Subsumption Demodulation in First-Order Theorem Proving

Citation for the original published paper (version of record):
Gleiss, B., Kovacs, L., Rath, J. (2020). Subsumption Demodulation in First-Order Theorem Proving. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 12166 LNAI: 297-315.
http://dx.doi.org/10.1007/978-3-030-51074-9_17

(article starts on next page)

Subsumption Demodulation in First-Order Theorem Proving

Bernhard Gleiss[1], Laura Kovács[1,2], and Jakob Rath[1](✉)

[1] TU Wien, Vienna, Austria
jakob.rath@tuwien.ac.at
[2] Chalmers University of Technology, Gothenburg, Sweden

Abstract. Motivated by applications of first-order theorem proving to software analysis, we introduce a new inference rule, called subsumption demodulation, to improve support for reasoning with conditional equalities in superposition-based theorem proving. We show that subsumption demodulation is a simplification rule that does not require radical changes to the underlying superposition calculus. We implemented subsumption demodulation in the theorem prover Vampire, by extending Vampire with a new clause index and adapting its multi-literal matching component. Our experiments, using the TPTP and SMT-LIB repositories, show that subsumption demodulation in Vampire can solve many new problems that could so far not be solved by state-of-the-art reasoners.

1 Introduction

For the efficiency of organizing proof search during saturation-based first-order theorem proving, simplification rules are of critical importance. Simplification rules are inference rules that do not add new formulas to the search space, but simplify formulas by deleting (redundant) clauses from the search space. As such, simplification rules reduce the size of the search space and are crucial in making automated reasoning efficient.

When reasoning about properties of first-order logic with equality, one of the most common simplification rules is demodulation [10] for rewriting (and hence simplifying) formulas using unit equalities $l \simeq r$, where l, r are terms and $\simeq$ denotes equality. As a special case of superposition, demodulation is implemented in first-order provers such as E [14], Spass [21] and Vampire [10]. Recent applications of superposition-based reasoning, for example to program analysis and verification [5], demand however new and efficient extensions of demodulation to reason about and simplify upon conditional equalities $C \rightarrow l \simeq r$, where C is a first-order formula. Such conditional equalities may, for example, encode software properties expressed in a guarded command language, with C denoting a guard (such as a loop condition) and $l \simeq r$ encoding equational properties over program variables. We illustrate the need of considering generalized versions of demodulation in the following example.

N. Peltier and V. Sofronie-Stokkermans (Eds.): IJCAR 2020, LNAI 12166, pp. 297–315, 2020.
https://doi.org/10.1007/978-3-030-51074-9_17

Example 1. Consider the following formulas expressed in the first-order theory of integer linear arithmetic:

$$\begin{array}{l} f(i) \simeq g(i) \\ 0 \leq i < n \rightarrow P(f(i)) \end{array} \tag{1}$$

Here, i is an implicitly universally quantified logical variable of integer sort, and n is an integer-valued constant. First-order reasoners will first clausify formulas (1), deriving:

$$\begin{array}{l} f(i) \simeq g(i) \\ 0 \not\leq i \vee i \not< n \vee P(f(i)) \end{array} \tag{2}$$

By applying demodulation over (2), the formula $0 \not\leq i \vee i \not< n \vee P(f(i))$ is rewritten[1] using the unit equality $f(i) \simeq g(i)$, yielding the clause $0 \not\leq i \vee i \not< n \vee P(g(i))$. That is, $0 \leq i < n \rightarrow P(g(i))$ is derived from (1) by one application of demodulation.

Let us now consider a slightly modified version of (1), as below:

$$\begin{array}{l} 0 \leq i < n \rightarrow f(i) \simeq g(i) \\ 0 \leq i < n \rightarrow P(f(i)) \end{array} \tag{3}$$

whose clausal representation is given by:

$$\begin{array}{l} 0 \not\leq i \vee i \not< n \vee f(i) \simeq g(i) \\ 0 \not\leq i \vee i \not< n \vee P(f(i)) \end{array} \tag{4}$$

It is again obvious that from (3) one can derive the formula $0 \leq i < n \rightarrow P(g(i))$, or equivalently the clause:

$$0 \not\leq i \vee i \not< n \vee P(g(i)) \tag{5}$$

Yet, *one cannot anymore apply demodulation-based simplification over* (4) *to derive such a clause*, as (4) contains no unit equality. $\square$

In this paper we propose a generalized version of demodulation, called *subsumption demodulation*, allowing to rewrite terms and simplify formulas using rewriting based on conditional equalities, such as in (3). To do so, we extend demodulation with subsumption, that is with deciding whether (an instance of a) clause C is a submultiset of a clause D. In particular, the non-equality literals of the conditional equality (i.e., the condition) need to subsume the unchanged literals of the simplified clause. This way, subsumption demodulation can be applied to non-unit clauses and is not restricted to have at least one premise clause that is a unit equality. We show that subsumption demodulation is a simplification rule of the superposition framework (Sect. 4), allowing for example to derive the clause (5) from (4) in one inference step. By properly adjusting clause indexing and multi-literal matching in first-order theorem provers, we provide an efficient implementation of subsumption demodulation in VAMPIRE (Sect. 5) and

[1] Assuming that g is simpler/smaller than f.

evaluate our work against state-of-the-art reasoners, including E [14], Spass [21], CVC4 [3] and Z3 [7] (Sect. 6).

Related Work. While several approaches generalize demodulation in superposition-based theorem proving, we argue that subsumption demodulation improves existing methods either in terms of applicability and/or efficiency. The AVATAR architecture of first-order provers [19] splits general clauses into components with disjoint sets of variables, potentially enabling demodulation inferences whenever some of these components become unit equalities. Example 1 demonstrates that subsumption demodulation applies in situations where AVATAR does not: in each clause of (4), all literals share the variable i and hence none of the clauses from (4) can be split using AVATAR. That is, AVATAR would not generate unit equalities from (4), and therefore cannot apply demodulation over (4) to derive (5).

