Subsumption Demodulation in First-Order Theorem Proving

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Jakob Rath, BSc
Matrikelnummer 0825352

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr.techn. Laura Kovács, MSc
Mitwirkung: Projektass. Dipl.-Ing. Bernhard Gleiss, BSc

Wien, 10. September 2019

Jakob Rath Laura Kovács

Subsumption Demodulation in First-Order Theorem Proving

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Jakob Rath, BSc
Registration Number 0825352

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr.techn. Laura Kovács, MSc
Assistance: Projektass. Dipl.-Ing. Bernhard Gleiss, BSc

Vienna, 10th September, 2019

Jakob Rath　　Laura Kovács

Erklärung zur Verfassung der Arbeit

Jakob Rath, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. September 2019

Jakob Rath

Acknowledgements

I am very grateful to my advisor Laura Kovács and her PhD student Bernhard Gleiss for the supervision of this work and for taking their time for numerous meetings over the past months. Their input and support during our discussions was essential in the preparation of this thesis.

Additionally, I am grateful for the financial support I have received due to a stipend organized by my advisor.

Many thanks also to Giles Reger, Martin Riener, Martin Suda, and Andrei Voronkov for various helpful comments on Vampire and its code. Martin Suda also answered questions concerning the experimental setup, and Giles Reger and Geoff Sutcliffe helped me access the relevant parts of StarExec, for which I thank them.

Finally, I would like to thank my parents, family and friends for their continued support and encouragement during my studies at TU Wien.

Kurzfassung

Die Sicherstellung der Korrektheit von Software ist eine wachsende Herausforderung in der modernen Gesellschaft. Die formale Software-Verifikation versucht, basierend auf logischem Schließen, verschiedene Anforderungen an die Korrektheit von Software mathematisch zu beweisen. Da jedoch das manuelle Beweisen solcher Aussagen mühsam und fehleranfällig ist, ist es erforderlich diese Aufgabe zu automatisieren, beispielsweise durch die Verwendung von automatischen Beweisern.

Moderne automatische Beweiser für Logik erster Stufe mit Gleichheit sind bereits leistungsstarke Werkzeuge des automatischen Schließens. Allerdings stoßen die verfügbaren Systeme bei Problemen, die aus der Software-Verifikation kommen, noch häufig an ihre Grenzen. Insbesondere kann eine große Anzahl von bedingten Gleichungen Schwierigkeiten bereiten.

Zur Verbesserung dieser Situation präsentieren wir in der vorliegenden Diplomarbeit eine neue Inferenzregel namens *Subsumption Demodulation* für Logik erster Stufe. Die grundlegende Idee von *Subsumption Demodulation* besteht darin, bedingte Gleichungen heranzuziehen um Terme in Klauseln, die die Bedingungen der Gleichung erfüllen, zu vereinfachen.

Wir implementieren und evaluieren *Forward Subsumption Demodulation* mit Hilfe des auf dem Superpositionskalkül basierenden automatischen Beweisers Vampire. Erste Ergebnisse zeigen, dass Beweiser für Logik erster Stufe mit *Forward Subsumption Demodulation* neue Probleme lösen können, die bisher nicht von existierenden automatischen Beweisern gelöst werden konnten.

Abstract

Ensuring correctness of software is becoming increasingly important in modern society, but still poses a difficult challenge. Reasoning-based software verification addresses this challenge by proving various requirements on software correctness. Performing such proofs manually is however tedious and error-prone, calling for the need to automate software correctness proofs, for example by using automated theorem provers.

State-of-the-art first-order theorem provers for first-order logic with equality are powerful automated reasoning tools. However, there are still limitations concerning problem encodings that arise from software verification. In particular, a large number of conditional equalities can be problematic.

In order to improve the situation, in this thesis we introduce a new inference rule in first-order theorem proving, called *subsumption demodulation*. The idea of subsumption demodulation is to use conditional equalities to simplify terms in clauses where the conditions are satisfied.

We implement and evaluate forward subsumption demodulation using the superposition-based theorem prover Vampire. Our initial results show that forward subsumption demodulation in first-order theorem proving can solve many new problems that could so far not be solved by existing automated reasoners.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Due to the rise of digitalization, an increasing number of important tasks in modern society are carried out by computer programs. Unfortunately, software errors happen on a daily basis in all kinds of computer systems and may lead to severe consequences such as high financial loss or even fatal accidents.

