop.

Algorithm 3 SAT-based subsumption in saturation

$f_{\mathtt{S}} \leftarrow \bot$ ▷ If $f_{\mathtt{S}}$ gets true then subsumption is guaranteed to fail

$f_{\mathtt{SR}} \leftarrow \bot$ ▷ If $f_{\mathtt{SR}}$ gets true then subsumption resolution is guaranteed to fail

procedure SUBSUMPTION(S, M)

  $f_{\mathtt{SR}} \leftarrow$ PRUNESUBSUMPTIONRESOLUTION(S, M)

  $f_{\mathtt{S}} \leftarrow f_{\mathtt{SR}} \vee$ PRUNESUBSUMPTION(S, M)

  ▷ Corollary 3 ensures that $f_{\mathtt{SR}} \Rightarrow f_{\mathtt{S}}$.

  if $f_{\mathtt{SR}}$ **then**

    return `NoSubsumption`

  ▷ If only subsumption fails, we still need to fill the match set.

  $\Pi \leftarrow \Pi(S, M)$

  $f_{\mathtt{S}} \leftarrow f_{\mathtt{S}} \vee \neg(33)$ ▷ Computed when filling Π

  $f_{\mathtt{SR}} \leftarrow f_{\mathtt{SR}} \vee \neg(34) \vee \neg(35) \vee \neg(36)$ ▷ Also computed when filling Π

  if $f_{\mathtt{S}}$ **then**

    return `NoSubsumption`

  $\mathcal{E}_{\mathtt{S}} \leftarrow$ ENCODECONSTRAINTS(Π)

  if $\mathcal{E}_{\mathtt{S}} \models \bot$ **then** ▷ SAT solver returns unsatisfiable

    return `NoSubsumption`

  else

    return `Subsumed`

In this approach, finding the substitutions of subsumption (resolution) comes with a significant computation burden. Further, as subsumption checks do not succeed most of the time, the match sets $\Pi(S_k, M)$ must be cached or recomputed. Therefore, when integrating our SAT-based solving of subsumption (resolution) in saturation using Algorithm 2, we use pruning-based preprocessing and build match sets before checking subsumption and subsumption resolution. Our Algorithm 2 yields thus a new, SAT-based forward simplification loop for subsumption (resolution) in saturation. Algorithm 2 uses Algorithm 3 to possibly prune both subsumption and subsumption resolution and then set up a complete match set. Even though subsumption alone does not require the negative polarity substitutions, these substitutions are computed for subsumption resolution. Then, Algorithm 4 benefits from the work done by subsumption, since it only requires to create the propositional clause set.

Remark 2 In Algorithm 2, when a subsumption resolution check was successful, no other is performed, but the algorithm still searches for a subsumption. In this case, only a partial match set is necessary and subsumption will not fill negative polarity matches.

The index used to provide candidate clauses returns clauses on a literal by literal manner. That is, for each literal $m \in M$, the index returns clauses that have at least one literal that is a generalisation of m. However, for subsumption resolution, we also get clauses with a generalisation of complemented literals $\neg m_j$. In this case, we do not need to check for subsumption, and only subsumption resolution is performed. Yet, subsumption resolution still sets up the match set.

Algorithm 4 SAT-based subsumption resolution in saturation—with subsumption already set up via Algorithm 3

```
procedure SubsumptionResolution(S, M)
                                        ▷ upon Algorithm 3 failing to subsume
                                        ▷ the match set Π is already set up
  if f_SR then
    return NoSubsumptionResolution
  enc ← chooseEncoding(Π, S, M)         ▷ choose the best encoding (see Sect 8)
  E_SR ← encodeConstraints(enc, Π)
  if ∃τ. τ ⊨ E_SR then                  ▷ τ is a model of E_SR found by the solver
    return buildConclusion(τ, M)        ▷ conclusion of subsumption resolution
  return NoSubsumptionResolution
```

Backward simplification Backward simplifications use newly generated clauses S to simplify the current clause set F. Given a newly generated clause S, backward subsumption (resolution) thus checks whether S subsumes some clauses $M \in F$ (or can remove a literal from M). In this case, performing subsumption resolution right after subsumption is almost free. Indeed, since backward simplifications do not stop after simplifying one clause, the only cost of performing subsumption resolution right after subsumption is to setup the full match set, rather than simply setting up the positive polarity matches.

Extensions of subsumption variants in saturation Our SAT-based approach for solving subsumption (resolution) in saturation is very flexible. Indeed, the SAT solver can handle

different types of matches to the same literal pair, yielding further extensions of the standard subsumption and subsumption resolution framework.

In the case of *symmetric predicates*, such as equality, two different substitutions are possible. Consider the literals $s_i := x = y$ and $m_j := c = f(c)$. To match these two literals, one can either use the substitution $\{x \mapsto c, y \mapsto f(c)\}$ or $\{x \mapsto f(c), y \mapsto c\}$. In this case, both substitutions would be added to the match set $\Pi(s_i, m_j)$ of s_i, m_j. That is, the matches $(\{x \mapsto c, y \mapsto f(c)\}, +, b^+_{i,j})$ and $(\{x \mapsto f(c), y \mapsto c\}, +, b'^+_{i,j})$ are added to $\Pi(s_i, m_j)$. In our implementation of the match set, it provides a list of matches $(b^\pm_{i,j}, \Sigma^\pm_{i,j})$ with either i or j fixed. When enumerating over this list to build the clauses, we ignore the second index. If several variables have the same index (i, j), the system will not be broken. Therefore, even when adding more than one match to the same literal pair, the SAT encoding remains the same. In addition, both substitutions are distinct, since otherwise one of the literals of s_i or m_j is a tautology and the respective clause would be removed. Handling of symmetric predicates brings great practical improvements, see Remark 3.

In the case of subsumption resolution, one may use the *most general unifier* on the resolution literal m', if the variable set of m' is disjoint from the variables in $M \backslash \{m'\}$. However, within the splitting approach of the AVATAR framework [38] of first-order theorem proving, the prover would split upon the main premise M; hence, using most general unifiers on the literal m' of M would not be triggered.

8 Solving heuristics for subsumption variants

Section 7 introduced efficient algorithms for integrating SAT-based subsumption reasoning in saturation. In this section, we further improve our methods from Sect. 7 by identifying and fine-tuning the key parameters of our SAT-based subsumption algorithm in saturation. Doing so, we (i) impose a solving timeout on particularly difficult subsumption and subsumption resolution instances (Sect. 8.1), and (ii) devise a framework for choosing the best SAT encodings for subsumption resolution (Sect. 8.2).

8.1 Cutting off the SAT search

We present how to fine-tune a timeout strategy for our SAT solver from Sect. 6, in order to prevent getting it stuck on unnecessary/difficult subsumption instances, while solving still as many positive instances as possible.

8.1.1 Measuring SAT solver progress

In general, the solver behaviour should be as deterministic as possible to ensure results are consistent and reproducible. Elapsed wall-clock time depends on many factors such as the type of machine and current load, and elapsed CPU time and number of CPU instructions easily change when refactoring code. As such, these measures are unsuitable when a deterministic solver behavior, and respective progress measure, is expected.

For evaluating our SAT solving approach in saturation, we therefore follow the Kissat methodology [39]: we count the number of elapsed *ticks*, which is a rough approximation of the number of memory cache lines accessed during unit propagation and conflict analysis.

