# An Efficient Subsumption Test Pipeline for BS(LRA) Clauses

Martin Bromberger, Lorenz Leutgeb, Christoph Weidenbach


## To cite this version:

Martin Bromberger, Lorenz Leutgeb, Christoph Weidenbach. An Efficient Subsumption Test Pipeline for BS(LRA) Clauses. IJCAR 2022 - International Joint Conference in Automated Reasoning, Aug 2022, Haifa, Israel. pp.147-168, 10.1007/978-3-031-10769-6_10. hal-03881893

## HAL Id: hal-03881893

## https://inria.hal.science/hal-03881893v1

Submitted on 2 Dec 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# An Efficient Subsumption Test Pipeline for BS(LRA) Clauses

Martin Bromberger[1] [0000-0001-7256-2190],
Lorenz Leutgeb[1,2] [0000-0003-0391-3430], and
Christoph Weidenbach[1] [0000-0001-6002-0458]

[1] Max Planck Institute for Informatics,
Saarland Informatics Campus, Saarbrücken, Germany
{mbromber,lorenz,weidenb}@mpi-inf.mpg.de
[2] Graduate School of Computer Science,
Saarland Informatics Campus, Saarbrücken, Germany

**Abstract** The importance of subsumption testing for redundancy elimination in first-order logic automatic reasoning is well-known. Although the problem is already NP-complete for first-order clauses, the meanwhile developed test pipelines efficiently decide subsumption in almost all practical cases. We consider subsumption between first-oder clauses of the Bernays-Schönfinkel fragment over linear real arithmetic constraints: BS(LRA). The bottleneck in this setup is deciding implication between the LRA constraints of two clauses. Our new *sample point heuristic* preempts expensive implication decisions in about 94% of all cases in benchmarks. Combined with filtering techniques for the first-order BS part of clauses, it results again in an efficient subsumption test pipeline for BS(LRA) clauses.

**Keywords:** Bernays-Schönfinkel Fragment · Linear Arithmetic · Redundancy Elimination · Subsumption

## 1 Introduction

The elimination of redundant clauses is crucial for the efficient automatic reasoning in first-order logic. In a resolution [50,5] or superposition setting [4,44], a newly inferred clause might be subsumed by a clause that is already known (*forward subsumption*) or it might subsume a known clause (*backward subsumption*). Although the SCL calculi family [1,21,11] does not require forward subsumption tests, a property also inherent to the propositional CDCL (Conflict Driven Clause Learning) approach [55,34,41,8,63], backward subsumption and hence subsumption remains an important test in order to remove redundant clauses.

In this work we present advances in deciding subsumption for constrained clauses, specifically employing the Bernays-Schönfinkel fragment as foreground logic, and linear real arithmetic as background theory, BS(LRA). BS(LRA) is of

particular interest because it can be used to model *supervisors*, i.e., components in technical systems that control system functionality. An example for a supervisor is the electronic control unit of a combustion engine. The logics we use to model supervisors and their properties are called *SupERLogs*—(Sup)ervisor (E)ffective(R)easoning (Log)ics. SupERLogs are instances of function-free first-order logic extended with arithmetic [18], which means BS(LRA) is an example of a SupERLog.

Subsumption is an important redundancy criterion in the context of hierarchic clausal reasoning [6,35,20,37,11]. At the heart of this paper is a new technique to speed up the treatment of linear arithmetic constraints as part of deciding subsumption. For every clause, we store a solution of its associated constraints, which is used to quickly falsify implication decisions, acting as a filter, called the *sample point heuristic*. In our experiments with various benchmarks, the technique is very effective: It successfully preempts expensive implication decisions in about 94% of cases. We elaborate on these findings in Section 4.

For example, consider three BS clauses, none of which subsumes another:

$$C_1 \coloneqq P(a, x) \qquad C_2 \coloneqq \neg P(y, z) \lor Q(y, z, b) \qquad C_3 \coloneqq \neg R(b) \lor Q(a, x, b)$$

Let $C_4$ be the resolvent of $C_1$ and $C_2$ upon the atom $P(a, x)$, i.e., $C_4 \coloneqq Q(a, z, b)$. Now $C_4$ backward-subsumes $C_3$ with matcher $\sigma \coloneqq \{z \mapsto x\}$, i.e. $C_4\sigma \subset C_3$, thus $C_3$ is redundant and can be eliminated. Now, consider an extension of the above clauses with some simple LRA constraints following the same reasoning:

$$\begin{aligned} C_1' &\coloneqq x \geq 1 \parallel P(a, x) \\ C_2' &\coloneqq z \geq 0 \parallel \neg P(y, z) \lor Q(y, z, b) \\ C_3' &\coloneqq x \geq 0 \parallel \neg R(b) \lor Q(a, x, b) \end{aligned}$$

where $\parallel$ is interpreted as an implication, i.e., clause $C_1'$ stands for $\neg x \geq 1 \lor P(a, x)$ or simply $x < 1 \lor P(a, x)$. The respective resolvent on the constrained clauses is $C_4' \coloneqq z \geq 0, z \geq 1 \parallel Q(a, z, b)$ or after constraint simplification $C_4' \coloneqq z \geq 1 \parallel Q(a, z, b)$ because $z \geq 1$ implies $z \geq 0$. For the constrained clauses, $C_4'$ does no longer subsume $C_3'$ with matcher $\sigma \coloneqq \{z \mapsto x\}$, because $z \geq 0$ does not LRA-imply $z \geq 1$. Now, if we store the sample point $x = 0$ as a solution for the constraint of clause $C_3'$, this sample point already reveals that $z \geq 0$ does not LRA-imply $z \geq 1$. This constitutes the basic idea behind our sample point heuristic. In general, constraints are not just simple bounds as in the above example, and sample points are solutions to the system of linear inequalities of the LRA constraint of a clause.

Please note that our test on LRA constraints is based on LRA theory implication and not on a syntactic notion such as subsumption on the first-order part of the clause. In this sense it is "stronger" than its first-order counterpart. This fact is stressed by the following example, taken from [26, Ex. 2], which shows that first-order implication does not imply subsumption. Let

$$\begin{aligned} C_1 &\coloneqq \neg P(x, y) \lor \neg P(y, z) \lor P(x, z) \\ C_2 &\coloneqq \neg P(a, b) \lor \neg P(b, c) \lor \neg P(c, d) \lor P(a, d) \end{aligned}$$

Then we have $C_1 \rightarrow C_2$, but again, for all $\sigma$ we have $C_1\sigma \not\subseteq C_2$: Constructing $\sigma$ from left to right we obtain $\sigma \coloneqq \{x \mapsto a, y \mapsto b, z \mapsto c\}$, but $P(a,c) \notin C_2$. Constructing $\sigma$ from right to left we obtain $\sigma \coloneqq \{z \mapsto d, x \mapsto a, y \mapsto c\}$, but $\neg P(a,c) \notin C_2$.

*Related Work* Treatment of questions regarding the complexity of deciding subsumption of first-order clauses [27] dates back more than thirty years. Notions of subsumption, varying in generality, are studied in different sub-fields of theorem proving, whereas we restrict our attention to first-order theorem proving. Modern implementations typically decide multiple thousand instances of this problem per second: In [62, Sec. 2], Voronkov states that initial versions of Vampire "seemed to [...] deadlock" without efficient implementations to decide (forward) subsumption.

In order to reduce the number of clauses out of a set of clauses to be considered for pairwise subsumption checking, the best known practice in first-order theorem proving is to use (imperfect) indexing data structures as a means for pre-filtering and research concerning appropriate techniques is plentiful, see [46,27,56,45,39,40,33,28,61,29,30,59,48,24,47,49,53,52,54,25,43] for an evaluation of these techniques. Here we concentrate on the efficiency of a subsumption check between two clauses and therefore do not take indexing techniques into account. Furthermore, the implication test between two linear arithmetic constraints is of a semantic nature and is not related to any syntactic features of the involved constraints and can therefore hardly be filtered by a syntactic indexing approach.

In addition to pre-filtering via indexing, almost all above mentioned implementations of first-order subsumption tests rely on additional filters on the clause level. The idea is to generate an abstraction of clauses together with an ordering relation such that the ordering relation is necessary to hold between two clauses in order for one clause to subsume the other. Furthermore, the abstraction as well as the ordering relation should be efficiently computable. For example, a necessary condition for a first-order clause $C_1$ to subsume a first-order clause $C_2$ is $|\operatorname{vars}(C_1)| \geq |\operatorname{vars}(C_2)|$, i.e., the number of different variables in $C_1$ must be larger or equal than the number of variables in $C_2$. Further and additional abstractions included by various implementations rely on the size of clauses, number of ground literals, depth of literals and terms, occurring predicate and function symbols. For the BS(LRA) clauses considered here, the structure of the first-order BS part, which consists of predicates and flat terms (variables and constants) only, is not particularly rich.