The local rewriting approach of [20] requires rewriting equality literals to be maximal[2] in clauses. However, following [10], for efficiency reasons we consider equality literals to be "smaller" than non-equality literals. In particular, the equality literals of clauses (4) are "smaller" than the non-equality literals, preventing thus the application of local rewriting in Example 1.

To the extent of our knowledge, the ordering restrictions on non-unit rewriting [20] do not ensure redundancy, and thus the rule is not a simplification inference rule. Subsumption demodulation includes all necessary conditions and we prove it to be a simplification rule. Furthermore, we show how the ordering restrictions can be simplified which enables an efficient implementation, and then explain how such an implementation can be realized.

We further note that the contextual rewriting rule of [1] is more general than our rule of subsumption demodulation, and has been first implemented in the Saturate system [12]. Yet, efficiently automating contextual rewriting is extremely challenging, while subsumption demodulation requires no radical changes in the existing machinery of superposition provers (see Sect. 5).

To the best of our knowledge, except Spass [21] and Saturate, no other state-of-the-art superposition provers implement variants of conditional rewriting. Subterm contextual rewriting [22] is a refined notion of contextual rewriting and is implemented in Spass. A major difference of subterm contextual rewriting when compared to subsumption demodulation is that in subsumption demodulation the discovery of the substitution is driven by the side conditions whereas in subterm contextual rewriting the side conditions are evaluated by checking the validity of certain implications by means of a reduction calculus. This reduction calculus recursively applies another restriction of contextual rewriting called recursive contextual ground rewriting, among other standard reduction rules. While subterm contextual rewriting is more general, we believe that the benefit of subsumption demodulation comes with its relatively easy and efficient integration within existing superposition reasoners, as evidenced also in Sect. 6.

[2] w.r.t. clause ordering.

Local contextual rewriting [9] is another refinement of contextual rewriting implemented in Spass. In our experiments it performed similarly to subterm contextual rewriting.

Finally, we note that SMT-based reasoners also implement various methods to efficiently handle conditional equalities [6,13]. Yet, the setting is very different as they rely on the DPLL(T) framework [8] rather than implementing superposition.

Contributions. Summarizing, this paper brings the following contributions.

– To improve reasoning in the presence of conditional equalities, we introduce the new inference rule *subsumption demodulation*, which generalizes demodulation to non-unit equalities by combining demodulation and subsumption (Sect. 4).
– Subsumption demodulation does not require radical changes to the underlying superposition calculus. We implemented subsumption demodulation in the first-order theorem prover Vampire, by extending Vampire with a new clause index and adapting its multi-literal matching component (Sect. 5).
– We compared our work against state-of-the-art reasoners, using the TPTP and SMT-LIB benchmark repositories. Our experiments show that subsumption demodulation in Vampire can solve 11 first-order problems that could so far not be solved by any other state-of-the-art provers, including Vampire, E, Spass, CVC4 and Z3 (Sect. 6).

2 Preliminaries

For simplicity, in what follows we consider standard first-order logic with equality, where equality is denoted by $\simeq$. We support all standard boolean connectives and quantifiers in the language. Throughout the paper, we denote terms by l, r, s, t, variables by x, y, constants by c, d, function symbols by f, g and predicate symbols by P, Q, R, all possibly with indices. Further, we denote literals by L and clauses by C, D, again possibly with indices. We write $s \not\simeq t$ to denote the formula $\neg s \simeq t$. A literal $s \simeq t$ is called an *equality literal*. We consider clauses as multisets of literals and denote by $\subseteq_M$ the subset relation among multisets. A clause that only consists of one equality literal is called a *unit equality*.

An expression E is a term, literal, or clause. We write $E[s]$ to mean an expression E with a particular occurrence of a term s. A *substitution*, denoted by σ, is any finite mapping of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $n > 0$. Applying a substitution σ to an expression E yields another expression, denoted by $E\sigma$, by simultaneously replacing each x_i by t_i in E. We say that $E\sigma$ is an instance of E. A *unifier* of two expressions E_1 and E_2 is a substitution σ such that $E_1\sigma = E_2\sigma$. If two expressions have a unifier, they also have a *most general unifier (mgu)*. A *match* of expression E_1 to expression E_2 is a substitution σ such that $E_1\sigma = E_2$. Note that any match is a unifier (assuming the sets of variables in E_1 and E_2 are disjoint), but not vice-versa, as illustrated below.

Example 2. Let E_1 and E_2 be the clauses $Q(x, y) \vee R(x, y)$ and $Q(c, d) \vee R(c, z)$, respectively. The only possible match of $Q(x, y)$ to $Q(c, d)$ is $\sigma_1 = \{x \mapsto c, y \mapsto d\}$. On the other hand, the only possible match of $R(x, y)$ to $R(c, z)$ is $\sigma_2 = \{x \mapsto c, y \mapsto z\}$. As σ_1 and σ_2 are not the same, there is no match of E_1 to E_2. Note however that E_1 and E_2 can be unified; for example, using $\sigma_3 = \{x \mapsto c, y \mapsto d, z \mapsto d\}$.

Superposition Inference System. We assume basic knowledge in first-order theorem proving and superposition reasoning [2,11]. We adopt the notations and the inference system of superposition from [10]. We recall that first-order provers perform inferences on clauses using inference rules, where an *inference* is usually written as:

$$\frac{C_1 \qquad \ldots \qquad C_n}{C}$$

with $n \geq 0$. The clauses $C_1, \ldots, C_n$ are called the premises and C is the conclusion of the inference above. An inference is *sound* if its conclusion is a logical consequence of its premises. An inference rule is a set of (concrete) inferences and an inference system is a set of inference rules. An inference system is *sound* if all its inference rules are sound.