8.1.2 Empirical observations

In our experiments (see Sect. 9), we evaluated our approach using the TPTP problem library [40]. Here, we logged the number of ticks the SAT solver performs on each problem and whether its search was successful. Figure 1a shows how the success rate of subsumption drops close to zero when our SAT solver runs longer. This effect is even more noticeable with subsumption resolution, as can be seen on Fig. 1b. We note that the performance jumps of Fig. 1a, b when crossing 10^k ticks are due to the non-linear scale used when aggregating data. We keep two significant digits to reduce the size of the files. Therefore, when jumping from $9.9 \cdot 10^{k-1}$ to $1.0 \cdot 10^k$, the size of the interval is multiplied by 10, hence a greater number of instances are gathered, and the line is discontinuous.

For improved solving progress, we aim to estimate a good trade-off between losing solutions by stopping the search early and the number of ticks saved. To do so, (i) we compute the number of ticks that the SAT solver has performed on instances that would be timed out; (ii) subtract the number of ticks ran before the timeout; and (iii) divide the result by the total number of ticks. Figure 2 shows that, when using a cutoff of 150, less than 1% of the successful instances are lost, while around 50% of ticks are saved for subsumption and 35% for subsumption resolution. Interestingly, when using a cutoff of 5000, we loose less than 0.01% of problems while still saving 10% of ticks.

8.2 Choosing SAT encodings for subsumption resolution

Section 5 introduced two different encodings for subsumption resolution over (S, M). The direct encoding $\mathcal{E}^d_{\text{SR}}(S, M)$ has $O(|S|^2 \cdot |M|^2)$ clauses and $O(|S| \cdot |M|)$ variables, while the indirect encoding $\mathcal{E}^i_{\text{SR}}(S, M)$ contains $O(|S| \cdot |M|)$ clauses with $O(|S| \cdot (|M| + 1))$ variables. Intuitively, the direct encoding $\mathcal{E}^d_{\text{SR}}(S, M)$ should to be more light weight and faster for smaller instances of subsumption resolution, whereas the indirect encoding $\mathcal{E}^i_{\text{SR}}(S, M)$ should scale better on harder instances. In this section, we present a procedure to choose which encoding to use for a given instance of subsumption resolution.

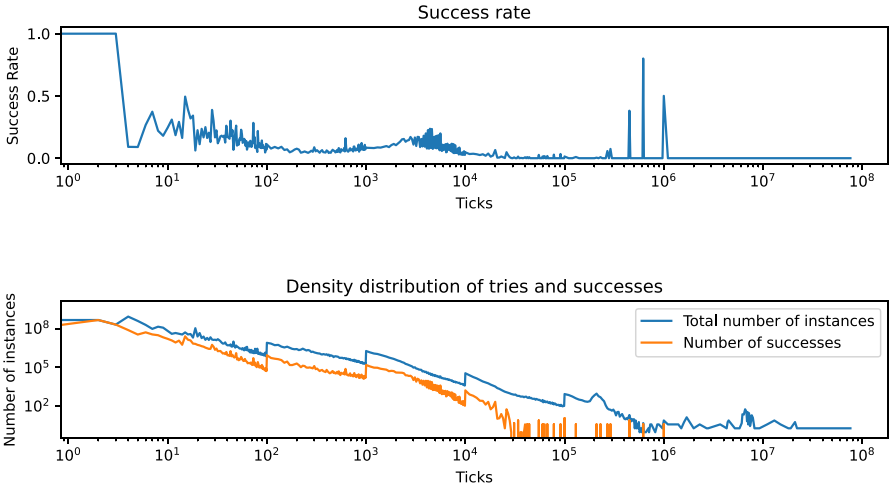
8.2.1 Problem setup

We focus on the problem of **choosing SAT encodings of subsumption resolution**. We approximate this problem via a random distribution $\mathcal{D}(y|x)$, where

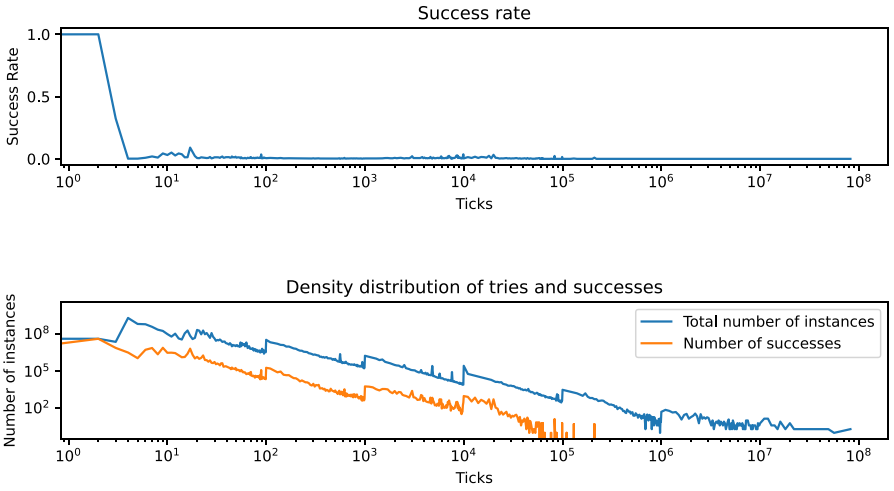
- input x, drawn from another distribution $\mathcal{X}$, is a vector of features $x_1, \ldots, x_n$;
- output y is a pair of values (y_0, y_1), where y_0 is the encoding and SAT solving time of the direct encoding $\mathcal{E}^d_{\text{SR}}(S, M)$ and y_1 is the encoding and SAT solving time of the indirect encoding $\mathcal{E}^i_{\text{SR}}(S, M)$.

Objective function Let a function family $\mathcal{F}$ be a set of functions $f : \mathbb{R}^n \to \{0, 1\}$. We define our objective function over $\mathcal{D}(y|x)$ and $\mathcal{X}$ as follows:

$$\arg\min_{f \in \mathcal{F}} \mathbb{E}_{\substack{x \sim \mathcal{X} \\ (y_0, y_1) \sim \mathcal{D}(\cdot|x)}} [y_{f(x)}] \tag{37}$$



(a) Solving subsumption instances.



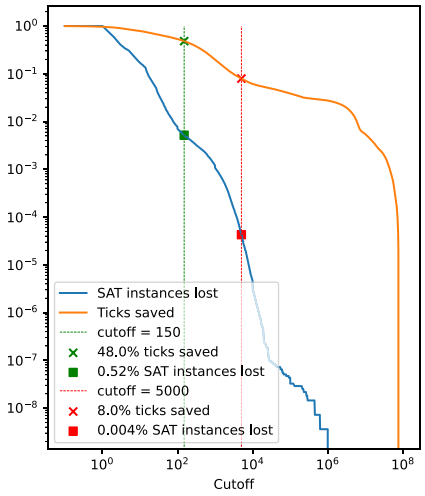
(b) Solving subsumption resolution instances using a direct SAT encoding

Fig. 1 Success rates of the SAT solver depending on the number of ticks (ticks are displayed on the horizontal axes). The problems taking the longest time are less likely to succeed

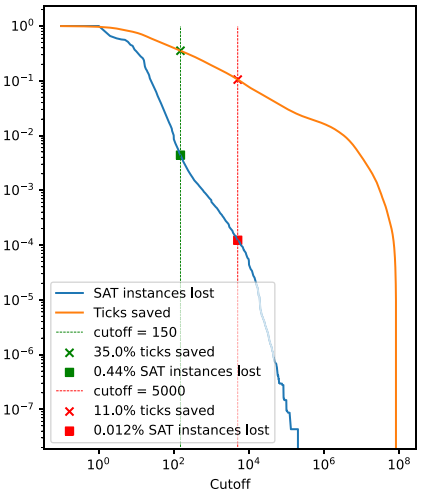
Intuitively, our objective function (37) computes a classifier f whose choice, given a set of features, minimises the expected run time of the respective SAT encoding and solving of subsumption resolution.