The exploration of sample points has already been studied in the context of first-order clauses with arithmetic constraints. In [17,36] it was used to improve the performance of iSAT [23] on testing non-linear arithmetic constraints. In general, iSAT tests satisfiability by interval propagation for variables. If intervals get "too small" it typically gives up, however sometimes the explicit generation of a sample point for a small interval can still lead to a certificate for satisfiability. This technique was successfully applied in [17], but was not used for deciding subsumption of constrained clauses.

*Motivation* The main motivation for this work is the realization that computing implication decisions required to treat constraints of the background theory presents the bottleneck of an BS(LRA) subsumption check in practice. Inspired by the success of filtering techniques in first-order logic, we devise an exceptionally effective filter for constraints and adopt well-known first-order filters to the BS fragment. Our sample point heuristic for LRA could easily be generalized to other arithmetic theories as well as full first-order logic.

*Structure* The paper is structured as follows. After a section defining BS(LRA) and common notions and notation, Section 2, we define redundancy notions and our sample point heuristic in Section 3. Section 4 justifies the success of the sample point heuristic by numerous experiments in various application domains of BS(LRA). The paper ends with a discussion of the obtained results, Section 5. Binaries, utility scripts, benchmarking instances used as input, and the output used for evaluation may be obtained online [13].

## 2 Preliminaries

We briefly recall the basic logical formalisms and notations we build upon [10]. Our starting point is a standard many-sorted first-order language for BS with *constants* (denoted $a, b, c$), without non-constant function symbols, with *variables* (denoted $w, x, y, z$), and *predicates* (denoted $P, Q, R$) of some fixed *arity*. *Terms* (denoted $t, s$) are variables or constants. An *atom* (denoted $A, B$) is an expression $P(t_1, \dots, t_n)$ for a predicate $P$ of arity $n$. A *positive literal* is an atom $A$ and a *negative literal* is a negated atom $\neg A$. We define $\text{comp}(A) = \neg A$, $\text{comp}(\neg A) = A$, $|A| = A$ and $|\neg A| = A$. Literals are usually denoted $L, K, H$. Formulas are defined in the usual way using quantifiers $\forall$, $\exists$ and the boolean connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$, and $\equiv$.

A *clause* (denoted $C, D$) is a universally closed disjunction of literals $A_1 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee \neg B_m$. Clauses are identified with their respective multisets and all standard multiset operations are extended to clauses. For instance, $C \subseteq D$ means that all literals in $C$ also appear in $D$ respecting their number of occurrences. A clause is *Horn* if it contains at most one positive literal, i.e. $n \leqslant 1$, and a *unit clause* if it has exactly one literal, i.e. $n + m = 1$. We write $C^+$ for the set of positive literals, or *conclusions* of $C$, i.e. $C^+ \coloneqq \{A_1, \dots, A_n\}$ and respectively $C^-$ for the set of negative literals, or *premises* of $C$, i.e. $C^- \coloneqq \{\neg B_1, \dots, \neg B_m\}$. If $Y$ is a term, formula, or a set thereof, $\text{vars}(Y)$ denotes the set of all variables in $Y$, and $Y$ is *ground* if $\text{vars}(Y) = \emptyset$.

The *Bernays-Schönfinkel Clause Fragment* (BS) in first-order logic consists of first-order clauses where all involved terms are either variables or constants. The *Horn Bernays-Schönfinkel Clause Fragment* (HBS) consists of all sets of BS Horn clauses.

A *substitution* $\sigma$ is a function from variables to terms with a finite domain $\text{dom}(\sigma) = \{x \mid x\sigma \neq x\}$ and codomain $\text{codom}(\sigma) = \{x\sigma \mid x \in \text{dom}(\sigma)\}$. We denote substitutions by $\sigma, \delta, \rho$. The application of substitutions is often written

postfix, as in $x\sigma$, and is homomorphically extended to terms, atoms, literals, clauses, and quantifier-free formulas. A substitution $\sigma$ is *ground* if $\text{codom}(\sigma)$ is ground. Let $Y$ denote some term, literal, clause, or clause set. A substitution $\sigma$ is a *grounding* for $Y$ if $Y\sigma$ is ground, and $Y\sigma$ is a *ground instance* of $Y$ in this case. We denote by $\text{gnd}(Y)$ the set of all ground instances of $Y$, and by $\text{gnd}_B(Y)$ the set of all ground instances over a given set of constants $B$. The *most general unifier* $\text{mgu}(Z_1, Z_2)$ of two terms/atoms/literals $Z_1$ and $Z_2$ is defined as usual, and we assume that it does not introduce fresh variables and is idempotent.

We assume a standard many-sorted first-order logic model theory, and write $\mathcal{A} \vDash \phi$ if an interpretation $\mathcal{A}$ satisfies a first-order formula $\phi$. A formula $\psi$ is a logical consequence of $\phi$, written $\phi \vDash \psi$, if $\mathcal{A} \vDash \psi$ for all $\mathcal{A}$ such that $\mathcal{A} \vDash \phi$. Sets of clauses are semantically treated as conjunctions of clauses with all variables quantified universally.

### 2.1 Bernays-Schönfinkel with Linear Real Arithmetic

The extension of BS with linear real arithmetic, BS(LRA), is the basis for the formalisms studied in this paper. We consider a standard *many-sorted* first-order logic with one first-order sort $\mathcal{F}$ and with the sort $\mathcal{R}$ for the real numbers. Given a clause set $N$, the interpretations $\mathcal{A}$ of our sorts are fixed: $\mathcal{R}^{\mathcal{A}} = \mathbb{R}$ and $\mathcal{F}^{\mathcal{A}} = \mathbb{F}$. This means that $\mathcal{F}^{\mathcal{A}}$ is a Herbrand interpretation, i.e., $\mathbb{F}$ is the set of first-order constants in $N$, or a single constant out of the signature if no such constant occurs. Note that this is not a deviation from standard semantics in our context as for the arithmetic part the canonical domain is considered and the first-order sort has the finite model property over the occurring constants (note that equality is not part of BS).

Constant symbols, arithmetic function symbols, variables, and predicates are uniquely declared together with their respective sort. The unique sort of a constant symbol, variable, predicate, or term is denoted by the function $\text{sort}(Y)$ and we assume all terms, atoms, and formulas to be well-sorted. We assume *pure* input clause sets, which means the only constants of sort $\mathcal{R}$ are (rational) numbers. This means the only constants that we do allow are rational numbers $c \in \mathbb{Q}$ and the constants defining our finite first-order sort $\mathcal{F}$. Irrational numbers are not allowed by the standard definition of the theory. The current implementation comes with the caveat that only integer constants can be parsed. Satisfiability of pure BS(LRA) clause sets is semi-decidable, e.g., using *hierarchic superposition* [6] or *SCL(T)* [11]. Impure BS(LRA) is no longer compact and satisfiability becomes undecidable, but its restriction to ground clause sets is decidable [22].

All arithmetic predicates and functions are interpreted in the usual way. An interpretation of BS(LRA) coincides with $\mathcal{A}^{\text{LRA}}$ on arithmetic predicates and functions, and freely interprets free predicates. For pure clause sets this is well-defined [6]. Logical satisfaction and entailment is defined as usual, and uses similar notation as for BS.

*Example 1.* The clause $y < 5 \lor x' \neq x + 1 \lor \neg S_0(x, y) \lor S_1(x', 0)$ is part of a timed automaton with two clocks $x$ and $y$ modeled in BS(LRA). It represents

a transition from state $S_0$ to state $S_1$ that can be traversed only if clock $y$ is at least 5 and that resets $y$ to 0 and increases $x$ by 1.

Arithmetic terms are constructed from a set $\mathcal{X}$ of *variables*, the set of integer constants $c \in \mathbb{Z}$, and binary function symbols $+$ and $-$ (written infix). Additionally, we allow multiplication $\cdot$ if one of the factors is an integer constant. Multiplication only serves us as syntactic sugar to abbreviate other arithmetic terms, e.g., $x + x + x$ is abbreviated to $3 \cdot x$. Atoms in BS(LRA) are either *first-order atoms* (e.g., $P(13, x)$) or *(linear) arithmetic atoms* (e.g., $x < 42$). Arithmetic atoms are denoted by $\lambda$ and may use the predicates $\leq, <, \neq, =, >, \geq$, which are written infix and have the expected fixed interpretation. We use $\triangleleft$ as a placeholder for any of these predicates. Predicates used in first-order atoms are called *free*. *First-order literals* and related notation is defined as before. *Arithmetic literals* coincide with arithmetic atoms, since the arithmetic predicates are closed under negation, e.g., $\neg(x \geq 42) \equiv x < 42$.

