SAT-Based Subsumption Resolution

Robin Coutelier[1](✉), Laura Kovács[2], Michael Rawson[2], and Jakob Rath[2]

[1] U. Liège, Liège, Belgium
robin.coutelier@student.uliege.be
[2] TU Wien, Vienna, Austria

Abstract. Subsumption resolution is an expensive but highly effective simplifying inference for first-order saturation theorem provers. We present a new SAT-based reasoning technique for subsumption resolution, without requiring radical changes to the underlying saturation algorithm. We implemented our work in the theorem prover Vampire, and show that it is noticeably faster than the state of the art.

1 Introduction

Saturation-based proof search is a popular approach to first-order theorem proving [6,14,18]. In addition to efficient inference systems [8,1], saturation provers also implement *redundancy elimination* to reduce the size of the search space. Redundancy elimination deletes clauses from the search space by showing them to be logical consequences of other (smaller) clauses, and therefore redundant. However, checking whether a first-order formula is implied by another first-order formula is undecidable, and so eliminating redundant clauses is in general undecidable too. In practice, saturation systems apply cheaper conditions for redundancy elimination, such as removing equational tautologies by congruence closure or deleting subsumed clauses by establishing multiset inclusion. Recently, SAT solving has been applied to efficiently detect and remove subsumed clauses [10]. *We extend SAT-based reasoning in first-order theorem proving to a combination of subsumption and resolution, **subsumption resolution** [2] (Section 4).*

Both subsumption and subsumption resolution are NP-complete [4]. To improve efficiency in practice, we (i) encode subsumption resolution as SAT formulas over (match) set constraints (Section 5) and (ii) directly integrate CDCL SAT solving for checking subsumption resolution in first-order theorem proving (Section 6). We implement our approach in the theorem prover Vampire [6], improving the state-of-the-art in first-order reasoning (Section 7).

Related Work. Subsumption and subsumption resolution are some of the most powerful and frequently used redundancy criteria in saturation-based provers. Subsumption resolution is supported as *contextual literal cutting* in [14], along with efficient approaches for detecting multiset inclusions among clauses [6,18,13]. Special cases of unit deletion as a by-product of subsumption tests are also proposed in [16]. Much attention has been given to refinements of *term indexing* [16,13] to drastically reduce the set of candidate clauses checked for subsumption. Recently, these approaches have been complemented by SAT solving [10],

reducing subsumption checking to SAT. Our work generalises this approach by solving for both subsumption and subsumption resolution via SAT.

SAT solvers have been applied widely to first-order theorem proving, including but not limited to AVATAR [17], instance-based methods [5], heuristic grounding [14], global subsumption [12] and combinations thereof [11], but using SAT solvers for classical subsumption methods is under-explored. To the best of our knowledge, SAT solving for subsumption resolution has so far not been addressed in the landscape of automated reasoning.

2 Illustrative Examples and Main Contributions

Let us illustrate a few challenges of subsumption resolution, which motivate our approach to solving it (Section 4). Given a pair of clauses L and M, denoted as (L, M), the problem is to decide whether M can be simplified by L via a special case of logical consequence. In Figure 1 we show examples where it is not obvious for which pairs (L_i, M_i) subsumption resolution can be applied.

$L_1 := p(x_1, x_2) \vee p(f(x_2), x_3)$ $M_1 := p(g(y_1), c) \vee \neg p(f(c), e)$	$L_2 := p(x_1) \vee q(x_2)$ $M_2 := \neg p(y) \vee \neg q(c)$
$L_3 := p(x_1) \vee q(x_1, x_2) \vee \neg p(x_2)$ $M_3 := \neg p(y) \vee q(y, y)$	$L_4 := p(x_1) \vee q(x_2) \vee r(x_3)$ $M_4 := \neg p(y_1) \vee q(c)$

Fig. 1: Illustrative examples.

In fact, subsumption resolution can only be applied to (L_1, M_1). Later, we show how our approach determines that M_1 can be shortened in the presence of L_1 (Example 3.1), but also how the remaining pairs cannot apply subsumption resolution (Examples 5.1, 5.2, and 4.1). For example, (L_4, M_4) is filtered by *pruning* to bypass the SAT routine altogether.

Our Contributions.

1. We cast the problem of subsumption resolution over pairs of first-order formulas (L, M) as a SAT problem (Theorem 5.1), ensuring any instance of subsumption resolution is a model of this SAT problem.
2. We tailor encodings of subsumption resolution (Sections 5.1–5.2) for effective SAT-based subsumption resolution (Algorithm 1).
3. We integrate our approach into the saturation loop, solving for subsumption and subsumption resolution simultaneously (Section 6).
4. We implement our work in the theorem prover VAMPIRE and showcase our practical gains in first-order proving (Section 7).

3 Preliminaries

We assume familiarity with first-order logic with equality. We include standard Boolean connectives and quantifiers in the language, and the constants $\top, \bot$ for truth and falsehood. We use x, y, z for first-order variables, c, d, e for constants, f, g for functions, p, q, r for atoms, l, m for literals, and L, M for clauses, all potentially with indices. If L is a clause $l_1 \vee \ldots \vee l_n$, we sometimes consider it as a multiset of its literals l_i, and write $|L|$ for its cardinality (i.e. the number n of literals in L). The empty clause is denoted $\square$. Free variables are universally quantified. An expression E is a term, atom, literal, clause, or formula.