Features For any classification problem, identifying relevant features is important. We chose the following features for our classifier f computed by (37):

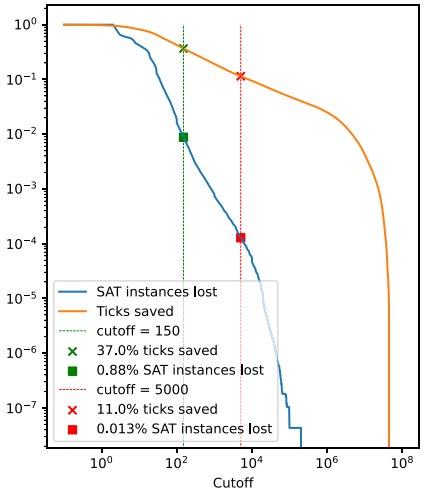
1. the number n of literals of the main premise M;



(a) Subsumption



(b) SR direct encoding



(c) SR indirect encoding

Fig. 2 Trade-off between positive instances lost when cutting off, and the number of ticks saved

2. the number k of literals of the side premise S;
3. the "sparsity" of the match set $\Pi(S, M)$, computed as: $\frac{|\Pi|}{k \cdot n}$, where $|\Pi|$ denotes the size of the match set $\Pi(S, M)$.

The relevance of the respective lengths k, n of the premises S, M is fairly self-explanatory, as the numbers of clauses of both SAT encodings grow differently with the number of literals of S, M. The sparsity of the match set $\Pi(S, M)$ is a measure of how many matches are found between literals of the main and side premises M, S. Sparsity of the match set is a

good indicator of the difficulty of the subsumption resolution problem. Indeed, if the match set $\Pi(S, M)$ is very sparse, then the subsumption resolution problem is easy: there are few matches to consider and the purely propositional clauses are already very constrained. On the other hand, if the match set $\Pi(S, M)$ is dense, then the subsumption resolution problem is hard.

Remark 3 The sparsity of the match set may be greater than 1. Indeed, in practice, we perform matching modulo the symmetry of equality (see Sect. 7). In such cases, one could use more than one match for a given literal pair.

8.2.2 Model architecture

The problem described in Sect. 8.2.1 is formalized as a classification problem in (37). Indeed, given a set of features x, we classify our problems sample into one of two classes: using the direct encoding $\mathcal{E}^d_{\text{SR}}(S, M)$ (class value 0) or using the indirect encoding $\mathcal{E}^i_{\text{SR}}(S, M)$ (class value 1). For solving the problem of Sect. 8.2.1, we select the SAT encoding of subsumption resolution over (S, M) that is likely to be solved the fastest way. Our classification procedure should thus be fast to compute at runtime. We therefore use a decision tree as our classifier, where our decision tree is a set of `if then else` expressions.

We used the `scikit-learn` [41] library to train our decision tree.

8.2.3 Building the dataset

Sampling We construct the set of samples (x, y) from the distributions $\mathcal{X}$, $\mathcal{D}_0$ and $\mathcal{D}_1$, respectively corresponding to the subsumption resolution input (S, M) drawn from the distribution $\mathcal{X}$ and their direct $\mathcal{E}^d_{\text{SR}}(S, M)$ and indirect encodings $\mathcal{E}^i_{\text{SR}}(S, M)$ modeling the distribution $\mathcal{D}_0(y_0 \mid x)$ and $\mathcal{D}_1(y_1 \mid x)$ respectively. To do so, we recorded the saturation running time of any subsumption resolution inference that reaches the SAT solving procedure. Indeed, if the subsumption instance is pruned, both encoding will behave exactly the same and the sample is irrelevant. We also recorded the features of the subsumption resolution check, that is, the length of the premises, and the sparsity. Each problem is run twice, once with the direct encoding $\mathcal{E}^d_{\text{SR}}(S, M)$, and another with the indirect encoding $\mathcal{E}^i_{\text{SR}}(S, M)$. As a result, we obtain two sets of samples samples (x, y), one for each encoding of subsumption resolution. We pair these samples to form (x, y_0, y_1).

Condensing the dataset Decision trees cannot be trained online, nor with mini-batches. Traditionally, when facing a large dataset, the classical method is to segment it into small batches, and train the model on randomly sampled batches [42]. However, this approach is not supported within the decision trees of the `scikit-learn` library. We therefore build a new dataset by summing the run times of all the samples that have the same features. That is, we build a new dataset $\mathcal{S}$ of $(x, \hat{y}_0, \hat{y}_1)$ samples, where x describes the feature and $\hat{y}_0$ and $\hat{y}_1$ are the respective sums of the run times of all the samples that have the same features x.

Modified objective function We adjust our objective function to our new dataset $\mathcal{S}$, as follows:

$$\arg\min_{f \in \mathcal{F}} \sum_{(x,\hat{y}_0,\hat{y}_1)\in\mathcal{S}} \left[|\hat{y}_0 - \hat{y}_1| * \left(f(x) - H(\hat{y}_0 - \hat{y}_1)\right)^2 \right] \tag{38}$$

where H is the step function, i.e., $H(a) = 1$ if $a \geq 0$, and $H(a) = 0$ otherwise.

The optimisation problem (38) is an empirical version of (37). Intuitively, (38) introduces more weight to samples with a large difference of efficiency between both SAT encodings $|\hat{y}_0 - \hat{y}_1|$. A choice of a SAT encoding of subsumption resolution is considered "wrong" if (i) $f(x)$ predicted 0 and the indirect encoding $\mathcal{E}^i_{\mathtt{SR}}(S, M)$ is faster than the direct encoding $\mathcal{E}^d_{\mathtt{SR}}(S, M)$; or (ii) $f(x)$ predicted 1 and the indirect encoding $\mathcal{E}^i_{\mathtt{SR}}(S, M)$ is slower than the direct encoding $\mathcal{E}^d_{\mathtt{SR}}(S, M)$. That is, $f(x) - H(\hat{y}_0 - \hat{y}_1)$ is 1 or -1 on wrong choices of SAT encodings, and 0 on correct choices.