As a recent example for safety issues, multiple accidents caused by autonomous vehicles have been reported in 2018. In at least one case the accident was directly attributed to the car's software[1] (the system mistakenly ignored a pedestrian who was run over as a consequence).

Furthermore, security issues are becoming increasingly serious due to large parts of vital infrastructure being controlled by software. Of particular concern are large-scale electrical power outages that might be caused by attacks on the software controlling power plants or the power grid.

These issues impose a need for rigorous methods to ensure the correctness of software systems. Unfortunately, manual verification of software is tedious, error-prone and only possible for the smallest of programs. Compounding the problem, modern software systems are constantly increasing in size and complexity, requiring that verification can be done *automatically*.

The most prominent approaches such as model checking and deductive verification are logic-based and rely on automated theorem proving. In this paradigm, the behaviour of the program and its desired properties are expressed as logical formulas, often by automatic translation from the program's source code. This translation is done in such a

[1] https://arstechnica.com/tech-policy/2018/05/report-software-bug-led-to-death-in-ubers-self-driving-crash/, accessed on 2019-05-20

way that the validity of the formulas implies that the program behaves as desired. The reasoning task is then delegated to specialized reasoning tools such as SAT solvers, SMT solvers, or first-order theorem provers.

First-order logic is a precise and powerful specification language, and the standard for axiomatisation of theories in mathematics. The semantics of first-order logic are well understood, and it is often used as compilation target for more specialized formalisms.

The essential feature separating first-order logic from propositional logic are quantifiers over object variables. Quantifiers are crucial, for example, to express properties over the computer memory, an arbitrary number of loop executions, or recursion. For program verification in particular, also data types such as natural numbers, integers and arrays are useful for modelling the semantics of programming languages. *Because of this, the thesis at hand focuses on first-order properties that may contain quantifiers, equality, uninterpreted functions and predicates, as well as natural numbers, integers, and arrays.*

As a motivating example, consider the following program, which takes two integer arrays a and b as input and copies all non-negative values of b into a.

```
int i := 0
int j := 0
while i < length(b) do
   if b[i] >= 0 then
      a[j] := b[i]
      i := i + 1
      j := j + 1
   else
      i := i + 1
   end
end
```

This program can be translated into a formula in first-order logic according to an axiomatisation of the programming language semantics. Using this translation, we want to prove certain properties about the program's behaviour. For example, we might want to prove the following postconditions:

- $\forall k(0 \leq k \land k < j \to a[k] \geq 0)$, expressing that the first j elements of a are non-negative, or
- $\forall k(0 \leq k \land k < j \to \exists \ell(0 \leq \ell \land \ell < i \land a[k] = b[\ell]))$, expressing that each of the first j elements of a is equal to some element of b.

Most current state-of-the-art theorem provers for first-order logic with equality are based on saturation and the superposition calculus. These systems are very efficient for general first-order reasoning involving quantifiers and equality, but the situation is unsatisfactory when theories such as natural numbers or integers come into play, or when the problem encoding contains a large number of conditional equalities.

1.2 Problem Statement

A major obstacle in automated reasoning is the enormous size of the proof search space. One of the most important concepts to control the growth of the search space in saturation-based theorem proving are the so-called *simplification rules*. The benefit of such inference rules is that they do not add new formulas to the currently known set of formulas, but instead simplify one of these formulas.

For reasoning about equality, a crucial simplification rule is *demodulation*, which allows the prover to use unit equalities such as $l = r$ to simplify parts of another formula F, i.e., replacing $F[l]$ by $F[r]$, assuming r is "simpler" than l in some sense[2].

Consider the following example, where `S` stands for an arbitrary statement in the programming language at hand.

```
i := i + 1;
S;
```

$$\longrightarrow \quad \begin{array}{l} i_2 \simeq i_1 + 1 \\ [\![\texttt{S}]\!] \end{array}$$

The program snippet on the left may be translated into the two formulas on the right. The assignment `i := i + 1` from the source program is translated into the unit equality $i_2 \simeq i_1 + 1$ in the first-order formalization, where i_1 and i_2 represent the value of `i` before and after execution of the statement, respectively. The prover can use this equality to rewrite terms occuring in $[\![\texttt{S}]\!]$. Since assignments are common and essential in imperative programming, this indicates demodulation is very important for this kind of problems.