Substitutions and matches. A substitution σ is a (partial) mapping from variables to terms. The result of applying a substitution σ to an expression E is denoted $\sigma(E)$ and is the expression obtained by simultaneously replacing each variable x in E by $\sigma(x)$. For example, the application of $\sigma \coloneqq \{x \mapsto f(c)\}$ to the clause $L \coloneqq \{p(x), q(x, y)\}$ yields $\sigma(L) = \{p(f(c)), q(f(c), y)\}$. Note that $\sigma(L)$ is a logical consequence of L.

A *matching substitution*, in short a *match*, between literals l and m is a substitution σ such that $\sigma(l) = m$. For example, the match of $p(x)$ onto $p(f(c))$ is $\{x \mapsto f(c)\}$. Two matches are *compatible* and can be combined in the same substitution iff they do not assign different terms to the same variable. For example, the substitutions $\{x \mapsto f(c), y \mapsto g(d)\}$ and $\{x \mapsto f(c), z \mapsto h(e)\}$ are compatible, but $\{x \mapsto f(c)\}$ and $\{x \mapsto g(c)\}$ are not.

Saturation and redundancy. Many first-order systems apply the superposition calculus [1] in a saturation loop [8]. Given an input set F of clauses, saturation iteratively derives logical consequences and adds them to F. By soundness and completeness of superposition, if $\square$ is derived the system can report unsatisfiability of F; if $\square$ is not encountered and no further clauses can be derived, the system reports satisfiability of F.

Saturation is more efficient when F is as small as possible. For this reason, saturation-based provers also employ *simplifying* inferences. Simplifying inferences reduce the number or size of clauses in F. This is formalised using the following notion of *redundancy*: a ground clause M is redundant in a set of ground clauses F if M is a logical consequence of clauses in F that are strictly smaller than M w.r.t. a fixed simplification ordering $\succ$. A non-ground clause M is redundant in a set of clauses F if each ground instance of M is redundant in the set of ground instances of F. If M is redundant in F, then M can be removed from F while retaining completeness.

Subsumption. A clause L *subsumes* a distinct clause M iff there is a substitution σ such that

$$\sigma(L) \subseteq_M M \tag{1}$$

where $\subseteq_M$ denotes multiset inclusion. We also say that M is *subsumed* by L. Note that subsumed clauses are redundant.

Removing subsumed clauses M from the search space F is implemented through a simplifying rule, checking condition (1) over pairs of clauses (L, M)

from F. Matches between every literal in L to some literal in M are checked; if a compatible set of matches is found, then M can be removed from F.

Subsumption resolution. Subsumption resolution aims to remove one redundant literal from a clause. Clauses M and L are said to be the main and side premise of subsumption resolution, respectively, iff there is a substitution σ, a set of literals $L' \subseteq L$ and a literal $m' \in M$ such that

$$\sigma(L') = \{\neg m'\} \quad \text{and} \quad \sigma(L \setminus L') \subseteq M \setminus \{m'\}. \tag{2}$$

If so, M can be replaced by $M \setminus \{m'\}$. Subsumption resolution is hence the rule

$$(\texttt{SR}) \quad \frac{L \qquad \cancel{M}}{M \setminus \{m'\}}$$

We indicate the deletion of a clause M by drawing a line through it ($\cancel{M}$), and 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 proof search dynamics.

Example 3.1. Consider L_1, M_1 of Figure 1. Subsumption resolution is applied by using the substitution $\sigma \coloneqq \{x_1 \mapsto g(y_1), x_2 \mapsto c, x_3 \mapsto e\}$. Note that $\sigma(L_1) = p(g(y_1), c) \vee p(f(c), e)$. $\sigma(L_1)$ and M_1 can be resolved to obtain $p(g(y_1), c)$. The clause $p(g(y_1), c)$ subsumes M_1, since it is a sub-multiset of M_1. We have

$$\frac{p(x_1, x_2) \vee p(f(x_2), x_3) \qquad \cancel{p(g(y_1), c) \vee \neg p(f(c), e)}}{p(g(y_1), c)}$$

4 SAT-based Subsumption Resolution

We describe the main steps of our SAT-based approach for deciding the applicability of subsumption resolution on a pair (L, M) of clauses. The core of our work solves (2) by finding match substitutions between literals in L and M. Our technique is summarised in Algorithm 1.

Pruning. The first step of Algorithm 1 *prunes* pairs (L, M) of clauses that cannot be simplified by subsumption resolution due to a syntactic restriction over symbols in L and M, *viz.* whether the set of predicates in L is a subset of the predicates in M. If not, then there is a literal in L that cannot be matched to any literal in M, and hence subsumption resolution cannot be applied.

Example 4.1. The clause pair (L_4, M_4) from Figure 1 is pruned by Algorithm 1: the set of predicates in L_4 and M_4 are respectively $\{p, q, r\}$ and $\{p, q\}$, implying that the literal $r(x_3)$ of L_4 cannot be matched to any literal in M_4.

