Formal Methods in Computer-Aided Design 2022

First-Order Subsumption via SAT Solving

Jakob Rath*, Armin Biere†, Laura Kovács*

*TU Wien, Vienna, Austria
{jakob.rath,laura.kovacs}@tuwien.ac.at

†University of Freiburg, Freiburg im Breisgau, Germany
biere@cs.uni-freiburg.de

***Abstract*—Automated reasoners, such as SAT/SMT solvers and first-order provers, are becoming the backbones of applications of formal methods, for example in automating deductive verification, program synthesis, and security analysis. Automation in these formal methods 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, 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 integration of a dedicated SAT solver provides a remedy towards efficient implementation of first-order subsumption and related rules, and thus further increasing scalability of first-order theorem proving towards applications of formal methods. Our experimental results demonstrate that, by using a tailored SAT solver within first-order reasoning, we gain a large speed-up in state-of-the-art benchmarks.**

***Index Terms*—first-order subsumption, multi-literal matching, automated theorem proving, satisfiability checking**

I. Introduction

Most formal verification approaches use automated reasoners in their backend to, for example, discharge verification conditions [22], [10], [15], produce/block counter-examples [20], [29], [1], or enforce security and privacy properties [30], [25], [4], [32]. All these approaches crucially depend on the efficiency of the underlying reasoning procedures, ranging from SAT/SMT solving [6], [12], [3] to first-order proving [41], [21], [34], [11]. In this paper we focus on automated first-order theorem proving with the aim of improving efficiency towards proving first-order (program) properties.

The leading concept behind the proof-search algorithms used by state-of-the-art first-order theorem provers is saturation [34], [21]. 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 [9], can be considered examples of saturation algorithms. The key idea behind saturation-based proof search is to reduce the problem of proving validity of a first-order formula A to the problem of establishing unsatisfiability of $\neg A$ by using a sound inference system, most commonly using the superposition inference system [28]. That is, instead of proving A, we refute $\neg A$, by selecting and applying inferences from the superposition calculus. In this paper, we focus on saturation algorithms using the superposition calculus.

Saturation with Redundancy. During saturation, the first-order prover keeps a set of *usable clauses* $C_1, \dots C_k$, 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 non-negative integers; and *reverse* is a unary function symbol reversing a list. As such, clause (1) asserts that the reverse of a list x of non-negative integers is also a list of non-negative 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* [31], 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

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_22

This article is licensed under a Creative Commons Attribution 4.0 International License

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 be used to identify the application of simplification rules during saturation, improving thus the efficiency of saturation algorithms implementing the superposition calculus extended with simplification rules*, as discussed next.

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 [2].

The intuition behind 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 [35]: (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 [27], [33], [35]. 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 [19]. 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 may remain problematic (due to NP-hardness). For example, while profiling subsumption in the world-leading theorem prover VAMPIRE [21], we observed subsumption applications, and in particular calls to the literal-matching algorithm of step (ii), that consume more than 20 seconds 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*.

Our Contributions. In this paper, we bring the following main contributions.

(1) We propose a *SAT-based encoding for capturing potential applications of subsumption* in first-order theorem proving (Section III). A solution to our SAT-based encoding gives a concrete application of subsumption, allowing the first-order prover to apply that instance of subsumption as a simplification rule during saturation. Our encoding uses so-called substitution constraints to formalize matching of literals within the premises (i.e., subset relation among literals of premises). Our encoding can be extended to other simplification rules, in particular when applying simplifications using the combination of subsumption with binary resolution (i.e., subsumption resolution).

(2) We introduce a *lean SAT solving approach tailored to substitution constraints*, by adjusting unit propagation and conflict resolution towards efficient handling of such constraints. (Section IV). We introduce a tailored encoding of substitution constraints in SAT solving, advocating the direct use of our SAT solver for deciding application of subsumption within first-order proving.

(3) We implemented our SAT-based subsumption approach as a new SAT solver in the VAMPIRE theorem prover (Section V). We empiricially evaluate our approach on the standard benchmark library TPTP (Section VI). Our experiments demonstrate that using SAT solving for deciding and applying subsumption brings clear improvements in the saturation process of first-order proving, for example improving the (time) performance of the prover by a factor of 2.

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

II. Preliminaries

Let $\mathcal{V}$ denote a countably infinite set of *first-order* variables. We consider standard multi-sorted first-order logic with variables $\mathcal{V}$, and support all standard boolean connectives (see later) and quantifiers in the language. Throughout the paper, we write x, y, z for *first-order* variables, c, d for constants, f, g for function symbols, and p, q 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 just atoms, are predicates applied to terms. Atoms and negated atoms are also called first-order *literals*, and denoted by L, M. First-order *clauses*, or simply just clauses, are disjunctions of literals, denoted by C, D. All our notation 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 C being $L_1 \vee L_2 \vee \ldots \vee L_n$ is considered to be the multiset $\{L_1, L_2, \ldots, L_n\}$. For example, the clause $p \vee \neg q \vee p$ is the multiset $\{p, \neg q, p\}$.