However, if the assignment appears inside a loop, it will be translated to a *guarded equality*, as shown by the (simplified) example below.

```
while P(i) do
   i := i + 1;
   S;
end
```

$$\longrightarrow \quad \begin{array}{l} P(i_1) \to i_2 \simeq i_1 + 1 \\ P(i_1) \to [\![\texttt{S}]\!] \end{array}$$

Here, `P` stands for some unary predicate in the programming language and P for its translation into first-order logic. Crucially, the same assignment `i := i + 1` as before is not translated into the conditional equality $P(i_1) \to i_2 \simeq i_1 + 1$. Since this is not a unit equality anymore, demodulation does not apply.

Problem statement. This thesis focuses on improving automated reasoning in first-order theories, in particular by advancing equality reasoning in first-order theorem proving by generalizing demodulation to non-unit equalities.

[2]The notation $F[\cdot]$ denotes a formula with a single *hole* in place of a term, and $F[t]$ denotes the formula with the hole filled by the term t.

1.3 Methodology

We re-use well-known results of superposition-based theorem proving, most notably the superposition calculus itself and the notions of redundancy, term/literal/clause orderings, and saturation. To ease inclusion into existing superposition-based provers, one of our goals was to develop improvements that do not require radical changes to the underlying superposition calculus.

To improve first-order reasoning in the presence of conditional equalities, we introduce the new inference rule *subsumption demodulation*, which generalizes demodulation to non-unit equalities by combining demodulation and subsumption. The basic idea of subsumption demodulation is to use conditional equalities to simplify terms in clauses where the conditions are satisfied.

Furthermore, we implement forward subsumption demodulation in the state-of-the-art superposition-based theorem prover Vampire. First, we present a reference implementation with a focus on understandability and minimal changes to the existing Vampire code base. Next, we discuss certain optimizations that lead to a second, optimized implementation.

Finally, we evaluate our implementation of forward subsumption demodulation on problems from the TPTP v7.2.0 problem library [Sut17] and the SMT-COMP 2019 competition [BDdM+13].

1.4 Contributions

The contributions of the thesis are as follows:

- Design of a new inference rule, called *subsumption demodulation*, in superposition-based theorem proving, as presented in Chapter 3.
- Prove soundness of subsumption demodulation and prove that it is simplification rule in Section 3.4.
- Implement forward subsumption demodulation in the theorem prover Vampire as described in Chapter 4.
- Evaluate and report on experimental results using forward subsumption demodulation in Chapter 5.

  Initial results show that our implementation solved 59 unique problems from the TPTP v7.2.0 problem library and 157 unique problems from the SMT-COMP 2019 Single Query Track[3]. From TPTP, our implementation was able to solve 6 problems that have not been solved before by another automated theorem prover, according to the TPTP solutions database.

[3]Configuration "fsdv2 combined", taking results for "unsat" and "sat" together.

CHAPTER 2 Preliminaries

In this chapter, we recall the basics of first-order logic and introduce some important concepts used in automated theorem proving.

2.1 First-Order Logic with Equality

For the purpose of this thesis, we are interested in standard first-order logic with equality. We assume basic knowledge of this logic as can be found in textbooks on the subject. In this section, we mostly fix the terminology and notation.

Let $\mathcal{V} = \{x, y, z, \dots\}$ be a countable set of variables. The language is determined by its signature $\Sigma = \langle \mathcal{F}, \mathcal{P} \rangle$, consisting of a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$. Every function symbol and predicate symbol is associated with an arity. Function symbols of arity 0 are also called constant symbols. We reserve $\simeq$ to denote the object-level equality and take care to distinguish it from the meta-level equality $=$. We consider $\simeq$ a part of the language, but not part of Σ.

Terms are constructed inductively from variables, constant symbols, and application of function symbols of non-zero arity. Let $\mathcal{T}$ denote the set of terms.

Atoms are built by applying predicate symbols or equality ($\simeq$) to the appropriate number of terms. *Formulas* are built inductively from atoms using the connectives $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), and $\rightarrow$ (implication) as well as the universal quantifier $\forall$ and the existential quantifier $\exists$.

Additionally, for all terms s, t we identify the equality atoms $s \simeq t$ and $t \simeq s$, which is justified because the interpretation of equality is symmetric.

