Blocked Clauses in First-Order Logic*

Benjamin Kiesl[1], Martin Suda[1], Martina Seidl[2], Hans Tompits[1], Armin Biere[2]

[1] Institute for Information Systems, Vienna University of Technology, Austria
[2] Institute for Formal Models and Verification, JKU Linz, Austria

Abstract

Blocked clauses provide the basis for powerful reasoning techniques used in SAT, QBF, and DQBF solving. Their definition, which relies on a simple syntactic criterion, guarantees that they are both redundant and easy to find. In this paper, we lift the notion of blocked clauses to first-order logic. We introduce two types of blocked clauses, one for first-order logic with equality and the other for first-order logic without equality, and prove their redundancy. In addition, we give a polynomial algorithm for checking whether a clause is blocked. Based on our new notions of blocking, we implemented a novel first-order preprocessing tool. Our experiments showed that many first-order problems in the TPTP library contain a large number of blocked clauses. Moreover, we observed that their elimination can improve the performance of modern theorem provers, especially on satisfiable problem instances.

1 Introduction

Modern theorem provers often use dedicated *preprocessing methods* to speed up the proof search [13, 18]. As most of these provers are based on proof systems that require formulas to be in conjunctive normal form (CNF), a wide range of established preprocessing methods performs simplifications on the CNF representation of the input formula. Preprocessing on the CNF level is well explored for propositional logic [10] and has been successfully integrated into SAT solvers such as MiniSat [7], Glucose [1], or Lingeling [4]. But, although generalizations of several propositional preprocessing methods have been utilized by first-order theorem provers, one particularly successful concept has, to the best of our knowledge, not yet found its way to first-order logic: the simple yet powerful concept of *blocked clauses* [20]. In this paper, we address this issue and lift the notion of blocked clauses to first-order logic.

Informally, a clause C is *blocked* by one of its literals in a propositional CNF formula F if all resolvents of C upon this literal are tautologies [20]. A blocked clause is redundant in the sense that neither its deletion from nor its addition to F affects the satisfiability or unsatisfiability of F. Blocked clauses provide the basis for the propositional preprocessing techniques of *blocked-clause elimination* (BCE), *blocked-clause addition* (BCA), and *blocked-clause decomposition* (BCD).

Blocked-clause elimination considerably boosts solver performance by simulating several other, more complicated preprocessing techniques [15]. But not only SAT solvers benefit from BCE; even greater performance improvements are achieved when generalizations of BCE are used for solving problems beyond the complexity class NP such as reasoning over quantified Boolean formulas (QBF) [11] or dependency quantified Boolean formulas (DQBF) [34]. When performed in a careful manner, however, also the *addition* of certain small blocked clauses has shown to be useful [16].

Finally, blocked-clause decomposition [9] is a technique that splits a CNF formula into two parts that can in turn be solved via blocked-clause elimination. Applications of blocked-clause decomposition are, for instance, the identification of backbone variables, the detection of implied equivalences, and gate extraction. Moreover, the winner of the SAT-Race 2015 competition, abcdSAT [5], is based on blocked-clause decomposition.

*This work has been supported by the Austrian Science Fund (FWF) under projects W1255-N23, S11408-N23, S11409-N23, and the ERC Starting Grant 2014 SYMCAR 639270.

The generalization of blocked clauses to the first-order case is not straightforward and poses several challenges; in particular, the following two are crucial: First, the involvement of unification in first-order resolution brings some intricacies with it that are absent in propositional logic. A careful choice of resolvents is therefore essential for ensuring the redundancy of blocked clauses. Second, in the presence of equality, further problems are caused by the fact that Herbrand's Theorem has to be adapted in order to account for the peculiarities of equality. Our approach successfully resolves these issues.

The main contributions of this paper are the following: (1) We present blocked clauses for first-order logic and prove their redundancy given that equality is not present. (2) We introduce equality-blocked clauses, a refined notion of blocked clauses that guarantees redundancy even in the presence of equality. (3) We give a polynomial algorithm for deciding whether a clause is blocked. (4) To demonstrate one potential application of blocked clauses, we implement a tool that performs blocked-clause elimination and evaluate its impact on the performance of modern first-order theorem provers.

This paper is structured as follows. After introducing the necessary preliminaries in Section 2, we shortly recapitulate the propositional notion of blocked clauses and lift it to first-order logic in Section 3. In Section 4, we introduce equality-blocked clauses and prove that they are redundant even when the equality predicate is present. We discuss the complexity of deciding the blockedness of a clause in Section 5. Finally, in Section 6, we present our implementation of blocked-clause elimination, relate it to other first-order preprocessing techniques, and evaluate its impact on first-order theorem provers.

2 Preliminaries

We assume the reader to be familiar with the basics of first-order logic. As usual, formulas of a first-order language $\mathcal{L}$ are built using predicate symbols, function symbols, and constants from some given alphabet together with logical connectives, quantifiers, and variables. We use the letters $a, b, c, \dots$ for constants and $x, y, z, u, v, \dots$ for variables (possibly with subscripts). The equality predicate symbol $\approx$ is used in infix notation and we write $x \not\approx y$ for $\neg(x \approx y)$. An expression (i.e., a term, literal, formula, etc.) is *ground* if it contains no variables.

A *literal* is an atom or the negation of an atom and a disjunction of literals is a *clause*. For a literal L and an atom P, we define $\bar{L} = \neg P$ if $L = P$ and $\bar{L} = P$ if $L = \neg P$. In the former case, L is of *positive polarity*; in the latter case, it is of *negative polarity*. A formula is in *conjunctive normal form* (*CNF*) if it is a conjunction of clauses. W.l.o.g., clauses are assumed to be variable disjoint. Variables of a CNF formula are implicitly universally quantified. We treat CNF formulas as sets of clauses and clauses as multisets of literals. If not stated otherwise, we assume formulas to be in CNF. A clause is a *tautology* if it contains both L and $\bar{L}$ for some literal L.

We use the standard notions of *interpretation*, *model*, *validity*, *satisfiability*, *logical equivalence*, and *satisfiability equivalence*. The predicate symbol $\approx$ is special as it must be **interpreted** as the identity relation over the domain under consideration. A *propositional assignment* is a mapping from ground atoms to the truth values 1 (*true*) and 0 (*false*). Accordingly, a set of ground clauses is *propositionally satisfiable* if there exists a propositional assignment that satisfies F under the usual semantics for the logical connectives. An assignment α' is obtained of an assignment α by *flipping* the truth value of a literal L if α' agrees with α on all atoms except for that of L to which it assigns the opposite truth value. We sometimes write propositional assignments as sequences of literals where a positive (negative) polarity of a literal indicates that its corresponding atom is assigned to true (false, respectively).

A *substitution* is a mapping from variables to terms that agrees with the identity function on all but finitely many variables. Let σ be a substitution. The domain, $dom(\sigma)$, of σ is the set of variables for which $\sigma(x) \neq x$. The range, $ran(\sigma)$, of σ is the set $\{\sigma(x) \mid x \in dom(\sigma)\}$. A substitution is *ground* if its range consists only of ground terms. As common, $E\sigma$ denotes the result of applying σ to the expression E. If $E\sigma$ is ground, it is a *ground instance* of E. Juxtaposition of substitutions denotes their composition, i.e., $\sigma\tau$ stands for $\tau \circ \sigma$. The substitution σ is a *unifier* of the expressions $E_1, \dots, E_n$ if $E_1\sigma = \cdots = E_n\sigma$. For substitutions σ and τ, we say that σ is *more general* than τ if there exists a substitution λ such that $\sigma\lambda = \tau$. Furthermore, σ is a *most general unifier* (mgu) of $E_1, \dots, E_n$ if, for every unifier τ of $E_1, \dots, E_n$, σ is more general than τ. In the rest of the paper, we make use of two popular variants of Herbrand's Theorem (cf. [8]):

Theorem 1. *A formula F that does not contain the equality predicate is satisfiable iff every finite set of ground instances of clauses in F is propositionally satisfiable.*

Furthermore, a formula F that contains the equality predicate is satisfiable iff $F \cup \mathcal{E}_\mathcal{L}$ is satisfiable without the restriction that $\approx$ must be interpreted as the identity relation, where $\mathcal{E}_\mathcal{L}$ denotes the following set of *equality axioms* for the language $\mathcal{L}$ under consideration (cf. [8]):

(E1) $x \approx x$;

(E2) for each n-ary function symbol f in $\mathcal{L}$, $x_1 \not\approx y_1 \vee \cdots \vee x_n \not\approx y_n \vee f(x_1, \dots, x_n) \approx f(y_1, \dots, y_n)$;

(E3) for each n-ary predicate symbol P in $\mathcal{L}$, $x_1 \not\approx y_1 \vee \cdots \vee x_n \not\approx y_n \vee \neg P(x_1, \dots, x_n) \vee P(y_1, \dots, y_n)$.

Hence, the following variant of Herbrand's Theorem for formulas with equality follows:

Theorem 2. *A formula F that contains the equality predicate is satisfiable iff every finite set of ground instances of clauses in $F \cup \mathcal{E}_\mathcal{L}$ is propositionally satisfiable.*

Next, we formally introduce the redundancy of clauses. Intuitively, a clause C is redundant w.r.t. a formula F if neither its addition to F nor its removal from F changes the satisfiability or unsatisfiability of F [12]:

Definition 1. *A clause C is* redundant *w.r.t. a formula F if $F \setminus \{C\}$ and $F \cup \{C\}$ are satisfiability equivalent.*

Note that this notion of redundancy does not require *logical* equivalence of $F \setminus \{C\}$ and $F \cup \{C\}$ and that it is different from the Bachmair-Ganzinger notion of redundancy that is usually employed within the context of ordered resolution [3]. It provides the basis for both clause elimination and clause addition procedures. Note also that the redundancy of a clause C w.r.t. a formula F can be shown by proving that the satisfiability of $F \setminus \{C\}$ implies the satisfiability of $F \cup \{C\}$.