Modern first-order theorem provers implement the *superposition inference system* for first-order logic with equality. This inference system is parametrized by a *simplification ordering* over terms and a *literal selection function* over clauses. In what follows, we denote by $\succ$ a simplification ordering over terms, that is $\succ$ is a well-founded partial ordering satisfying the following three conditions:

- *stability under substitutions*: if $s \succ t$, then $s\theta \succ t\theta$;
- *monotonicity*: if $s \succ t$, then $l[s] \succ l[t]$;
- *subterm property*: $s \succ t$ whenever t is a proper subterm of s.

The simplification ordering $\succ$ on terms can be extended to a simplification ordering on literals and clauses, using a multiset extension of orderings. For simplicity, the extension of $\succ$ to literals and clauses will also be denoted by $\succ$. Whenever $E_1 \succ E_2$, we say that E_1 is bigger than E_2 and E_2 is smaller than E_1 w.r.t. $\succ$. We say that an equality literal $s \simeq t$ is *oriented*, if $s \succ t$ or $t \succ s$. The literal extension of $\succ$ asserts that negative literals are always bigger than their positive counterparts. Moreover, if $L_1 \succ L_2$, where L_1 and L_2 are positive, then $\neg L_1 \succ L_1 \succ \neg L_2 \succ L_2$. Finally, equality literals are set to be smaller than any literal using a predicate different than $\simeq$.

A *selection function* selects at least one literal in every non-empty clause. In what follows, selected literals in clauses will be underlined: when writing $\underline{L} \vee C$, we mean that (at least) L is selected in $L \vee C$. In what follows, we assume that selection functions are *well-behaved* w.r.t. $\succ$: either a negative literal is selected or all maximal literals w.r.t. $\succ$ are selected.

In the sequel, we fix a simplification ordering $\succ$ and a well-behaved selection function and consider the superposition inference system, denoted by Sup, parametrized by these two ingredients. The inference system Sup for first-order logic with equality consists of the inference rules of Fig. 1, and it is both sound and refutationally complete. That is, if a set S of clauses is unsatisfiable, then the empty clause (that is, the always false formula) is derivable from S in Sup.

– Resolution and Factoring

$$\frac{\underline{L} \vee C_1 \qquad \underline{\neg L'} \vee C_2}{(C_1 \vee C_2)\sigma} \qquad\qquad \frac{\underline{L} \vee \underline{L'} \vee C}{(L \vee C)\sigma}$$

where L is not an equality literal and $\sigma = mgu(L, L')$

– Superposition

$$\frac{\underline{s \simeq t} \vee C_1 \qquad \underline{L[s']} \vee C_2}{(C_1 \vee L[t] \vee C_2)\theta}$$

$$\frac{\underline{s \simeq t} \vee C_1 \qquad \underline{l[s'] \simeq l'} \vee C_2}{(C_1 \vee l[t] \simeq l' \vee C_2)\theta} \qquad \frac{\underline{s \simeq t} \vee C_1 \qquad \underline{l[s'] \not\simeq l'} \vee C_2}{(C_1 \vee l[t] \not\simeq l' \vee C_2)\theta}$$

where s' not a variable, L is not an equality, $\theta = mgu(s, s')$, $t\theta \not\succeq s\theta$ and $l'\theta \not\succeq l[s']\theta$

– Equality Resolution and Equality Factoring

$$\frac{\underline{s \not\simeq s'} \vee C}{C\theta} \qquad\qquad \frac{s \simeq t \vee \underline{s' \simeq t'} \vee C}{(s \simeq t \vee t \not\simeq t' \vee C)\theta}$$

where $\theta = mgu(s, s')$, $t\theta \not\succeq s\theta$ and $t'\theta \not\succeq t\theta$

Fig. 1. The superposition calculus Sup.

3 Superposition-Based Proof Search

We now overview the main ingredients in organizing proof search within first-order provers, using the superposition calculus. For details, we refer to [2,10,11].

Superposition-based provers use *saturation algorithms*: applying all possible inferences of Sup in a certain order to the clauses in the search space until (i) no more inferences can be applied or (ii) the empty clause has been derived. A simple implementation of a saturation algorithm would however be very inefficient as applications of all possible inferences will quickly blow up the search space.

Saturation algorithms can however be made efficient by exploiting a powerful concept of *redundancy*: deleting so-called redundant clauses from the search space by preserving completeness of Sup. A clause C in a set S of clauses (i.e., in the search space) is *redundant* in S, if there exist clauses $C_1, \ldots, C_n$ in S, such that $C \succ C_i$ and $C_1, \ldots, C_n \vDash C$. That is, a clause C is redundant in S if it is a logical consequence of clauses that are smaller than C w.r.t. $\succ$. It is known that redundant clauses can be removed from the search space without affecting completeness of superposition-based proof search. For this reason, saturation-based theorem provers, such as E, Spass and Vampire, not only generate new clauses but also delete redundant clauses during proof search by using both *generating* and *simplifying* inferences.

Simplification Rules. A *simplifying inference* is an inference in which one premise C_i becomes redundant after the addition of the conclusion C to the search space, and hence C_i can be deleted. In what follows, we will denote deleted clauses by drawing a line through them and refer to simplifying inferences as *simplification rules*. The premise C_i that becomes redundant is called the *main premise*, whereas other premises are called *side premises* of the simplification rule. Intuitively, a simplification rule simplifies its main premise to its conclusion by using additional knowledge from its side premises. Inferences that are not simplifying are called *generating*, as they generate and add a new clause C to the search space.

In saturation-based proof search, we distinguish between *forward* and *backward* simplifications. During forward simplification, a newly derived clause is simplified using previously derived clauses as side clauses. Conversely, during backward simplification a newly derived clause is used as a side clause to simplify previously derived clauses.