Algorithm 1 SAT-based subsumption resolution over pair (L, M) of clauses

```
ms ← createMatchSet()
solver ← createSatSolver(ms)
procedure SubsumptionResolution(L, M)
    if pruned(L, M) then
        return NoSubsumptionResolution
    if fillMatchSet(ms, L, M) is false then
        return NoSubsumptionResolution
    encodeConstraints(solver, ms)
    if solver.solve() is SAT then
        return buildConclusion(solver.getSolution(), M)    ▷ conclusion of subsumption resolution
    return NoSubsumptionResolution
```

Match set. The *match set* of Algorithm 1 computes matching substitutions over literals of L and M. The match set ms consists of a sparse matrix that assigns each literal pair $(l_i, m_j) \in L \times M$ a substitution $\sigma_{i,j}$ such that $\sigma_{i,j}(l_i) = m_j$ or $\sigma_{i,j}(l_i) = \neg m_j$. In addition, a polarity $P_{i,j}$ is also assigned to (l_i, m_j), as follows: we set polarity $P_{i,j} = +$ if $\sigma_{i,j}(l_i) = m_j$ and $P_{i,j} = -$ if $\sigma_{i,j}(l_i) = \neg m_j$. This matrix is sparse because in general not all literal pairs $(l_i, m_j) \in L \times M$ can be matched. Additionally, it is again possible to prune (L, M) while filling the match set: if a row of the match set is empty, then there is some literal in L that cannot be matched to any literal in M. In this case, subsumption resolution cannot use L to simplify M, so the pair (L, M) is pruned.

SAT solver. The *solver* of Algorithm 1 is the CDCL-based SAT solver introduced previously [10], which supports reasoning over matching substitutions in addition to standard propositional reasoning. This solver also features direct support for *AtMostOne* constraints. Solver performance was tuned for subsumption, which we retain for subsumption resolution. Each propositional variable v is associated with a substitution σ_v, and the solver ensures that all substitutions σ_v, for which v is assigned $\top$ in the current model, are compatible. Conceptually, a global substitution σ satisfying the invariant $\sigma = \bigcup\{\sigma_v \mid v = \top\}$ is kept in the SAT solver. In the following, we will write this binding as $v \Rightarrow \sigma_v \subseteq \sigma$.

Example 4.2. Suppose propositional variables v_1 and v_2 are associated with substitutions $\sigma_1 := \{x \mapsto y\}$ and $\sigma_2 := \{x \mapsto z\}$, respectively. As σ_1 and σ_2 are incompatible, the solver will block assigning $v_1 = \top$ and $v_2 = \top$ simultaneously since it would break the above invariant.

Encoding constraints. Given the match set of (L, M), we formalise the subsumption resolution problem (2) as the conjunction of four constraints over matching substitutions. Our formalisation is given in Theorem 5.1 and is complete in the following sense: subsumption resolution can be applied over (L, M) iff each constraint of Theorem 5.1 is satisfiable. Application of subsumption reso-

lution is tested via satisfiability checking over our constraints from Theorem 5.1. Encodings of our subsumption resolution constraints are given in Section 5.

Building the conclusion. If a model is found for the constraints encoding subsumption resolution, the conclusion $M \setminus \{m'\}$ of SR is built using the model.

5 Subsumption Resolution and SAT Encodings

As mentioned in Section 4, we turn the application of subsumption resolution SR over (L, M) into the satisfiability checking problem of Algorithm 1. We give our formalisation of SR in Theorem 5.1, followed by two encodings to SAT (Section 5.1–5.2) and adjustments to subsumption (Section 5.3).

Theorem 5.1 (Subsumption Resolution Constraints). *Clauses M and L are the main and side premise, respectively, of an instance of the subsumption resolution rule SR iff there exists a substitution σ that satisfies the following four properties:*

$$\textbf{\textit{existence}} \qquad \exists i\, j.\, \sigma(l_i) = \neg m_j \tag{3}$$

$$\textbf{\textit{uniqueness}} \qquad \exists j'.\, \forall i\, j.\, \big(\sigma(l_i) = \neg m_j \Rightarrow j = j'\big) \tag{4}$$

$$\textbf{\textit{completeness}} \qquad \forall i.\, \exists j.\, \big(\sigma(l_i) = \neg m_j \vee \sigma(l_i) = m_j\big) \tag{5}$$

$$\textbf{\textit{coherence}} \qquad \forall j.\, \big(\exists i.\, \sigma(l_i) = m_j \Rightarrow \forall i.\, \sigma(l_i) \neq \neg m_j\big) \tag{6}$$

We relate these constraints to the definition of subsumption resolution (2). The **existence** property (3) requires a literal m_j in M such that a literal l_i of L can be matched to $\neg m_j$, ensuring the existence of the resolution literal in SR. **Uniqueness** (4) asserts that the resolution literal m_j of SR is unique, required because SR performs only a single resolution step. **Completeness** (5) requires each literal in L be matched either to the complement of a resolution literal, or to a literal in M. Since each (complementary) literal in L is matched to one (resolution) literal of M, the completeness property ensures that the conclusion of SR subsumes M. Finally, **coherence** (6) states that all literals in M must be matched by literals in L with uniform polarity. This implies that all literals of L other than the resolution literal are present in the conclusion of SR. We note that these constraints can be used to recreate Example 3.1.

Example 5.1. The clause pair (L_2, M_2) of Figure 1 does not satisfy the uniqueness property: both the match between $p(x_1)$ and $\neg p(y)$ and the match between $q(x_2)$ and $\neg q(c)$ are negative and so no substitution can satisfy all constraints simultaneously. Therefore, subsumption resolution cannot be applied over (L_2, M_2).

Example 5.2. The clause pair (L_3, M_3) violates the coherence property for all possible σ, since a negative map from $p(x_1)$ to $\neg p(y)$ cannot coexist with a positive map from $\neg p(x_2)$ to $\neg p(y)$. Subsumption resolution cannot be performed over (L_3, M_3).