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 function $\sigma\colon \mathcal{V} \to \mathcal{T}$ such that $\sigma(x) \neq x$ only for finitely many $x \in \mathcal{V}$. The function σ is extended to arbitrary expressions E by simultaneously replacing each variable x in E by $\sigma(x)$. We say an expression E_1 can be matched to expression E_2 if there exists a substitution σ such that $\sigma(E_1) = E_2$.

Saturation and Subsumption. Most first-order theorem provers, see e.g. [41], [21], [34], implement saturation with redundancy, using the superposition calculus [2]. A clause C subsumes a clause D iff there exists a substitution σ such that $\sigma(C) \subseteq D$, where C and D are treated as multisets of literals. *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 [21], [34], also known as contextual literal cutting or self-subsuming resolution, is the combination of subsumption with binary resolution; and subsumption demodulation [16] results from combining subsumption with demodulation/rewriting.

SAT Solving. Let $\mathcal{B}$ be a countably infinite set of *boolean* variables. We denote boolean variables by b, possibly with indices. We use the standard boolean connectives $\wedge$, $\vee$, $\rightarrow$, $\neg$, and write $\top$ for the boolean constant *true* as well as $\bot$ for the boolean constant *false*. A boolean *literal*, denoted l, is a variable b or its negation $\neg b$. A boolean *clause* is a disjunction of literals. As before, we drop the qualifier *boolean* when it is clear from the context.

Modern SAT solvers are based on conflict-driven clause learning (CDCL) [24], with the core procedures *decide*, *unit-propagate*, and *resolve-conflict*. The 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-literals scheme [26] 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 0, the conflict follows unconditionally from the input clauses and the solver returns "unsatisfiable" (UNSAT). Otherwise, by analyzing how the literals in the conflicting clause have been assigned, the solver may derive and learn a conflict lemma, undo some decisions, and continue solving.

III. Substitution Constraints and Subsumption

Recall that a first-order clause C subsumes a clause D iff there exists a substitution σ such that $\sigma(C) \subseteq D$, where $\subseteq$ is to be understood as multiset inclusion. In what follows, we refer by *clausal subsumption* between C and D to the case when clause C subsumes clause D. Similarly, *literal subsumption* between L and M refers to the case when literal L subsumes literal M. We note that deciding literal subsumption, that is whether a literal L subsumes a literal M, can be done in almost linear time, by constructing a substitution (if it exists) σ s.t. $\sigma(L) = M$; in this case, the value of $\sigma(x)$ is uniquely determined by L and M for each variable x occurring in L. However, when working with arbitrary, and not necessarily unit, clauses C, D, deciding clausal subsumption between C, D is NP-complete for the following reason: for each literal L_i of C, one of the literals M_{j_i} of D needs to be chosen in such a way that a substitution σ *simultaneously matches* each L_i with its respective M_{j_i}; that is, $\sigma(L_i) = M_{j_i}$ for all i. Towards addressing NP-completeness of clausal subsumption, in this section we introdude *substitution constraints* (Section III-A), allowing us to formulate clausal subsumption as a SAT problem over substitution constraints (Section III-B). Based on this SAT-encoding of subsumption, we further present an effective approach towards using subsumption in saturation in Section IV.

A. Substitution Constraints

We first introduce *substitution constraints* to be further used in deciding clausal subsumption.

Definition 1 (Substitution Constraints): A *substitution constraint* Γ is a partial function from $\mathcal{V}$ to $\mathcal{T}$, denoted as

$$(x_1, \ldots, x_k) \rhd (t_1, \ldots, t_k),$$

where $k \geq 0$, $x_i \in \mathcal{V}$ are pairwise different, and $t_i \in \mathcal{T}$. The set $\operatorname{dom}(\Gamma) := \{x_1, \ldots, x_k\}$ is called the *domain* of Γ. We further write $\Gamma(x_i) = t_i$ for $i \in \{1, \ldots, k\}$.

A substitution $\sigma\colon \mathcal{V} \to \mathcal{T}$ *satisfies the substitution constraint* Γ, written $\sigma \models \Gamma$, iff $\sigma(x_i) = t_i$ for all $i \in \{1, \ldots, k\}$.

Two substitution constraints Γ_1, Γ_2 are *compatible* if there exists a substitution σ that satisfies both Γ_1 and Γ_2, that is, if $\Gamma_1(x) = \Gamma_2(x)$ for all variables $x \in \operatorname{dom}(\Gamma_1) \cap \operatorname{dom}(\Gamma_2)$.

As already discussed, literal subsumption between two literals L and M can easily be determined (as there is only one literal, i.e. L, that needs to be matched, i.e. to M). The substitution constraint corresponding to the literal subsumption between L and M is denoted by $\Gamma(L, M)$ and is defined below.