Evaluating the model We introduce a metric called the *advantage* of the model over a function to evaluate the performance of our classifier f from (38). We introduce three baseline classifiers to compare our model to:

1. the *direct encoding* $d(x) = 0$ always chooses the direct encoding $\mathcal{E}^d_{\mathtt{SR}}(S, M)$ for sample x;
2. the *indirect encoding* $i(x) = 1$ always chooses the indirect encoding $\mathcal{E}^i_{\mathtt{SR}}(S, M)$ for sample x;
3. the *perfect model* $p_{\mathcal{S}}(x)$ always chooses the fastest encoding for sample x, being defined as:

$$p_{\mathcal{S}}(x) = \begin{cases} 0 \text{ if } \exists(x, \hat{y}_0, \hat{y}_1) \in \mathcal{S} \wedge \hat{y}_0 < \hat{y}_1 \\ 1 \text{ otherwise} \end{cases} \tag{39}$$

We then set the *advantage* of the model f over a function g on a dataset $\mathcal{S}$ as:

$$Adv(f, g, \mathcal{S}) = \frac{\sum_{(x,\hat{y}_0,\hat{y}_1)\in\mathcal{S}} \left[\hat{y}_{g(x)}\right]}{\sum_{(x,\hat{y}_0,\hat{y}_1)\in\mathcal{S}} \left[\hat{y}_{f(x)}\right]} \tag{40}$$

Naturally, the higher the advantage $Adv(f, g, \mathcal{S})$ is, the better the model f performs. Note that advantage over the perfect model is always less than or equal to 1.

8.2.4 Choosing the depth of the decision tree

Training, validation and test sets We divided our dataset into a test set and a set of pairs of training and validation sets. More precisely, we chose to segment our dataset $\mathcal{S}$ into 11 segments, namely $\mathcal{S}_0, \ldots, \mathcal{S}_{10}$. Here, $\mathcal{S}_0$ is kept for the final testing phase while the remaining 10 segments of $\mathcal{S}$ are used to generate pairs $(\mathcal{S}_i, \bigcup_{j\neq i} \mathcal{S}_j)$ for $i = 1, \ldots, 10$.

Choosing the right depth Decision trees have the ability to match arbitrary functions if they are deep enough and the training set is sufficiently large. However, this is not desirable for two reasons: (i) the deeper the tree is, the more code will have to be added; and (ii) the deeper the tree is, the more susceptible it is to overfitting. We therefore need to find a proper depth for our decision tree. To do so, for each $i = 1, \ldots, 10$, we train a decision tree for each depth $d = 1, \ldots, 15$ on the set $\bigcup_{j\neq i} \mathcal{S}_j$ and evaluate the performance on the validation set $\mathcal{S}_i$. Figure 3 shows that the performance gains are mostly achieved by trees of depth lower than 3. As such, we empirically chose to use a decision tree of depth 3 in our framework,

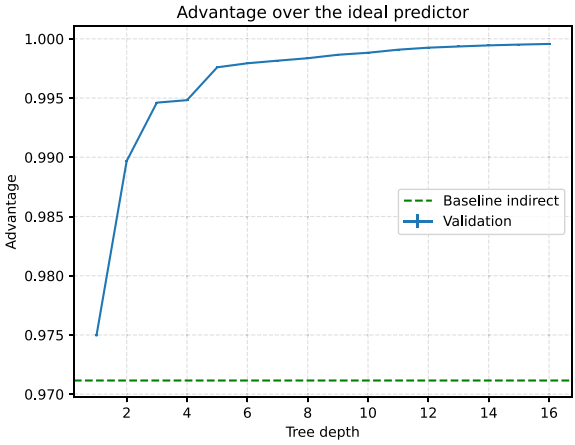


Fig. 3 Advantage of the model over the perfect model for different depths. The green dashed line shows the baseline advantage of the indirect SAT encoding over the perfect model

Figure 4c summarizes the decision tree resulting from the training. As only two leaves prefer the direct encoding, this tree can be summarised and optimised into the following pseudo-code on Fig. 5.

Evaluating the model Once our decision tree is trained, we can evaluate its performance on the test set. Because of the very large dataset available and the limited number of features, no overfitting was observed. The predictor has an advantage of 1.024 over the indirect encoding, and 0.995 over the ideal predictor. This method could be further improved by adding new encodings, or by increasing the feature space.

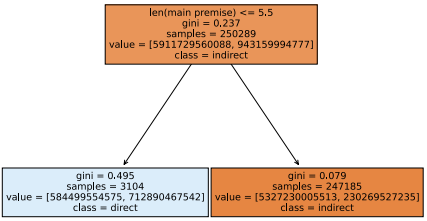
9 Experimental results

We implemented our SAT-based framework for solving subsumption and subsumption resolution in the Vampire theorem prover [16]. We next discuss the evaluation and results of our approach.

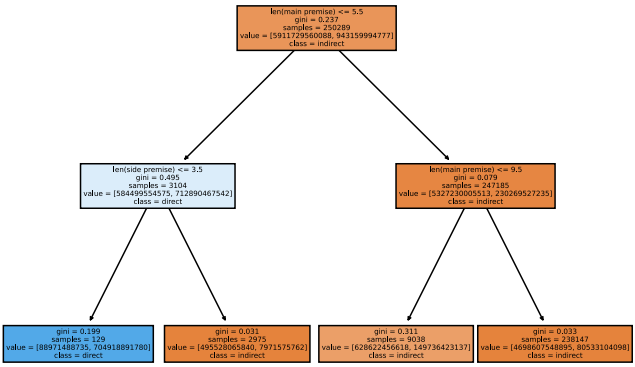
Benchmarks We use the TPTP library [40] (version 8.1.2) as the benchmark source for our experiments. This version of the TPTP library contains altogether 25,257 problems in various languages. Out of these examples, 24,973 problems have been included in our evaluation of SAT-based subsumption and subsumption resolution in Vampire. The remaining TPTP problems that we did not use for our experiments requires features that Vampire currently does not support (e.g., higher-order logic with theories).

Experimental Setup All our experiments were carried out on a cluster at TU Wien, where each compute node contains two AMD Epyc 7502 processors, each of which has 32 CPU cores running at 2.5 GHz. Each compute node is equipped with 1008 GiB of physical memory that is split into eight memory nodes of 126 GiB each, with eight logical CPUs assigned to each node. We used the `runexec` engine from the benchmarking framework BenchExec [43] to assign each benchmark process to a different CPU core and its corresponding memory node, aiming to balance the load across memory nodes. Further, we used GNU Parallel [44] to schedule 32 benchmark processes in parallel.

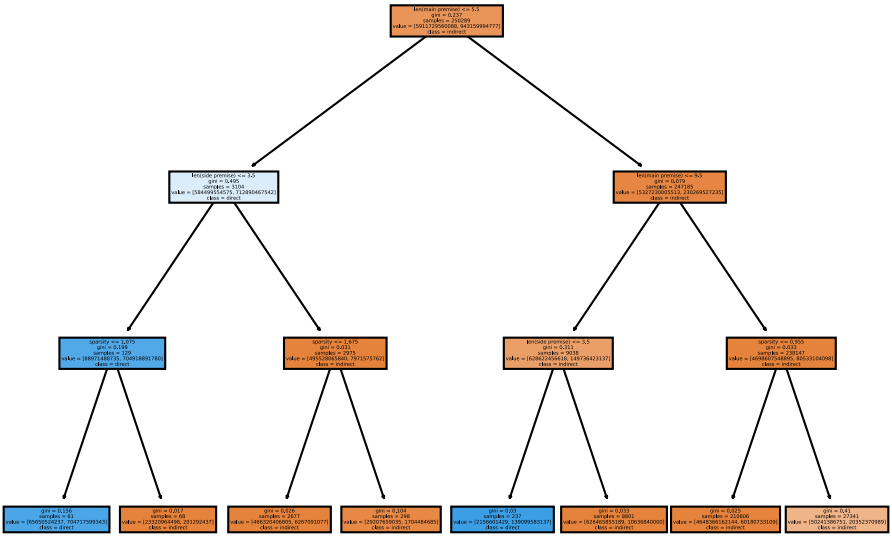
Ensuring consistent progress For several of the subsequent experiments, we perform relatively expensive computation and/or logging in addition to the measured solving