BS(LRA) clauses are defined as for BS but using BS(LRA) atoms. We often write clauses in the form $\Lambda \parallel C$ where $C$ is a clause solely built of free first-order literals and $\Lambda$ is a multiset of LRA atoms called the *constraint* of the clause. A clause of the form $\Lambda \parallel C$ is therefore also called a *constrained clause*. The semantics of $\Lambda \parallel C$ is as follows:

$$\Lambda \parallel C \quad \text{iff} \quad \big(\bigwedge_{\lambda \in \Lambda} \lambda\big) \rightarrow C \quad \text{iff} \quad \big(\bigvee_{\lambda \in \Lambda} \neg\lambda\big) \vee C$$

For example, the clause $x > 1 \vee y \neq 5 \vee \neg Q(x) \vee R(x, y)$ is also written $x \leq 1, y = 5 || \neg Q(x) \vee R(x, y)$. The negation $\neg(\Lambda \parallel C)$ of a constrained clause $\Lambda \parallel C$ where $C = A_1 \vee \cdots \vee A_n \vee \neg B_1 \vee \cdots \vee \neg B_m$ is thus equivalent to $(\bigwedge_{\lambda \in \Lambda} \lambda) \wedge \neg A_1 \wedge \cdots \wedge \neg A_n \wedge B_1 \wedge \cdots \wedge B_m$. Note that since the neutral element of conjunction is $\top$, an empty constraint is thus valid, i.e. equivalent to true.

An *assignment* for a constraint $\Lambda$ is a substitution (denoted $\beta$) that maps all variables in $\text{vars}(\Lambda)$ to real numbers $c \in \mathbb{R}$. An assignment is a *solution* for a constraint $\Lambda$ if all atoms $\lambda \in (\Lambda\beta)$ evaluate to true. A constraint $\Lambda$ is *satisfiable* if there exists a solution for $\Lambda$. Otherwise it is *unsatisfiable*. Note that assignments can be extended to $C$ by also mapping variables of the first-order sort accordingly.

A clause or clause set is *abstracted* if its first-order literals contain only variables or first-order constants. Every clause $C$ is equivalent to an abstracted clause that is obtained by replacing each non-variable arithmetic term $t$ that occurs in a first-order atom by a fresh variable $x$ while adding an arithmetic atom $x \neq t$ to $C$. We assume abstracted clauses for theory development, but we prefer non-abstracted clauses in examples for readability, e.g., a unit clause $P(3, 5)$ is considered in the development of the theory as the clause $x = 3, y = 5 \parallel P(x, y)$. In the implementation, we mostly prefer abstracted clauses except that we allow integer constants $c \in \mathbb{Z}$ to appear as arguments of first-order literals. In some cases, this makes it easier to recognize whether two clauses can be matched or not. For instance, we see by syntactic comparison that the two unit clauses $P(3, 5)$ and $P(0, 1)$ have no substitution $\sigma$ such that $P(3, 5) = P(0, 1)\sigma$. For the abstracted

versions on the other hand, $x = 3, y = 5 \parallel P(x, y)$ and $u = 0, v = 1 \parallel P(u, v)$ we can find a matching substitution for the first-order part $\sigma := \{u \mapsto x, v \mapsto y\}$ and would have to check the constraints semantically to exclude the matching.

*Hierarchic Resolution* One inference rule, foundational to most algorithms for solving constrained first-order clauses, is *hierarchic resolution* [6]:

$$\frac{\Lambda_1 \parallel L_1 \vee C_1 \quad \Lambda_2 \parallel L_2 \vee C_2 \quad \sigma = \text{mgu}(L_1, \text{comp}(L_2))}{(\Lambda_1, \Lambda_2 \parallel C_1 \vee C_2)\sigma}$$

The conclusion is called *hierarchic resolvent* (of the two clauses in the premise). A *refutation* is the sequence of resolution steps that produces a clause $\Lambda \parallel \bot$ with $\mathcal{A}^{\text{LRA}} \models \Lambda\delta$ for some grounding $\delta$. Hierarchic resolution is sound and refutationally complete for the BS(LRA) clauses considered here, since every set $N$ of BS(LRA) clauses is *sufficiently complete* [6], because all constatnts of the arithemtic sort are numbers. Hence *hierarchic resolution* is sound and refutationally complete for $N$ [6,7]. *Hierarchic unit resolution* is a special case of hierarchic resolution, that only combines two clauses in case one of them is a unit clause. Hierarchic unit resolution is sound and complete for HBS(LRA) [6,7], but not even refutationally complete for BS(LRA).

Most algorithms for Bernays-Schnönfinkel, first-order logic, and beyond utilize resolution. The SCL(T) calculus for HBS(LRA) uses hierarchic resolution in order to learn from the conflicts it encounters during its search. The hierarchic superposition calculus on the other hand derives new clauses via hierarchic resolution based on an ordering. The goal is to either derive the empty clause or a saturation of the clause set, i.e., a state from which no new clauses can be derived. Each of those algorithms must derive new clauses in order to progress, but their subroutines also get progressively slower as more clauses are derived. In order to increase efficiency, it is necessary to eliminate clauses that are obsolete. One measure that determines whether a clause is useful or not is *redundancy*.

*Redundancy* In order to define redundancy for constrained clauses, we need an $\mathcal{H}$*-order*, i.e., a well-founded, total, strict ordering $\prec$ on ground literals such that literals in the constraints (in our case arithmetic literals) are always smaller than first-order literals. Such an ordering can be lifted to constrained clauses and sets thereof by its respective multiset extension. Hence, we overload any such order $\prec$ for literals, constrained clauses, and sets of constrained clause if the meaning is clear from the context. We define $\preceq$ as the reflexive closure of $\prec$ and $N^{\preceq \Lambda \parallel C} := \{D \mid D \in N \text{ and } D \preceq \Lambda \parallel C\}$. An instance of an LPO [15] with appropriate precedence can serve as an $\mathcal{H}$-order.

**Definition 2 (Clause Redundancy).** *A ground clause* $\Lambda \parallel C$ *is* redundant *with respect to a set* $N$ *of ground clauses and an* $\mathcal{H}$*-order* $\prec$ *if* $N^{\preceq \Lambda \parallel C} \models \Lambda \parallel C$*. A clause* $\Lambda \parallel C$ *is* redundant *with respect to a clause set* $N$ *and an* $\mathcal{H}$*-order* $\prec$ *if for all* $\Lambda' \parallel C' \in \text{gnd}(\Lambda \parallel C)$ *the clause* $\Lambda' \parallel C'$ *is redundant with respect to* $\text{gnd}(N)$*.*

If a clause $\Lambda \parallel C$ is redundant with respect to a clause set $N$, then it can be removed from $N$ without changing its semantics. Determining clause redundancy is an undecidable problem [11,63]. However, there are special cases of redundant clauses that can be easily checked, e.g., tautologies and subsumed clauses. Techniques for tautology deletion and subsumption deletion are the most common elimination techniques in modern first-order provers.

A *tautology* is a clause that evaluates to true independent of the predicate interpretation or assignment. It is therefore redundant with respect to all orders and clause sets; even the empty set.

**Corollary 3 (Tautology for Constrained Clauses).** *A clause $\Lambda \parallel C$ is a tautology if the existential closure of $\neg(\Lambda \parallel C)$ is unsatisfiable.*

Since $\neg(\Lambda \parallel C)$ is essentially ground (by existential closure and skolemization), it can be solved with an appropriate SMT solver, i.e., an SMT solver that supports unquantified uninterpreted functions coupled with linear real arithmetic. In [2], it is recommended to check only the following conditions for tautology deletion in hierarchic superposition:

**Corollary 4 (Tautology Check).** *A clause $\Lambda \parallel C$ is a tautology if the existential closure of $\Lambda$ is unsatisfiable or if $C$ contains two literals $L_1$ and $L_2$ with $L_1 = \mathrm{comp}(L_2)$.*

The advantage is that the check on the first-order side of the clause is still purely syntactic and corresponds to the tautology check for pure first-order logic. Nonetheless, there are tautologies that are not captured by Corollary 4, e.g., $x = y \parallel P(x) \vee \neg P(y)$. The SCL(T) calculus on the other hand requires no tautology checks because it never learns tautologies as part of its conflict analysis [1,21,11]. This property is also inherent to the propositional CDCL (Conflict Driven Clause Learning) approach [55,34,41,8,63].