5.1 Direct SAT Encoding of Subsumption Resolution

We present our encoding of subsumption resolution constraints as a SAT problem, allowing us to use Algorithm 1 for deciding the application of `SR`. In the sequel we consider the clauses L, M as in Theorem 5.1.

Compatibility. We introduce indexed propositional variables $b^+_{i,j}$ and $b^-_{i,j}$ to represent $\sigma(l_i) = m_j$ and $\sigma(l_i) = \neg m_j$ respectively, which we use to track compatible matching substitutions between literals of L and M. More precisely, a propositional variable is created if and only if the corresponding match is possible (i.e., in the formulas below, if no match exist, replace the corresponding propositional variable by $\bot$). As it is not possible to have simultaneously a substitution $\sigma_{i,j}(l_i) = m_j$ and $\sigma_{i,j}(l_i) = \neg m_j$, we also write $b_{i,j}$ to mean either $b^+_{i,j}$ or $b^-_{i,j}$ when the polarity of the match is irrelevant. Following Section 4, the variables are bound to their substitutions:

SAT-based compatibility $$\bigwedge_i \bigwedge_j [b_{i,j} \Rightarrow \sigma_{i,j} \subseteq \sigma] \tag{7}$$

`SR` **constraints.** Constraints (3)–(6) of Theorem 5.1 employ *bounded* quantification over the finite number of literals in L, M. Expanding these quantifiers over their respective domains, we translate them into the following SAT formulas:

SAT-based existence $$\bigvee_i \bigvee_j b^-_{i,j} \tag{8}$$

SAT-based uniqueness $$\bigwedge_j \bigwedge_i \bigwedge_{i' \geq i} \bigwedge_{j' > j} \neg b^-_{i,j} \vee \neg b^-_{i',j'} \tag{9}$$

SAT-based completeness $$\bigwedge_i \bigvee_j b_{i,j} \tag{10}$$

SAT-based coherence $$\bigwedge_j \bigwedge_i \bigwedge_{i'} \neg b^+_{i,j} \vee \neg b^-_{i',j} \tag{11}$$

`SR` **as SAT problem.** Based on the above, application of subsumption resolution is decided by the satisfiability of (7)∧(8)∧(9)∧(10)∧(11). This SAT formula extended with substitutions represents the result of $encodeConstraint()$ in Algorithm 1 and is used further in Algorithm 3. When this formula is satisfiable, we construct the substitution σ required for `SR` by

$$\sigma = \bigcup \{\sigma_{i,j} \mid b_{i,j} = \top\}.$$

From the model of the SAT solver, we extract the first literal $b^-_{i,j}$ assigned $\top$, from which we conclude that the jth literal in M is the resolution literal of `SR`. As such, application of `SR` over L and M results in replacing M by $M \setminus \{m_j\}$.

Remark 5.1. Implicitly, all l_i literals are mapped to at most one literal m_j. Indeed, if there were several literals m_j such that $\sigma(l_i) = m_j$ or $\sigma(l_i) = \neg m_j$, then either the respective matches are not compatible (guarded by the compatibility property (7)), there are identical literals in M, or M is a tautology (which is not allowed).

Remark 5.2. While we defined $b_{i,j}$ to be true if, and *only* if, $\sigma_{i,j} \subseteq \sigma$, we only encode the sufficient condition $b_{i,j} \Rightarrow \sigma_{i,j} \subseteq \sigma$. The completeness property (10) together with Remark 5.1 state that each l_i must have exactly one match to some m_j or $\neg m_j$. Therefore, if $\sigma_{i,j} \subseteq \sigma$ then the respective $b_{i,j}$ must be true and the condition also becomes necessary: $b_{i,j} \Leftarrow \sigma_{i,j} \subseteq \sigma$.

Example 5.3. Consider the pair (L_1, M_1) of Figure 1. The match set ms of Algorithm 1 is:

$$\sigma_{i,j} = \begin{bmatrix} \{x_1 \mapsto g(y_1), x_2 \mapsto c\} & \{x_1 \mapsto f(c), x_2 \mapsto e\} \\ \bot & \{x_1 \mapsto c, x_2 \mapsto e\} \end{bmatrix} \quad P_{i,j} = \begin{bmatrix} + & - \\ & - \end{bmatrix}$$

Since $\sigma_{2,1}$ is incompatible with any substitution, $b_{2,1} = \bot$ need not be defined. This also allows to disregard SAT clauses that are trivially satisfied. The existence (8) and completeness (10) properties cannot have empty clauses: this is easily detected while filling the match set, and the instance of `SR` is pruned. Adding falsified literals in these constraints is unnecessary. The uniqueness (9) and coherence (11) properties have only negative polarity literals and therefore there is no need to add clauses containing $b_{2,1}$. In light of the previous comment, we use variables $b^+_{1,1}$, $b^-_{1,2}$ and $b^-_{2,2}$ and encode `SR` using the following constraints:

$$\begin{aligned}
&b^+_{1,1} \Rightarrow \{x_1 \mapsto g(y_1), x_2 \mapsto c\} \subseteq \sigma && \textbf{SAT-based compatibility} \text{ of } b^+_{1,1} \\
&b^-_{1,2} \Rightarrow \{x_1 \mapsto f(c), x_2 \mapsto e\} \subseteq \sigma && \textbf{SAT-based compatibility} \text{ of } b^-_{1,2} \\
&b^-_{2,2} \Rightarrow \{x_2 \mapsto c, x_3 \mapsto e\} \subseteq \sigma && \textbf{SAT-based compatibility} \text{ of } b^-_{2,2} \\
&b^-_{1,2} \vee b^-_{2,2} && \textbf{SAT-based existence} \\
&b^+_{1,1} \vee b^-_{1,2} && \textbf{SAT-based completeness}, i = 1 \\
&b^-_{2,2} && \textbf{SAT-based completeness}, i = 2
\end{aligned}$$