Finally, given two clauses $C = L_1 \vee \cdots \vee L_k \vee C'$ and $D = N_1 \vee \cdots \vee N_l \vee D'$ such that the literals $L_1, \dots, L_k, \bar{N}_1, \dots, \bar{N}_l$ are unifiable by an mgu σ, the clause $C'\sigma \vee D'\sigma$ is said to be a *resolvent* of C and D. If $k = l = 1$, it is a *binary resolvent* of C and D upon L_1.

3 Blocked Clauses

In this section, we first recapitulate the notion of blocked clauses used in propositional logic. We then illustrate complications that arise when lifting blocked clauses to first-order logic. As main result of the section, we introduce blocked clauses for first-order logic and prove that they are redundant if the equality predicate is not present. Throughout this section, we therefore consider only clauses and formulas without the equality predicate.

In propositional logic, a clause C is *blocked* by a literal $L \in C$ in a CNF formula F if all binary resolvents of C upon L with clauses from $F \setminus \{C\}$ are tautologies. A clause C is blocked in a formula F if C is blocked in F by one (or more) of its literals.

Example 1. *The clause $C = \neg P \vee Q$ is blocked by $\neg P$ in $F = \{P \vee \neg Q, \neg Q \vee R\}$: The only resolvent of C upon $\neg P$ is the tautology $Q \vee \neg Q$, obtained by resolving with $P \vee \neg Q$.*

Under the restriction—common in propositional logic—that clauses must not contain multiple occurrences of the same literal, it can be shown that blocked clauses are redundant: Let C be blocked by $L \in C$ in a formula F. Then, every assignment that satisfies $F \setminus \{C\}$ but falsifies C can be turned into a satisfying assignment of C by simply flipping the truth value of L, i.e., by inverting the truth value of its atom. This flipping does not falsify any of the clauses in $F \setminus \{C\}$ that contains $\bar{L}$, because of the fact that every binary resolvent of C upon L is a tautology: A clause that contains $\bar{L}$ either is itself a tautology or it contains a literal $R \neq \bar{L}$ such that $\bar{R} \in C$. In the latter case, since C and thus $\bar{R}$ was assumed to be false before the flipping of the truth value of L, R also stays true afterwards.

Example 2. *Consider again C and F from Example 1. The assignment $P \neg Q R$ satisfies $F \setminus \{C\}$ but falsifies C. By flipping the truth value of $\neg P$, we obtain the assignment $\neg P \neg Q R$ that satisfies $F \cup \{C\}$. The only clause that could have possibly been falsified, namely $P \vee \neg Q$, stays true since it contains $\neg Q$ which was true before the flipping.*

As can be seen in the next example, redundancy is not guaranteed when clauses are allowed to contain multiple occurrences of the same literal. Although the example might seem pathological at first, it will help to illustrate an inherent complication arising in first-order logic:

Example 3. *Let $C = P \vee P$ and $F = \{\neg P \vee \neg P\}$. Clearly, $F \setminus \{C\} = F$ is satisfiable whereas $F \cup \{C\}$ is not. There is one binary resolvent of C upon P, namely the tautology $P \vee \neg P$, hence C is blocked by P in F. However, turning a satisfying assignment of F (i.e., one that falsifies P) into one of C by flipping the truth value of P falsifies $\neg P \vee \neg P$.*

In first-order logic, the requirement that all binary resolvents of C upon L are valid[1] fails to guarantee redundancy, even when clauses are not allowed to contain multiple occurrences of the same literal. The reason is that similar issues as in Example 3 might occur on the ground level after certain literals are instantiated through unification:

Example 4. *Consider $C = P(x,y) \vee P(y,x)$ and $F = \{\neg P(u,v) \vee \neg P(v,u)\}$. Two binary resolvents can be derived from C upon $P(x,y)$ and both are valid: The resolvent $P(v,u) \vee \neg P(v,u)$, obtained by using the mgu $\{x \mapsto u, y \mapsto v\}$ of $P(x,y)$ and $P(u,v)$, and the resolvent $P(u,v) \vee \neg P(u,v)$, obtained by using the mgu $\{x \mapsto v, y \mapsto u\}$ of $P(x,y)$ and $P(v,u)$. However, the formula $F \setminus \{C\} = F$ is clearly satisfiable whereas $F \cup \{C\}$ is not. To see this, observe that there exists no satisfying assignment for the two ground instances $P(c,c) \vee P(c,c)$ and $\neg P(c,c) \vee \neg P(c,c)$ of C and $\neg P(u,v) \vee \neg P(v,u)$, respectively. Since $P(u,v)$ and $P(v,u)$ both unify with $P(x,y)$ on the ground level, we face the same problem as in Example 3.*

Such examples are often used for illustrating that binary resolution alone does not guarantee completeness of the resolution principle (see, e.g., [8] or [21]). Analogously, we have to test the validity of more than just its binary resolvents in order to guarantee the redundancy of a clause. But there is no need to test all possible resolvents. As we will see, it is enough to consider the following ones:

Definition 2. *Let $C = L \vee C'$ and $D = N_1 \vee \cdots \vee N_l \vee D'$ with $l > 0$ be clauses such that the literals $L, \bar{N}_1, \ldots, \bar{N}_l$ are unifiable by an mgu σ. Then, $C'\sigma \vee D'\sigma$ is called L-*resolvent *of C and D.*

Definition 3. *A clause C is* blocked *by a literal $L \in C$ in a formula F if all L-resolvents of C with clauses in $F \setminus \{C\}$ are valid.*

For instance, in Example 4, the clause C is *not* blocked by $L = P(x,y)$ in F. In addition to the two valid binary resolvents—which are both L-resolvents—already considered in the example, there is

[1] As common in first-order logic, we use the notion of *validity* instead of *tautologyhood*. In the absence of equality, a clause is valid if and only if it contains two complementary literals $L, \bar{L}$.

another L-resolvent of C and $\neg P(u,v) \vee \neg P(v,u)$, namely $P(x,x)$ (which is not valid), obtained by unifying $P(x,y)$, $P(u,v)$, and $P(v,u)$ via the mgu $\{y \mapsto x, u \mapsto x, v \mapsto x\}$. Example 5 shows a clause that is blocked according to Definition 3:

Example 5. *Let $C = P(x,y) \vee \neg P(y,x) \vee Q(b)$ and $F = \{\neg P(a,b) \vee P(b,a), \neg Q(b)\}$. Then, $L = P(x,y)$ blocks C in F since there is only a single L-resolvent of C upon $P(x,y)$, namely $\neg P(b,a) \vee Q(b) \vee P(b,a)$, obtained by using the mgu $\{x \mapsto a, y \mapsto b\}$ of the literals $P(x,y)$ and $P(a,b)$, and this resolvent is valid.*

Similar to the propositional case, where a satisfying assignment of $F \setminus \{C\}$ (with C being blocked in F) can be turned into one of $F \cup \{C\}$ by flipping the truth value of the blocking literal, we can satisfy ground instances of blocked clauses in first-order logic. For instance, in Example 5, the assignment $\alpha = \neg P(a,b)P(b,a)\neg Q(b)$ satisfies $F \setminus \{C\}$ (which is already ground) but falsifies the ground instance $P(a,b) \vee \neg P(b,a) \vee Q(b)$ of C. By flipping the truth value of $P(a,b)$ we obtain $\alpha' = P(a,b)P(b,a)\neg Q(b)$—a satisfying assignment of this ground instance that still satisfies F.

Lemma 3. *Let C be blocked by L in F, and α a propositional assignment that falsifies a ground instance $C\lambda$ of C. Then, the assignment α', obtained from α by flipping the truth value of $L\lambda$, satisfies all the ground instances of clauses in $F \setminus \{C\}$ that are satisfied by α.*

Proof. Let $D\tau$ be a ground instance of a clause $D \in F \setminus \{C\}$ and suppose α satisfies $D\tau$. If $D\tau$ does not contain $\bar{L}\lambda$ it is trivially satisfied by α'. Assume therefore that $\bar{L}\lambda \in D\tau$ and let $N_1, \dots, N_l$ be all the literals in D such that $N_i\tau = \bar{L}\lambda$ for $1 \leq i \leq l$. Then, the substitution $\lambda\tau = \lambda \cup \tau$ (note that C and D are variable disjoint by assumption) is a unifier of $L, \bar{N}_1, \dots, \bar{N}_l$. Since C is blocked by L in F, the L-resolvent $(C \setminus \{L\})\sigma \vee (D \setminus \{N_1, \dots, N_l\})\sigma$, with σ being an mgu of $L, \bar{N}_1, \dots, \bar{N}_l$, is valid. As σ is most general, it follows that $\sigma\gamma = \lambda\tau$ for some substitution γ. Hence,

$$\begin{array}{lll} & (C \setminus \{L\})\sigma\gamma & \vee (D \setminus \{N_1, \dots, N_l\})\sigma\gamma \\ = & (C \setminus \{L\})\lambda\tau & \vee (D \setminus \{N_1, \dots, N_l\})\lambda\tau \\ = & (C \setminus \{L\})\lambda & \vee (D \setminus \{N_1, \dots, N_l\})\tau \end{array}$$

is valid. Thus, since α falsifies $C\lambda$, it must satisfy a literal $L'\tau \in (D \setminus \{N_1, \dots, N_l\})\tau$. But, as all the literals in $(D \setminus \{N_1, \dots, N_l\})\tau$ are different from $\bar{L}\lambda$, flipping the truth value of $L\lambda$ does not affect the truth value of $L'\tau$. It follows that α' satisfies $L'\tau$ and thus it satisfies $D\tau$. □

A falsified ground instance $C\lambda$ of C can therefore be satisfied without falsifying any ground instances of clauses in $F \setminus \{C\}$ by simply flipping the truth value of $L\lambda$. Still, it could happen that this flipping falsifies other ground instances of C itself, namely those in which the only satisfied literals are complements of $L\lambda$. As it turns out, this is not a serious problem. Consider the following example:

Example 6. *Given C and F from Example 5, let $P(a,b) \vee \neg P(b,a) \vee Q(b)$ and $P(b,a) \vee \neg P(a,b) \vee Q(b)$ be the two ground instances[2] of C that are not valid. As shown above, the satisfying assignment $\alpha = \neg P(a,b)P(b,a)\neg Q(b)$ of F can be turned into the satisfying assignment $\alpha' = P(a,b)P(b,a)\neg Q(b)$ of $P(a,b) \vee \neg P(b,a) \vee Q(b)$ by flipping the truth value of $P(a,b)$. Now, α' falsifies the other ground instance $P(b,a) \vee \neg P(a,b) \vee Q(b)$ of C.*

But, by flipping the truth value of yet another instance of the blocking literal—this time that of $P(b,a)$—we can also satisfy $P(b,a) \vee \neg P(a,b) \vee Q(b)$. We don't need to worry that this flipping falsifies $P(a,b) \vee \neg P(b,a) \vee Q(b)$ again—the instance $P(a,b)$ of the blocking literal cannot be falsified by making a literal of the form $P(\dots)$ true. The resulting assignment $\alpha'' = P(a,b)P(b,a)\neg Q(b)$ is then a satisfying assignment of all ground instances of clauses in $F \cup \{C\}$.