Definition 2 (Substitution Constraints for Literals): Let L and M be two literals. If there exists a substitution σ such that $\sigma(L) = M$, the *substitution constraint* $\Gamma(L, M)$ *for literals L and M* is

$$\Gamma(L, M) \coloneqq (x_1, \dots, x_k) \rhd (t_1, \dots, t_k),$$

where $\mathcal{V}(L) = \{x_1, \dots, x_k\}$ and $\sigma(x_i) = t_i$ for all $i \in \{1, \dots, k\}$. Otherwise, L cannot be matched to M and the *substitution constraint* $\Gamma(L, M)$ *for literals L and M* is

$$\Gamma(L, M) \coloneqq \bot.$$

Example 1: Consider the following first-order literals:

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

We obtain the following substitution constraints:

$$\begin{aligned}
\Gamma(L_1, M_1) &= (x_1, x_2, x_3) \rhd (f(c), d, y_1) \\
\Gamma(L_1, M_2) &= (x_1, x_2, x_3) \rhd (f(d), c, c) \\
\Gamma(L_2, M_1) &= \bot \\
\Gamma(L_2, M_2) &= (x_2, x_4) \rhd (d, c)
\end{aligned}$$

The constraints $\Gamma(L_1, M_1)$ and $\Gamma(L_1, M_2)$ are incompatible, as these constraints map, for example, x_1 to different values. The constraints $\Gamma(L_1, M_1)$ and $\Gamma(L_2, M_2)$ are compatible, as both constraints require their only shared variable x_2 to be mapped to d.

To encode clausal subsumption, we need to combine substitution constraints using boolean connectives, and boolean variables. For this reason, we now define the semantics of boolean combinations of substitution constraints.

Definition 3 (Boolean Combination of Substitution Constraints): Let F be a formula using standard boolean connectives, whose atoms are boolean variables and substitution constraints. An interpretation $I = (\alpha, \sigma)$ for such a formula is a pair of a standard boolean assignment $\alpha \colon \mathcal{B} \to \{\top, \bot\}$ and a substitution $\sigma \colon \mathcal{V} \to \mathcal{T}$.

For a boolean variable b, we define $I \models b$ iff $\alpha(b) = \top$. For a substitution constraint Γ, we define $I \models \Gamma$ iff $\sigma \models \Gamma$. For formulas F with a top-level connective of $\wedge$, $\vee$, $\to$, or $\neg$, we define $I \models F$ inductively in the standard way. For boolean constants, $I \models \top$ and $I \not\models \bot$.

Remark 1: The formula F can also be translated into an SMT formula using the theory of equality and uninterpreted functions (EUF), where substitution constraints are replaced by conjunctions of equality literals. Let T denote the set of terms t appearing on the right-hand side of some substitution constraint in F. We then introduce fresh constant symbols $\{c_t \mid t \in T\}$, and replace each substitution constraint $\Gamma = (x_1, \dots, x_k) \rhd (t_1, \dots, t_k)$ in F by $x_1 = c_1 \wedge \dots \wedge x_k = c_k$. To obtain correct semantics of substitution compatibility, we also need to add

$$\bigwedge_{t,u \in T, t \neq u} c_t \neq t_u, \tag{2}$$

asserting that constants representing different terms in F cannot be equal.

However, for clausal subsumption in a first-order theorem prover, it is vital that the process of encoding subsumption in SAT, as well as the setting up of our SAT solver for handling this encoding are as lean as possible (see Section V). Hence, we did not employ a standard SMT solver with the EUF-based encoding discussed above, but instead opted to directly add support for substitution constraints to our SAT solver. The advantage of our SAT-based approach is that we use less boolean literals, and we avoid using all-different constraints for terms, such as (2).

B. SAT-Encoding of Clausal Subsumption

We now present our formalization to express clausal subsumption between clauses C and D as a SAT problem over substitution constraints. To this end, assume that clause C is $L_1 \vee L_2 \vee \dots \vee L_n$, whereas D is $M_1 \vee M_2 \vee \dots \vee M_m$. Recall that deciding whether C subsumes D reduces to the problem of deciding whether there exists a substitution σ such that $\sigma(C) \subseteq D$, where "$\subseteq$" denotes multiset inclusion (over multisets of literals).

For arbitrary literals L_i and M_j, deciding the existence of a substitution σ with $\sigma(L_i) = M_j$ can easily be done. Yet, for clausal subsumption we are left with the challenge of finding a substitution σ such that, for each L_i, we have one of the M_j such that $\sigma(L_i) = M_j$. To address this challenge, we introduce new boolean variables b_{ij} to encode possible matchings of L_i to M_j, given by $\sigma(L_i) = M_j$. Additionally, we use Definition 2 to derive the substitution constraints $\Gamma(L_i, M_j)$. Based on the boolean variables b_{ij} and substitution constraints $\Gamma(L_i, M_j)$, we formalize clausal subsumption between C and D by ensuring its three properties: (i) each literal L_i in C is matched to a literal M_j in D, (ii) the same substitution σ is used for each of these matchings, and (iii) $C\sigma \subseteq D$ is multiset inclusion. Our formalization of clausal subsumption between C and D is given as follows.

(i) We first define the following clauses, capturing that each literal L_i from C must be matched to (at least one) literal M_j of D:

$$\bigwedge_{1 \leq i \leq n} b_{i1} \vee b_{i2} \vee \dots \vee b_{im} \tag{3}$$

(ii) We connect the boolean variables b_{ij} to the substitution constraints $\Gamma(L_i, M_j)$ through the following clauses:

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m} b_{ij} \to \Gamma(L_i, M_j). \tag{4}$$

These clauses employ the substitution constraints $\Gamma(L_i, M_j)$ to ensure the same substitution σ is used for matching L_i and M_j simultaneously, for all i, j.