The uniqueness (9) and coherence (11) properties are trivial here because the problem is simple: all $b^-_{i,j}$ have the same j, and no literal m_j can be mapped with different polarities. By using SAT solving from Algorithm 1 over the above SAT constraints, we obtain the SAT model $b^+_{1,1} \wedge \neg b^-_{1,2} \wedge b^-_{2,2}$, with $b^-_{2,2}$ the first literal assigned $\top$ with negative polarity. The application of `SR` over (L_1, M_1) yields the conclusion $M \setminus \{m_2\} = p(g(y_1), c)$, replacing M.

5.2 Indirect SAT Encoding of Subsumption Resolution

SAT-based formulas (9) and (11) may yield many constraints, with worst-case complexity $O(|L|^2|M|^2)$. In practice such situations rarely occur, since the match set ms is sparsely populated. Nevertheless, to alleviate this worst-case complexity, we further constrain the approach of Section 5.1. We introduce structuring propositional variables c_j such that c_j is $\top$ iff there exists a literal l_i with $\sigma(l_i) = \neg m_j$, which we encode as:

SAT-based structurality $$\bigwedge_j \left[\neg c_j \vee \bigvee_i b^-_{i,j}\right] \wedge \bigwedge_j \bigwedge_i \left(c_j \vee \neg b^-_{i,j}\right) \quad (12)$$

`SR` **as revised SAT problem.** While the compatibility property (7) remains unchanged, the `SR` constrains of Theorem 5.1 are revised as given below.

SAT-based revised existence $$\bigvee_j c_j \quad (13)$$

SAT-based revised uniqueness $$AtMostOne(\{c_j, j = 1, ..., |M|\}) \quad (14)$$

SAT-based revised completeness $$\bigwedge_i \bigvee_j b_{i,j} \quad (15)$$

SAT-based revised coherence $$\bigwedge_j \bigwedge_i \left(\neg c_j \vee \neg b^+_{i,j}\right) \quad (16)$$

Similarly to Section 5.1, application of subsumption resolution is decided via Algorithm 1 by checking satisfiability of (7) $\wedge$ (12) $\wedge$ (13) $\wedge$ (14) $\wedge$ (15) $\wedge$ (16). Using the above SAT formula as the result of *encodeConstraint*() in Algorithm 1, the worst-case behaviour is eliminated in exchange for $O(|M|)$ propositional variables, c_j. While the direct encoding of Section 5.1 is more efficient on small problems as it requires fewer variables and constraints, the indirect encoding of this section is expected to behave better on larger problems (see Section 7).

Remark 5.3. Note that the uniqueness property (14) is handled via *AtMostOne* constraints, based on the approach of [10]. If a variable c_j is set to $\top$, then our SAT *solver* in Algorithm 1 infers that all other variables $c_{j'}$ are set to $\bot$.

Example 5.4. Consider again the clause pair (L_1, M_1) of Figure 1. Compared to Example 5.3, our revised encoding of `SR` requires one additional variable c_2, as m_2 in Example 5.3 is used with negative polarity. The revised constraints are:

$b^+_{1,1} \Rightarrow \{x_1 \mapsto g(y_1), x_2 \mapsto c\} \subseteq \sigma$	**SAT-based compatibility** of $b^+_{1,1}$
$b^-_{1,2} \Rightarrow \{x_1 \mapsto f(c), x_2 \mapsto e\} \subseteq \sigma$	**SAT-based compatibility** of $b^-_{1,2}$
$b^-_{2,2} \Rightarrow \{x_2 \mapsto c, x_3 \mapsto e\} \subseteq \sigma$	**SAT-based compatibility** of $b^-_{2,2}$
$\neg c_2 \vee b^-_{1,2} \vee b^-_{2,2}$	**SAT-based structurality** of c_2
$c_2 \vee \neg b^-_{1,2}$	**SAT-based structurality** of c_2
$c_2 \vee \neg b^-_{2,2}$	**SAT-based structurality** of c_2
c_2	**SAT-based revised existence**
$AtMostOne(\{c_2\})$	**SAT-based revised uniqueness**
$b^+_{1,1} \vee b^-_{1,2}$	**SAT-based revised completeness**, $i = 1$
$b^-_{2,2}$	**SAT-based revised completeness**, $i = 2$

The SAT solver returns $b^+_{1,1} \wedge \neg b^-_{1,2} \wedge b^-_{2,2} \wedge c_2$ as a solution to the above SAT problem, from which the application of SR yields a similar result to that of Example 5.3.

Remark 5.4. We note that our method naturally supports commutative predicates, such as equality. Let $\simeq$ denote object-level equality. Suppose we have literals $l_i := a \simeq b$ and $m_j := c \simeq d$. Two propositional variables with associated matching substitutions $\sigma_{i,j}$ and $\sigma'_{i,j}$ are introduced, where $\sigma_{i,j}$ matches $a \simeq b$ against $c \simeq d$ and $\sigma'_{i,j}$ matches $a \simeq b$ against $d \simeq c$. If zero or one matches exist, then the problem behaves exactly like the non-symmetric case. If both matches exist, then $\sigma_{i,j}$ and $\sigma'_{i,j}$ must be incompatible: otherwise, c and d would be identical terms and the trivial literal m_j would have been eliminated. Therefore, our SAT-based encodings for subsumption resolution do not need to be adapted and behave as expected.