[2] With respect to the (here) finite Herbrand universe $\{a, b\}$.

The proof of the following lemma is based on the idea of repeatedly making instances of the blocking literal true. We remark that—thanks to this lemma—the definition of a blocked clause can safely ignore resolvents of the clause C with itself. It is not a priori obvious that these resolvents can be ignored when lifting the propositional notion since "on the ground level" two different instances of C may become premises of a resolution step. (For this exact reason, Khasidashvili and Korovin [18] restrict their attention to non-self-referential predicates with their predicate elimination technique, a lifting of variable elimination [6].)

Lemma 4. *Let C be blocked in F and let F' and F_C be finite sets of ground instances of clauses in $F \setminus \{C\}$ and $\{C\}$, respectively. Then, every assignment that propositionally satisfies F' can be turned into one that satisfies $F' \cup F_C$.*

Proof. Let C be blocked by L in F and let α be a satisfying assignment of F'. Assume furthermore that α does not satisfy F_C, i.e., there exist ground instances of C that are falsified by α. By Lemma 4, for every falsified ground instance $C\lambda$ of C, we can turn α into a satisfying assignment of $C\lambda$ by flipping the truth value of $L\lambda$. Moreover, this flipping does not falsify any clauses in F'. The only clauses that could possibly be falsified are other ground instances of C that contain the literal $\bar{L}\lambda$.

But, once an instance $L\tau$ of the blocking literal L is true in a ground instance $C\tau$ of C, this ground instance cannot (later) be falsified by making other instances of L true (since it has, of course, the same polarity as L). As there are only finitely many clauses in F_C, we can therefore turn α into a satisfying assignment of $F' \cup F_C$ by repeatedly making ground instances of C true by flipping the truth values of their instances of the blocking literal L. □

Theorem 5. *If a clause is blocked in a formula F, it is redundant w.r.t. F.*

Proof. Let C be blocked by L in F and suppose $F \setminus \{C\}$ is satisfiable. We show that $F \cup \{C\}$ is satisfiable. By Herbrand's theorem (Theorem 1), it suffices to show that every finite set of ground instances of clauses in $F \cup \{C\}$ is propositionally satisfiable. Let therefore F' and F_C be finite sets of ground instances of clauses in $F \setminus \{C\}$ and $\{C\}$, respectively. Clearly, F' must be propositionally satisfiable for otherwise $F \setminus \{C\}$ were not satisfiable. By Lemma 4, every satisfying propositional assignment of F' can be turned into one of $F' \cup F_C$. It follows that $F \cup \{C\}$ is satisfiable. □

4 Equality-Blocked Clauses

In the following, we first illustrate why the blocking notion from the previous section fails to guarantee redundancy in the presence of equality. We then introduce a refined notion of blocking, *equality-blocking*, and prove that equality-blocked clauses are redundant even if the equality predicate is present.

Example 7. *Let $C = P(a)$ and $F = \{a \approx b, \neg P(b)\}$. Since $P(a)$ and $P(b)$ are not unifiable, there are no resolvents of C, hence $P(a)$ trivially blocks C in F. But, F is clearly satisfiable whereas $F \cup \{C\}$ is not.*

In Example 7, every model of F must assign the same truth value to $P(a)$ and $P(b)$. Hence, when trying to turn a model of F into one of $F \cup \{C\}$ by flipping the truth value of $P(a)$, we implicitly flip the truth value of $P(b)$ although $P(a)$ and $P(b)$ are not unifiable.

Thus, in the presence of equality, it is not enough to consider only the clauses that are resolvable with C. We need to take all clauses that contain a literal of the form $\bar{L}(\dots)$ into account. In order to do so, we make use of *flattening* as introduced by Khasidashvili and Korovin [18]:

Definition 4. *Let $C = L(t_1, \dots, t_n) \vee C'$.* Flattening *the literal $L(t_1, \dots, t_n)$ in C yields the clause $C^- = \bigvee_{1 \leq i \leq n} x_i \not\approx t_i \vee L(x_1, \dots, x_n) \vee C'$, with $x_i, \dots, x_n$ being fresh variables not occurring in C.*

Example 8. *Flattening the literal $P(f(x), c, c)$ in clause $P(f(x), c, c) \lor Q(c)$ yields the new clause $x_1 \not\approx f(x) \lor x_2 \not\approx c \lor x_3 \not\approx c \lor P(x_1, x_2, x_3) \lor Q(c)$.*

The clause resulting from flattening $L(t_1, \dots, t_n)$ in $L(t_1, \dots, t_n) \lor C'$ is equivalent to an implication of the form $(x_1 \approx t_1 \land \dots \land x_n \approx t_n) \rightarrow (L(x_1, \dots, x_n) \lor C')$. Thus, flattening preserves equivalence. Using flattening, we can define *flat resolvents*. Intuitively, flat resolvents are obtained by first flattening literals and then resolving them. This enables us to resolve literals that might otherwise not be unifiable.

Definition 5. *Let $C = L \lor C'$ and $D = N_1 \lor \dots \lor N_l \lor D'$ with $l > 0$ be clauses such that the literals $L, \bar{N}_1, \dots, \bar{N}_l$ have the same predicate symbol and polarity. Let furthermore C^- and D^- be obtained from C and D, respectively, by flattening $L, N_1, \dots, N_l$ and denote the flattened literals by $L^-, N_1^-, \dots, N_l^-$. The resolvent*

$$(C^- \setminus \{L^-\})\sigma \ \lor (D^- \setminus \{N_1^-, \dots, N_l^-\})\sigma$$

of C^- and D^-, with σ being an mgu of $L^-, \bar{N}_1^-, \dots, \bar{N}_l^-$, is a flat L-resolvent *of C and D.*

Note that the unifier $\bigcup_{i=1}^{l} \{y_{ij} \mapsto x_j \mid 1 \leq j \leq n\}$ of $L(x_1, \dots, x_n)$, $\bar{N}_1(y_{11}, \dots, y_{1n})$, $\dots$, $\bar{N}_l(y_{l1}, \dots, y_{ln})$ is a most general unifier (cf. [2]).

Example 9. *Let $C = P(a)$ and $D = \neg P(b)$ (cf. Example 7). By flattening $P(a)$ in C and $\neg P(b)$ in D we obtain $C^- = x_1 \not\approx a \lor P(x_1)$ and $D^- = y_1 \not\approx b \lor \neg P(y_1)$, respectively. Their resolvent $x_1 \not\approx a \lor x_1 \not\approx b$ (which is not valid) is a flat $P(a)$-resolvent of C and D.*

The following definition prohibits blocking by an equality literal. This is because equality must be treated specially in our extension of the flipping argument (see below). After this intuitive discussion, we formally define equality-blocking as follows:

Definition 6. *A clause C is* equality-blocked *by a literal $L \in C$ in a formula F if the predicate of L is not $\approx$ and all flat L-resolvents of C with clauses in $F \setminus \{C\}$ are valid.*

Note that in the presence of equality, clauses without complementary literals, like $x \approx x$, can be valid. Before we prove redundancy, we consider the following example that stems from a first-order encoding of an AI-benchmark problem known as "Who killed Aunt Agatha?" [24] and that illustrates the power of equality-blocked clauses:

Example 10. *Let F be the following set of four clauses: $\{L(a), L(b), L(c), \neg L(x) \lor x \approx a \lor x \approx b \lor x \approx c\}$. Intuitively, the clauses $L(a)$, $L(b)$, and $L(c)$ encode that there are three living individuals: Agatha, Butler, and Charles. The clause $\neg L(x) \lor x \approx a \lor x \approx b \lor x \approx c$ encodes that these three individuals are the only living individuals. We can observe that all four clauses are equality-blocked in F. For instance, let $C = L(a)$. There exists one flat $L(a)$-resolvent of C: the valid clause $x_1 \not\approx a \lor x_1 \not\approx x \lor x \approx a \lor x \approx b \lor x \approx c$, obtained by resolving the clause $x_1 \not\approx a \lor L(x_1)$ with $y_1 \not\approx x \lor \neg L(y_1) \lor x \approx a \lor x \approx b \lor x \approx c$.*

In order to show that equality-blocked clauses are redundant, we introduce the notion of *equivalence flipping*. Intuitively, equivalence flipping of a ground literal $L(t_1, \dots, t_n)$ turns a propositional assignment α into an assignment α' by inverting the truth value of $L(t_1, \dots, t_n)$ as well as that of all $L(s_1, \dots, s_n)$ for which α satisfies $t_1 \approx s_1, \dots, t_n \approx s_n$.