(iii) As clausal subsumption uses *multiset* inclusion over the respective multisets of literals of C and D, we encode the requirement that each literal of D may only be matched at most once:

$$\bigwedge_{1 \le j \le m} \text{AtMostOne}(b_{1j}, \dots, b_{nj}), \tag{5}$$

where $\text{AtMostOne}(b_{1j}, \dots, b_{nj})$ is true iff zero or one of $b_{1j}, \dots, b_{nj}$ are true.

Together, the constraints (3), (4), (5) fully capture clausal subsumption, yielding the following result.

Theorem 1 (Clausal Subsumption as SAT): Clausal subsumption between clauses C and D is given by the conjunction of (3), (4), and (5). That is, C subsumes D iff (3) $\wedge$ (4) $\wedge$ (5) is satisfiable.

Note that for deciding clausal subsumption between C and D, we only need to *establish satisfiability of* (3) $\wedge$ (4) $\wedge$ (5) in Theorem 1: one substitution σ such that $C\sigma \subseteq D$ is sufficient for deciding that C subsumes D, implying that D can be deleted from the set of usable clauses during saturation. Hence, while clausal subsumption (3) $\wedge$ (4) $\wedge$ (5) captures all substitutions σ for which $C\sigma \subseteq D$, for deciding whether C subsumes D we are interested to find only one satisfying instance of (3) $\wedge$ (4) $\wedge$ (5). As a result, application of clausal subsumption in saturation can be decided by solving the satisfiability of (3) $\wedge$ (4) $\wedge$ (5).

Example 2: Consider the literals defined in Example 1 and clauses $C = L_1 \vee L_2$ and $D = M_1 \vee M_2$. The encoding of clausal subsumption between C and D resulting from Theorem 1 is the conjunction of the following clauses:

$$\begin{aligned}
&b_{11} \vee b_{12}\\
&b_{21} \vee b_{22}\\
&b_{11} \rightarrow (x_1, x_2, x_3) \rhd (f(c), d, y_1)\\
&b_{12} \rightarrow (x_1, x_2, x_3) \rhd (f(d), c, c)\\
&b_{21} \rightarrow \bot\\
&b_{22} \rightarrow (x_2, x_4) \rhd (d, c)\\
&\neg b_{11} \vee \neg b_{21}\\
&\neg b_{12} \vee \neg b_{22}
\end{aligned}$$

This set of clauses is satisfiable, as witnessed by the model that assigns b_{11} and b_{22} to true, b_{12} and b_{21} to false, and $\sigma(x_1) = c$, $\sigma(x_2) = f(d)$, $\sigma(x_3) = y_1$, $\sigma(x_4) = c$. We conclude that the first-order clause C subsumes D.

Remark 2 (Subsumption Resolution): Our encoding of clausal subsumption can be adjusted to also decide the application of other simplification rules in saturation, when these rules implement variants of subsumption. To this end, we have extended the SAT encoding (3)$\wedge$(4)$\wedge$(5) of clausal subsumption to the inference rule *subsumption resolution*. In addition to clausal subsumption, subsumption resolution also uses instances of binary resolution. Hence, for finding substitutions σ such that subsumption resolution between clauses C and D can be applied (and D deleted from the set of usable clauses), we extended the clauses (3) $\wedge$ (4) $\wedge$ (5) with additional constraints capturing application of resolution, while also adjusting the encoding of (3) $\wedge$ (4) $\wedge$ (5) to set inclusion between literals of C and D (instead of multiset inclusion from subsumption).

Remark 3 (At-Most-One Constraints): We conclude this section by noting that a correct but naive solution to encode $\text{AtMostOne}(b_{1j}, \dots, b_{nj})$ in (5) would be the following:

$$\bigwedge_{1 \le i_1 < i_2 \le n} \neg b_{i_1 j} \vee \neg b_{i_2 j}. \tag{6}$$

More efficient encodings using at-most-one constraints (see, e.g., [13]) can be used instead of (6). In our work however, we opted to add direct support for at-most-one constraints when reasoning about (5) (see Section IV).

IV. Effective Subsumption via Lean SAT Solving

In Section III we showed that the application of subsumption, as an inference rule in saturation, can be reduced to the satisfiability problem of the formula (3) $\wedge$ (4) $\wedge$ (5) using substitution constraints (Theorem 1). In this section we describe our approach for solving (3) $\wedge$ (4) $\wedge$ (5).

A straightforward approach towards handling (3) $\wedge$ (4) $\wedge$ (5) could come with translating (3)$\wedge$(4)$\wedge$(5) into only propositional clauses; yet, such a translation would either require additional propositional variables to encode at-most-one constraints or would come with a quadratic number of propositional clauses [13]; similarly for substitution constraints.