We use the notation $F[t]$ to denote an occurrence of the term t in the formula F. For convenience, we regard any subsequent usage of $F[s]$ for some term s as the formula F where t has been replaced by s. This notation is slightly imprecise, but it can be made precise in the following way: define $F[\cdot]$ as a formula with a single *hole* in place of a term, and then $F[t]$ as $F[\cdot]$ with the hole filled by the term t. We allow similar notation for terms, atoms, literals, and clauses in place of the formula F.

The semantics of first-order logic with equality are defined in the usual way. An *interpretation structure* $\mathcal{M} = \langle \mathcal{U}, I, \alpha \rangle$ consists of a non-empty domain $\mathcal{U}$, an interpretation function I assigning meaning to the non-logical symbols in Σ, and a variable assignment $\alpha \colon \mathcal{V} \to \mathcal{U}$. The interpretation function I can be uniquely extended to terms. Truth of a formula F is then defined inductively and relative to an interpretation structure $\mathcal{M}$. We say $\mathcal{M}$ is a *model* of F and write $\mathcal{M} \models F$ if F is true in $\mathcal{M}$. A formula F is called *satisfiable* if a model of F exists, and *unsatisfiable* otherwise. A formula F is called *valid* if every interpretation structure is a model of F.

The following lemma is a well-known result:

Lemma 1 (Equivalent Replacement)**.** *Let $\mathcal{M}$ be an interpretation structure and s, t be terms such that $\mathcal{M} \models s \simeq t$, i.e., $I(s) = I(t)$.*

Then $\mathcal{M} \models F[s]$ iff $\mathcal{M} \models F[t]$ for any formula F.

A *literal* is either an atom or a negated atom. We define *clauses* as finite multisets of literals. Semantically, a clause is equivalent to (the universal closure of) the disjunction of its literals. As such, we may write the clause consisting of literals $L_1, L_2 \dots, L_n$ also in the form $L_1 \vee L_2 \vee \cdots \vee L_n$. The empty clause, denoted by $\square$, is false in every interpretation structure. A formula is said to be in *conjunctive normal form* if it is a conjunction of disjunctions of literals, i.e., it is essentially a set of clauses.

While most first-order theorem provers accept arbitrary first-order formulas as input, they usually work only with clauses internally. This approach is justified because every formula can be converted into a satisfiability-equivalent formula in conjunctive normal form in (almost) linear time [PG86].

A *substitution* $\theta = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ is a finite mapping from variables to terms. Given a term t, the term $t\theta$ is the result of simultaneously performing the replacements $x_1 \mapsto t_1, \dots, x_n \mapsto t_n$ on t. We use the same notation for applying substitutions to atoms, literals, formulas, and clauses.

We say a substitution θ is *more general* than a substitution σ if there exists a substitution ϱ such that for every term t we have $(t\theta)\varrho = t\sigma$. We say a substitution θ *unifies* the terms s and t if $s\theta = t\theta$. In this case, θ is a *unifier* of s and t. We further call θ a *most general unifier* (*mgu*), if it is more general than any other unifier of s and t. A well-known result is that two unifiable terms have an mgu that is unique up to renaming of variables. This concept can be extended naturally from terms to atoms and literals.

A term, atom, literal, formula, or clause is called *ground* if it does not contain any variables.

We assume a fixed language throughout the thesis and will not mention it explicitly unless necessary. Furthermore, we will only consider finite signatures. In the sequel, we denote by a, b, c, d, e constant symbols, f, g, h function symbols, P, Q, R predicate symbols, x, y, z variables, except where noted otherwise and possibly with additional subscripts and superscripts. As for variables in the meta-language, we will generally use l, r, s, t, u for terms, A, B for atoms, L, M for literals, C, D for clauses, and σ, θ for substitutions.

2.2 Clause Orderings

In this thesis, we are interested in computer-supported proving in first-order logic with equality. To automate reasoning in first-order logic, a clause ordering is a key ingredient. A more detailed introduction of orderings as defined in this section has been given by Baader and Nipkow in the context of term rewriting [BN98].

Let us consider a set X, the *domain* of the ordering. A *binary relation* over X is a subset of the cartesian product $X \times X$.