Definition 7. *Let α be a propositional assignment and $L(t_1, \dots, t_n)$ a ground literal with predicate symbol P other than $\approx$. The assignment α', obtained by* equivalence flipping *the truth value of $L(t_1, \dots, t_n)$, is defined as follows:*

$$\alpha'(A) = \begin{cases} 1 - \alpha(A) & \text{if } A = P(s_1, \dots, s_n) \text{ and} \\ & \quad \alpha(t_i \approx s_i) = 1 \text{ for all } 1 \leq i \leq n, \\ \alpha(A) & \text{otherwise.} \end{cases}$$

Obviously, equivalence flipping preserves the truth of instances of the equality axioms, leading to the equality counterpart of Lemma 3:

Lemma 6. *Let C be equality-blocked by L in F, and α a propositional assignment that satisfies all ground instances of the equality axioms, $\mathcal{E}_\mathcal{L}$, but falsifies a ground instance $C\lambda$ of C. Then, the assignment α', obtained from α by equivalence flipping the truth value of $L\lambda$, satisfies all the ground instances of clauses in $\mathcal{E}_\mathcal{L} \cup F \setminus \{C\}$ that are satisfied by α.*

Proof. Let $L = L(t_1, \ldots, t_n)$ and $C = L \vee C'$ and suppose α falsifies a ground instance $C\lambda$ of C. By definition, the only clauses that are affected by the equivalence flipping of $L(t_1, \ldots, t_n)\lambda$ are clauses of the form $D\tau$, with $D \in F \setminus \{C\}$ and $\bar{L}(s_1, \ldots, s_n)\tau \in D\tau$ such that $\alpha(t_i\lambda \approx s_i\tau) = 1$ for $1 \leq i \leq n$.

Let $D\tau$ be such a clause and let $\bar{L}(s_1, \ldots, s_n), \ldots, \bar{L}(r_1, \ldots, r_n)$ be all literals in D such that α satisfies $t_i\lambda \approx s_i\tau, \ldots, t_i\lambda \approx r_i\tau$ for $1 \leq i \leq n$. To simplify the presentation, we assume that $\bar{L}(s_1, \ldots, s_n)$ and $\bar{L}(r_1, \ldots, r_n)$ are all such literals. The proof for another number of such literals is analogous. We observe that D is of the form $\bar{L}(s_1, \ldots, s_n) \vee \bar{L}(r_1, \ldots, r_n) \vee D'$.

Since C is equality-blocked by $L(t_1, \ldots, t_n)$ in F, all flat $L(t_1, \ldots, t_n)$-resolvents of C are valid. Therefore, the flat $L(t_1, \ldots, t_n)$-resolvent

$$R = (C' \vee D' \vee \bigvee_{1 \leq i \leq n} x_i \not\approx t_i \vee y_i \not\approx s_i \vee z_i \not\approx r_i)\sigma$$

is valid, where σ is an mgu of the literals $L(x_1, \ldots, x_n), L(y_1, \ldots, y_n)$, and $L(z_1, \ldots, z_n)$, which were obtained by respectively flattening $L(t_1, \ldots, t_n)$, $\bar{L}(s_1, \ldots, s_n)$, and $\bar{L}(r_1, \ldots, r_n)$. Assume w.l.o.g. that $\sigma = \{y_i \mapsto x_i \mid 1 \leq i \leq n\} \cup \{z_i \mapsto x_i \mid 1 \leq i \leq n\}$. Then,

$$R = C' \vee D' \vee \bigvee_{1 \leq i \leq n} x_i \not\approx t_i \vee x_i \not\approx s_i \vee x_i \not\approx r_i.$$

As R is valid, the assignment α must satisfy all ground instances of R. Consider therefore the following substitution γ that yields a ground instance $R\gamma$ of R:

$$\gamma(x) = \begin{cases} t_i\lambda & \text{if } x \in \{x_1, \ldots, x_n\}, \\ x\lambda & \text{if } x \in var(C'), \\ x\tau & \text{if } x \in var(D'). \end{cases}$$

We observe that the ground instance $R\gamma$ of R is the clause

$$C'\lambda \vee D'\tau \vee \bigvee_{1 \leq i \leq n} t_i\lambda \not\approx t_i\lambda \vee t_i\lambda \not\approx s_i\tau \vee t_i\lambda \not\approx r_i\tau$$

which must be satisfied by α. Now, all the $t_i\lambda \not\approx t_i\lambda$ are clearly falsified by α. Furthermore, by assumption, α falsifies all the $t_i\lambda \not\approx s_i\tau$ and all the $t_i\lambda \not\approx r_i\tau$ as well as $C'\lambda$. But then, α must satisfy at least one of the literals in $D'\tau$. Since none of the literals in $D'\tau$ are affected by equivalence flipping the truth value of $L\lambda$, $D'\tau$ must be satisfied by α'. It follows that α' satisfies $D\tau$. □

Using Lemma 6 instead of Lemma 3 and replacing the notion of flipping by that of equivalence flipping, the proof of the following lemma is analogous to the one of Lemma 4:

Lemma 7. *Let C be a clause that is equality-blocked in F. Let furthermore F' and F_C be finite sets of ground instances of clauses in $F \setminus \{C\}$ and $\{C\}$, respectively. Then, every assignment that propositionally satisfies all the ground instances of $\mathcal{E}_\mathcal{L}$ as well as F' can be turned into one that satisfies $F' \cup F_C$ and all the ground instances of $\mathcal{E}_\mathcal{L}$.*

Using Lemma 7 and the equality variant of Herbrand's Theorem (Theorem 2), the proof of the following theorem is similar to that of Theorem 5:

Theorem 8. *If a clause is equality-blocked in a formula F, it is redundant w.r.t. F.*

Proof. Let C be equality-blocked in F. Assuming that $F \setminus \{C\}$ is satisfiable, we conclude that there exists an assignment α that propositionally satisfies all ground instances of clauses in $(F \setminus \{C\}) \cup \mathcal{E}_{\mathcal{L}}$ (by using Theorem 2 together with compactness). Using Lemma 7, one can then show that every finite set of ground instances of clauses in $F \cup \{C\} \cup \mathcal{E}_{\mathcal{L}}$ can be satisfied by modifying α. It follows, again by Theorem 2, that $F \cup \{C\}$ is satisfiable. □

5 Complexity of Detecting Blocked Clauses

In this section, we show that deciding whether a clause C is blocked (or equality-blocked) by a literal L in a formula F can be decided in polynomial time. From the definitions of blocking (Definition 3) and equality-blocking (Definition 6) this is not obvious, because a direct implementation of these definitions would require to test exponentially many (flat) L-resolvents of C for validity. Although the number of clauses in $F \setminus \{C\}$ with which C could possibly be resolved is linearly bounded by the size of F, there can be exponentially many L-resolvents of C with a single clause $D \in F \setminus \{C\}$. For example, consider the clause $C = L \lor C'$ and assume that $F \setminus \{C\}$ contains a clause $D = N_1 \lor \cdots \lor N_n \lor D'$ such that the literals $L, \bar{N}_1, \ldots, \bar{N}_n$ are unifiable. We then have one L-resolvent of C and D for every non-empty subset of $N_1, \ldots, N_n$. Therefore, there are $2^n - 1$ such L-resolvents whose validity we have to check. To show how this can be done in polynomial time, we first argue that the validity of a (flat) L-resolvent is decidable in polynomial time and then show that it actually suffices to check the validity of only polynomially many L-resolvents.

In the case without equality, checking the validity of an L-resolvent basically amounts to looking for a complementary pair of literals. Assume we want to check the validity of an L-resolvent $(C' \lor D')\sigma$ of clauses $C' \lor L$ and $N_1 \lor \cdots \lor N_n \lor D'$ where σ is an mgu of $L, \bar{N}_1, \ldots, \bar{N}_n$. Although the size of σ can be exponential in the worst case, this exponential blow up can be avoided by not computing σ explicitly but only computing the *unification closure* [17] of $L, \bar{N}_1, \ldots, \bar{N}_n$, which can be done in polynomial time. The unification closure is basically an equivalence relation under which two literals are considered equivalent if they are unified by a most general unifier of $L, \bar{N}_1, \ldots, \bar{N}_n$. Checking the validity of $(C' \lor D')\sigma$ then boils down to checking whether $C' \lor D'$ contains two literals that are complementary w.r.t. the unification closure.

In the case of equality-blocking, we work with flat L-resolvents. Computing a flat L-resolvent is easy since—as pointed out in the section on equality-blocked clauses—there exists a trivial (and small) mgu of the flattened literals. Furthermore, a flat L-resolvent R is valid if and only if the negation of its universal closure $\neg\forall R$ is unsatisfiable. After skolemization (which introduces fresh constants for the variables of R), the formula $\neg\forall R$ becomes a conjunction of ground (equational) literals and can therefore be efficiently decided by a congruence-closure algorithm (cf. [29]).

Algorithm 1 shows a polynomial-time procedure for checking whether all L-resolvents of a candidate clause $C = L \lor C'$ and a partner clause D are valid. We do this here for the non-equational case and leave the details of the equational case for the appendix (see Appendix A). With this procedure deciding whether a clause C is blocked in a formula F can be done in polynomial time by iterating over all the potential blocking literals $L \in C$ and all the partner clauses $D \in F \setminus \{C\}$. Practical details on how to efficiently implement this top-level iteration will be discussed in Section 6.1.