Due to the particular distribution of subsumption instances (see Section V), the encoding must be lightweight to be practically feasible. To overcome the increase in propositional variables/clauses to be used for deciding clausal subsumption in an efficient manner, we support substitution constraints (4) and and at-most-one constraints (5) directly in SAT solving, and introduce a lean SAT solving approach tailored to subsumption properties. In particular, we adjust unit propagation and conflict resolution in CDCL-based SAT solving for handling propositional formulas with substitution constraints. This way, we integrate our lean SAT solving methodology directly into the saturation process of first-order proving (Section V), instead of interfacing first-order proving with an existing off-the-shelf SAT solver. Such a direct integration allows us to efficiently identify and apply subsumption during proof search (see Section VI).

a) Using Substitution Constraints in SAT Solving: For handling substitution constraints in clausal subsumption, we attach a substitution constraint $\Gamma(L_i, M_j)$ to each freshly introduced boolean variable b_{ij} in (3), which is equivalent to adding the constraint $b_{ij} \rightarrow \Gamma(L_i, M_j)$ of (4).

b) Unit Propagation with Substitution Constraints: Consider now the clauses $b_{ij} \rightarrow \Gamma(L_i, M_j)$ using substitution constraints, with $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$, from clausal subsumption (3) ∧ (4) ∧ (5). Semantically, these constraints are equivalent to the following set of binary clauses:

$$\begin{aligned}\{\neg b_{ij} \vee \neg b_{i'j'} \mid\ & i, i' \in \{1 \ldots n\}, j, j' \in \{1 \ldots m\},\\ & (i,j) \neq (i',j'),\\ & \exists x \in \operatorname{dom}(\Gamma(L_i, M_j)) \cap \operatorname{dom}(\Gamma(L_{i'}, M_{j'}))\\ & \text{s.t. } \Gamma(L_i, M_j)(x) \neq \Gamma(L_{i'}, M_{j'})(x)\},\end{aligned} \tag{7}$$

which intuitively encodes that no two incompatible substitution constraints may be true at the same time.

In our work, instead of creating the binary clauses of (7) explicitly, we introduce support for substitution constraints as an *additional (unit) propagator* in SAT solving: whenever a boolean variable b_{ij} is assigned to true, our SAT solver processes the associated bindings for the first-order variables from $\operatorname{dom}(\Gamma(L_i, M_j))$, and propagates all boolean variables $b_{i'j'}$ to false that are associated with conflicting bindings for variables $\operatorname{dom}(\Gamma(L_i, M_j)) \cap \operatorname{dom}(\Gamma(L_{i'}, M_{j'}))$; in other words, all $b_{i'j'}$ whose associated substitution constraints are incompatible with $\Gamma(L_i, M_j)$. This propagation is done exhaustively once b_{ij} is assigned to true and before standard unit propagation in SAT solving would be applied. Thus we ensure that no conflict can occur at this point: if there were a conflict, that would mean a $b_{i'j'}$ with conflicting bindings has already been assigned to true; in this case however, we would have already propagated b_{ij} to false when assigning $b_{i'j'}$. An exception in handling conflicts occurs with the initial propagation before starting the CDCL loop of SAT solving; in this case, we may get a conflict if two unit clauses with conflicting substitution constraints have been added, however, in that case the SAT solver is at decision level 0 and can terminate with reporting unsatisfiability (UNSAT) of (3) ∧ (4) ∧ (5).

c) Conflict Resolution with Substitution Constraints: During conflict resolution in our SAT engine, we proceed as if the binary clauses (7) were part of the clause database, i.e., as if the binary clause $\neg b_{ij} \vee \neg b_{i'j'}$ were the reason for propagating $b_{i'j'}$. Therefore we only need to store the literal b_{ij} as the reason for unit propgation. Substitution constraints during conflict resolution thus do not need specialized treatment in our SAT solving approach.

d) At-Most-One Constraints: During unit propagation and conflict resolution, our at-most-one constraints (5) are treated as if we had the corresponding binary clauses from (6), saving the overhead from creating additional clauses and variables.

Remark 4: While we presented our approach in the context of solving (3) ∧ (4) ∧ (5), our SAT solving approach naturally supports arbitrary boolean clauses and at-most-one constraints, as well as substitution constraints in the form $b \rightarrow \Gamma$ (where b is a boolean variable and Γ a substitution constraint).

V. SAT-Based Subsumption in First-Order Theorem Proving

We implemented our lean SAT-based approach of Section IV as a new extension to the theorem prover Vampire. While Vampire already implements highly optimized algorithms for checking subsumption, these algorithms are built on a standard, backtracking-based search procedure: using a static variable ordering and limited amount of unit propagation, without learning from conflicts. Hence, the full power of SAT-based reasoning with unit propagation and conflict resolution is not yet supported for subsumption. We overcome this limitation by integrating our SAT-based approach for clausal subsumption directly in Vampire. Our implementation consists of about 5000 lines of C++ code and is available at https://github.com/JakobR/vampire/tree/sat-subsumption.

a) Implementing Subsumption: When establishing satisfiability of (3) ∧ (4) ∧ (5), we can observe two different types of subsumption instances:

(i) easy subsumption instances, where not much SAT-based search is required (very few or even no decisions/conflicts), For such instances the overhead of setting up the clausal encoding of (3) ∧ (4) ∧ (5) largely determines the total running time of our SAT solver.