Demodulation. One example of a simplification rule is *demodulation*, or also called *rewriting by unit equalities*. Demodulation is the following inference rule:

$$\frac{l \simeq r \qquad \cancel{L[t] \vee C}}{L[r\sigma] \vee C}$$

where $l\sigma = t$, $l\sigma \succ r\sigma$ and $L[t] \vee C \succ (l \simeq r)\sigma$, for some substitution σ.

It is easy to see that demodulation is a simplification rule. Moreover, demodulation is a special case of a superposition inference where one premise of the inference is deleted. However, unlike a superposition inference, demodulation is not restricted to selected literals.

Example 3. Consider the clauses $C_1 = f(f(x)) \simeq f(x)$ and $C_2 = P(f(f(c))) \vee Q(d)$. Let σ be the substitution $\sigma = \{x \mapsto c\}$. By the subterm property of $\succ$, we have $f(f(c)) \succ f(c)$. Further, as equality literals are smaller than non-equality literals, we have $P(f(f(c))) \vee Q(d) \succ f(f(c)) \simeq f(c)$. We thus apply demodulation and C_2 is simplified into the clause $C_3 = P(f(c)) \vee Q(d)$:

$$\frac{f(f(x)) \simeq f(x) \qquad \cancel{P(f(f(c))) \vee Q(d)}}{P(f(c)) \vee Q(d)}$$ □

Deletion Rules. Even when simplification rules are in use, deleting more/other redundant clauses is still useful to keep the search space small. For this reason, in addition to simplifying and generating rules, theorem provers also use *deletion rules*: a *deletion rule* checks whether clauses in the search space are redundant due to the presence of other clauses in the search space, and removes redundant clauses from the search space.

Given clauses C and D, we say C subsumes D if there is some substitution σ such that $C\sigma$ is a submultiset of D, that is $C\sigma \subseteq_M D$. *Subsumption* is the deletion rule that removes subsumed clauses from the search space.

Example 4. Let $C = P(x) \vee Q(f(x))$ and $D = P(f(c)) \vee P(g(c)) \vee Q(f(c)) \vee Q(f(g(c))) \vee R(y)$ be clauses in the search space. Using $\sigma = \{x \mapsto g(c)\}$, it is easy to see that C subsumes D, and hence D is deleted from the search space. □

4 Subsumption Demodulation

In this section we introduce a new simplification rule, called subsumption demodulation, by extending demodulation to a simplification rule over conditional equalities. We do so by combining demodulation with subsumption checks to find simplifying applications of rewriting by non-unit (and hence conditional) equalities.

4.1 Subsumption Demodulation for Conditional Rewriting

Our rule of subsumption demodulation is defined below.

Definition 1 (Subsumption Demodulation). Subsumption demodulation *is the inference rule:*

$$\frac{l \simeq r \vee C \qquad L[t] \vee D}{L[r\sigma] \vee D} \tag{6}$$

where:

1. $l\sigma = t$,
2. $C\sigma \subseteq_M D$,
3. $l\sigma \succ r\sigma$, *and*
4. $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$.

We call the equality $l \simeq r$ in the left premise of (6) *the* rewriting equality *of subsumption demodulation.*

Intuitively, the side conditions 1 and 2 of Definition 1 ensure the soundness of the rule: it is easy to see that if $l \simeq r \vee C$ and $L[t] \vee D$ are true, then $L[r\sigma] \vee D$ also holds. We thus conclude:

Theorem 1 (Soundness). *Subsumption demodulation is sound.*

On the other hand, side conditions 3 and 4 of Definition 1 are vital to ensure that subsumption demodulation is a simplification rule (details follow in Sect. 4.2).

Detecting possible applications of subsumption demodulation involves (i) selecting one equality of the side clause as rewriting equality and (ii) matching each of the remaining literals, denoted C in (6), to some literal in the main clause. Step (i) is similar to finding unit equalities in demodulation, whereas step (ii) reduces to showing that C subsumes parts of the main premise. Informally speaking, subsumption demodulation combines demodulation and subsumption, as discussed in Sect. 5. Note that in step (ii), matching allows any instantiation of C to $C\sigma$ via substitution σ; yet, we we do *not* unify the side and main premises of subsumption demodulation, as illustrated later in Example 7. Furthermore, we need to find a term t in the unmatched part $D \setminus C\sigma$ of the main premise, such that t can be rewritten according to the rewriting equality into $r\sigma$.

As the ordering $\succ$ is partial, the conditions of Definition 1 must be checked a posteriori, that is after subsumption demodulation has been applied with a fixed substitution. Note however that if $l \succ r$ in the rewriting equality, then $l\sigma \succ r\sigma$ for any substitution, so checking the ordering a priori helps, as illustrated in the following example.

Example 5. Let us consider the following two clauses:

$$\begin{aligned} C_1 &= f(g(x)) \simeq g(x) \vee Q(x) \vee R(y) \\ C_2 &= P(f(g(c))) \vee Q(c) \vee Q(d) \vee R(f(g(d))) \end{aligned}$$

By the subterm property of $\succ$, we conclude that $f(g(x)) \succ g(x)$. Hence, the rewriting equality, as well as any instance of it, is oriented.

Let σ be the substitution $\sigma = \{x \mapsto c, y \mapsto f(g(d))\}$. Due to the previous paragraph, we know $f(g(c)) \succ g(c)$ As equality literals are smaller than non-equality ones, we also conclude $P(f(g(c))) \succ f(g(c)) \simeq g(c)$. Thus, we have $P(f(g(c))) \vee Q(c) \vee Q(d) \vee R(f(g(d))) \succ f(g(c)) \simeq g(c) \vee Q(c) \vee R(f(g(d)))$ and we can apply subsumption demodulation to C_1 and C_2, deriving clause $C_3 = P(g(c)) \vee Q(c) \vee Q(d) \vee R(f(g(d)))$.

We note that demodulation cannot derive C_3 from C_1 and C_2, as there is no unit equality. □

Example 5 highlights limitations of demodulation when compared to subsumption demodulation. We next illustrate different possible applications of subsumption demodulation using a fixed side premise and different main premises.