Definition 1 (Strict Partial Ordering)**.** A binary relation $>$ over X is called a *strict partial ordering* if it is

(i) irreflexive: $\forall x \in X : x \not> x$, and

(ii) transitive: $\forall x, y, z \in X : x > y \wedge y > z \implies x > z$. ■

Definition 2 (Total)**.** A strict partial ordering $>$ over X is called *total*, or a *strict total ordering*, if all elements of X are comparable, i.e., for all $x, y \in X$ with $x \neq y$ either $x > y$ or $y > x$ holds. ■

Definition 3 (Well-Founded)**.** We say a strict partial ordering $>$ over X is *well-founded* if there is no infinite descending sequence of elements, i.e., there is no infinite sequence $x_1, x_2, \ldots \in X$ such that $x_1 > x_2 > \cdots$. ■

2.2.1 Simplification Orderings

Definition 4 (Simplification Ordering)**.** We say a strict partial ordering $\succ$ on terms is a *simplification ordering* if it satisfies the following properties:

(i) Stability under substitutions: $s \succ t \implies s\theta \succ t\theta$ for all terms s, t and all substitutions θ.

(ii) Monotonicity: $s \succ t \implies u[s] \succ u[t]$ for all terms s, t, u.

(iii) Subterm property: $s \succ t$ whenever t is a proper subterm of s. ■

Every simplification ordering $\succ$ on terms over a finite signature is well-founded [BN98, Theorem 5.4.8].

Simplification orderings are often total on ground terms, but no simplification ordering can be total on non-ground terms. This is easy to see by considering the variable x and the ground term c: Assume $x \succ c$, then by stability under substitutions also $c \succ c$, which contradicts irreflexivity. Assume $c \succ x$, then again by stability under substitions also $c \succ f(c)$, which contradicts the subterm property.

A widely used simplification ordering is the Knuth-Bendix ordering (KBO) [KB83, BN98], which is parameterized by a weight function and a precedence ordering $\gg$ on the signature. We will only consider the case where the weight of a term t is simply the number of symbols in t, but in general this can be customized by using a different weight function. We omit a detailed definition of KBO in this thesis, but will nevertheless give some properties and examples. KBO is a simplification ordering and total on ground terms. It can be computed in linear time [Löc06].

Example 1. Assume a symbol precedence of $g \gg f \gg d \gg c$ and a weight of 1 for each symbol, and let $\succ_{kbo}$ denote the induced KBO.

- $g(c,d) \succ_{kbo} f(c)$ because $g(c,d)$ has weight 3 which is larger than weight 2 of $f(c)$.
- $g(d,d) \succ_{kbo} g(c,d)$ because $d \gg c$.
- $g(x,y) \not\succ_{kbo} g(x,x)$ because x appears twice on the right side but only once on the left side.
- $g(x,x) \not\succ_{kbo} g(x,y)$ because y appears on the right side but not on the left side. ■

2.2.2 Multiset Orderings

Formally, a multiset C of elements from a domain X is a map $C\colon X \to \mathbb{N}$, where $C(x)$ is the number of occurrences of $x \in X$ in C. A multiset C is *finite* if there are only finitely many $x \in X$ with $C(x) > 0$. Note that clauses are always finite multisets. We also write $x \in C$ when $C(x) > 0$.

Let C and D be multisets. We define the *multiset sum* $C \uplus D$ by $C \uplus D(x) = C(x) + D(x)$ for all $x \in X$. We write $C \subseteq_M D$ to express that C is a submultiset of D, i.e., that $C(x) \leq D(x)$ for all $x \in X$.

Definition 5 (Multiset Extension)**.** Given a strict partial ordering $>$ on X, its *multiset extension* $>_M$ is defined as follows. For multisets C and D over X, $C >_M D$ holds if and only if

1. $C \neq D$, and
2. for all $x \in X$ with $D(x) > C(x)$ there is some $y \in X$ with $C(y) > D(y)$ such that $y > x$. ■

As a different and maybe more intuitive characterization, $C >_M D$ holds iff we can reach D by (repeatedly) replacing an element of C by a finite number of smaller elements.

If $>$ is well-founded, then also its multiset extension $>_M$ is well-founded. If $>$ is total, then also its multiset extension $>_M$ is total.

We will require the following result.

Lemma 2. *Let C, D, E be multisets and let $>_M$ be the multiset extension of some strict partial ordering $>$. Then $C >_M D$ iff $C \uplus E >_M D \uplus E$.*

Proof. By definition, $C >_M D$ is equivalent to stating that (i) there is some $x \in X$ such that $C(x) \neq D(x)$ and (ii) for all $x \in X$ with $D(x) > C(x)$ there is some $y \in X$ with $C(y) > D(y)$ such that $y > x$.