## 3 Subsumption for Constrained Clauses

A *subsumed* constrained clause is a clause that is redundant with respect to a single clause in our clause set. Formally, subsumption is defined as follows.

**Definition 5 (Subsumption for Constrained Clauses [2]).** *A constrained clause $\Lambda_1 \parallel C_1$* subsumes *another constrained clause $\Lambda_2 \parallel C_2$ if there exists a substitution $\sigma$ such that $C_1\sigma \subseteq C_2$, $\mathrm{vars}(\Lambda_1\sigma) \subseteq \mathrm{vars}(\Lambda_2)$, and the universal closure of $\Lambda_2 \rightarrow (\Lambda_1\sigma)$ holds in LRA.*

Eliminating redundant clauses is crucial for the efficient operation of an automatic first-order theorem prover. Although subsumption is considered one of the easier redundancy relationships that we can check in practice, it is still a hard problem in general:

**Lemma 6 (Complexity of Subsumption in the BS Fragment).** *Deciding subsumption for a pair of BS clauses is NP-complete.*

*Proof.* Containment in NP follows from the fact that the size of subsumption matchers is limited by the subsumed clause and set inclusion of literals can be decided in polynomial time. For the hardness part, consider the following polynomial-time reduction from 3-SAT. Take a propositional clause set where all clauses have length three. Now introduce a 6-place predicate $R$ and encode each propositional variable $P$ by a first-order variable $x_P$. Then a propositional clause $L_1 \vee L_2 \vee L_3$ can be encoded by an atom $R(x_{P_1}, p_1, x_{P_2}, p_2, x_{P_3}, p_3)$ where $p_i$ is 0 if $L_i$ is negative and 1 otherwise and $P_i$ is the predicate of $L_i$. This way the clause set $N$ can be represented by a single BS clause $C_N$. Now construct a clause $D$ that contains all atoms representing the way a clause of length three can become true by ground atoms over $R$ and constants 0, 1. For example, it contains atoms like $R(0, 0, \ldots)$ and $R(1, 1, \ldots)$ representing that the first literal of a clause is true. Actually, for each such atom $R(0, 0, \ldots)$ the clause $D$ contains $|C_N|$ copies. Finally, $C_N$ subsumes $D$ if and only if $N$ is satisfiable. □

In order to be efficient, modern theorem provers need to decide multiple thousand subsumption checks per second. In the pure first-order case, this is possible because of indexing and filtering techniques that quickly decide most subsumption checks [46,27,56,45,39,40,33,28,61,29,30,59,48,62,24,47,49,53,52,54,25].

For BS(LRA) (and FOL(LRA)), there also exists research on how to perform the subsumption check in general [2,36], but the literature contains no dedicated indexing or filtering techniques for the constraint part of the subsumption check. In this section and as the main contribution of this paper, we present the first such filtering techniques for BS(LRA). But first, we explain how to solve the subsumption check for constrained clauses in general.

**First-Order Check** The first step of the subsumption check is exactly the same as in first-order logic without arithmetic. We have to find a substitution $\sigma$, also called a *matcher*, such that $C_1\sigma \subseteq C_2$. The only difference is that it is not enough to compute one matcher $\sigma$, but we have to compute all matchers for $C_1\sigma \subseteq C_2$ until we find one that satisfies the implication $\Lambda_2 \rightarrow (\Lambda_1\sigma)$. For instance, there are two matchers for the clauses $C_1 := x + y \geq 0 \parallel Q(x, y)$ and $C_2 := x < 0, y \geq 0 \parallel Q(x, x) \vee Q(y, y)$. The matcher $\{x \mapsto y\}$ satisfies the implication $\Lambda_2 \rightarrow (\Lambda_1\sigma)$ and $\{y \mapsto x\}$ does not. Our own algorithm for finding matchers is in the style of Stillman except that we continue after we find the first matcher [58,27].

**Implication Check** The universal closure of the implication $\Lambda_2 \rightarrow (\Lambda_1\sigma)$ can be solved by any SMT solver for the respective theory after we negate it. Note that the resulting formula

$$\exists x_1, \ldots, x_n.\ \Lambda_2 \wedge \neg(\Lambda_1\sigma) \qquad \text{where } \{x_1, \ldots, x_n\} = \text{vars}(\Lambda_2) \tag{1}$$

is already in clause normal form and that the formula can be treated as ground since existential variables can be handled as constants. Intuitively, the universal closure $\Lambda_2 \rightarrow (\Lambda_1\sigma)$ asserts that the set of solutions satisfying $\Lambda_2$ is a subset of

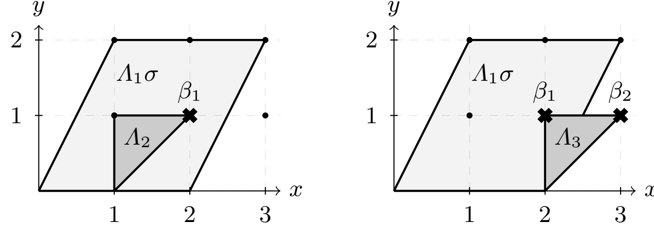


**Figure 1.** Solutions of the constraints $\Lambda_1\sigma$, $\Lambda_2$, and $\Lambda_3$ depicted as polytopes

the set of solutions satisfying $\Lambda_1\sigma$. This means a solution to its negation (1) is a solution for $\Lambda_2$, but not for $\Lambda_1\sigma$, thus a counterexample of the subset relation.

*Example 7.* Let us now look at an example to illustrate the role that formula (1) plays in deciding subsumption. In our example, we have three clauses: $\Lambda_1 \parallel C_1$, $\Lambda_2 \parallel C_2$, and $\Lambda_3 \parallel C_2$, where $C_1 := \neg P(x,y) \vee Q(u,z)$, $C_2 := \neg P(x,y) \vee Q(2,x)$, $\Lambda_1 := y \geq 0\ ,\ y \leq u\ ,\ y \leq x{+}z\ ,\ y \geq x{+}z{-}2{\cdot}u$, $\Lambda_2 := x \geq 1\ ,\ y \leq 1\ ,\ y \geq x{-}1$, and $\Lambda_3 := x \geq 2\ ,\ y \leq 1\ ,\ y \geq x - 2$. Our goal is to test whether $\Lambda_1 \parallel C_1$ subsumes the other two clauses. As our first step, we try to find a substitution $\sigma$ such that $C_1\sigma \subseteq C_2$. The most general substitution fulfilling this condition is $\sigma := \{z \mapsto x, u \mapsto 2\}$. Next, we check whether $\Lambda_1\sigma$ is implied by $\Lambda_2$ and $\Lambda_3$. Normally, we would do so by solving the formula (1) with an SMT solver, but to help our intuitive understanding, we instead look at their solution sets depicted in Figure 1. Note that $\Lambda_1\sigma$ simplifies to $\Lambda_1\sigma := y \geq 0\ ,\ y \leq 2\ ,\ y \leq 2{\cdot}x\ ,\ y \geq 2{\cdot}x{-}4$. Here we see that the solution set for $\Lambda_2$ is a subset of $\Lambda_1\sigma$. Hence, $\Lambda_2$ implies $\Lambda_1\sigma$, which means that $\Lambda_2 \parallel C_2$ is subsumed by $\Lambda_1 \parallel C_1$. The solution set for $\Lambda_3$ is not a subset of $\Lambda_1\sigma$. For instance, the assignment $\beta_2 := \{x \mapsto 3, y \mapsto 1\}$ is a counterexample and therefore a solution to the respective instance of formula (1). Hence, $\Lambda_1 \parallel C_1$ does not subsume $\Lambda_3 \parallel C_2$.