Example 6. Consider the clause $C_1 = f(g(x)) \simeq g(y) \vee Q(x) \vee R(y)$. Only the first literal $f(g(x)) \simeq g(y)$ is a positive equality and as such eligible as rewriting equality. Note that $f(g(x))$ and $g(y)$ are incomparable w.r.t. $\succ$ due to occurrences of different variables, and hence whether $f(g(x))\sigma \succ g(y)\sigma$ depends on the chosen substitution σ.

(1) Consider the clause $C_2 = P(f(g(c))) \vee Q(c) \vee R(c)$ as the main premise. With the substitution $\sigma_1 = \{x \mapsto c, y \mapsto c\}$, we have $f(g(x))\sigma_1 \succ g(x)\sigma_1$ as $f(g(c)) \succ g(c)$ due to the subterm property of $\succ$, enabling a possible application of subsumption demodulation over C_1 and C_2.

(2) Consider now $C_3 = P(g(f(g(c)))) \vee Q(c) \vee R(f(g(c)))$ as the main premise and the substitution $\sigma_2 = \{x \mapsto c, y \mapsto f(g(c))\}$. We have $g(y)\sigma_2 \succ f(g(x))\sigma_2$, as $g(f(g(c)) \succ f(g(c))$. The instance of the rewriting equality is oriented differently in this case than in the previous one, enabling a possible application of subsumption demodulation over C_1 and C_3.
(3) On the other hand, using the clause $C_4 = P(f(g(c))) \vee Q(c) \vee R(z)$ as the main premise, the only substitution we can use is $\sigma_3 = \{x \mapsto c, y \mapsto z\}$. The corresponding instance of the rewriting equality is then $f(g(c)) \simeq g(z)$, which cannot be oriented in general. Hence, subsumption demodulation cannot be applied in this case, even though we can find the matching term $f(g(c))$ in C_4. □

As mentioned before, the substitution σ appearing in subsumption demodulation can only be used to instantiate the side premise, but not for unifying side and main premises, as we would not obtain a simplification rule.

Example 7. Consider the clauses:

$$C_1 = f(c) \simeq c \vee Q(d)$$
$$C_2 = P(f(c)) \vee Q(x)$$

As we cannot match $Q(d)$ to $Q(x)$ (although we could match $Q(x)$ to $Q(d)$), subsumption demodulation is not applicable with premises C_1 and C_2. □

4.2 Simplification Using Subsumption Demodulation

Note that in the special case where C is the empty clause in (6), subsumption demodulation reduces to demodulation and hence it is a simplification rule. We next show that this is the case in general:

Theorem 2 (Simplification Rule). *Subsumption demodulation is a simplification rule and we have:*

$$\frac{l \simeq r \vee C \qquad \cancel{L[t] \vee D}}{L[r\sigma] \vee D}$$

where:

1. $l\sigma = t$,
2. $C\sigma \subseteq_M D$,
3. $l\sigma \succ r\sigma$, *and*
4. $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$.

Proof. Because of the second condition of the definition of subsumption demodulation, $L[t] \vee D$ is clearly a logical consequence of $L[r\sigma] \vee D$ and $l \simeq r \vee C$. Moreover, from the fourth condition, we trivially have $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$. It thus remains to show that $L[r\sigma] \vee D$ is smaller than $L[t] \vee D$ w.r.t. $\succ$. As

$t = l\sigma \succ r\sigma$, the monotonicity property of $\succ$ asserts that $L[t] \succ L[r\sigma]$, and hence $L[t] \vee D \succ L[r\sigma] \vee D$. This concludes that $L[t] \vee D$ is redundant w.r.t. the conclusion and left-most premise of subsumption demodulation. $\square$

Example 8. By revisiting Example 5, Theorem 2 asserts that clause C_2 is simplified into C_3, and subsumption demodulation deletes C_2 from the search space. $\square$

4.3 Refining Redundancy

The fourth condition defining subsumption demodulation in Definition 1 is required to ensure that the main premise of subsumption demodulation becomes redundant. However, comparing clauses w.r.t. the ordering $\succ$ is computationally expensive; yet, not necessary for subsumption demodulation. Following the notation of Definition 1, let D' such that $D = C\sigma \vee D'$. By properties of multiset orderings, the condition $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$ is equivalent to $L[t] \vee D' \succ (l \simeq r)\sigma$, as the literals in $C\sigma$ occur on both sides of $\succ$. This means, to ensure the redundancy of the main premise of subsumption demodulation, we only need to ensure that there is a literal from $L[t] \vee D$ such that this literal is bigger that the rewriting equality.

Theorem 3 (Refining Redundancy). *The following conditions are equivalent:*

(R1) $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$
(R2) $L[t] \vee D' \succ (l \simeq r)\sigma$

As mentioned in Sect. 4.1, application of subsumption demodulation involves checking that an ordering condition between premises holds (side condition 4 in Definition 1). Theorem 3 asserts that we only need to find a literal in $L[t] \vee D'$ that is bigger than the rewriting equality in order to ensure that the ordering condition is fulfilled. In the next section we show that by re-using and properly changing the underlying machinery of first-order provers for demodulation and subsumption, subsumption demodulation can efficiently be implemented in superposition-based proof search.

5 Subsumption Demodulation in Vampire

We implemented subsumption demodulation in the first-order theorem prover Vampire. Our implementation consists of about 5000 lines of C++ code and is available at:

https://github.com/vprover/vampire/tree/subsumption-demodulation

As for any simplification rule, we implemented the forward and backward versions of subsumption demodulation separately. Our new Vampire options controlling subsumption demodulation are `fsd` and `bsd`, both with possible values `on` and `off`, to respectively enable forward and backward subsumption demodulation.