(a) Depth 1



(b) Depth 2



(c) Depth 3

Fig. 4 Decision trees of different depths. The orange nodes choose the indirect encoding, and the blue nodes choose the direct encoding

process. While this instrumentation does not affect the measurements per se, it will reduce the progress the solver can make in the saturation algorithm within a fixed duration of

Fig. 5 Decision tree from Fig. 4c in pseudo-code

```
if (|S| <= 3):
  if (|M| <= 5 and  sparsity <= 1.075):
    return direct
  elif (|M| <= 9):
    return direct
return indirect
```

wall-clock time. To avoid this effect, we first performed a run of Vampire without any expensive instrumentation and a time limit of 60 s, and report for each TPTP problem the number of times the forward simplification loop has been called. For all subsequent Vampire runs that involve instrumentation, we do not impose a time limit, but instead terminate after performing the previously reported number of forward simplification loops.

9.1 Measuring speed improvements for subsumption

We first measured the cost of subsumption checks in isolation. A similar evaluation has previously been done for indexing techniques in first-order provers, see [25].

Methods Considered We first ran Vampire with a timeout of 60 s on each TPTP problem, while logging each subsumption check into a file. Each of these files then contains a sequence of subsumption checks, which we call the *subsumption log* for a problem. This preparatory step led to a large number of benchmarks that are representative for subsumption checks that appear during actual proof search. These benchmarks occupy 1.79 TiB of disk space in compressed form, and contain about 278 billion subsumption checks in total. About 0.6 % of these subsumption checks are satisfiable (1.7 billion), while the rest is unsatisfiable. We note that we removed 5530 TPTP problems from this experiment, because Vampire was unable to parse back the output it generated during the logging phase. format for higher-order problems that we have been unable to correct in time. However, the successfully replayed subsumptions amount to about 258 billion subsumption checks (93 % of the collected).

Next, we executed the checks listed in each subsumption log and measured the total running times, once for the existing backtracking-based subsumption algorithm of Vampire, and once for our SAT-based subsumption approach in Vampire.

Results The results of this experiment are given in Fig. 6 and Table 1. Each mark in Fig. 6 represents one subsumption log from a TPTP problem, and compares the total running time of executing all subsumption checks contained in the log with the old backtracking-based algorithm vs. the new SAT-based algorithm. The dashed line indicates equal runtime, hence, our SAT-based approach was faster for marks below the line. In Table 1, we give the cumulative time used for subsumption. For the six TPTP problems `LCL673+1.015`, `LCL673+1.020`, `NLP023+1`, `NLP023-1`, `NLP024+1`, and `NLP024-1`, the old backtracking-based subsumption algorithm of Vampire did not terminate within a time limit of 1200s; these problems are not included in the cumulative sum.

Overall, our results show a clear improvement of the running time of subsumption in Vampire, yielding an improvement by a factor of 2.5.

Table 1 Total time spent on subsumption checks, summed over 19437 TPTP problems

Prover	Subsumption	Boost
$\text{VAMPIRE}_{\text{M}}$	35.86 *h*	
$\text{VAMPIRE}_{\text{SAT}}$	13.68 *h*	2.62 x

Note that VAMPIRE_{M} timed out on 6 problems during subsumption replay; these have not been included in the total

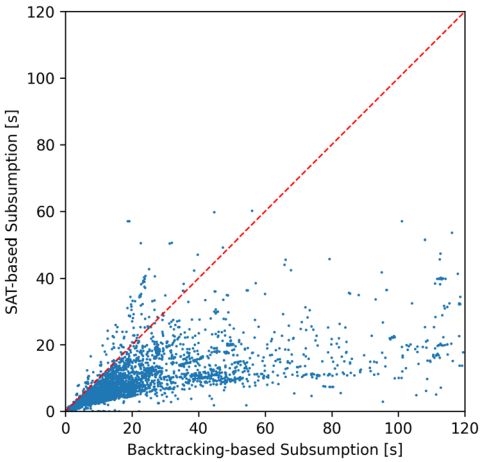


Fig. 6 Total running time (in seconds) of backtracking-based vs. SAT-based subsumption, where each mark represents a TPTP problem. For marks below the dashed line, our SAT-based approach was faster

9.2 Measuring speed improvements for subsumption resolution

Whereas subsumption instances can be separated from the rest of the execution, efficient subsumption resolution cannot. As explained in Sect. 6, subsumption resolution is applied in a simplification loop that optimised the setup between subsumption and subsumption resolution. It would thus be an unfair comparison to isolate subsumption resolution from subsumption. This is why we decided to measure the runtime of subsumption and subsumption resolution together in a forward simplification loop.

Our experimental procedure is summarized in Algorithm 5 and commented below.

Algorithm 5 Evaluation of SAT-based subsumption resolution

```
procedure ForwardSimplifyWrapper(M, F)
    s ← StartTimer()
    r ← ForwardSimplify(M, F)          ▷ Benchmarked method
                                       ▷ Prevent modification of F
    e ← EndTimer()
    WriteToFile(e − s)
    r′ ← Oracle(M, F)
    CheckCoherence(r, r′)              ▷ Empiric check
    return r′
```

- The conclusion clause of the subsumption resolution rule SR is not necessarily unique. Therefore, different versions of subsumption resolution, including our work based on direct and indirect SAT encodings, may not return the same conclusion clause of SR. Hence, applying different versions of subsumption resolution over the same clauses may change the saturation process.
- Saturation with our SAT-based subsumption resolution takes advantage of subsumption checking (see Algorithms 2–4). Therefore, only checking subsumption resolution on pairs of clauses is not a fair nor viable comparison, as isolating subsumption checks from subsumption resolution is not what we aimed for (due to efficiency).
- CPU cache influences results. For example, two consecutive runs of Algorithm 2 may be up to 25% faster on second execution, due to cache effects.
- CheckCoherence(r, r') is an empiric check that ensures that the result of the oracle is compatible with the result of the benchmarked method.

Oracle The oracle used in our experiments is the fastest method overall. The motivation of this choice is to maximise the number of sample points compared to the total computation time. Indeed, a slower oracle will prevent Vampire to progress faster. The oracle therefore runs the dynamic encoding (heuristic encoding selection) with loop optimisation.

Methods considered We compared the following versions of Vampire:

- Vampire$_M$—the master branch of Vampire (commit a47e1dca9), without SAT-based subsumption and subsumption resolution;
- Vampire$_D$—the SAT-based version of Vampire using the direct encoding $\mathcal{E}^d_{\mathtt{SR}}$;
- Vampire$_I$—the SAT-based version of Vampire using the indirect encoding $\mathcal{E}^i_{\mathtt{SR}}$;
- Vampire$_H$—the SAT-based version of Vampire using the heuristic discussed in Sect. 8.2;
- Vampire$^*_{\mathcal{E}}$—using the loop optimisation discussed in Sect. 7 with the encoding $\mathcal{E}$ (note that the loop optimisation does not apply to the non-SAT version);
- Vampire-cutoff-n—uses a cutoff at n ticks, as discussed in Sect. 8.1.