(ii) hard subsumption instances, whose application is determined by a significant number of unit propagation and/or conflict resolution steps in SAT solving.

We recall that the overall goal of our work is to improve subsumption checking in first-order theorem proving. For this, we complemented Vampire with a SAT-based approach to decide application of subsumption. Note that the majority of the subsumption instances encountered during a typical first-order proving attempt are of type (i), with instances of type (ii) appearing occasionally, depending on the input formula. Still, the total running time is often dominated by type (ii) instances, and these are the target of our SAT-based approach. We must however be careful to not become slower on type (i) instances, thus motivating our choice of a lean, dedicated SAT-solver embedded into Vampire.

In many of the trivial instances of (3) ∧ (4) ∧ (5), the unsatisfiabiliy (UNSAT) of these instances can be discovered already during the encoding of (3) ∧ (4) ∧ (5) (whenever an empty clause would be added). To save time on these instances, in our implementation we defer the construction of watch lists and other data structures until entering the solving loop of our SAT engine (if at all).

We note that the number of subsumption instances, especially easy ones of type (i), during first-order proving can become quite large, often in the order of millions of instances in a 60 s run of a theorem prover. Allocating and deallocating a new SAT solver instance for each SAT-based subsumption query can thus become expensive (see Section VI); therefore, in our implementation we keep the same solver instance around, and re-use it for different queries. In particular, we keep the memory for data structures (such as clause storage, watch lists, trail, and others), instead of reallocating it for each query.

b) Unit Propagation: To achieve efficient unit propagation, our SAT solver for clausal subsumption watches two literals of each clause [26]. However, for at-most-one constraints the situation is different. Consider the constraint $\text{AtMostOne}(l_1, \ldots, l_k)$ for some $k \geq 3$ (note that for $k \leq 2$ we either drop the constraint or add a binary clause instead). As soon as any l_i is assigned true, all l_j with $j \neq i$ must be false to avoid violating the constraint, and are propagated thus. Hence, the solver watches *all* literals of at-most-one constraints.

VI. Experiments

We evaluated our new SAT-based implementation for clausal subsumption in Vampire (see Section V). In our experiments, we were interested (i) to measure the performance improvements we gain through our approach, as well as (ii) to assess the advantage of re-using our SAT solver objects, and thus having our SAT solver directly integrated the first-order proving process of Vampire.

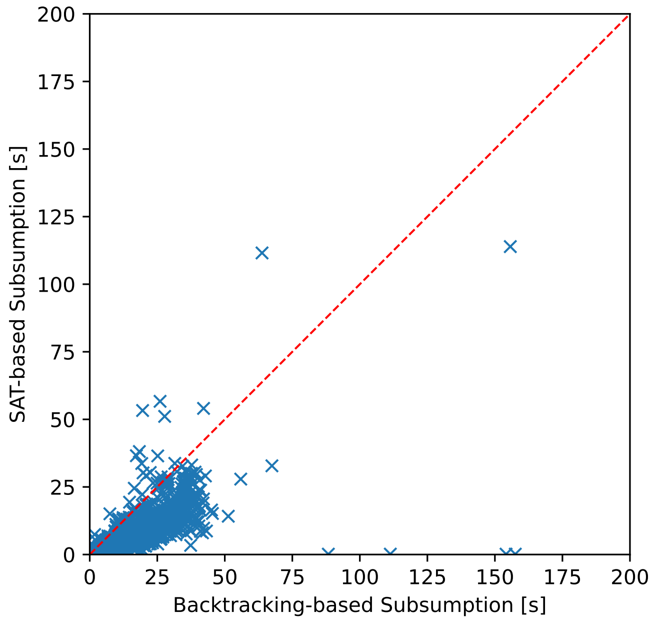
Benchmarks. The basis for our benchmarks is formed by the TPTP library [36] (version 7.5.0), which is a standard benchmark library in the theorem proving community. The TPTP library contains altogether 24,098 problems in various languages, out of which 16,312 problems have been included in our evaluation of SAT-based subsumption in Vampire. The remaining TPTP problems that we did not use for our experiments either use features that Vampire currently does not support (e.g., higher-order logic with theories), or did not involve subsumption checks.

Experimental Setup. All our experiments were carried out on a cluster at TU Wien, where the compute nodes contain 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 tool `runexec` from the benchmarking framework BenchExec [5] to assign each benchmark process to a different CPU core and its corresponding memory node, while aiming to balance the load evenly across memory nodes. Further, we used GNU Parallel [38] to schedule 32 benchmark processes in parallel.

Experimental Results on Measuring Speed Improvements. We emphasize that using a SAT-based approach for deciding clausal subsumption will, in theory, not prove problems that were not provable before. If a problem is provable while using saturation with redundancy, and hence with subsumption, then it is also provable using saturation without redundancy, and vice versa. However, in practice, saturation with redundancy (hence with subsumption) will improve the prover's performance in finding a proof. As such, the aim of our work is to speed up the application of subsumption in saturation. For this reason, we set up our first experiment to measure the cost of subsumption checks in isolation. A similar evaluation has previously been done for indexing techniques in first-order provers, see [27].