As discussed in Sect. 4, subsumption demodulation uses reasoning based on a combination of demodulation and subsumption. Algorithm 1 details our implementation for *forward subsumption demodulation*. In a nutshell, given a clause D as main premise, (forward) subsumption demodulation in Vampire consists of the following main steps:

1. *Retrieve candidate clauses* C as side premises of subsumption demodulation (line 1 of Algorithm 1). To this end, we design a new clause index with imperfect filtering, by modifying the subsumption index in Vampire, as discussed later in this section.
2. *Prune candidate clauses* by checking the conditions of subsumption demodulation (lines 3–7 of Algorithm 1), in particular selecting a rewriting equality and matching the remaining literals of the side premise to literals of the main premise. After this, prune further by performing a posteriori checks for orienting the rewriting equality E, and checking the redundancy of the given main premise D. To do so, we revised multi-literal matching and redundancy checking in Vampire (see later).
3. *Build simplified clause* by simplifying and deleting the (main) premise D of subsumption demodulation using (forward) simplification (line 8 of Algorithm 1).

Our implementation of *backward subsumption demodulation* requires only a few changes to Algorithm 1: (i) we use the input clause as side premise C of backward subsumption demodulation and (ii) we retrieve candidate clauses D as potential main premises of subsumption demodulation. Additionally, (iii) instead of returning a single simplified clause D', we record a replacement clause for each candidate clause D where a simplification was possible.

Clause Indexing for Subsumption Demodulation. We build upon the indexing approach [15] used for subsumption in Vampire: the subsumption index in Vampire stores and retrieves candidate clauses for subsumption. Each clause is indexed by exactly one of its literals. In principle, any literal of the clause can be chosen. In order to reduce the number of retrieved candidates, the best literal is chosen in the sense that the chosen literal maximizes a certain heuristic (e.g., maximal weight). Since the subsumption index is not a perfect index (i.e., it may retrieve non-subsumed clauses), additional checks on the retrieved clauses are performed.

Using the subsumption index of Vampire as the clause index for forward subsumption demodulation would however omit retrieving clauses (side premises) in which the rewriting equality is chosen as key for the index, omitting this way a possible application of subsumption demodulation. Hence, we need a new clause index in which the best literal can be adjusted to be the rewriting equality. To

Algorithm 1. Forward Subsumption Demodulation – FSD

```
Input  : Clause $D$, to be used as main premise
Output: Simplified clause $D'$ if (forward) subsumption demodulation is possible
// Retrieve candidate side premises
candidates := $FSDIndex.Retrieve(D)$
for each $C \in candidates$ do
    while $m = FindNextMLMatch(C, D)$ do
        $\sigma' := m.GetSubstitution()$
        $E := m.GetRewritingEquality()$
        // $E$ is of the form $l \simeq r$, for some terms $l, r$
        if exists term $t$ in $D \setminus C\sigma'$ and substitution $\sigma \supseteq \sigma'$ s.t. $t = l\sigma$ then
            if $CheckOrderingConditions(D, E, t, \sigma)$ then
                $D' := BuildSimplifiedClause(D, E, t, \sigma)$
                return $D'$
            end
        end
    end
end
```

address this issue, we added a new clause index, called the *forward subsumption demodulation index (FSD index)*, to Vampire, as follows: we index potential side premises either by their best literal (according to the heuristic), the second best literal, or both. If the best literal in a clause C is a positive equality (i.e., a candidate rewriting equality) but the second best is not, C is indexed by the second best literal, and vice versa. If both the best and second best literal are positive equalities, C is indexed by both of them. Furthermore, because the FSD index is exclusively used by forward subsumption demodulation, this index only needs to keep track of clauses that contain at least one positive equality.

In the backward case, we can in fact reuse Vampire's index for backward subsumption. Instead we need to query the index by the best literal, the second best literal, or both (as described in the previous paragraph).

Multi-literal Matching. Similarly to the subsumption index, our new subsumption demodulation index is not a perfect index, that is it performs imperfect filtering for retrieving clauses. Therefore, additional post-checks are required on the retrieved clauses. In our work, we devised a multi-literal matching approach to:

- choose the rewriting equality among the literals of the side premise C, and
- check whether the remaining literals of C can be uniformly instantiated to the literals of the main premise D of subsumption demodulation.

There are multiple ways to organize this process. A simple approach is to (i) first pick any equality of a side premise C as the rewriting equality of subsumption demodulation, and then (ii) invoke the existing multi-literal matching machinery of Vampire to match the remaining literals of C with a subset of literals of D. For the latter step (ii), the task is to find a substitution σ such that $C\sigma$ becomes a submultiset of the given clause D. If the choice of the rewriting equality in step (i) turns out to be wrong, we backtrack. In our work, we revised the existing multi-literal matching machinery of Vampire to a new multi-literal matching approach for subsumption demodulation, by using the steps (i)-(ii) and interleaving equality selection with matching.

We note that the substitution σ in step (ii) above is built in two stages: first we get a partial substitution σ' from multi-literal matching and then (possibly) extend σ' to σ by matching term instances of the rewriting equality with terms of $D \setminus C\sigma$.

Example 9. Let D be the clause $P(f(c,d)) \vee Q(c)$. Assume that our (FSD) clause index retrieves the clause $C = f(x,y) \simeq y \vee Q(x)$ from the search space (line 1 of Algorithm 1). We then invoke our multi-literal matcher (line 3 of Algorithm 1), which matches the literal $Q(x)$ of C to the literal $Q(c)$ of D and selects the equality literal $f(x,y) \simeq y$ of C as the rewriting equality for subsumption demodulation over C and D. The matcher returns the choice of rewriting equality and the partial substitution $\sigma' = \{x \mapsto c\}$. We arrive at the final substitution $\sigma = \{x \mapsto c, y \mapsto d\}$ only when we match the instance $f(x,y)\sigma'$, that is $f(c,y)$, of the left-hand side of the rewriting equality to the literal $f(c,d)$ of D. Using σ, subsumption demodulation over C and D will derive $P(d) \vee Q(c)$, after ensuring that D becomes redundant (line 8 of Algorithm 1). $\square$

We further note that multi-literal matching is an NP-complete problem. Our multi-literal matching problems may have more than one solution, with possibly only some (or none) of them leading to successful applications of subsumption demodulation. In our implementation, we examine all solutions retrieved by multi-literal matching. We also experimented with limiting the number of matches examined after multi-literal matching but did not observe relevant improvements. Yet, our implementation in Vampire also supports an additional option allowing the user to specify an upper bound on how many solutions of multi-literal matching should be examined.