Results In Table 2, the average and standard deviation of the runtime of the forward simplification loop have been logged for the considered versions. The column **Boost** is the ratio between the average runtime of Vampire$_M$ and the method considered. From the table, we can see that the simplest version of our algorithm, that is, the direct encoding without loop

Table 2 Average time spent in the forward simplify loop. VAMPIRE^*_H is the fastest method, closely followed by the VAMPIRE^*_I

Prover	Average	Std. Dev.	Boost
VAMPIRE_M	33.63 µs	1839.25 µs	1.00
VAMPIRE_D	28.74 µs	1245.88 µs	1.17
VAMPIRE_I	28.36 µs	243.38 µs	1.19
VAMPIRE_H	28.16 µs	233.87 µs	1.19
VAMPIRE^*_D	25.38 µs	1241.86 µs	1.32
VAMPIRE^*_I	24.93 µs	196.38 µs	1.35
VAMPIRE^*_H	24.73 µs	190.69 µs	1.36

The versions $\text{VAMPIRE}^*_{\mathcal{E}}$ integrate the loop optimisation discussed in Sect. 7 into $\text{VAMPIRE}_{\mathcal{E}}$

optimisation, already performs better than the old backtracking-based algorithm. Introducing the indirect encoding creates a large drop in variance, indicating that $\mathcal{E}^i_{\text{SR}}$ is more stable and scalable. The loop optimisation further improved performance by sharing work in the encoding setup. Finally, choosing the encoding based on the heuristic discussed in Sect. 8.2 brings another small improvement boost. Overall, we obtained an improvement in performance by a factor of 1.36 on the simplification loop.

When considering these results with our previous analysis of subsumption alone (Sect. 9.1), it is worth mentioning that they are not comparable. While the evaluation method in Sect. 9.1 allows a direct comparison of the backtracking-based subsumption implementation to the SAT-based approach, such an evaluation is not suitable for subsumption resolution, especially when considering the optimized simplification loop (Sect. 7). Indeed, our second benchmarking technique (this section) includes all components of the simplification loop, including obtaining candidate clauses. In particular, this means we also measure improvements obtained by optimizing the simplification loop itself.

Remark 4 In [29], we observed a large drop in variance from the standard to the optimised simplification loop. Improving the memory usage of the pruning algorithms in Sect. 6.2 greatly reduced this unexpected behaviour. In [29], we used a standard C++ vector that was cleared between pruning runs. However, some problems in the TPTP library have very large signatures. On these instances, the subsumption execution time has been greatly impacted by the calls to our simpler pruning algorithm from [29]. Namely, in the standard saturation loop, pruning was executed once for subsumption and once for subsumption resolution. As discussed in Sect. 6, a large proportion of subsumption pruning checks are unnecessary if the subsumption resolution pruning criterion fails first. Our fast implementations of pruning from Sect. 6.2 greatly reduced this effect from [29].

Figure 7 shows the cumulative number of forward simplification loops performed in less then t μs for some methods. We can visually see that our method performs better than the previous implementation even for the easier instances, and further increasing its advantage on harder instances. The loop optimisation shows most its strength in the 10 to 20 μs region before almost getting caught up by the non-optimised loop. The reason is that the optimisation only improves the polynomial setup of the algorithm, that becomes less relevant as the exponential nature of the problem takes over.

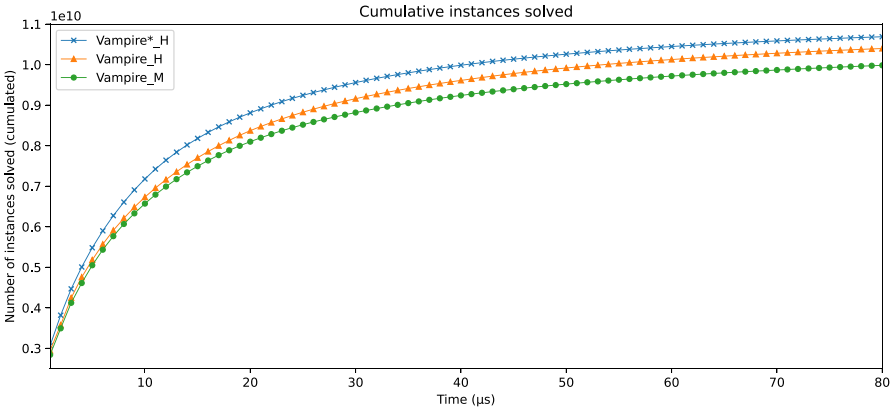


Fig. 7 Cumulative instances of applying subsumption resolution, using the TPTP examples. A point (t, n) on the graph means that n forward simplify loops were executed in less than t μs. The higher the curve, the faster the Vampire version is. The difference between the different encoding being small relative to the difference the optimisation brings, we only displayed the dynamic encoding to avoid superposition of plot lines

9.3 Overall Vampire runs

We finally analysed the the number of problems Vampire solves depending on various implementations of the subsumption and subsumption resolution procedure. Table 3 summarizes our findings and we draw the following conclusions.

- Each SAT-based configuration of subsumption solves more problems than the previous, backtracking-based implementation of subsumption, showing the superiority of our method in solving subsumption and subsumption resolution.
- Our heuristic approach using decision trees of Sect. 8 solves slightly more problems than the other SAT-based only methods of Sect. 7. We remark that we trained our decision trees on a dataset built from the exact problems we are testing our methods against, with the purpose of maximizing the number of solved problems. We note that our methodology might suffer from (minimial) overfitting: we used a very rigid classification algorithm with a very low potential for overfitting. It is unlikely that a decision tree with such a low depth and few features will learn how to solve specific problems, but not learn general trends.
- Our cutoff method from Sect. 8.1 did not bring great improvements. While this result may sound discouraging, we believe it actually strengthens our contributions from Sect. 7. Indeed, it shows that only finding the simple subsumption and subsumption resolution instances is not an effective strategy. While our methods from Sect. 7 might not be the fastest for small clauses, they scale well with the complexity of the problem.
- The saturation loop optimisation techniques, e.g. forward simplifcation from Sect. 7, bring the largest increase in number of problems solved. This follows our intuition built from Table 2. We however note that our loop optimisation techniques may lose slightly more problems than their un-optimised loop versions. This is because our loop optimisation methods may perform some unnecessary and potentially hard subsumption resolutions, slightly increasing the likelihood of being stuck on difficult combinatorial problems.

Table 3 Number of TPTP problems solved by the considered versions of Vampire

Prover	Total solved	Gain/loss
Vampire_M	10,728	Baseline
Vampire_D	10,762	(+62, −28)
Vampire_I	10,760	(+63, −31)
Vampire_H	10,764	(+64, −28)
$\text{Vampire–cutoff-5000}_H$	10,766	(+65, −27)
$\text{Vampire–cutoff-150}_H$	10,739	(+56, −45)
Vampire^*_D	10,791	(+94, −31)
Vampire^*_I	10,785	(+92, −35)
Vampire^*_H	10,794	(+97, −31)
$\text{Vampire–cutoff-5000}^*_H$	10,790	(+97, −35)
$\text{Vampire–cutoff-150}^*_H$	10,768	(+93, −53)

The run was made using the options `-sa otter -av off` with a timeout of 60 s. The **Gain/Loss** column reports the difference of solved instances compared to Vampire_M. The versions $\text{Vampire}^*_{\mathcal{E}}$ integrate the loop optimisation discussed in Sect. 7 into $\text{Vampire}_{\mathcal{E}}$

10 Related work

Subsumption and subsumption resolution are some of the most powerful and frequently used redundancy criteria in saturation-based first-order theorem proving.