5.3 SAT Constraints for Subsumption

In the new framework of Algorithm 1, the formulation suggested by [10] was adjusted to work with subsumption resolution. Algorithm 1 needs very little adaptation for subsumption: the *encodeConstraint*() method uses the encoding below, and the conclusion needs not be built as only the satisfiability of the formulas is relevant. The re-written SAT encoding becomes:

subsumption compatibility $$\bigwedge_i \bigwedge_j \left(b^+_{i,j} \Rightarrow \sigma_{i,j} \subseteq \sigma\right) \tag{17}$$

subsumption completeness $$\bigwedge_i \bigvee_j b^+_{i,j} \tag{18}$$

multiplicity conservation $$\bigwedge_j AtMostOne(\{b^+_{i,j}, i = 1, ..., |L|\}) \tag{19}$$

Note that the set of propositional variables used in our SAT-based formulas (17)–(19) encoding subsumption is a subset of the variables used by our SAT-based subsumption resolution constraints.

Pruning for subsumption. The pruning technique described in Section 4 can be adapted into a stronger form for subsumption. In this case, we will check for multi-set inclusion between multi-sets of (predicates, polarity) pairs.

6 SAT-based Subsumption Resolution in Saturation

In this section we discuss the integration of our SAT-based subsumption resolution approach within saturation-based proof search.

Forward/backward simplifications. For the purpose of efficient reasoning, saturation algorithms use two main variants of simplification inferences implementing redundancy. *Forward* simplifications are applied on a newly generated

Algorithm 2 SAT-based subsumption in saturation

```
ms ← createMatchSet()
solver ← createSatSolver(ms)
procedure Subsumption(L, M)
    F_S, F_SR ← pruned(L, M)
    ▷ F_S (resp. F_SR) gets true if subsumption (resp. subsumption resolution) cannot succeed
    fillMatchSet(ms, L, M)        ▷ Build the whole match set, and update F_S and F_SR
    if F_S then                   ▷ subsumption cannot be applied
        return NoSubsumption
    encodeConstraints(solver, ms) ▷ SAT-constraints of Section 5.3
    if solver.solve() is SAT then
        return Subsumed
    else
        return NoSubsumption
```

Algorithm 3 SAT-based subsumption resolution in saturation
– with subsumption already set up via Algorithm 2

```
procedure SubsumptionResolution(L, M)
                                  ▷ upon Algorithm 2 failing to subsume
                                  ▷ the match set is already set up
    if F_SR then
        return NoSubsumptionResolution
    encodeConstraints(solver, ms) ▷ SAT constraints of Section 5.1 or Section 5.2
    if solver.solve() is SAT then
        return buildConclusion(solver.getSolution(), M)   ▷ conclusion of subsumption resolution
    return NoSubsumptionResolution
```

clause M to check whether M can be simplified by an existing clause L. *Backward* simplifications use a newly generated clause L to check whether L can simplify existing clauses M. Backward simplification tends to be more expensive.

SAT-based subsumption resolution in saturation. Since subsumption is a stronger form of simplification, subsumption is checked before subsumption resolution. This means that subsumption resolution is applied only if subsumption fails for all candidate premises. We integrate Algorithm 1 within saturation so that it is used both for subsumption and subsumption resolution.

Algorithms 2–3 display a variation of the integration of our SAT-based approach for checking subsumption resolution during saturation. Since most of the setup of subsumption is also required for subsumption resolution, both simplification rules are set up at the same time. As such, whenever turning to subsumption resolution, the same match set ms from Algorithm 2 can be reused, while also taking advantage of pruning steps performed during subsumption.

We modified the forward simplification algorithm as described in Algorithm 4. In this new setting, checking the same pair (L, M) for subsumption directly fol-

Algorithm 4 Forward simplification with SAT-based subsumption resolution

```
procedure ForwardSimplify(M, F)
    M* ← NoSubsumptionResolution
    for L ∈ F \ {M} do
        if subsumption(L, M) is Subsumed then            ▷ using Algorithm 2
            F ← F \ {M}
            return ⊤                                ▷ M is subsumed and removed
        if M* = NoSubsumptionResolution then
            M* ← subsumptionResolution(L, M)             ▷ using Algorithm 3
    if M* ≠ NoSubsumptionResolution then
        F ← F \ {M} ∪ {M*}     ▷ M* is the conclusion of subsumption resolution
between L and M
        return ⊤
    return ⊥
```