By simple algebraic manipulation, these two statements are equivalent to (i') there is some $x \in X$ such that $C(x) + E(x) \neq D(x) + E(x)$ and (ii') for all $x \in X$ with $D(x) + E(x) > C(x) + E(x)$ there is some $y \in X$ with $C(y) + E(y) > D(y) + E(y)$ such that $y > x$.

Because $C \uplus E(x) = C(x) + E(x)$, the statements (i') and (ii') together are equivalent to $C \uplus E >_M D \uplus E$. □

2.2.3 Clause Orderings

Let $\succ_{\mathcal{T}}$ be a simplification ordering on terms. In this section, we discuss how to extend $\succ_{\mathcal{T}}$ to an ordering $\succ_{\mathcal{C}}$ on clauses.

In the first step, we define the ordering $\succ_{\mathcal{A}}$ on atoms. Equality atoms are compared via multiset extension: $s_1 \simeq s_2 \succ_{\mathcal{A}} t_1 \simeq t_2$ iff $\{s_1, s_2\} \succ'_{\mathcal{T}} \{t_1, t_2\}$, where $\succ'_{\mathcal{T}}$ is the multiset extension of $\succ_{\mathcal{T}}$.

To compare non-equality atoms, we introduce a new function symbol f_P for each $P \in \mathcal{P}$ and a new constant symbol c. Non-equality atoms $P(t_1, \ldots, t_n)$ are then, for the purpose of comparison, transformed into equalities $f_P(t_1, \ldots, t_n) \simeq c$ and treated like equality atoms.

The literal ordering $\succ_{\mathcal{L}}$ essentially interleaves negated atoms above their positive counterparts, i.e., $A \succ_{\mathcal{A}} B$ implies $\neg A \succ_{\mathcal{L}} A \succ_{\mathcal{L}} \neg B \succ_{\mathcal{L}} B$. Formally, $\succ_{\mathcal{L}}$ can be defined as multiset extension of $\succ_{\mathcal{A}}$ where negative literals $\neg A$ are identified with the multiset $\{A, A\}$ and positive literals A are identified with the multiset $\{A\}$.

Finally, as clauses are multisets of literals, we define the clause ordering $\succ_{\mathcal{C}}$ simply as the multiset extension of $\succ_{\mathcal{L}}$.

The clause ordering $\succ_{\mathcal{C}}$ is well-founded because it is essentially the three-fold multiset extension of the well founded ordering $\succ_{\mathcal{T}}$. Well-foundedness of $\succ_{\mathcal{C}}$ is essential for the completeness of superposition. Moreover, $\succ_{\mathcal{C}}$ inherits stability under substitutions from $\succ_{\mathcal{T}}$.

In the sequel, we will write $\succ$ for $\succ_{\mathcal{T}}$, $\succ_{\mathcal{A}}$, $\succ_{\mathcal{L}}$, and $\succ_{\mathcal{C}}$, unless the distinction is important and it is not clear from the context which one is meant. Furthermore, from now on we assume that the underlying simplification ordering is KBO and that all equality literals are smaller than all non-equality literals.

2.3 The Superposition Calculus

Most state-of-the-art automatic theorem provers for first-order logic with equality are based on the superposition calculus, a sound and refutationally complete inference system for this logic. Soundness means that for every inference in the calculus, the conclusion is logically entailed by the premises. Refutational completeness means that for each unsatisfiable set S of input clauses, there is a derivation of $\square$ from clauses in S using the inference rules of the calculus.

Unfortunately, there is no canonical treatment of the superposition calculus in the literature. In this thesis, we follow the presentation by Kovács and Voronkov [KV13]. Further details can be found in texts by Bachmair and Ganzinger [BG94, BG98, BG01] as well as Nieuwenhuis and Rubio [NR01].

In practice, it is essential to refine superposition with clause orderings and literal selection, resulting in *ordered superposition with selection*. The clause ordering $\succ$ has been introduced in the previous Section 2.2 and is used in the side conditions of inference rules to exclude unnecessary inferences.

Literal selection governs which literals may be used in inferences. We underline literals in the premises of inference rules to indicate that they must be selected. The literal selection must be *well-behaved* with respect to the ordering: either a negative literal or all maximal positive literals must be selected.