Redundancy Checking. To ensure redundancy of the main premise D after the subsumption demodulation inference, we need to check two properties. First, the instance $E\sigma$ of the rewriting equality E must be oriented. This is a simple ordering check. Second, the main premise D must be larger than the side premise C. Thanks to Theorem 3, this latter condition is reduced to finding a literal among the unmatched part of the main premise D that is bigger than the instance $E\sigma$ of the rewriting equality E.

Example 10. In case of Example 9, the rewriting equality E is oriented and hence $E\sigma$ is also oriented. Next, the literal $P(f(c,d))$ is bigger than $E\sigma$, and hence D is redundant w.r.t. C and D'. $\square$

6 Experiments

We evaluated our implementation of subsumption demodulation in Vampire on the problems of the TPTP [17] (version 7.3.0) and SMT-LIB [4] (release 2019-05-06) repositories. All our experiments were carried out on the StarExec cluster [16].

Benchmark Setup. From the 22,686 problems in the TPTP benchmark set, Vampire can parse 18,232 problems.[3] Out of these problems, we only used those problems that involve equalities as subsumption demodulation is only applicable in the presence of (at least one) equality. As such, we used 13,924 TPTP problems in our experiments.

On the other hand, when using the SMT-LIB repository, we chose the benchmarks from categories LIA, UF, UFDT, UFDTLIA, and UFLIA, as these benchmarks involve reasoning with both theories and quantifiers and the background theories are the theories that Vampire supports. These are 22,951 SMT-LIB problems in total, of which 22,833 problems remain after removing those where equality does not occur.

Comparative Experiments with Vampire. As a first experimental study, we compared the performance of subsumption demodulation in Vampire for different values of `fsd` and `bsd`, that is by using forward (FSD) and/or backward (BSD) subsumption demodulation. To this end, we evaluated subsumption demodulation using the CASC and SMTCOMP schedules of Vampire's portfolio mode. In order to test subsumption demodulation with the portfolio mode, we added the options `fsd` and/or `bsd` to *all* strategies of Vampire. While the resulting strategy schedules could potentially be further improved, it allowed us to test FSD/BSD with a variety of strategies.

Table 1. Comparing Vampire with and without subsumption demodulation on TPTP, using Vampire in portfolio mode.

Configuration	Total	Solved	New (SAT + UNSAT)
Vampire	13,924	9,923	–
Vampire, with FSD	13,924	9,757	20 (3 + 17)
Vampire, with BSD	13,924	9,797	14 (2 + 12)
Vampire, with FSD and BSD	13,924	9,734	30 (6 + 24)

Our results are summarized in Tables 1 and 2. The first column of these tables lists the Vampire version and configuration, where Vampire refers to Vampire in its portfolio mode (version 4.4). Lines 2–4 of these tables use our new Vampire, that is our implementation of subsumption demodulation in Vampire. The

[3] The other problems contain features, such as higher-order logic, that have not been implemented in Vampire yet.

Table 2. Comparing VAMPIRE with and without subsumption demodulation on SMT-LIB, using VAMPIRE in portfolio mode.

Configuration	Total	Solved	New (SAT + UNSAT)
VAMPIRE	22,833	13,705	–
VAMPIRE, with FSD	22,833	13,620	55 (1 + 54)
VAMPIRE, with BSD	22,833	13,632	48 (0 + 48)
VAMPIRE, with FSD and BSD	22,833	13,607	76 (0 + 76)

column "Solved" reports, respectively, the total number of TPTP and SMT-LIB problems solved by the considered VAMPIRE configurations. Column "New" lists, respectively, the number of TPTP and SMT-LIB problems solved by the version with subsumption demodulation but not by the portfolio version of VAMPIRE. This column also indicates in parentheses how many of the solved problems were satisfiable/unsatisfiable.

While in total the portfolio mode of VAMPIRE can solve more problems, we note that this comes at no surprise as the portfolio mode of VAMPIRE is highly tuned using the existing VAMPIRE options. In our experiments, we were interested to see whether subsumption demodulation in VAMPIRE can solve problems that cannot be solved by the portfolio mode of VAMPIRE. Such a result would justify the existence of the new rule because the set of problems that VAMPIRE can solve in principle is increased. In future work, the portfolio mode should be tuned by also taking into account subsumption demodulation, which then ideally leads to an overall increase in performance. The columns "New" of Tables 1 and 2 give indeed practical evidence of the impact of subsumption demodulation: there are 30 new TPTP problems and 76 SMT-LIB problems[4] that the portfolio version of VAMPIRE cannot solve, but forward and backward subsumption demodulation in VAMPIRE can.

New Problems Solved Only by Subsumption Demodulation. Building upon our results from Tables 1 and 2, we analysed how many new problems subsumption demodulation in VAMPIRE can solve when compared to other state-of-the-art reasoners. To this end, we evaluated our work against the superposition provers E (version 2.4) and SPASS (version 3.9), as well as the SMT solvers CVC4 (version 1.7) and Z3 (version 4.8.7). We note however, that when using our 30 new problems from Table 1, we could not compare our results against Z3 as Z3 does not natively parse TPTP. On the other hand, when using our 76 new problems from Table 2, we only compared against CVC4 and Z3, as E and SPASS do not support the SMT-LIB syntax.