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()
    writeInFile(e − s)
    r′ ← Oracle(M, F)
    checkCoherence(r, r′)                                  ▷ Empiric check
    return r′
```

lowed by subsumption resolution enables us to use Algorithms 2–3 efficiently. Algorithm 4 pays the price of checking subsumption resolution even if subsumption may succeed, but in practice inefficiencies in this respect are seen rarely.

Role of indices. When applying inferences that require terms or literals to unify or match, modern automated first-order theorem provers typically use *term indices* [9] to consider only viable candidates within the set of clauses. Subsumption and subsumption resolution is no exception. Our testbed system Vampire currently uses a substitution tree to index clauses for matching by their literals (Section 7).

7 Implementation and Experiments

We implemented and integrated our SAT-based subsumption resolution approach in the saturation-based first-order theorem prover Vampire [6][3].

Versions compared. We use following versions of Vampire in our evaluation:
- Vampire_M is the *master* branch without SAT-based subsumption resolution;

[3] https://github.com/vprover/vampire/tree/robin_c-subsumption_resolution

- Vampire$_I$ is the SAT-based subsumption resolution with the *indirect* encoding of Section 5.2 and a standard forward simplification algorithm with Algorithm 1 — that is, Algorithm 4 is not used here;
- Vampire$_I^*$ uses the *indirect* encoding with Algorithms 2–4;
- Vampire$_D^*$ uses the *direct* encoding of Section 5.1 and Algorithms 2–4.

Experimental setting. To evaluate our work, we used the examples of the TPTP library (version 8.1.2) [15]. In our evaluation, 24 926 problems were used out of the 25 257 TPTP problems; the remaining problems are not supported by Vampire (e.g., problems with both higher-order operators and polymorphism).

Our experimental evaluation was done on a machine with two 32-core AMD Epyc 7502 CPUs clocked at 2.5 GHz and 1006 GiB of RAM (split into 8 memory nodes of 126 GiB shared by 8 cores). Each benchmark problem was run with the options `-sa otter -t 60`, meaning that we used the Otter saturation algorithm [7] with a 60-second time-out. We use the Otter strategy because it is the most aggressive in terms of simplification and therefore runs the most subsumption resolutions. We turned off the AVATAR framework (`-av off`) in order to have full control over SAT-based reasoning in Vampire.

Evaluation setup. Our evaluation process is summarised in Algorithm 5, incorporating the following notes.

- 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 3–4). 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 4 may be up to 25% faster on second execution, due to cache effects.

For the reasons above, we decided to measure the run time of a complete execution of Algorithm 4. To prevent the branches to change, an `Oracle` is used to choose the path to follow. The `Oracle` is based on our indirect SAT encoding (Vampire$_I^*$). This way, the same computation graph is used for all evaluated methods. To prevent cache preheating, we run the `Oracle` after the respective evaluated method. This way the cache is in a normal state for the evaluated method. To measure the run time of Algorithm 4, a `Wrapper` method was built on top of the `Forward Simplify` procedure of Algorithm 4. This `Wrapper` replaces the `Forward Simplify` loop in Vampire with minimal changes to the code. To empirically verify the correctness of our results, we used the `Wrapper` to compare the result of the evaluated method with the result of the `Oracle`.

Experimental details and analysis. Figure 2 lists the cumulative instances solved by the respective Vampire versions, highlighting the strength of forward simplifications for effective saturation.

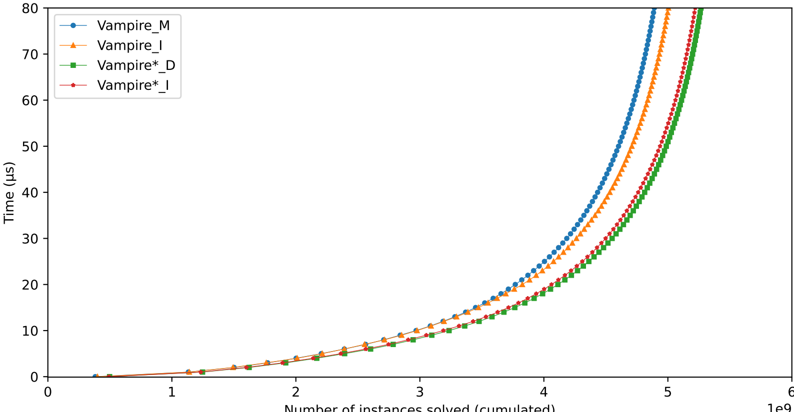


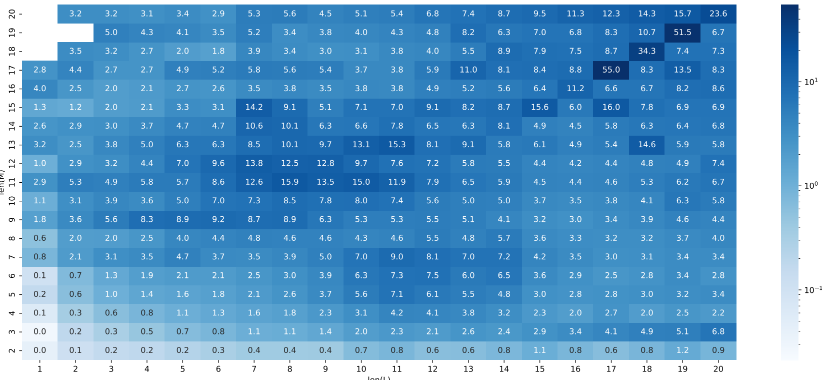
Fig. 2: Cumulative instances of applying subsumption resolution, using the TPTP examples. A point (n, t) on the graph means that n forward simplify loops were executed in less than t μs. The flatter the curve, the faster the VAMPIRE version is.

Prover	Average	Std. Dev.	Speedup
VAMPIRE$_M$	42.63 μs	1609.06 μs	0 %
VAMPIRE$_I$	40.13 μs	1554.52 μs	6.2 %
VAMPIRE$^*_D$	34.39 μs	1047.85 μs	23.9 %
VAMPIRE$^*_I$	34.55 μs	250.25 μs	23.4 %

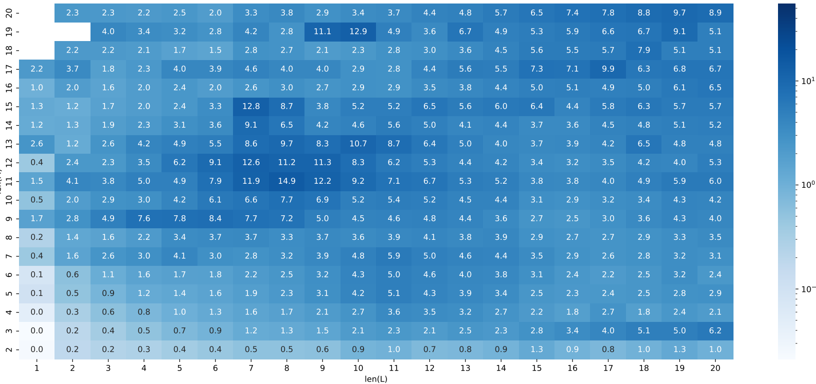
Table 1: Average time spent in the `Forward Simplify` loop. VAMPIRE$^*_D$ is the fastest method, closely followed by the VAMPIRE$^*_I$. However, the indirect encoding is much more stable and has a lower variance.