**Excess Variables** Note that in general it is not sufficient to find a substitution $\sigma$ that matches the first-order parts to also match the theory constraints: $C_1\sigma \subseteq C_2$ does not generally imply $\mathrm{vars}(\Lambda_1\sigma) \subseteq \mathrm{vars}(\Lambda_2)$. In particular, if $\Lambda_1$ contains variables that do not appear in the first-order part $C_1$, then these must be projected to $\Lambda_2$. We arrive at a variant of (1), that is $\exists x_1, \ldots, x_n \forall y_1, \ldots, y_m.\ \Lambda_2 \wedge \neg(\Lambda_1\sigma)$ where $\{x_1, \ldots, x_n\} = \mathrm{vars}(\Lambda_2)$ and $\{y_1, \ldots, y_m\} = \mathrm{vars}(\Lambda_1) \setminus \mathrm{vars}(C_1)$. Our solution to this problem is to normalize all clauses $\Lambda \parallel C$ by eliminating all *excess variables* $\mathcal{Y} := \mathrm{vars}(\Lambda) \setminus \mathrm{vars}(C)$ such that $\mathrm{vars}(\Lambda) \subseteq \mathrm{vars}(C)$ is guaranteed. For linear real arithmetic this is possible with quantifier elimintation techniques, e.g., Fourier-Motzkin elimination (FME). Although these techniques typically cause the size of $\Lambda$ to increase exponentially, they often behave well in practice. In fact, we get rid of almost all excess variables in our benchmark examples with simplification techniques based on Gaussian elimination with execution time linear in the number of LRA atoms. Given the precondition $\mathcal{Y} = \emptyset$ achieved by such elimination techniques, we can compute $\sigma$ as matcher for the first-order parts and then

directly use it for testing whether the universal closure of $\Lambda_2 \rightarrow (\Lambda_1\sigma)$ holds. An alternative solution to the issue of excess variables has been proposed: In [2], the substitution $\sigma$ is decomposed as $\sigma = \delta\tau$, where $\delta$ is the first-order matcher and $\tau$ is a *theory matcher*, i.e. $\mathrm{dom}(\tau) \subseteq \mathcal{Y}$ and $\mathrm{vars}(\mathrm{codom}(\tau)) \subseteq \mathrm{vars}(\Lambda_2)$. Then, exploiting Farkas' lemma, the computation of $\tau$ is reduced to testing the feasibility of a linear program (restricted to matchers that are affine transformations).

The reduction to solving a linear program offers polynomial worst-case complexity but in practice typically behaves worse than solving the variant with quantifier alternations using an SMT solver such as Z3 [42,36].

**Filtering First-Order Literals** Even though deciding implication of theory constraints is in practice more expensive than constructing a matcher and deciding inclusion of first-order literals, we still incorporate some lightweight filters for our evaluation. Inspired by Schulz [54] we choose three features, so that every feature $f$ maps clauses to $\mathbb{N}_0$, and $f(C_1) \leqslant f(C_2)$ is necessary for $C_1\sigma \subseteq C_2$.

The features are: $|C^+|$, the number of positive first-order literals in $C$, $|C^-|$, the number of negative first-order literals in $C$, and $\lfloor C \rfloor$, the number of occurrences of constants in $C$.

**Sample Point Heuristic** The majority of subsumption tests fail because we cannot find a fitting substitution for their first-order parts. In our experiments, between 66.5% and 99.9% of subsumption tests failed this way. This means our tool only has to check in less than 33.5% of the cases whether one theory constraint implies the other. Despite this, our tool spends more time on implication checks than on the first-order part of the subsumption tests without filtering on the constraint implication tests. The reason is that constraint implication tests are typically much more expensive than the first-order part of a subsumption test. For this reason, we developed the *sample point heuristic* that is much faster to execute than a full constraint implication test, but still filters out the majority of implications that do not hold (in our experiments between 93.8% and 100%).

The idea behind the sample point heuristic is straightforward. We store for each clause $\Lambda \,\|\, C$ a sample solution $\beta$ for its theory constraint $\Lambda$. Before we execute a full constraint implication test, we simply evaluate whether the sample solution $\beta$ for $\Lambda_2$ is also a solution for $\Lambda_1\sigma$. If this is not the case, then $\beta$ is a solution for (1) and a counterexample for the implication. If $\beta$ is a solution for $\Lambda_1\sigma$, then the heuristic returns unknown and we have to execute a full constraint implication test, i.e., solve the SMT problem (1).

Often it is possible to get our sample solutions for free. Theorem provers based on hierarchic superposition typically check for every new clause $\Lambda \,\|\, C$ whether $\Lambda$ is satisfiable in order to eliminate tautologies. This means we can already use this tautology check to compute and store a sample solution for every new clause without extra cost. We only need to pick a solver for the check that returns a solution as a certificate of satisfiability. Although the SCL(T) calculus never learns any tautologies, it is also possible to get a sample solution for free as part of its conflict analysis [11].

*Example 8.* We revisit Example 7 to illustrate the sample point heuristic. During the tautology check for $\Lambda_2 \,\|\, C_2$ and $\Lambda_3 \,\|\, C_2$, we determined that $\beta_1 := \{x \mapsto 2, y \mapsto 1\}$ is a sample solution for $\Lambda_2$ and $\beta_2 := \{x \mapsto 3, y \mapsto 1\}$ a sample solution for $\Lambda_3$. Since $\Lambda_2$ implies $\Lambda_1\sigma$, all sample solutions for $\Lambda_2$ automatically satisfy $\Lambda_1\sigma$. This is the reason why the sample point heuristic never filters out an implication that actually holds, i.e., it returns unknown when we test whether $\Lambda_2$ implies $\Lambda_1\sigma$. The assignment $\beta_2$ on the other hand does not satisfy $\Lambda_1\sigma$. Hence, the sample point heuristic correctly claims that $\Lambda_3$ does not imply $\Lambda_1\sigma$. Note that we could also have chosen $\beta_1$ as the sample point for $\Lambda_3$. In this case, the sample point heuristic would also return unknown for the implication $\Lambda_3 \to \Lambda_1\sigma$ although the implication does not hold.

**Trivial Cases** Subsumption tests become much easier if the constraint $\Lambda_i$ of one of the participating clauses is empty. We use two heuristic filters to exploit this fact. We highlight them here because they already exclude some subsumption tests before we reach the sample point heuristic in our implementation.

The *empty conclusion heuristic* exploits that $\Lambda_1$ is valid if $\Lambda_1$ is empty. In this case, all implications $\Lambda_2 \to (\Lambda_1\sigma)$ hold because $\Lambda_1\sigma$ evaluates to true under any assignment. So by checking whether $\Lambda_1 = \emptyset$, we can quickly determine whether $\Lambda_2 \to (\Lambda_1\sigma)$ holds for some pairs of clauses. Note that in contrast to the sample point heuristic, this heuristic is used to find valid implications.

The *empty premise test* exploits that $\Lambda_2$ is valid if $\Lambda_2$ is empty. In this case, an implication $\Lambda_2 \to (\Lambda_1\sigma)$ may only hold if $\Lambda_1\sigma$ simplifies to the empty set as well. This is the case because any inequality in the canonical form $\sum_{i=1}^{n} a_i x_i \triangleleft c$ either simplifies to true (because $a_i = 0$ for all $i = 1, \ldots, n$ and $0 \triangleleft c$ holds) and can be removed from $\Lambda_1\sigma$, or the inequality eliminates at least one assignment as a solution for $\Lambda_1\sigma$ [51]. So if $\Lambda_2 = \emptyset$, we check whether $\Lambda_1\sigma$ simplifies to the empty set instead of solving the SMT problem (1).

**Pipeline** We call our approach a *pipeline* since it combines multiple procedures, which we call *stages*, that vary in complexity and are independent in principle, for the overall aim of efficiently testing subsumption. Pairs of clauses that "make it through" all stages, are those for which the subsumption relation holds. The pipeline is designed with two goals in mind: (1.) To reject as many pairs of clauses as early as possible, and (2.) to move stages further towards the end of the pipeline the more expensive they are.

The pipeline consists of six stages, all of which are mentioned above. We divide the pipeline into two phases, the *first-order phase* (FO-phase) consisting of two stages, and the *constraint phase* (C-phase), consisting of four stages. First-order filtering rejects all pairs of clauses for which $f(C_1) > f(C_2)$ holds. Then, matching constructs all matchers $\sigma$ such that $C_1\sigma \subseteq C_2$. Every matcher is individually tested in the constraint phase. Technically, this means that the input of all following stages is not just a pair of clauses, but a triple of two clauses and a matcher. The constraint phase then proceeds with the empty conclusion heuristic and the empty premise test to accept (resp. reject) all trivial cases of

**Algorithm 1:** Saturation prover used for evaluation

```
    Input  : A set N of clauses.
    Output : ⊥ or "unknown".
 1  U := {C ∈ N | |C| = 1}
 2  while U ≠ ∅ do
 3  |  M := ∅
 4  |  foreach C ∈ U do M := M ∪ resolvents(C, N)
 5  |  if ⊥ ∈ M then return ⊥
 6  |  reduce M using N          (forward subsumption)
 7  |  if M = ∅ then return "unknown"
 8  |  reduce N using M          (backward subsumption)
 9  |  U := {C ∈ M | |C| = 1}
10  |  N := N ∪ M
11  end
12  return "unknown"
```

the constraint implication test. The next stage is the sample point heuristic. If the sample solution $\beta_2$ for $\Lambda_2$ is no solution for $\Lambda_1$ (i.e. $\not\models \Lambda_1\sigma\beta_2$), then the matcher $\sigma$ is rejected. Otherwise (i.e. $\models \Lambda_1\sigma\beta_2$), the implication test $\Lambda_2 \rightarrow (\Lambda_1\sigma)$ is performed by solving the SMT problem (1) to produce the overall result of the pipeline and finally determine whether subsumption holds.