Subsumption While efficient literal- and clause-indexing techniques have been proposed [26, 45], optimising the matching step among multisets of literals, and hence clauses, has so far not been addressed. Our work shows that SAT solving methods can provide efficient solutions in this respect, further improving first-order theorem proving.

A related approach that integrates multi-literal matching into indexing is given in [24], using code trees. Code trees organise potentially subsuming clauses into a tree-like data structure with the aim of sharing some matching effort for similar clauses. However, the underlying matching algorithm uses a fixed branching order and does not learn from conflicts, and will thus run into the same issues on hard subsumption instances as the standard backtracking-based matching.

The specialised subsumption algorithm DC [46] is based on the idea of separating the clause S into variable-disjoint components and testing subsumption for each component separately. However, the notion of subsumption considered in that work is defined using subset inclusion, rather than multiset inclusion. For subsumption based on multiset inclusion, the subsumption test for one variable-disjoint component is no longer independent of the other components.

An improved version of [46] comes with IDC [47], whereupon each recursion level is checked whether each literal of S by itself subsumes M under the current partial substitution, which is a necessary condition for subsumption. The backtracking-based subsumption algorithm of Vampire uses this optimisation as well, and our SAT-based approach also implements it as propagation over substitution constraints.

By combining subsumption and resolution into one simplification rules, subsumption resolution is supported as contextual literal cutting in [17], along with efficient approaches for detecting multiset inclusions among clauses [15, 26, 48]. Special cases of unit deletion as a by-product of subsumption tests are also proposed in [45], with

further refinements of term indexing to drastically reduce the set of candidate clauses checked for subsumption (resolution).

SAT- and SMT-based techniques have previously been applied to the setting of first-order saturation-based proof search, e.g. in the form of the Avatar architecture [38]. These techniques are, however, independent from our work, as they apply the SAT- or SMT-solver over an abstraction of the input problem, while in our work we use a SAT solver to speed up certain inferences.

Some solvers, such as the pseudo-boolean solver MiniCard [49] and the ASP solver Clasp [50], support cardinality constraints natively, in a similar way to our handling of at-most-one constraints. Our encoding, however, requires only at-most-one constraints instead of arbitrary cardinality constraints, thus simplifying the implementation.

Note that clausal subsumption can also be seen as a constraint satisfaction problem (CSP). In this view, the boolean variables b^+_{ij} in our subsumption encoding represent the different choices of a non-boolean CSP variable, corresponding to the so-called *direct encoding* of a CSP variable [51]. A well-known heuristic in CSP solving is the minimum remaining values heuristic: always assign the CSP variable that has the fewest possible choices remaining. We adapted this heuristic to our embedded SAT solver and used it to solve subsumption instances [28]; however, it does not fit the subsumption resolution encodings well, especially the indirect encoding. Moreover, the advantage over the well-known variable-move-to-front (VMTF) heuristic [52] is minor even for subsumption, which is why we now always use VMTF for variable selection in the SAT solver.

We finally remark that redundancy is explored in SAT-based equivalence checking [53], by using first-order and QBF reasoning for subsumption checks [23]. In particular, first-order backward subsumption [24] has become a key preprocessing techniques in SAT solving, in particular in bounded variable elimination [54, 55]. Our work complements this line of research by showcasing that SAT solving also improves solving variants of first-order subsumption, not just the other way around.

11 Conclusion

We promote tailored SAT solving to solve clausal subsumption and subsumption resolution in first-order theorem proving. We introduce substitution constraints to encode subsumption constraints as SAT instance. For solving such instances, we adjust unit propagation and conflict resolution in SAT solving towards a specialized treatment of substitution constraints and at-most-one constraints. Crucially, our encoding together with our SAT solver enables efficient setup of subsumption and subsumption resolution instances. We show that the resulting SAT solver can directly be integrated within the saturation loop of first-order theorem proving, solving both subsumption and subsumption resolution. Our experimental results indicate that SAT-based subsumption and subsumption resolution significantly improves the performance of first-order proving. Extending subsumption with theory reasoning with equality, possibly in the presence of (arithmetic) first-order theories, is an interesting task for future work. We believe this would open up potentially new venues for using SMT solving instead of SAT solving for subsumption reasoning.

Acknowledgements We thank Pascal Fontaine (University of Liège, Belgium) for fruitful discussions. We acknowledge partial support from the ERC Consolidator Grant ARTIST 101002685, the FWF SFB project SpyCoDe F8504, the Austrian FWF project W1255-N23, the WWTF ICT22-007 Grant ForSmart, and the

TU Wien Trustworthy Autonomous Cyber-Physical Systems Doctoral College. Initial results on this work have been established during a research internship of Robin Coutelier at TU Wien, while he was still a master student at the University of Liège, Belgium, This research was funded also in part by the Austrian Science Fund (FWF) [10.55776/F85, 10.55776/W1255], for open access purposes; the authors have applied a CC BY public copyright license to any accepted manuscript version arising from this submission.

Funding Open access funding provided by TU Wien (TUW).