Let Sup denote the inference system formed by the following inference rules. Sup is a complete superposition inference system.

- Superposition into non-equalities:

$$\frac{\underline{l \simeq r} \vee C \qquad \underline{L[s]} \vee D}{(L[r] \vee C \vee D)\theta}$$

where

1. θ is the most general unifier of the terms l and s,
2. s is not a variable,
3. $r\theta \not\succeq l\theta$, and
4. L is not an equality literal.

- Superposition into equalities:

$$\frac{\underline{l \simeq r} \lor C \qquad \underline{t[s] \simeq u} \lor D}{(t[r] \simeq u \lor C \lor D)\theta} \qquad\qquad \frac{\underline{l \simeq r} \lor C \qquad \underline{t[s] \not\simeq u} \lor D}{(t[r] \not\simeq u \lor C \lor D)\theta}$$

  where

  1. θ is the most general unifier of the terms l and s,
  2. s is not a variable,
  3. $r\theta \not\succeq l\theta$, and
  4. $u\theta \not\succeq t\theta$.

- Equality Resolution:

$$\frac{\underline{s \not\simeq t} \lor C}{C\theta}$$

  where θ is the most general unifier of the terms s and t.

- Equality Factoring:

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

  where

  1. θ is the most general unifier of the terms s and s',
  2. $t\theta \not\succeq s\theta$, and
  3. $t'\theta \not\succeq t\theta$.

- Positive Factoring:

$$\frac{\underline{A} \lor \underline{B} \lor C}{(A \lor C)\theta}$$

  where θ is the most general unifier of the atoms A and B.

- Binary Resolution:

$$\frac{\underline{A} \lor C \qquad \underline{\neg B} \lor D}{(C \lor D)\theta}$$

  where θ is the most general unifier of the atoms A and B.

2.4 Redundancy

In this section, we define the concept of *redundancy*, which formalizes when clauses are unnecessary and can be deleted. Note that redundancy is always relative to a certain set of clauses. Informally, a clause is redundant in a set of clauses S if it is a logical consequence of smaller clauses that are already in S. Further details can be found in the literature [BG98].

Definition 6 (Ground Redundancy)**.** A ground clause C is *(ground-)redundant* in a set of ground clauses S if there are $C_1, \ldots, C_n \in S$ such that

- $C_1, \ldots, C_n \models C$, and
- $C \succ C_i$ for all $i \in \{1, \ldots, n\}$. ■

Since non-ground clauses represent all their ground instances, we lift redundancy in the following way. By itself, Definition 6 is not suitable for non-ground clauses because variables lead to incomparability, which would block many useful redundancies. However, note that the conditions from Definition 6 are sufficient for lifted redundancy due to the stability of simplification orderings under substitutions.

Definition 7 (Lifted Redundancy)**.** A clause C is *redundant* in a set of clauses S if all ground instances of C are (ground-)redundant w.r.t. ground instances of clauses in S. ■

Example 2. Consider the following superposition inference.

$$\frac{f(x) \simeq x \qquad f(c) \simeq g(c, y)}{c \simeq g(c, y)}$$

We show that the right premise $f(c) \simeq g(c, y)$ is redundant with respect to the conclusion $c \simeq g(c, y)$ and the left premise $f(x) \simeq x$.

- Obviously, the right premise is entailed by the two other clauses.
- The right premise is larger than the conclusion because of $f(c) \succ c$, which is due to the subterm property of simplification orderings. Because of stability under substitutions, also all ground instances of the right premise are larger than the corresponding instance of the conclusion.
- Every ground instance of the right premise is of the form $f(c) \simeq g(c, t)$ for some ground term t, and is larger than $f(c) \simeq c$, which is a ground instance of the left premise. This is due to $g(c, t) \succ c$, which again follows from the subterm property of simplification orderings.

  However, note that the non-ground clauses $f(x) \simeq x$ and $f(c) \simeq g(c, y)$ are incomparable because they contain different variables.

In general, premises of superposition inferences are not redundant. Here, we have seen an instance of the special case called *demodulation*, which will be introduced in Section 3.1. ■

Redundant clauses can be deleted from the search space without affecting completeness of the inference process.

Redundancy in full generality is undecidable. However, theorem provers implement several relatively cheap criteria to test for special cases of redundancy, aiming to prune the search space as much as possible. This leads to the notions of simplification rules and deletion rules.