Table 3 summarizes our findings. First, 11 of our 30 "new" TPTP problems can only be solved using forward and backward subsumption demodulation in VAMPIRE; none of the other systems were able solve these problems.

[4] The list of these new problems is available at https://gist.github.com/JakobR/605a7b7db0101259052e137ade54b32c.

Table 3. Comparing Vampire with subsumption demodulation against other solvers, using the "new" TPTP and SMT-LIB problems of Tables 1 and 2 and running Vampire in portfolio mode.

Solver/configuration	TPTP problems	SMT-LIB problems
Baseline: Vampire, with FSD and BSD	30	76
E with `--auto-schedule`	14	–
Spass (default)	4	–
Spass (local contextual rewriting)	6	–
Spass (subterm contextual rewriting)	5	–
CVC4 (default)	7	66
Z3 (default)	–	49
Only solved by Vampire, with FSD and BSD	11	0

Second, while all our 76 "new" SMT-LIB problems can also be solved by CVC4 and Z3 together, we note that out of these 76 problems there are 10 problems that CVC4 cannot solve, and similarly 27 problems that Z3 cannot solve.

Comparative Experiments without AVATAR. Finally, we investigated the effect of subsumption demodulation in Vampire without AVATAR [19]. We used the default mode of Vampire (that is, without using a portfolio approach) and turned off the AVATAR setting. While this configuration solves less problems than the portfolio mode of Vampire, so far Vampire is the only superposition-based theorem prover implementing AVATAR. Hence, evaluating subsumption demodulation in Vampire without AVATAR is more relevant to other reasoners. Further, as AVATAR may often split non-unit clauses into unit clauses, it may potentially simulate applications of subsumption demodulation using demodulation. Table 4 shows that this is indeed the case: with both `fsd` and `bsd` enabled, subsumption demodulation in Vampire can prove 190 TPTP problems and 173 SMT-LIB examples that the default Vampire without AVATAR cannot solve. Again, the column "New" denotes the number of problems solved by the respective configuration but not by the default mode of Vampire without AVATAR.

Table 4. Comparing Vampire in default mode and without AVATAR, with and without subsumption demodulation.

	TPTP problems			SMT-LIB problems		
Configuration	Total	Solved	New (SAT + UNSAT)	Total	Solved	New (SAT + UNSAT)
Vampire	13,924	6,601	–	22,833	9,608	–
Vampire (FSD)	13,924	6,539	152 (13 + 139)	22,833	9,597	134 (1 + 133)
Vampire (BSD)	13,924	6,471	112 (12 + 100)	22,833	9,541	87 (0 + 87)
Vampire (FSD + BSD)	13,924	6,510	190 (15 + 175)	22,833	9,581	173 (1 + 172)

7 Conclusion

We introduced the simplifying inference rule subsumption demodulation to improve support for reasoning with conditional equalities in superposition-based first-order theorem proving. Subsumption demodulation revises existing machineries of superposition provers and can therefore be efficiently integrated in superposition reasoning. Still, the rule remains expensive and does not pay off for all problems, leading to a decrease in total number of solved problems by our implementation in Vampire. However, this is justified because subsumption demodulation also solves many new examples that existing provers, including first-order and SMT solvers, cannot handle. Future work includes the design of more sophisticated approaches for selecting rewriting equalities and improving the imperfect filtering of clauses indexes.

Acknowledgements. This work was funded by the ERC Starting Grant 2014 SYMCAR 639270, the ERC Proof of Concept Grant 2018 SYMELS 842066, the Wallenberg Academy Fellowship 2014 TheProSE, the Austrian FWF research project W1255-N23, the OMAA grant 101öu8, and the WWTF ICT15-193 grant.

References

1. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. Logic Comput. **4**(3), 217–247 (1994)
2. Bachmair, L., Ganzinger, H., McAllester, D.A., Lynch, C.: Resolution theorem proving. In: Handbook of Automated Reasoning, pp. 19–99 (2001)
3. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). www.SMT-LIB.org
5. Barthe, G., Eilers, R., Georgiou, P., Gleiss, B., Kovács, L., Maffei, M.: Verifying relational properties using trace logic. In: Proceedings of FMCAD, pp. 170–178 (2019)
6. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. Fields Logic Comput. **II**, 24–51 (2015)
7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): fast decision procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_14
9. Hillenbrand, T., Piskac, R., Waldmann, U., Weidenbach, C.: From search to computation: redundancy criteria and simplification at work. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics: Essays in Memory of Harald Ganzinger, vol. 7797, pp. 169–193. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37651-1_7

10. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
11. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Handbook of Automated Reasoning, pp. 371–443 (2001)
12. Nivela, P., Nieuwenhuis, R.: Saturation of first-order (constrained) clauses with the saturate system. In: Kirchner, C. (ed.) Rewriting Techniques and Applications, vol. 690, pp. 436–440. Springer, Heidelberg (1993). https://doi.org/10.1007/978-3-662-21551-7_33
13. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 453–474. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_24
14. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 495–507. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
15. Sekar, R., Ramakrishnan, I.V., Voronkov, A.: Term indexing. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1853–1964. Elsevier Science Publishers B.V. (2001)
16. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a cross-community infrastructure for logic solving. In: Proceedings of IJCAR, pp. 367–373 (2014)
17. Sutcliffe, G.: The TPTP problem library and associated infrastructure. From CNF to TH0, TPTP v6.4.0. J. Autom. Reasoning **59**(4), 483–502 (2017)
18. Tange, O.: GNU Parallel 2018. Ole Tange (2018)
19. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 696–710. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_46
20. Weidenbach, C.: Combining superposition, sorts and splitting. In: Handbook of Automated Reasoning, pp. 1965–2013 (2001)
21. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 140–145. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_10
22. Weidenbach, C., Wischnewski, P.: Contextual rewriting in SPASS. In: Proceedings of PAAR (2008)