The inputs of Algorithm 1 are a candidate clause $C = L \lor C'$ and a partner clause $D = N_1 \lor \ldots \lor N_n \lor D'$, where the literals $N_1, \ldots, N_n$ are all the literals of D which pairwise unify with $\bar{L}$. It is easy to see that the running time of the procedure is quadratic in n: We perform n iterations of the for loop

Algorithm 1 Testing validity of L-resolvents

Input:

A candidate clause $C = L \vee C'$ and a partner clause $D = N_1 \vee \ldots \vee N_n \vee D'$,
where the literals $N_1, \ldots, N_n$ are all the literals of D which pairwise unify with $\bar{L}$

Output:

Indication whether all L-resolvents of $L \vee C'$ and D are valid

1: **for** $k \leftarrow 1, \ldots, n$ **do**
2:     $N \leftarrow \{N_k\}$
3:     **while** L is unifiable with the literals in $\bar{N}$ via an mgu σ **do**
4:         Let K contain all pairs of complementary literals in the L-resolvent $C'\sigma \vee (D \setminus N)\sigma$
5:         **if** $K = \emptyset$ **then**
6:             **return** NO
7:         **if** every pair of complementary literals in K contains a literal $N_i\sigma$ **then**
8:             $N \leftarrow N \cup \{N_i \mid N_i\sigma \text{ is part of a complementary pair}\}$
9:         **else**
10:             **break** (the while loop)
11: **return** YES

and at most n iterations of the inner while loop (since there are no more than $n-1$ literals N_i that can be added to N in line 8). Therefore, only quadratically many L-resolvents are explicitly tested for validity. By proving that Algorithm 1 is a sound and complete procedure for testing whether all L-resolvents of a candidate clause C with a partner clause D are valid, we show that this is sufficient:

Theorem 9. *Algorithm 1 returns YES if and only if all L-resolvents of C with D are valid.*

Proof. For the $\Leftarrow$-direction, assume that all L-resolvents of C with D are valid, i.e., they all contain at least one pair of complementary literals. It follows that line 6 is never executed and therefore the algorithm returns YES.

For the $\Rightarrow$-direction, let $C = C' \vee L$ and let $R = C'\sigma \vee (D \setminus N)\sigma$ be an L-resolvent of C and D with σ being an mgu of L and $\bar{N}$ (note that N is a *set* of literals). If the algorithm has explicitly tested R for validity (in line 5), then the statement clearly holds. Assume thus that R has not been explicitly tested for validity. Now, let N' be a maximal subset of N for which the validity of the L-resolvent $R' = C'\sigma' \vee (D \setminus N')\sigma'$ (with σ' being an mgu of L and $\bar{N}'$) has been explicitly tested.[3] Clearly, such an N' must exist since the algorithm explicitly tests the validity of all binary resolvents upon L (in the first iteration of the while-loop, for every iteration of the for-loop). As the algorithm returned YES, we know that R' must be valid.

From N' being maximal it follows that R' contains a complementary pair of literals $P(t_1, \ldots, t_n)\sigma'$ and $\neg P(s_1, \ldots, s_n)\sigma'$ such that $P(t_1, \ldots, t_n)$ and $\neg P(s_1, \ldots, s_n)$ are both not contained in N: otherwise the algorithm would have continued by testing the validity of an L-resolvent $C'\sigma'' \vee (D \setminus N'')\sigma''$ with $N' \subset N'' \subseteq N$ (by extending N' in line 8 and then testing validity in the next iteration of the while-loop). It follows that $P(t_1, \ldots, t_n)\sigma$ and $\neg P(s_1, \ldots, s_n)\sigma$ are both contained in R. Now, since σ unifies L with $\bar{N}$, it unifies L with $\bar{N}'$. Moreover, since σ' is a most general unifier of L and $\bar{N}'$, it is more general than σ and therefore there exists a substitution γ such that $\sigma'\gamma = \sigma$. But then, since $P(t_1, \ldots, t_n)\sigma' = P(s_1, \ldots, s_n)\sigma'$ it follows that $P(t_1, \ldots, t_n)\sigma = P(t_1, \ldots, t_n)\sigma'\gamma = P(s_1, \ldots, s_n)\sigma'\gamma = P(s_1, \ldots, s_n)\sigma$ and thus R is valid. □

[3] In other words, N' should be a subset of N for which there exists no other subset N'' of N such that (1) $N' \subset N''$, and (2) the validity of an L-resolvent $C'\sigma'' \vee (D \setminus N'')\sigma''$ has been explicitly tested by the algorithm in line 5.

In conclusion, we have shown that it suffices to perform polynomially many validity checks—each of which can be performed in polynomial time—to decide whether a clause is blocked.

6 Blocked-Clause Elimination in First-Order Logic

In this section, we present the implementation and empirical evaluation of a first-order preprocessing tool that performs one possible application of blocked clauses, namely blocked-clause elimination (BCE). We further discuss how BCE eliminates pure predicates and how it is related to the existing preprocessing technique of *unused definition elimination* (UDE) by Hoder et al. [13].

6.1 Implementation

We implemented blocked-clause elimination and equality-blocked-clause elimination for first-order logic as a preprocessing step in the automated theorem prover VAMPIRE [19].[4] This preprocessing step can be activated by providing the command line flag `-bce on`. Depending on whether the formula at hand contains the equality predicate or not, VAMPIRE then performs either the elimination of equality-blocked clauses or blocked clauses. It will be performed as the last step in the preprocessing pipeline, because it relies on the input being in CNF. After the preprocessing, instead of proceeding to proving the formula—which is the default behavior—VAMPIRE can be instructed to output the final set of clauses by specifying `--mode clausify` on the command line.

The top level organization of our elimination procedure, which is the same for both blocked-clause elimination and equality-blocked-clause elimination, is inspired by the approach adopted in the propositional case by Järvisalo et al. (c.f. [14], section 7). For efficiency, we maintain an index for accessing a literal within a clause by its predicate symbol and polarity. The main data structure is a priority queue of candidates (L, C) where L is a potential blocking literal in a clause C. We prioritize for processing those candidates (L, C) which have fewer potential resolution partners estimated by the number of clauses indexed with the same predicate symbol and the opposite polarity as L.[5]

At the beginning, every (non-equational) literal L in a clause C gives rise to a candidate (L, C). We always pick the next candidate (L, C) from the queue and iterate over potential resolution partners D. If we discover that a (flat) L-resolvent of C and D is not valid, further processing of (L, C) is postponed and the candidate is "remembered" by the partner clause D. If, on the other hand, all the (flat) L-resolvents with all the possible partners D have been found valid, the clause C is declared blocked and the candidates remembered by C are "resurrected" and put back to the queue. Their processing will be resumed by iterating over those partners which have not been tried yet.

Although, as we have shown in Section 5, testing whether all the (flat) L-resolvents of a clause C and a partner clause D are valid can be done in polynomial time, our implementation uses for efficiency reasons an approximate solution, which only computes binary (flat) resolvents. Then, before testing the resolvent for validity, we remove from it all the literals that (1) are unifiable with $\bar{L}\sigma$ in the blocking case, or (2) have the same predicate symbol and polarity as $\bar{L}$ in the equality-blocking case. This still ensures redundancy and significantly improves the performance.

For testing validity of flat L-resolvents in the equality case, we experimented with a complete congruence-closure procedure which turned out to be too inefficient. Our current implementation only "normalizes" in a single pass all (sub-)terms of the literals in the flat resolvent using the equations from

[4] A statically compiled x86_64 executable of VAMPIRE used in our experiments can be obtained from `http://forsyte.at/wp-content/uploads/vampire_bce.zip`.

[5] We remark that, similarly to the propositional case, blocked-clause elimination in first-order logic is confluent. This means that the resulting set of clauses is always the same regardless of the elimination order. The ordering of candidates in our queue can therefore influence the computation time, but not the output of our procedure.

the flattening, but ignores (dis-)equations originally present in the two clauses and does not employ the congruence rule recursively. Our experiments show that even this limited version is effective.

6.2 Relation to Pure Predicate Elimination and Unused Definition Elimination

In the propositional setting, blocked-clause elimination is known to simulate on the CNF-level several refinements of the standard CNF encoding for circuits [15]. Somewhat analogously, we observe that in the first-order setting BCE simulates *pure predicate elimination* (PPE) and, under certain conditions, also *unused definition elimination* (UDE), a formula-level simplification described by Hoder et al. [13]. This section briefly recalls these two techniques and explains their relation to BCE. Apart from being of independent interest, the observations made in this section are also relevant for interpreting the experimental results presented in Section 6.3.

We say that a predicate symbol P is *pure* in a formula F if, in F, all occurrences of literals with predicate symbol P are of the same polarity. If a clause C contains a literal L with a pure predicate symbol P, then there are no L-resolvents of C, hence it is vacuously blocked. Therefore, blocked-clause elimination removes all clauses that contain pure predicates and thus simulates PPE.

UDE is a preprocessing method that removes so-called unused predicate definitions from general formulas (i.e., formulas that are not necessarily in CNF). Given a predicate symbol P and a general formula φ such that P does not occur in φ, a *predicate definition* is a formula

$$def(P, \varphi) = \forall \vec{x}.\ P(\vec{x}) \leftrightarrow \varphi(\vec{x}).$$

Assuming we have a predicate definition as a conjunct within a larger formula $\Psi = \psi \wedge def(P, \varphi)$, the definition is *unused* if P does not occur in ψ. (In fact, if P only occurs in ψ with a single polarity, then one of the two implications of the equivalence $def(P, \varphi)$, corresponding to that polarity, can be dropped by UDE.) UDE preserves satisfiability equivalence [13].

Note that UDE operates on the level of general formulas while BCE is only defined for formulas in CNF. Let therefore $def(P, \varphi)$ be an unused predicate definition in the formula $\Psi = \psi \wedge def(P, \varphi)$ as above and let $BCE(cnf(\Psi))$ be the result of eliminating all blocked clauses from a clause form translation $cnf(\Psi)$ of Ψ. We conjecture that for any "reasonably behaved" clausification procedure cnf (e.g., the well-known Tseitin encoding [33]), it holds that $BCE(cnf(\Psi)) \subseteq cnf(\psi)$ if φ does not contain quantifiers. In other words, BCE simulates UDE under the above conditions.