In preparation for this experiment, we ran Vampire, using the original backtracking-based subsumption implementation, with a timeout of 60 seconds on each TPTP problem while logging each subsumption (and subsumption resolution) check into a file. Each of these files contains a sequence of subsumption (and subsumption resolution) checks, which we call the *subsumption log* for a problem. This preparatory step yielded a large number of benchmarks that are representative for the checks appearing during actual proof search. These benchmarks

Figure 1. Total running time (in seconds) of backtracking-based vs. SAT-based subsumption, with detailed information about outliers in Table I. For marks below the dashed line, our SAT-based approach was faster.



occupy 1.75 TiB of disk space in compressed form, and contain approximately 114 billion subsumption checks in total. About 0.5 % of these subsumption checks are satisfiable (561 million), while the rest are unsatisfiable.

In addition to generating these benchmarks, we have profiled the portion of time spent by subsumption in Vampire. Over the TPTP problems used for our experiments and a time limit of 60 seconds, it ranges from 0 % (no subsumption checks) to more than 99 % (hard subsumption check), with a mean of 46 % and the median at 53 %.

Next, we executed the checks listed in each subsumption log and measured the total running times, once for the already existing subsumption algorithm of Vampire using backtracking, and once for our SAT-based subsumption approach in Vampire. The subsumption checks are benchmarked in a similar way as they would appear during a regular prover run, i.e., with the same caching of intermediate results. For increased reliability, each measurement was performed five times, and then taking the arithmetic mean.

The results of these experiments are given in Figure 1 and Table I. Each mark in Figure 1 represents one subsumption log from a TPTP problem, and compares the total running times 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 I, we give the cumulative times needed to set up the subsumption checks, to solve them, and the total time. Both the backtracking-based and our SAT-based subsumption algorithm can naturally be split up into a setup stage and a separate solving stage. The setup stage transforms the two

Table I
RUNNING TIME OF SUBSUMPTION CHECKS

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

input clauses into constraints while the solving stage searches for a solution to these constraints. Additionally the table gives detailed data for selected outliers (problems not in the bottom-left of Figure 1).

As shown in Figure 1 and Table I, our SAT-based algorithm for clausal subsumption gives a clear overall improvement of the running/proving time of VAMPIRE by a factor of 2.

Note that for some problems, the running time for the backtracking-based subsumption is higher than the original timeout of 60 s that has been used when collecting subsumption logs. The cause of this apparent discrepancy is that VAMPIRE was working on a hard subsumption instance when hitting the timeout, with the subsequent measurements in Table I showing the true cost. Problems such as `NLP023+1` are getting stuck in the backtracking-based subsumption algorithm, while our SAT-based approach would allow proof search to continue much further within the same time limit.

We also evaluated the impact of our custom variable selection heuristic (see last paragraph of Section VII) compared to the variable-move-to-front (VMTF) heuristic of SAT solvers [8], as VMTF is conjectured to perform well for SAT problems that are unsatisfiable, being part of the "unstable phase" described in [7]. Given that almost all subsumption instances are unsatisfiable, we were interested to see how our SAT-based approach performs compared to a VMTF heuristic. Our results in this respect are listed in the last line of Table I. While our custom heuristic shows slightly better solving times than VMTF, the difference is rather small.

Experimental Results on the Advantage of Re-Using SAT Solver Objects. We also assessed the importance of re-using the SAT solver object instead of re-allocating the solver for every subsumption query. The result is given in the second-to-last line of Table I, confirming the significance of having SAT-based subsumption directly integrated in VAMPIRE.

VII. RELATED WORK

Subsumption is one of the most important simplification rules in first-order theorem proving. While efficient literal- and clause-indexing techniques have been proposed [37], [33], optimizing the matching step among multisets of literals, and hence clauses, has so far not been addressed. In our work, we show 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 [35], using code trees. Code trees organize potentially subsuming clauses into a trie-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 specialized subsumption algorithm DC [18] is based on the idea of separating the clause C 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 that algorithm, called IDC [17], tests on each recursion level whether each literal of C by itself subsumes D under the current partial substitution, which is a necessary condition for subsumption. The backtracking-based subsumption algorithm of VAMPIRE uses this optimization as well, and our SAT-based approach also implements it as propagation over substitution constraints.

SAT- and SMT-based techniques have previously been applied to the setting of first-order saturation-based proof search, e.g., in form of the AVATAR architecture [39]. 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 [23] and the ASP solver Clasp [14], support cardinality constraints natively, in a similar way to our handling of AtMostOne constraints. Our encoding however requires only AtMostOne constraints instead of arbitrary cardinality constraints, thus simplifying the implementation.

We finally 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 $(3) \wedge (4) \wedge (5)$ represent the different choices of a non-boolean CSP variable, corresponding to the so-called *direct encoding* of a CSP variable [40]. 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 use it to solve subsumption instances (see Section V).