References

1. Leino KRM (2017) Accessible software verification with Dafny. IEEE Softw 34(6):94–97
2. Clochard M, Marché C, Paskevich A (2020) Deductive verification with ghost monitors. In: Proceedings of POPL, pp 2–1226
3. Georgiou P, Gleiss B, Kovács L (2020) Trace logic for inductive loop reasoning. In: Proceedings of FMCAD, pp 255–263
4. Komuravelli A, Gurfinkel A, Chaki S (2016) SMT-based model checking for recursive programs. Form Methods Syst Des 48(3):175–205
5. Padon O, McMillan KL, Panda A, Sagiv M, Shoham S (2016) Ivy: safety verification by interactive generalization. In: Proceedings of PLDI, pp 614–630
6. Asadi S, Blicha M, Hyvärinen AEJ, Fedyukovich G, Sharygina N (2020)Incremental verification by SMT-based summary repair. In: Proceedings of FMCAD, pp 77–82
7. Garcia-Contreras I, K, HGV, Shoham S, Gurfinkel A (2023) Fast approximations of quantifier elimination. In: Proceedings of CAV, pp 64–86 (2023). https://doi.org/10.1007/978-3-031-37703-7_4
8. Pick L, Fedyukovich G, Gupta A (2020) Automating modular verification of secure information flow. In: Proceedings of FMCAD, pp 158–168
9. Martínez G, Ahman D, Dumitrescu V, Giannarakis N, Hawblitzel C, Hritcu C, Narasimhamurthy M, Paraskevopoulou Z, Pit-Claudel C, Protzenko J, Ramananandro T, Rastogi A, Swamy N (2019) Meta-F*: proof automation with SMT, tactics, and metaprograms. In: Proceedings of ESOP, pp 30–59
10. Veronese L, Farinier B, Bernardo P, Tempesta M, Squarcina M, Maffei M (2023) WebSpec: towards machine-checked analysis of browser security mechanisms. In: SP, pp 2761–2779 . https://doi.org/10.1109/SP46215.2023.10179465
11. Brugger LS, Kovács L, Komel AP, Rain S, Rawson M (2023) CheckMate: automated game-theoretic security reasoning. In: CCS, pp 1407–1421. https://doi.org/10.1145/3576915.3623183
12. Biere A (2008) PicoSAT essentials. J Satisf Boolean Model Comput 4(2–4):75–97
13. De Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Proceedings of TACAS, pp 337–340
14. Barbosa H, Barrett CW, Brain M, Kremer G, Lachnitt H, Mann M, Mohamed A, Mohamed M, Niemetz A, Nötzli A, Ozdemir A, Preiner M, Reynolds A, Sheng Y, Tinelli C, Zohar Y (2022) CVC5: a versatile and industrial-strength SMT solver. In: Proceedings of TACAS, pp 415–442
15. Weidenbach C, Dimova D, Fietzke A, Kumar R, Suda M, Wischnewski P (2009) SPASS version 3.5. In: Proceedings of CADE, pp 140–145
16. Kovács L, Voronkov A (2013) First-order theorem proving and vampire. In: CAV, pp 1–35
17. Schulz S, Cruanes S, Vukmirovic P (2019) Faster, higher, stronger: E 2.3. In: Proceedings of CADE, pp 495–507
18. Cruanes S (2017) Superposition with structural induction. In: Proceedings of FroCoS, pp 172–188
19. Buchberger B (2006) Bruno Buchberger's PhD thesis 1965: an algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. J Symb Comput 41(3–4):475–511. https://doi.org/10.1016/j.jsc.2005.09.007
20. Nieuwenhuis R, Rubio A (2001) Paramodulation-based theorem proving. In: Handbook of automated reasoning, pp 371–443. https://doi.org/10.1016/b978-044450813-3/50009-6

21. Robinson JA (1965) A machine-oriented logic based on the resolution principle. J ACM 12(1):23–41. https://doi.org/10.1145/321250.321253
22. Bachmair L, Ganzinger H (1994) Rewrite-based equational theorem proving with selection and simplification. J Log Comput 4(3):217–247
23. Biere A (2004) Resolve and expand. In: Proceedings of SAT. https://doi.org/10.1007/11527695_5
24. Sekar R, Ramakrishnan IV, Voronkov A (2001) Term indexing. In: Handbook of automated reasoning, pp 1853–1964
25. Nieuwenhuis R, Hillenbrand T, Riazanov A, Voronkov A (2001) On the evaluation of indexing techniques for theorem proving. In: Proceedings of IJCAR, pp 257–271
26. Schulz S (2013) Simple and efficient clause subsumption with feature vector indexing. In: Automated reasoning and mathematics—essays in memory of William W. McCune, pp 45–67
27. Kapur D, Narendran P (1986) NP-completeness of the set unification and matching problems. In: Proceedings of IJCAR, pp 489–495
28. Rath J, Biere A, Kovács L (2022) First-order subsumption via SAT solving. In: FMCAD, p 160
29. Coutelier R, Kovács L, Rawson M, Rath J (2023) SAT-based subsumption resolution. In: Proceedings of CADE, pp 190–206. https://doi.org/10.1007/978-3-031-38499-8_11
30. Gleiss B, Kovács L, Rath J (2020) Subsumption demodulation in first-order theorem proving. In: Proceedings of the of IJCAR, pp 297–315
31. Eén N, Sörensson N (2003) An extensible SAT-solver. In: Proceedings of SAT, pp 502–518. https://doi.org/10.1007/978-3-540-24605-3_37
32. Biere A, Froleyks N, Wang W (2023) CadiBack: extracting backbones with CaDiCaL. In: Proceedings of SAT, pp 3–1312. https://doi.org/10.4230/LIPICS.SAT.2023.3
33. Fleury M, Biere A (2022) Mining definitions in Kissat with Kittens. Formal Methods Syst Des 60(3):381–404. https://doi.org/10.1007/S10703-023-00421-2
34. Marques-Silva J, Lynce I, Malik S (2021) Conflict-driven clause learning SAT solvers. In: Handbook of satisfiability. frontiers in artificial intelligence and applications, vol 336, pp 133–182. Chapter 4
35. Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: engineering an efficient SAT solver. In: Proceedings of DAC, pp 530–535
36. Frisch AM, Giannaros PA (2010) SAT encodings of the at-most-k constraint. some old, some new, some fast, some slow. In: Proceedings of WS on constraint modelling and reformulation
37. McCune W, Wos L (1997) Otter—the CADE-13 competition incarnations. J Autom Reason 18:211–220
38. Voronkov A (2014) AVATAR: the architecture for first-order theorem provers. In: Proceedings of CAV, pp 696–710. https://doi.org/10.1007/978-3-319-08867-9_46
39. Biere A, Fazekas K, Fleury M, Heisinger M (2020) CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In: Proceedings of SAT competition 2020: solver and benchmark descriptions, pp 50–53. http://hdl.handle.net/10138/318450
40. Sutcliffe G (2017) The TPTP problem library and associated infrastructure. From CNF to TH0, TPTP v6.4.0. J Autom Reason 59(4):483–502
41. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in python. J Mach Learn Res 12:2825–2830
42. Goodfellow IJ, Bengio Y, Courville AC (2016) Deep learning. Adaptive computation and machine learning. http://www.deeplearningbook.org/
43. Beyer D, Löwe S, Wendler P (2017) Reliable benchmarking: requirements and solutions. J. Softw Tools Technol Transf 21(1):1–29
44. Tange O (2018) GNU parallel 2018
45. Tammet T (1998) Towards efficient subsumption. In: Proceedings of CADE, pp 427–441
46. Gottlob G, Leitsch A (1985) On the efficiency of subsumption algorithms. J ACM 32(2):280–295
47. Gottlob G, Leitsch A (1985) Fast subsumption algorithms. In: Proceedings of EUROCAL '85, pp 64–77
48. Kovács L, Voronkov A (2013) First-order theorem proving and vampire. In: CAV, pp 1–35. https://doi.org/10.1007/978-3-642-39799-8_1
49. Liffiton MH, Maglalang JC (2012) A cardinality solver: more expressive constraints for free. In: Proceedings of SAT, pp 485–486
50. Gebser M, Kaminski R, Kaufmann B, Schaub T (2009) On the implementation of weight constraint rules in conflict-driven ASP solvers. In: Proceedings of ICLP, pp 250–264
51. Walsh T (2000) SAT v CSP. In: Proceedings of CP, pp 441–456
52. Ryan L (2004) Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University

53. Heule MJH, Kiesl B, Biere A (2020) Strong extension-free proof systems. J Autom Reason 64(3):533–554. https://doi.org/10.1007/S10817-019-09516-0
54. Eén N, Biere A (2005) Effective preprocessing in SAT through variable and clause elimination. In: Proceedings of SAT, vol 3569, pp 61–75. https://doi.org/10.1007/11499107_5
55. Biere A, Järvisalo M, Kiesl B (2021) Preprocessing in SAT solving. In: Handbook of satisfiability—second edition. Frontiers in artificial intelligence and applications, vol 336, pp 391–435. https://doi.org/10.3233/FAIA200992