The main idea behind the simulation would be to show that each clause stemming from the clausification of an unused definition $def(P, \varphi)$ is blocked on the literal corresponding to predicate P. Although further intuitions are omitted here due to lack of space,[6] the reason why the presence of quantifiers in the definition formula φ poses a problem can be highlighted on a simple example:

Example 11. *The predicate definition $def(P, \exists x.Q(x)) = P \leftrightarrow \exists x.Q(x)$ can be clausified as $\neg P \vee Q(c),\ P \vee \neg Q(x)$, where c is a Skolem constant corresponding to the existential quantifier. By resolving these two clauses on P we obtain the resolvent $Q(c) \vee \neg Q(x)$ which is not valid.*

6.3 Experimental Evaluation

We present an empirical evaluation of our implementation of blocked-clause elimination, which is part of the preprocessing pipeline of the automated theorem prover VAMPIRE [19]. In our experiments, we used the 15 942 first-order benchmark formulas of the latest TPTP library [31] (version 6.4.0). Of these benchmarks, 7898 were already in CNF, while the remaining 8044 general formulas needed to be clausified by VAMPIRE before being subjected to BCE. This clausification step was optionally preceded

[6] See Appendix B for an additional discussion.

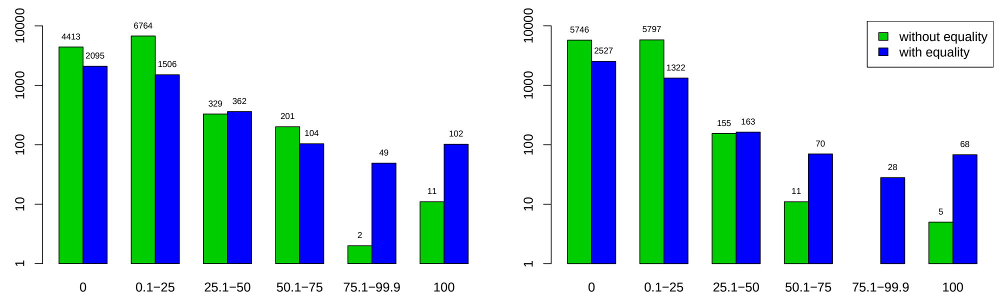

Figure 1: Histogram of problems by the percentage of blocked clauses eliminated. BCE applied to formulas after simple clausification (left) and after clausification preceded by PPE and UDE (right).

by VAMPIRE's implementation of PPE and UDE (see Section 6.2). $73\,\%$ of the benchmark formulas contain the equality predicate. In these formulas, we eliminated equality-blocked clauses while in the others we eliminated blocked clauses. All experiments were run on the StarExec compute cluster [30].

Occurrence of Blocked Clauses. Within a time limit of $300\,\mathrm{s}$ for parsing, classification (if needed), and subsequent blocked-clause detection and elimination our implementation was able to process all but one problem. Average/median time for detecting and eliminating blocked clauses was $0.238\,\mathrm{s}/0.001\,\mathrm{s}$.

In total, the benchmarks correspond to $299\,379\,591$ clauses. BCE removes $11.72\,\%$ of these clauses, while independently processing the problems with PPE and UDE before clausification leads to $7.66\,\%$ fewer clauses. Combining both methods yields a total reduction of $11.73\,\%$. Hence, the number of clauses which can be effectively removed by UDE but not by BCE or which can only be removed by BCE after some other clauses have been effectively removed by UDE is in the order of $0.01\,\%$.

Out of the $15\,941$ benchmarks, $59\,\%$ contain a blocked clause after simple clausification and $48\,\%$ of these benchmarks contain a blocked clause if first processed by PPE and UDE. Figure 1 shows the detailed distribution of eliminated blocked clauses. With PPE and UDE disabled, more than $25\,\%$ of the clauses could be eliminated in over 1000 problems. Moreover, 113 satisfiable formulas were directly solved by BCE, which means that BCE rendered the input empty. After applying PPE and UDE, which directly solve 46 problems, subsequent BCE can directly solve 73 other problems. There are two problems which can only be directly solved by the combination of PPE, UDE and BCE.

Impact on Proving Performance. To measure the effect of BCE on recent theorem provers, we considered the three best different[7] systems of the main FOF division of the 2016 CASC competition [32]: Vampire 4.0, E 2.0, and CVC4 1.5.1. Instead of running the provers in competition configurations, which are in all three cases based on a portfolio of strategies and thus lead to results that tend to be hard to interpret (c.f. [26]), we asked the respective developers to provide a single representative strategy good for proving theorems by their prover and then used these strategies in the experiment.[8]

We combined VAMPIRE as a clausifier with the three individual provers using the unix pipe construct. The clausification included PPE and UDE (enabled by default in VAMPIRE) and either did or did not include BCE. We set a time limit of $300\,\mathrm{s}$ for the whole combination, so the possible time overhead

[7] Actually, Vampire 4.1 was ranked second, but we did not include it, as it is just an updated version of Vampire 4.0.
[8] The strategies are listed in Appendix C.

Table 1: Effect of blocked-clause elimination (BCE) on theorem proving strategies. Bold: numbers of solved problems without BCE, positive (negative): problems gained (lost) by using BCE.

	unsatisfiable			satisfiable			total		
Vampire	**3172**	−28	+40	**458**	−0	+5	**3630**	−28	+45
E	**3097**	−20	+27	**363**	−1	+9	**3460**	−21	+36
CVC4	**2930**	−18	+37	**9**	−0	+68	**2939**	−18	+105

Table 2: Effect of blocked-clause elimination (BCE) on satisfiability checking strategies. Bold: numbers of solved problems without BCE, positive (negative): problems gained (lost) by using BCE.

	satisfiable			unsatisfiable			total		
Vampire	**531**	−0	+24	**719**	−4	+5	**1250**	−4	+29
iProver	**558**	−0	+1	**755**	−6	+4	**1313**	−6	+5
CVC4	**489**	−1	+28	**1724**	−24	+20	**2213**	−25	+48

incurred by BCE implied shorter time left for actual proving. We ran the systems on the 7619 problems established above on which BCE eliminates at least one clause.

Table 1 shows the numbers of solved problems without BCE and the difference when BCE is enabled. We can see that on satisfiable problems, BCE allows every prover to find more solutions; the most notable gain is observed with CVC4. BCE also enables each prover to solve new unsatisfiable problems, but there are problems that cannot be solved anymore (with the preselected strategy) when BCE is activated. Although the overall trend is that using BCE pays off, the existence of the lost problems is slightly puzzling. For a majority of them, the time taken to perform BCE is negligible and thus cannot explain the phenomenon. Moreover, proofs that would make use of a blocked clause, although they do sometimes occur, are quite rare.[9] Our current explanation thus appeals to the inherently "fragile" nature of the search spaces traversed by a theorem prover, in which the presence of a clause can steer the search towards a proof even if the clause does not itself directly take part in the proof in the end.

Strategies for Showing Satisfiability. Since the previous experiment indicates that BCE can be especially helpful on satisfiable problems, we decided to test how much it could improve strategies explicitly designed for establishing satisfiability, such as finite-model finding. This should be contrasted with the previous strategies, which focused on showing theoremhood. Here we selected three systems successful in the FNT (First-order form Non-Theorems) division of the 2016 CASC competition, namely Vampire 4.1, iProver 2.5, and CVC4 1.5.1 and again picked representative strategies for each, this time focusing on satisfiability detection.[10] The overall setup remained the same, with a time limit of 300 s.

Table 2 provides results of this experiment. We can see that Vampire and CVC4 detected significantly more satisfiable problems when BCE was used. On the other hand, iProver only solved one extra satisfiable problem with the help of BCE. The results on unsatisfiable problems, which are not specifically targeted by the selected strategies, were mixed, not showing a clear advantage of BCE.

[9] For 51 of the 3172 problems shown unsatisfiable by Vampire, the corresponding proof contained a blocked clause. However, none of these problems were among the 28 which Vampire did not solve after applying BCE.

[10] See Appendix D for the list of strategies selected for each system.

Mock Portfolio Construction. Understanding the value of a new technique within a theorem prover is very hard. The reason is that—in its most powerful configuration—a theorem prover usually employs a portfolio of strategies and each of these strategies may respond differently to the introduction of the new technique. In fact, a portfolio constructed without regard to the new technique is most likely suboptimal because the new technique may—due to interactions which are typically hard to predict—give rise to new successful strategies that could not be considered previously (c.f. [26], Section 4.4). In this final experiment, we tried to establish the value of BCE for the construction of a new strategy portfolio in VAMPIRE by emulating the typical first phase of the portfolio construction process, namely random sampling of the space of all strategies. Encouraged by the previous experiment, we focused on the construction of a portfolio specialized on detecting satisfiable problems.

We took a subset of 302 satisfiable problems from the TPTP library that were previously established hard for VAMPIRE, and that all contain at least one predicate which is different from equality. We randomly generated strategies by flipping values of various options that define how the prover attempts to establish satisfiability. Each strategy was cloned into two, one running with BCE as part of the preprocessing and the other without. Every such pair of strategies was then run on a randomly selected hard problem with a time limit of 120 s. In total, we ran 50 000 pairs.

Strategies using BCE succeeded 8414 times while strategies not using BCE succeeded 6766 times. There were 1796 cases where only the BCE variation succeeded on a problem compared to 148 cases where only the strategy without BCE succeeded. This demonstrates that BCE is a valuable addition to the set of VAMPIRE options and will likely be employed by a considerable fraction, if not all, of the strategies of the satisfiability checking CASC mode portfolio of the next version of the prover.

7 Conclusion

We lifted blocked clauses to first-order logic and showed that the presence of equality requires a refined notion of blocking for guaranteeing redundancy. We proved that checking blockedness is possible in polynomial time and, based on our theoretical results, implemented blocked-clause elimination for first-order logic to showcase a practical application of blocked clauses. In our evaluation, we observed that the elimination of blocked clauses is beneficial for modern provers in many cases, especially when dealing with satisfiable input formulas.