VIII. Conclusion

We advocate the use of lean dedicated SAT solving to solve clausal subsumption in first-order theorem proving. We introduce substitution constraints to encode subsumption as a SAT instance. For solving such instances, we adjust unit propagation and conflict resolution in SAT solving towards a tailored treatment of substitution constraints. Crucially, our encoding together with our tailored solver enables efficient setup of subsumption instances. Our experimental results indicate that SAT-based subsumption significantly improves the performance of first-order proving. Extending our work towards equality reasoning, and hence addressing subsumption demodulation, is an interesting task for future work. For doing so, we believe our substitution constraints would need to encode matching also on the term level, and thus not only on the literal level, in order to find suitable terms to rewrite.

Acknowledgements. We thank Bernhard Gleiss for fruitful initial discussions about this work. This work was supported by the ERC Consolidator Grant ARTIST 101002685 and the Austrian FWF project W1255-N23.

References

[1] Sepideh Asadi, Martin Blicha, Antti E. J. Hyvärinen, Grigory Fedyukovich, and Natasha Sharygina. Incremental Verification by SMT-based Summary Repair. In *Proc. of FMCAD*, pages 77–82, 2020.

[2] Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[3] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. CVC5: A Versatile and Industrial-Strength SMT Solver. In *Proc. of TACAS*, pages 415–442, 2022.

[4] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. Verifying Relational Properties using Trace Logic. In *Proc. of FMCAD*, pages 170–178, 2019.

[5] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable Benchmarking: Requirements and Solutions. *J. on Software Tools for Technology Transfer*, 21(1):1–29, 2017.

[6] Armin Biere. PicoSAT Essentials. *J. Satisf. Boolean Model. Comput.*, 4(2-4):75–97, 2008.

[7] Armin Biere. CaDiCaL at the SAT Race 2019. In *Proc. of SAT Race*, pages 8–9, 2019.

[8] Armin Biere and Andreas Fröhlich. Evaluating CDCL Variable Scoring Schemes. In *Proc. of SAT*, pages 405–422, 2015.

[9] Bruno Buchberger. 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, 2006.

[10] Martin Clochard, Claude Marché, and Andrei Paskevich. Deductive Verification with Ghost Monitors. *Proc. of POPL*, pages 2:1–2:26, 2020.

[11] Simon Cruanes. Superposition with Structural Induction. In *Proc. of FroCoS*, pages 172–188, 2017.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, pages 337–340, 2008.

[13] Alan M. Frisch and Paul A. Giannaros. SAT Encodings of the At-Most-k Constraint. Some Old, Some New, Some Fast, Some Slow. In *Proc. of WS on Constraint Modelling and Reformulation*, 2010.

[14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. On the Implementation of Weight Constraint Rules in Conflict-Driven ASP Solvers. In *Proc. of ICLP*, pages 250–264, 2009.

[15] Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace Logic for Inductive Loop Reasoning. In *Proc. of FMCAD*, pages 255–263, 2020.

[16] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption Demodulation in First-Order Theorem Proving. In *Proc. of IJCAR*, pages 297–315, 2020.

[17] Georg Gottlob and Alexander Leitsch. Fast Subsumption Algorithms. In *Proc. of EUROCAL '85*, pages 64–77, 1985.

[18] Georg Gottlob and Alexander Leitsch. On the Efficiency of Subsumption Algorithms. *J. of the ACM*, 32(2):280–295, 1985.

[19] Deepak Kapur and Paliath Narendran. NP-Completeness of the Set Unification and Matching Problems. In *Proc. of IJCAR*, pages 489–495, 1986.

[20] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-Based Model Checking for Recursive Programs. *Formal Methods Syst. Des.*, 48(3):175–205, 2016.

[21] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, pages 1–35, 2013.

[22] K. Rustan M. Leino. Accessible Software Verification with Dafny. *IEEE Softw.*, 34(6):94–97, 2017.

[23] Mark H. Liffiton and Jordyn C. Maglalang. A Cardinality Solver: More Expressive Constraints for Free. In *Proc. of SAT*, pages 485–486, 2012.

[24] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, pages 133–182. IOS Press, 2021.

[25] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *Proc. of ESOP*, pages 30–59, 2019.

[26] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of DAC*, pages 530–535, 2001.

[27] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the Evaluation of Indexing Techniques for Theorem Proving. In *Proc. of IJCAR*, pages 257–271, 2001.

[28] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

[29] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proc. of PLDI*, pages 614–630, 2016.

[30] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. Automating Modular Verification of Secure Information Flow. In *Proc. of FMCAD*, pages 158–168, 2020.

[31] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[32] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proc. of CCS*, pages 621–640, 2020.

[33] 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.

[34] Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, Higher, Stronger: E 2.3. In *Proc. of CADE*, pages 495–507, 2019.

[35] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Term Indexing. In *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier and MIT Press, 2001.

[36] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *J. of Automated Reasoning*, 59(4):483–502, 2017.

[37] Tanel Tammet. Towards Efficient Subsumption. In *Proc. of CADE*, pages 427–441, 1998.

[38] Ole Tange. *GNU Parallel 2018*. Ole Tange, March 2018.

[39] Andrei Voronkov. AVATAR: The Architecture for First-Order Theorem Provers. In *Proc. of CAV*, pages 696–710, 2014.

[40] Toby Walsh. SAT v CSP. In *Proc. of CP*, pages 441–456, 2000.

[41] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In *Proc. of CADE*, pages 140–145, 2009.