## 4 Experimentation

In order to evaluate our new approach on three benchmark instances, derived from BS(LRA) applications, all presented techniques and their combination in form of a pipeline were implemented in the theorem prover SPASS-SPL, a prototype for BS(LRA) reasoning.

Note that SPASS-SPL contains more than one approach for BS(LRA) reasoning, e.g., the Datalog hammer for HBS(LRA) reasoning [10]. These various modes of operation operate independently, and the desired mode is chosen via command-line option. The reasoning approach discussed here is the current default option. On the first-order side, SPASS-SPL consists of a simple saturation prover based on hierarchic unit resolution, see Algorithm 1. It resolves unit clauses with other clauses until either the empty clause is derived or no new clauses can be derived. Note that this procedure is only complete for Horn clauses. For arithmetic reasoning, SPASS-SPL relies on SPASS-SATT, our sound and complete CDCL(LA) solver for quantifier-free linear real and linear mixed/integer arithmetic [12]. SPASS-SATT implements a version of the dual simplex algorithm fine-tuned towards SMT solving [16]. In order to ensure soundness, SPASS-SATT represents all numbers with the help of the *arbitrary-precision arithmetic library* FLINT [31]. This means all calculations, including the implication test and the sample point heuristic, are always exact and thus free of numerical errors. The most relevant part of SPASS-SPL with regards to

**Table 1.** Overview of how many clause pairs advance in the pipeline (top to bottom)

| | | lc | bakery, tad | All |
|---|---|---|---|---|
| All | | 1 244 819k | 196 437k | 1 441 256k |
| FO | Filtering | 61.21% | 85.03% | 64.45% |
| | $f(C_1) \leq f(C_2)$ | 761 905k 61.2061% | 167 025k 85.0274% | 928 931k 64.4540% |
| | Matching | 0.02% | 39.83% | 7.18% |
| | $C_1\sigma \subseteq C_2$ | 131k 0.0106% | 66 531k 33.8694% | 66 664k 4.6254% |
| C | Empty (pre./con.) | 44.73% | 100.00% | 99.89% |
| | $\not\models \Lambda_1\sigma, \not\models \Lambda_2$ | 59k 0.0047% | 66 531k 33.8694% | 66 591k 4.6203% |
| | Sample point | 59.28% | 0.12% | 0.18% |
| | $\models \Lambda_1\sigma\beta_2$ | 35k 0.0028% | 82k 0.0416% | 117k 0.0081% |
| | Implication | 95.51% | 100.00% | 98.66% |
| Subsumes | | 33k 0.0027% | 82k 0.0416% | 115k 0.0080% |

**Table 2.** An overview of the accuracy of non-perfect pipeline stages

| Test | Specificity/Sensitivity | | | Pos./Neg. Predictive Value | | |
|---|---|---|---|---|---|---|
| Instances | lc | bakery, tad | All | lc | bakery, tad | All |
| FO Filtering | 0.38797 | 0.14979 | 0.35552 | 0.00013 | 0.00049 | 0.00020 |
| FO Matching | 0.99996 | 0.60196 | 0.92841 | 0.78456 | 0.00123 | 0.00275 |
| Empty Conclusion | 0.70973 | 0.00000 | 0.00103 | 0.54474 | 0.00123 | 0.00173 |
| Sample Point | 0.93864 | 1.00000 | 0.99998 | 0.95510 | 1.00000 | 0.98653 |

this paper is that it performs tautology and subsumption deletion to eliminate redundant clauses. As a preprocessing step, SPASS-SPL eliminates all tautologies from the set of input clauses. Similarly, the function resolvents$(C, N)$ (see Line 4 of Algorithm 1) filters out all newly derived clauses that are tautologies. Note that we also use these tautology checks to eliminate all excess variables and to store sample solutions for all remaining clauses. After each iteration of the algorithm, we also check for subsumed clauses. We first eliminate newly generated clauses by forward subsumption (see Line 6 of Algorithm 1), then use the remaining clauses for backward subsumption (see Line 8 of Algorithm 1).

*Benchmarks* Our benchmarking instances come out of three different applications. (1.) A supervisor for an automobile lane change assistant, formulated in the Horn fragment of BS(LRA) [10,9] (five instances, referred to as `lc` in aggregate). (2.) The formalization of reachability for non-deterministic timed automata, formulated in the non-Horn fragment of BS(LRA) [20] (one instance, referred to as `tad`). (3.) Formalizations of variants of mutual exclusion protocols, such as the bakery protocol [38], also formulated in the non-Horn fragment of BS(LRA) [19] (one instance, referred to as `bakery`). The machine used for benchmarking features an Intel Xeon W-1290P CPU (10 cores, 20 threads, up to 5.2 GHz) and 64 GiB DDR4-2933 ECC main memory. Runtime was limited to ten minutes, and memory usage was not limited.

**Table 3.** Evaluation of the sample point heuristic

| Instances | | `lc` | `bakery`, `tad` | All |
|---|---|---|---|---|
| Bottleneck | (C time ÷ FO time) | | | |
| | *without* sample point | 127 | 2757 | 14867 |
| | *with* sample point | 78 | 32 | 89 |
| Avg. pipeline runtime in $\mu s$ | | | | |
| | *without* sample point | 0.0315 | 89.9401 | 0.5189 |
| | *with* sample point | 0.0311 | 1.4150 | 0.2197 |
| Speedup | (C time *with* ÷ *without*) | 1.63 | 137.88 | 124.16 |
| Benefit-to-cost | (C time *taken* ÷ *saved*) | 6.74 | 181.72 | 163.72 |

*Evaluation* In Table 1 we give an overview of how many pairs of clauses advance how far in the pipeline (in thousands). Rows with grey background refer to a stage of the pipeline and show which portion of pairs of clauses were kept, relative to the previous stage. Rows with white background refer to (virtual) sets of clauses, their absolute size, and their size relative to the number of attempted tests, as well as the condition(s) established. The three groups of columns refer to groups of benchmark instances. Results vary greatly between `lc` and the aggregate of `bakery` and `tad`. In `lc` the relative number of subsumed clauses is significantly smaller (0.0027% compared to 0.0416%). FO Matching eliminates a large number of pairs in `lc`, because the number of predicate symbols, and their arity (`lc1`, ..., `lc4`: 36 predicates, arities up to 5; `lc5`: 53 predicates, arities up to 12) is greater than in `bakery` (11 predicates, all of arity 2) and `tad` (4 predicates, all of arity 2).

*Binary Classifiers* To evaluate the performance of each stage of the proposed test pipeline, we view each stage individually as a binary classifier on pairs of constrained clauses. The two classes we consider are “subsumes” (positive outcome) and “does not subsume” (negative outcome). Each stage of the pipeline computes a *prediction* on the *actual* result of the overall pipeline. We are thus interested in minimizing two kinds of errors: (1.) When one stage of the pipeline predicts that the subsumption test will succeed (the prediciton is positive) but it fails (the actual result is negative), called *false positive* (FP). (2.) When one stage of the pipeline predicts that the subsumption test will fail (the prediction is negative) but it succeeds (the actual result is positive), called *false negative* (FN). Dually, a correct prediction is called *true positive* (TP) and *true negative* (TN). For each stage, at least one kind of error is excluded by design: First-order filtering and the sample point heuristic never produce false negatives. The empty conclusion heuristic never produces false positives. The empty premise test is perfect, i.e. it neither produces false positives nor false negatives, with the caveat of not always being applicable. The last stage (implication test) decides the overall result of the pipeline, and thus is also perfect. For evaluation of binary classifiers, we use four different measures (two symmetric pairs):

$$\mathrm{SPC} = \mathrm{TN} \div (\mathrm{TN} + \mathrm{FP}) \qquad \mathrm{PPV} = \mathrm{TP} \div (\mathrm{TP} + \mathrm{FP}) \tag{2}$$

The first pair, *specificity* (SPC) and *positive predictive value*, see (2), is relevant only in presence of false postives (the measures approach 1 as FP approaches 0).

$$\text{SEN} = \text{TP} \div (\text{TP} + \text{FN}) \qquad\qquad \text{NPV} = \text{TN} \div (\text{TN} + \text{FN}) \tag{3}$$

The second pair, *sensitivity* (SEN) and *negative predictive value* (NPV), see (3), is relevant only in presence of false negatives (the measures approach 1 as FN approaches 0). Specificity (resp. sensitivity) might be considered the "success rate" in our setup. They answer the question: "Given the *actual* result of the pipeline is 'subsumed' (resp. 'not subsumed'), in how many cases does this stage *predict* correctly?" A specificity (resp. sensitivity) of 0.99 means that the classifier produces a false positive (resp. negative), i.e. a wrong prediction, in one out of one hundred cases. Both measures are independent of the prevalence of particular actual results, i.e. the measures are not biased by instances that feature many (or few) subsumed clauses. On the other hand, positive and negative predictive value are biased by prevalence. They answer the following question: "Given this stage of the pipeline *predicts* 'subsumed' (resp. 'not subsumed'), how likely is it that the *actual* result indeed is 'subsumed' (resp. 'not subsumed')?"