Remark 7.1. Our experimental summary in Figure 2 shows that the total number of `Forward Simplify` loops ran in 60 seconds. However, the average and standard deviation were computed only on the intersection of the problems solved. That is, only the `Forward Simplify` loops finished by all the methods are taken into account. Otherwise, if a hard problem is solved in, for instance, 1 000 000 μs by one method, and times out for another, the average for the better would increase a lot, but the weaker method would not be penalised. Table 1 summarises the average solving time of our evaluation.

Comparison of encodings. We correlated the constraint building and SAT solving time with the length of clauses, using the different encodings of Sections 5.1–5.2. Figure 3 shows that on larger clauses, the average computation time increases faster for the direct encoding than for the indirect encoding.



(a) Average time (μs) for creating/solving direct encoding constraints (Section 5.1).



(b) Average time (μs) for creating/solving indirect encoding constraints (Section 5.2).

Fig. 3: Average time (μs) spent on the creating and solving SAT-based subsumption resolution constraints.

Prover	**Total Solved**	**Gain/Loss**
Vampire$_M$	10 555	baseline
Vampire$^*_D$	10 667	(+141, −29)
Vampire$^*_I$	10 658	(+133, −30)

Table 2: Number of TPTP problems solved by the considered versions of Vampire. 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$_M$.

Experimental summary. Our experiments show that $\textsc{Vampire}^*_I$ yields the most stable approach for SAT-based subsumption resolution (Table 1), especially when it comes on solving large instances (Figure 3). Our results demonstrate the superiority of SAT-based subsumption resolution used with forward simplifications in saturation (e.g., $\textsc{Vampire}^*_D$ and $\textsc{Vampire}^*_I$), as concluded by Table 2.

8 Conclusion

We advocate SAT solving for improving saturation-based first-order theorem proving. We encode powerful simplification rules, in particular subsumption resolution, as SAT problems, triggering eager and efficient reasoning steps for the purpose of keeping proof search small. Our experiments with Vampire showcase the benefit of SAT-based subsumption. In the future, we aim to further extend simplification rules with SAT solving, in particular focusing on subsumption demodulation for equality reasoning [3].

Acknowledgements. This work is the result of a research internship hosted at TU Wien and defended at the University of Liège. The authors would like to thank Pascal Fontaine for valuable discussions and comments. We acknowledge funding from the ERC Consolidator Grant ARTIST 101002685 and the FWF SFB project SpyCoDe F8504.

References

1. Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *J. Log. Comput.*, 4(3):217–247, 1994.
2. Leo Bachmair and Harald Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 19–99. Elsevier and MIT Press, 2001.
3. Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption Demodulation in First-Order Theorem Proving. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 297–315. Springer, 2020.
4. Deepak Kapur and Paliath Narendran. NP-Completeness of the Set Unification and Matching Problems. In *IJCAR*, pages 489–495, 1986.
5. Konstantin Korovin. Inst-Gen — A Modular AApproach to Instantiation-Based Automated Reasoning. *Programming Logics: Essays in Memory of Harald Ganzinger*, pages 239–270, 2013.
6. Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, pages 1–35, 2013.
7. William McCune and Larry Wos. Otter— the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18:211–220, 1997.
8. Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

9. I. V. Ramakrishnan, R. Sekar, and Andrei Voronkov. Term Indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1853–1964. Elsevier and MIT Press, 2001.
10. Jakob Rath, Armin Biere, and Laura Kovács. First-Order Subsumption via SAT Solving. In *FMCAD*, page 160, 2022.
11. Giles Reger and Martin Suda. The Uses of SAT Solvers in Vampire. In *Vampire Workshop*, pages 63–69, 2015.
12. Giles Reger and Martin Suda. Global Subsumption Revisited (Briefly). In *Vampire @ IJCAR*, pages 61–73, 2016.
13. Stephan Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, pages 45–67, 2013.
14. Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, Higher, Stronger: E 2.3. In *CADE*, pages 495–507, 2019.
15. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.
16. Tanel Tammet. Towards Efficient Subsumption. In *CADE*, pages 427–441, 1998.
17. Andrei Voronkov. AVATAR: The Architecture for First-Order Theorem Provers. In *CAV*, pages 696–710, 2014.
18. Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In *CADE*, pages 140–145, 2009.