So far, we only investigated the impact of blocked-clause elimination as a stand-alone technique. From SAT and QBF research, however, it is known that blocked-clause elimination is even more powerful in combination with other preprocessing techniques [11] and so we expect this to be the case in first-order logic too. In particular, the combination of variable elimination and blocked-clause elimination has shown to be very effective in SAT solving [14]. It would therefore be interesting to analyze how the combination of first-order blocked-clause elimination with the predicate elimination technique of Khasidashvili and Korovin [18] affects the performance of theorem provers. Moreover, since blocked-clause elimination leads to even greater performance improvements when used not only before but also during SAT and QBF solving [22], a question arises how to integrate it more tightly into the theorem-proving process. Besides elimination, there are other applications for blocked clauses as well, like the addition of (small) blocked clauses or blocked-clause decomposition. We expect that such techniques, for which this paper lays the groundwork, can be helpful in the context of first-order theorem proving.

Acknowledgements

We thank Andrei Voronkov for performing the mock portfolio construction experiment.

References

[1] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Just-in-time compilation of knowledge bases. In *Proc. of the 23rd Int. Joint Conference on Artificial Intelligence* (*IJCAI 2013*), pages 447–453. AAAI Press, 2013.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Robinson and Voronkov [28], pages 19–99.

[4] Armin Biere. Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, volume B-2016-1 of *Dep. of Computer Science Series of Publications B*, pages 44–45. University of Helsinki, 2016.

[5] Jingchao Chen. Fast blocked clause decomposition with high quality. *CoRR*, abs/1507.00459, 2015.

[6] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. of the 8th Int. Conference on Theory and Applications of Satisfiability Testing* (*SAT 2005*), volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[7] Niklas Eén and Niklas Sörensson. The minisat page. `http://minisat.se`, 2016. Accessed: 2016-08-26.

[8] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 2 edition, 1996.

[9] Marijn Heule and Armin Biere. Blocked clause decomposition. In *Proc. of the 19th Int. Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (*LPAR-19*), volume 8312 of *LNCS*, pages 423–438. Springer, 2013.

[10] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for CNF formulas. In *Proc. of the 17th Int. Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (*LPAR-17*), volume 6397 of *LNCS*, pages 357–371. Springer, 2010.

[11] Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause elimination for SAT and QSAT. *Journal of Artificial Intelligence Research*, 53:127–168, 2015.

[12] Marijn Heule, Martina Seidl, and Armin Biere. Solution validation and extraction for QBF preprocessing. *Journal of Automated Reasoning*, pages 1–29, 2016.

[13] Krystof Hoder, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. Preprocessing techniques for first-order clausification. In *Proc. of the 12th Conference on Formal Methods in Computer-Aided Design* (*FMCAD 2012*), pages 44–51. IEEE, 2012.

[14] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *Proc. of the 16th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS 2010*), volume 6015 of *LNCS*, pages 129–144. Springer, 2010.

[15] Matti Järvisalo, Armin Biere, and Marijn Heule. Simulating circuit-level simplifications on CNF. *Journal of Automated Reasoning*, 49(4):583–619, 2012.

[16] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Proc. of the 6th Int. Joint Conference on Automated Reasoning* (*IJCAR 2012*), volume 7364 of *LNCS*, pages 355–370, Heidelberg, 2012. Springer.

[17] Paris C. Kanellakis and Peter Z. Revesz. On the relationship of congruence closure and unification. *Journal of Symbolic Computation*, 7(3/4):427–444, 1989.

[18] Zurab Khasidashvili and Konstantin Korovin. Predicate elimination for preprocessing in first-order theorem proving. In *Proc. of the 19th Int. Conference on Theory and Applications of Satisfiability Testing* (*SAT 2016*), volume 9710 of *LNCS*, pages 361–372. Springer, 2016.

[19] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *Proc. of the 25th Int. Conference on Computer Aided Verification* (*CAV 2013*), volume 8044 of *LNCS*, pages 1–35, 2013.

[20] Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97:149–176, 1999.

[21] Alexander Leitsch. *The Resolution Calculus*. Texts in Theoretical Computer Science. Springer, 1997.

[22] Florian Lonsing, Fahiem Bacchus, Armin Biere, Uwe Egly, and Martina Seidl. Enhancing search-based QBF

solving by dynamic blocked clause elimination. In *Proc. of the 20th Int. Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (*LPAR-20*), volume 9450 of *LNCS*, pages 418–433. Springer, 2015.

[23] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [28], pages 335–367.

[24] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.

[25] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[26] Giles Reger, Martin Suda, and Andrei Voronkov. The challenges of evaluating a new feature in Vampire. In *Proc. of the 1st and 2nd Vampire Workshops* (*Vampire@VSL 2014, Vampire@CADE 2015*), volume 38 of *EPiC Series in Computing*, pages 70–74. EasyChair, 2014.

[27] Giles Reger, Martin Suda, and Andrei Voronkov. New techniques in clausal form generation. In *Proc. of the 2nd Global Conference on Artificial Intelligence* (*GCAI 2016*), volume 41 of *EPiC Series in Computing*, pages 11–23. EasyChair, 2016.

[28] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[29] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[30] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In *Proc. of the 7th Int. Joint Conference on Automated Reasoning* (*IJCAR 2014*), volume 8562 of *LNCS*, pages 367–373. Springer, 2014.

[31] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[32] G. Sutcliffe and J. Urban. The CADE-25 Automated Theorem Proving System Competition - CASC-25. *AI Communications*, 29(3):423–433, 2016.

[33] G.S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer, 1983.

[34] Ralf Wimmer, Karina Gitina, Jennifer Nist, Christoph Scholl, and Bernd Becker. Preprocessing for DQBF. In *Proc. of the 18th Int. Conference on Theory and Applications of Satisfiability Testing* (*SAT 2015*), volume 9340 of *LNCS*, pages 173–190. Springer, 2015.

Appendix

A Polynomial Time Equality-Blocking Check

The purpose of this appendix is to argue that the ideas behind Algorithm 1 presented in Section 5 carry over to the case of equality-blocking, where we deal with flat L-resolvents and check validity in the presence of equality. Thus we will also obtain a polynomial time procedure for the equational case.

Because the top-level structure of the procedure along with the main arguments remain unchanged, we do not repeat them here and instead focus on highlighting the driving analogies between the non-equational and equational case. For this purpose let $C = L \vee C'$ be a candidate clause and $D = N_1 \vee \ldots \vee N_n \vee D'$ a partner clause. We assume, without the loss of generality, that $L = P(\vec{s})$ for a vector of terms $\vec{s}$ and, correspondingly, each $N_i = \neg P(\vec{t_i})$ for a vector of terms $\vec{t_i}$. A flat L-resolvent corresponding to a non-empty set of indexes $I \subseteq \{1, \ldots, n\}$ (i.e., a resolvent where the literals N_i with $i \in I$ are unified with $\bar{L}$) can be written as

$$R_I = C' \vee \bigvee_{i \in I} \vec{s} \not\approx \vec{t_i} \vee \bigvee_{i \notin I} N_i \vee D'$$

and it is valid if and only if the ground conjunction of units $\neg\forall R_I$ is unsatisfiable in the theory of uninterpreted functions.

If we compare how a transition from I to a larger set of indexes $J \supset I$ is reflected on the corresponding (flat) L-resolvents, we observe the following. In the non-equational case, R_J has fewer literals than R_I, because those corresponding to indexes $J \setminus I$ are missing, but is obtained using a unifier which is an instance of the one used for obtaining R_I. In the equational case, R_J has analogously fewer literals N_i, but has more literals of the form $\vec{s} \not\approx \vec{t_i}$. From the perspective of the "complemented" presentation, $\neg\forall R_J$ has fewer atomic literals $p(\vec{t_i})$ than $\neg\forall R_I$, but has a larger set of equations $\vec{s} \approx \vec{t_i}$.

We are now ready to describe the analog of Algorithm 1 for the equational case. First, the condition of the while loop (line 3) becomes "constant true", because in the equational case unification can never fail. Next, the condition "$K = \emptyset$" (line 5), which corresponds to "R_I is not valid", can be restated as $\neg\forall R_I$ is satisfiable and decided by a congruence closure algorithm. Finally, and this is the sole non-trivial part of the analogy, we need to realize that if $\neg\forall R_I$ is unsatisfiable it is either because already

$$\neg\forall \left(C' \vee \bigvee_{i \in I} \vec{s} \not\approx \vec{t_i} \vee D' \right)$$

is unsatisfiable and therefore $\neg\forall R_J$ will be unsatisfiable for any $J \supset I$ (this corresponds to the breaking the loop on line 10) or there is a single index $i \notin I$ such that

$$\neg\forall \left(C' \vee \bigvee_{i \in I} \vec{s} \not\approx \vec{t_i} \vee N_i \vee D' \right)$$

is unsatisfiable and therefore $\neg\forall R_J$ will be unsatisfiable for any $J \supset I$ for which $i \notin J$ (in this latter case, the loop continues as on line 8, with literal N_i added to the set N). This last observation, more specifically the fact that two distinct literals $N_i = P(\vec{t_i})$, $i \in I$ and $N_j = P(\vec{t_j})$, $j \in I$ cannot be both at the same time necessary for unsatisfiability of $\neg\forall R_I$ is left as an exercise for the reader.