In Table 2 we present for all non-perfect stages of the pipeline specificity (for those that produce false positives) and sensitivity (for those that produce false negatives) as well as the (positive/negative) predictive value. Note that the sample point heuristic has an exceptionally high specificity, still above 93% in the benchmarks where it performed worst. For the benchmarks `bakery` and `tad` it even performs perfectly. Combined, this gives a specificity of above 99.99%. Considering FO Filtering, we expect limited performance, since the structure of terms in BS is flat compared to the rich structure of terms as trees in full first-order logic. This is evidenced by a comparatively low specificity of 35%. However, this classifier is very easy to compute, so pays for itself. FO Matching is a much better classifier, at an aggregate sensitivity of 93%. Even though this classifier is NP-complete, this is not problematic in practice.

*Runtime* In Table 3 we focus on the runtime improvement achieved by the sample point heuristic. In the first two lines (Bottleneck), we highlight how much slower testing implication of constraints (the C-phase) is compared to treating the first-order part (the FO-phase). This is equivalent to the time taken for the C-phase per pair of clauses (that reach at least the first C-phase) divided by the time taken for the FO-phase per pair of clauses. We see that without the sample point heuristic, we can expect the constraint implication test to take hundreds to thousands of times longer than the FO-phase. Adding the sample point heuristic decreases this ratio to below one hundred. In the fourth line (avg. pipeline runtime) we do not give a ratio, but the average time it takes to compute the whole pipeline. We achieve millions of subsumption checks per second. In the fifth line (Speedup), we take the time that all C-phases combined take per pair of clauses that reach at least the first C-phase, and take the ratio to the same time without applying the sample point heuristic. In the sixth line (Benefit-to-cost), we consider the time taken to compute the sample point vs.

the time it saves. The benefit is about two orders of magnitude greater than the cost.

## 5 Conclusion

Our next step will be the integration of the subsumption test in the backward subsumption procedure of an SCL based reasoning procedure for BS(LRA) [11] which is currently under development.

There are various ways to improve the sample point heuristic. One improvement would be to store and check multiple sample points per clause. For instance, whenever the sample point heuristic fails and the implication test for $\Lambda_2 \rightarrow (\Lambda_1\sigma)$ also fails, store the solution to (1) as an additional sample point for $\Lambda_2$. The new sample point will filter out any future implication tests with $\Lambda_1\sigma$ or similar constraints. However, testing too many sample points might lead to costs outweighing benefits. A potential solution to this problem would be score-based garbage collection, as done in SAT solvers [57]. Another way to store and check multiple sample points per clause is to store a compact description of a set of points that is easy to check against. For instance, we can store the center point and edge length of the largest orthogonal hypercube contained in the solutions of a constraint, which is equivalent to infinitely many sample points. Computing the largest orthogonal hypercube for an LRA constraint is not much harder than finding a sample solution [14]. Checking whether a cube is contained in an LRA constraint works almost the same as evaluating a sample point [14].

Although we developed our sample point technique for the BS(LRA) fragment it is obvious that it will also work for the overall FOL(LRA) clause fragment, because this extension does not affect the LRA constraint part of clauses. From an automated reasoning perspective, satisfiability of the FOL(LRA) and BS(LRA) fragments (clause sets) is undecidable in both cases. Actually, satisfiability of a BS(LRA) clause set is already undecidable if the first-order part is restricted to a single monadic predicate [32]. The first-order part of BS(LRA) is decidable and therefore enables effective guidance for an overall reasoning procedure [11]. Form an application perspective, the BS(LRA) fragment already encompasses a number of used (sub)languages. For example, timed automata [3] and a number of extensions thereof are contained in the BS(LRA) fragment [60].

We also believe that the sample point heuristic will speed up the constraint implication test for FOL(LIA), first-order clauses over linear integer arithmetic, FOL(NRA), i.e., first-order clauses over non-linear real arithmetic, and other combinations of FOL with arithmetic theories. However, the non-linear case will require a more sophisticated setup due to the nature of test points in this case, e.g., a solution may contain root expressions.

*Acknowledgments* This work was partly funded by DFG grant 389792660 as part of TRR 248, see `https://perspicuous-computing.science`. We thank the anonymous reviewers for their thorough reading and detailed constructive comments. Martin Desharnais suggested some textual improvements.

## References


1. Alagi, G., Weidenbach, C.: NRCL - A model building approach to the bernays-schönfinkel fragment. In: FroCoS 2015. LNCS, vol. 9322, pp. 69–84. Springer (2015). https://doi.org/10.1007/978-3-319-24246-0_5
2. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition modulo linear arithmetic SUP(LA). In: FroCoS 2009. LNCS, vol. 5749, pp. 84–99. Springer (2009). https://doi.org/10.1007/978-3-642-04222-5_5
3. Alur, R., Dill, D.L.: A theory of timed automata. TCS **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
4. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. LOGCOM **4**(3), 217–247 (1994). https://doi.org/10.1093/logcom/4.3.217
5. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 19–99. Elsevier and MIT Press (2001). https://doi.org/10.1016/b978-044450813-3/50004-7
6. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierarchic first-order theories. AAECC **5**, 193–212 (1994). https://doi.org/10.1007/BF01190829
7. Baumgartner, P., Waldmann, U.: Hierarchic superposition revisited. In: Description Logic, Theory Combination, and All That - Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday. LNCS, vol. 11560, pp. 15–56. Springer (2019). https://doi.org/10.1007/978-3-030-22102-7_2
8. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
9. Bromberger, M., Dragoste, I., Faqeh, R., Fetzer, C., González, L., Krötzsch, M., Marx, M., Murali, H.K., Weidenbach, C.: A sorted datalog hammer for supervisor verification conditions modulo simple linear arithmetic (2022), https://arxiv.org/abs/2201.09769
10. Bromberger, M., Dragoste, I., Faqeh, R., Fetzer, C., Krötzsch, M., Weidenbach, C.: A datalog hammer for supervisor verification conditions modulo simple linear arithmetic. In: FroCoS 2021. LNCS, vol. 12941, pp. 3–24. Springer (2021). https://doi.org/10.1007/978-3-030-86205-3_1
11. Bromberger, M., Fiori, A., Weidenbach, C.: Deciding the bernays-schoenfinkel fragment over bounded difference constraints by simple clause learning over theories. In: VMCAI 2021. LNCS, vol. 12597, pp. 511–533. Springer (2021). https://doi.org/10.1007/978-3-030-67067-2_23
12. Bromberger, M., Fleury, M., Schwarz, S., Weidenbach, C.: SPASS-SATT - A CDCL(LA) solver. In: CADE-27, 2019. LNCS, vol. 11716, pp. 111–122. Springer (2019). https://doi.org/10.1007/978-3-030-29436-6_7
13. Bromberger, M., Leutgeb, L., Weidenbach, C.: An Efficient Subsumption Test Pipeline for BS(LRA) Clauses (2022). https://doi.org/10.5281/zenodo.6544456, Supplementary Material
14. Bromberger, M., Weidenbach, C.: Fast cube tests for LIA constraint solving. In: IJCAR 2016. LNCS, vol. 9706, pp. 116–132. Springer (2016). https://doi.org/10.1007/978-3-319-40229-1_9
15. Dershowitz, N.: Orderings for term-rewriting systems. TCS **17**, 279–301 (1982). https://doi.org/10.1016/0304-3975(82)90026-3

16. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer (2006). https://doi.org/10.1007/11817963_11
17. Eggers, A., Kruglov, E., Kupferschmid, S., Scheibler, K., Teige, T., Weidenbach, C.: Superposition modulo non-linear arithmetic. In: FroCoS 2011. LNCS, vol. 6989, pp. 119–134. Springer (2011). https://doi.org/10.1007/978-3-642-24364-6_9
18. Faqeh, R., Fetzer, C., Hermanns, H., Hoffmann, J., Klauck, M., Köhl, M.A., Steinmetz, M., Weidenbach, C.: Towards dynamic dependable systems through evidence-based continuous certification. In: ISoLA 2020. LNCS, vol. 12477, pp. 416–439. Springer (2020). https://doi.org/10.1007/978-3-030-61470-6_25
19. Fietzke, A.: Labelled superposition. Ph.D. thesis, Universität des Saarlandes (2014). https://doi.org/10.22028/D291-26569
20. Fietzke, A., Weidenbach, C.: Superposition as a decision procedure for timed automata. MICS **6**(4), 409–425 (2012). https://doi.org/10.1007/s11786-012-0134-5
21. Fiori, A., Weidenbach, C.: SCL clause learning from simple models. In: CADE-27, 2019. LNCS, vol. 11716, pp. 233–249. Springer (2019). https://doi.org/10.1007/978-3-030-29436-6_14
22. Fiori, A., Weidenbach, C.: SCL with theory constraints (2020), https://arxiv.org/abs/2003.04627
23. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. JSAT **1**(3-4), 209–236 (2007). https://doi.org/10.3233/sat190012
24. Ganzinger, H., Nieuwenhuis, R., Nivela, P.: Fast term indexing with coded context trees. JAR **32**(2), 103–120 (2004). https://doi.org/10.1023/B:JARS.0000029963.64213.ac
25. Gleiss, B., Kovács, L., Rath, J.: Subsumption demodulation in first-order theorem proving. In: IJCAR 2020. LNCS, vol. 12166, pp. 297–315. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_17
26. Gottlob, G.: Subsumption and implication. IPL **24**(2), 109–111 (1987). https://doi.org/10.1016/0020-0190(87)90103-7
27. Gottlob, G., Leitsch, A.: On the efficiency of subsumption algorithms. JACM **32**(2), 280–295 (1985). https://doi.org/10.1145/3149.214118
28. Graf, P.: Extended path-indexing. In: CADE-12, 1994. LNCS, vol. 814, pp. 514–528. Springer (1994). https://doi.org/10.1007/3-540-58156-1_37
29. Graf, P.: Substitution tree indexing. In: Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5-7, 1995, Proceedings. LNCS, vol. 914, pp. 117–131. Springer (1995). https://doi.org/10.1007/3-540-59200-8_52
30. Graf, P.: Term Indexing, LNCS, vol. 1053. Springer (1996). https://doi.org/10.1007/3-540-61040-5
31. Hart, W.B.: Fast library for number theory: An introduction. In: ICMS 2010. LNCS, vol. 6327, pp. 88–91. Springer (2010). https://doi.org/10.1007/978-3-642-15582-6_18
32. Horbach, M., Voigt, M., Weidenbach, C.: The universal fragment of presburger arithmetic with unary uninterpreted predicates is undecidable (2017), http://arxiv.org/abs/1703.01212
33. Jr., P.W.P., Brown, C.A.: Fast many-to-one matching algorithms. In: Rewriting Techniques and Applications, First International Conference, RTA-85, Dijon, France, May 20-22, 1985, Proceedings. LNCS, vol. 202, pp. 407–416. Springer (1985). https://doi.org/10.1007/3-540-15976-2_21

34. Jr., R.J.B., Schrag, R.: Using CSP look-back techniques to solve exceptionally hard SAT instances. In: Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996. LNCS, vol. 1118, pp. 46–60. Springer (1996). https://doi.org/10.1007/3-540-61551-2_65
35. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer (2007). https://doi.org/10.1007/978-3-540-74915-8_19
36. Kruglov, E.: Superposition modulo theory. Ph.D. thesis, Universität des Saarlandes (2013). https://doi.org/10.22028/D291-26547
37. Kruglov, E., Weidenbach, C.: Superposition decides the first-order logic fragment over ground theories. MICS **6**(4), 427–456 (2012). https://doi.org/10.1007/s11786-012-0135-4
38. Lamport, L.: A new solution of dijkstra's concurrent programming problem. CACM **17**(8), 453–455 (1974). https://doi.org/10.1145/361082.361093
39. McCune, W.: OTTER 2.0. In: CADE-10, 1990. LNCS, vol. 449, pp. 663–664. Springer (1990). https://doi.org/10.1007/3-540-52885-7_131
40. McCune, W.: Experiments with discrimination-tree indexing and path indexing for term retrieval. JAR **9**(2), 147–167 (1992). https://doi.org/10.1007/BF00245458
41. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC 2001. pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017
42. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: ETAPS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
43. Nieuwenhuis, R., Hillenbrand, T., Riazanov, A., Voronkov, A.: On the evaluation of indexing techniques for theorem proving. In: IJCAR 2001. LNCS, vol. 2083, pp. 257–271. Springer (2001). https://doi.org/10.1007/3-540-45744-5_19
44. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 371–443. Elsevier and MIT Press (2001). https://doi.org/10.1016/b978-044450813-3/50009-6
45. Ohlbach, H.J.: Abstraction tree indexing for terms. In: 9th European Conference on Artificial Intelligence, ECAI 1990, Stockholm, Sweden, 1990. pp. 479–484 (1990)
46. Overbeek, R.A., Lusk, E.L.: Data structures and control architectures for implementation of theorem-proving programs. In: CADE-5, 1980. LNCS, vol. 87, pp. 232–249. Springer (1980). https://doi.org/10.1007/3-540-10009-1_19
47. Ramakrishnan, I.V., Sekar, R.C., Voronkov, A.: Term indexing. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 1853–1964. Elsevier and MIT Press (2001). https://doi.org/10.1016/b978-044450813-3/50028-x
48. Riazanov, A., Voronkov, A.: Partially adaptive code trees. In: JELIA 2000. LNCS, vol. 1919, pp. 209–223. Springer (2000). https://doi.org/10.1007/3-540-40006-0_15
49. Riazanov, A., Voronkov, A.: Efficient instance retrieval with standard and relational path indexing. Information and Computation **199**(1-2), 228–252 (2005). https://doi.org/10.1016/j.ic.2004.10.012
50. Robinson, J.A.: A machine-oriented logic based on the resolution principle. JACM **12**(1), 23–41 (1965). https://doi.org/10.1145/321250.321253
51. Schrijver, A.: Theory of linear and integer programming. Wiley-Interscience series in discrete mathematics and optimization, Wiley (1999)

52. Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving. Elsevier Science (2004)
53. Schulz, S.: Fingerprint indexing for paramodulation and rewriting. In: IJCAR 2012. LNCS, vol. 7364, pp. 477–483. Springer (2012). https://doi.org/10.1007/978-3-642-31365-3_37
54. Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: Automated Reasoning and Mathematics - Essays in Memory of William W. McCune. LNCS, vol. 7788, pp. 45–67. Springer (2013). https://doi.org/10.1007/978-3-642-36675-8_3
55. Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: ICCAD 1996. pp. 220–227. IEEE Computer Society / ACM (1996). https://doi.org/10.1109/ICCAD.1996.569607
56. Socher, R.: A subsumption algorithm based on characteristic matrices. In: CADE-9, 1988. LNCS, vol. 310, pp. 573–581. Springer (1988). https://doi.org/10.1007/BFb0012858
57. Soos, M., Kulkarni, R., Meel, K.S.: Crystalball: Gazing in the black box of SAT solving. In: SAT 2019. LNCS, vol. 11628, pp. 371–387. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_26
58. Stillman, R.B.: The concept of weak substitution in theorem-proving. JACM **20**(4), 648–667 (1973). https://doi.org/10.1145/321784.321792
59. Tammet, T.: Towards efficient subsumption. In: CADE-15, 1998. LNCS, vol. 1421, pp. 427–441. Springer (1998). https://doi.org/10.1007/BFb0054276
60. Voigt, M.: Decidable $\exists^*\forall^*$ first-order fragments of linear rational arithmetic with uninterpreted predicates. JAR **65**(3), 357–423 (2021). https://doi.org/10.1007/s10817-020-09567-8
61. Voronkov, A.: The anatomy of vampire implementing bottom-up procedures with code trees. JAR **15**(2), 237–265 (1995). https://doi.org/10.1007/BF00881918
62. Voronkov, A.: Algorithms, datastructures, and other issues in efficient automated deduction. In: IJCAR 2001. LNCS, vol. 2083, pp. 13–28. Springer (2001). https://doi.org/10.1007/3-540-45744-5_3
63. Weidenbach, C.: Automated reasoning building blocks. In: Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings. LNCS, vol. 9360, pp. 172–188. Springer (2015). https://doi.org/10.1007/978-3-319-23506-6_12