B A Few More Ideas on the Simulation of UDE by BCE

We start by providing a formally more precise definition of UDE taken from [13]. Given a predicate symbol P, a formula φ such that P does not occur in φ and polarity $pol \in \{0, 1, -1\}$, a predicate definition is a formula

$$def(pol, P, \varphi) = \begin{cases} \forall\vec{x}.\, P(\vec{x}) \leftrightarrow \varphi(\vec{x}), & \text{if } pol = 0, \\ \forall\vec{x}.\, P(\vec{x}) \leftarrow \varphi(\vec{x}), & \text{if } pol = 1, \\ \forall\vec{x}.\, P(\vec{x}) \rightarrow \varphi(\vec{x}), & \text{if } pol = -1. \end{cases}$$

Assuming we have a predicate definition $def(pol, P, \varphi)$ as a conjunct within a larger formula

$$\Psi = \psi \wedge def(pol, P, \varphi), \tag{1}$$

UDE allows us (a) to drop the definition provided P does not occur in ψ or (b) to weaken (from an equivalence to an implication) a definition with $pol = 0$ to a one with $pol' \in \{1, -1\}$ provided P only occurs with polarity $-pol'$ in ψ. UDE preserves satisfiability of a formula [13].

We assume there is a clausification procedure cnf which takes as an input a first-order formula φ and transforms it into set of first-order clauses $cnf[\varphi]$. The transformation involves operations such as applying de Morgan and distributivity rules, expanding equivalences, performing skolemization of existential quantifiers and naming subformulas to prevent exponential blow-up [25, 23, 27]. Instead of trying to define in the most general terms what properties a "reasonably behaved" clausification procedure should satisfy and then showing that our claim holds for any such procedure, we present the main ingredients of our argument in a form of an informal proof script. A clausification procedure is "reasonably behaved" whenever this proof script can be used to show our result for it.

Let us now consider formula Ψ as in (1) and focus on the case (b) of UDE, where a definition with polarity $pol = 0$ can be weakened to one with $pol' = 1$, because P only occurs with polarity -1 in ψ. The other cases are similar or simpler. The claim that BCE simulates UDE on the CNF-level in this case means that there is a sequence of blocked-clause elimination steps turning $cnf[\psi \wedge def(0, P, \varphi)]$ to $cnf[\psi \wedge def(1, P, \varphi)]$. We would like to prove it along the following lines:

1. $$cnf[\psi \wedge def(0, P, \varphi)] = cnf[\psi] \cup cnf[def(1, P, \varphi)] \cup cnf[def(-1, P, \varphi)]$$
   using the fact that an equivalence is translated as a conjunction of two implications,[11]
2. every $C \in cnf[def(-1, P, \varphi)]$ is of the form $C = \neg P(\vec{x}) \vee C'$ for $C' \in cnf[\varphi(\vec{x})]$ using a property of the clausification procedure; moreover, C' does not contain any literal with predicate symbol P, because P does not occur in φ,
3. similarly, every $D \in cnf[def(1, P, \varphi)]$ is of the form $D = P(\vec{x}) \vee D'$ for $D' \in cnf[\neg\varphi(\vec{x})]$ and D' does not contain any literal with predicate symbol P,
4. $cnf[\psi]$ does not contain a clause with a positive occurrence of a literal with predicate symbol P by assumption,
5. for every $C' \in cnf[\varphi(\vec{x})]$ and every $D' \in cnf[\neg\varphi(\vec{x})]$ the clause $C' \vee D'$ is valid.

Finally, the argument would be closed by observing that every $\neg P(\vec{x}) \vee C' \in cnf[def(-1, P, \varphi)]$ is blocked on the literal $\neg P(\vec{x})$, because its only resolution partners are the clauses $P(\vec{x}) \vee D' \in cnf[def(1, P, \varphi)]$ and each of them leads to a resolvent which is valid by item 5.

While items 1 and 4 are easy to justify for any reasonable implementation of cnf, formula naming can interfere with the argument behind items 2 and 3, and, on top of that, item 5 does not work if a

[11] And the fact that universal quantifier are simply dropped when transforming a formula to CNF.

skolemisation step needs to be performed when clausifying $\varphi(\vec{x})$ or $\neg\varphi(\vec{x})$. Let us now look more closely at these two caveats.

A classification procedure may decide to *name* a subformula to prevent, in the worst case, exponential blow-up stemming from the distributivity of conjunctions over disjunctions (c.f. [33]). This is actually achieved by introducing a new predicate symbol – the name – and adding a predicate definition (!) for the name and the subformula. If a subformula χ of φ is named by cnf, items 2 and 3 no longer hold as stated, because clauses from the definition of χ will not be of the form $(\neg)P(\vec{x}) \vee C'$, but rather of the form $(\neg)R(\vec{y}) \vee C'$, where R is the name introduced for χ. This is ultimately not a problem, since clauses of $\mathit{cnf}[\mathit{def}(\mathit{pol}, R, \chi)]$ will (recursively) become blocked by the same argument, once R does not occur anywhere else in clausified formula. However, item 5 is endangered unless the clausification procedure introduces the same name for χ a subformula of φ and $\neg\varphi$. The following example illustrates the issue:

Example 12. *Let us consider the predicate definition $\mathit{def}(0, p, \varphi)$ for $\varphi = (a \wedge b) \vee c$. One possible clausification of the definition consists of the clauses:*

$$\mathit{cnf}_1[\mathit{def}(-1, p, \varphi)] = \{\neg p \vee a \vee c, \neg p \vee b \vee c\} \text{ and } \mathit{cnf}_1[\mathit{def}(1, p, \varphi)] = \{p \vee \neg a \vee \neg b, p \vee \neg c\}.$$

It is easy to check that, in particular, the clauses $\mathit{cnf}_1[\mathit{def}(-1, p, \varphi)]$ are blocked on the literal $\neg p$.

Naming the subformula $(a \wedge b)$ in the definition might lead to the following clausification:

$$\mathit{cnf}_2[\mathit{def}(0, p, r \vee c) \wedge \mathit{def}(0, r, a \wedge b)] = \{\neg p \vee r \vee c, p \vee \neg r, p \vee c, \neg r \vee a, \neg r \vee b, r \vee \neg a \vee \neg b\},$$

in which the clauses from $\mathit{def}(-1, p, r \vee c)$, namely the clause $\neg p \vee r \vee c$, are blocked on $\neg p$ and after their elimination the clauses from $\mathit{cnf}_2[\mathit{def}(-1, r, a \wedge b)]$, namely $\neg r \vee a$ and $\neg r \vee b$, become blocked on $\neg r$.

However, a clausification procedure which would, for instance, introduce a name for $(a \wedge b)$ only for the sake of $\mathit{def}(-1, p, \varphi)$ and not for $\mathit{def}(1, p, \varphi)$, or vice versa, or which would introduce two distinct names and corresponding definitions, one for the positive and one for the negative occurrence of the subformula, would still be correct, but the result could not simplified as claimed above by BCE. The clausification could then look, for instance, as follows: $\mathit{cnf}_3[\mathit{def}(0, p, \varphi)] = \mathit{cnf}_3[\mathit{def}(-1, p, r \vee c) \wedge \mathit{def}(-1, r, a \wedge b)] \cup \mathit{cnf}_3[\mathit{def}(1, p, (a \wedge b) \vee c)] = \{\neg p \vee r \vee c, \neg r \vee a, \neg r \vee b, p \vee \neg a \vee \neg b, p \vee \neg c\}$. Here, the clause $\neg p \vee r \vee c$ does not resolve to a valid clause with $p \vee \neg a \vee \neg b$.

While it is straightforward to show that item 5 holds for any reasonable clausification procedure cnf whenever only propositional rules are applied and formula names, if introduced, are shared between the two polarities of φ as discussed above, this item no longer holds when formula φ contains a quantifier and a skolemization step becomes necessary (as already also shown in the main text):

Example 13. *Consider a predicate definition $\mathit{def}(0, P, \varphi)$ for the formula: $\varphi(x) = \exists y.Q(x, y)$. The definition gets clausified as:*

$$\mathit{cnf}[\mathit{def}(0, P, \varphi)] = \{\neg P(x) \vee Q(x, f(x)), P(x) \vee \neg Q(x, y)\},$$

where f is a Skolem function introduced for the sake of the existential quantifier in φ. The resolvent $Q(x, f(x)) \vee \neg Q(x, y)$ of the two defining clauses on $(\neg)P(x)$ is not valid and so the first clause is not blocked on $\neg P(x)$. (Resolving on the second literal leads to a valid clause here, but recall we do not exclude the possibility of predicate Q occurring also elsewhere in the formula.)

To sum up, under certain reasonable conditions, which we did not specify formally, imposed on the clausification procedure, BCE on the CNF-level simulates UDE from the formula level provided the definition in question does not contain a quantifier. It should be clear, on the other hand, that BCE can eliminate more than just certain predicate definitions, simply because the input formula can already be in CNF to which UDE obviously cannot, in general, apply.

C Theorem Proving Strategies Used in the Experiment

C.1 Vampire 4.0

```
./vampire -t 300 -sa discount -awr 10
```

C.2 E 2.0

```
./eprover -s --simul-paramod --forward-context-sr \
--destructive-er-aggressive --destructive-er -tKBO6 \
-winvfreqrank -c1 -Ginvfreq -F1 \
-WSelectMaxLComplexAvoidPosPred \
-H'(1.ConjectureGeneralSymbolWeight\
(SimulateSOS,488,104,105,32,173,0,327,3.6,1.4,1),\
1.FIFOWeight(PreferProcessed),\
8.Clauseweight(PreferUnitGroundGoals,1,1,0.5),\
3.Refinedweight(PreferGoals,2,4,7,5,6.6),\
2.ConjectureRelativeSymbolWeight\
(ConstPrio,0.06,67,160,111,25,3.1,2.8,1))'
```

C.3 CVC4 1.5.1

```
./cvc4 --full-saturate-quant
```

D Strategies for Testing Satisfiability Used in the Experiment

D.1 Vampire 4.1

```
./vampire -t 300 -sa fmb
```

D.2 iProver 2.5

```
./iproveropt --sat_mode true --schedule none \
--sat_finite_models true
```

D.3 CVC4 1.5.1

```
./cvc4 --finite-model-find --fmf-inst-engine --sort-inference \
--uf-ss-fair
```