Definition 8 (Simplification Rule)**.** An inference rule is a *simplification rule* if one of its premises becomes redundant after adding the conclusion to the search space. ■

The premise which becomes redundant is called the *main premise*, and all others are called *side premises*.

Because the main premise is redundant after the inference, it can then be deleted. Intuitively, a simplification rule simplifies its main premise to its conclusion by using additional knowledge from its side premises. It is called simplification because the clause becomes smaller with respect to the clause ordering.

Inference rules that are not simplification rules are called *generating rules*, because they generate new clauses without deleting previous ones and thus enlarge the search space. For example, all the rules in the base calculus Sup introduced in Section 2.3 are generating rules.

Example 3. *Duplicate literal elimination* is a simplification rule without side premises:

$$\frac{\cancel{L \lor L \lor C}}{L \lor C}$$

As a notational convention, we sometimes strike out the main premise in simplification rules to indicate that it can be deleted after application of the rule. ■

Definition 9 (Deletion Rule)**.** A *deletion rule* removes a redundant clause from the search space. ■

Example 4. All valid formulas, also called *tautologies*, are redundant in any search space because they are entailed by the empty set of clauses. *(Simple) tautology deletion* is a deletion rule which removes clauses C containing complementary literals (i.e., whenever C contains both L and $\neg L$ for some atom L). ■

2.5 Saturation up to Redundancy

The basic idea of saturation-based theorem proving is to keep applying possible inferences in a controlled manner until either the empty clause is derived (which means we have a refutation of the input set of clauses) or no more new clauses can be derived (which means the input set of clauses is satisfiable). In practice, resource exhaustion is a common third outcome of saturation, in which case the status of the input set of clauses remains undecided. Note that since the unsatisfiability of first-order formulas is undecidable, we cannot expect termination in general even if the theorem prover had unlimited resources at its disposal.

In practice, it is not enough to simply derive new clauses because the search space grows extremely fast. Instead, redundant clauses are deleted in an attempt to keep the search space small. To this end, powerful simplification and deletion rules are essential.

This notion of saturation is called *saturation up to redundancy* which we formalize now in the concept of inference processes. For a more in-depth discussion, we refer to the literature [KV13, BG01, NR01].

A finite or infinite sequence of sets of clauses $S_0, S_1, \dots$ is called a Sup-*process* if for each pair (S_i, S_{i+1}) either

1. $S_{i+1} = S_i \cup \{C\}$ and Sup contains an inference

$$\frac{C_1 \quad \cdots \quad C_n}{C}$$

   such that $\{C_1, \dots, C_n\} \subseteq S_i$, or

2. $S_{i+1} = S_i \setminus \{C\}$ and C is redundant in S_i.

In particular, we are interested in Sup-processes that preserve the refutational completeness of Sup. To this end, we call a clause C *persistent* in a Sup-process if there is some i such that $C \in S_j$ for all $j \geq i$, i.e., after a certain point, C will never be deleted. Furthermore, a Sup-process is called *fair* if every possible Sup-inference with persistent clauses as premises is performed at some point.

Theorem 1 (Completeness)**.** *Let S_0 be a set of clauses and $S_0, S_1, \dots$ be a fair* Sup-*process. Then S_0 is unsatisfiable if and only iff some S_i contains the empty clause.*

Fair inference processes are implemented by *saturation algorithms*. Common saturation algorithms are Otter [McC94], Discount [DKS97], and Limited Resource Strategy [RV03].

Concerning simplification rules during saturation, we can distinguish between *forward* and *backward* simplifications. During forward simplification, a newly derived clause is simplified using previously derived clauses as side clauses. Conversely, during backward simplification a newly derived clause is used as side clause to simplify previously derived clauses.

Here, the exact meaning of "newly derived" and "previously derived" depends on the concrete saturation algorithm. For example, so-called *given-clause algorithms* do not perform forward simplifications immediately whenever a new clause is derived but only when it is selected for further processing. The clause thus under consideration is called the *given clause*. Further details can be found in the literature indicated above.

CHAPTER 3

Subsumption Demodulation

In this chapter, we generalize demodulation to non-unit equalities.

3.1 Demodulation

Definition 10. *Demodulation* [KV13], also known as *rewriting by unit equalities*, is the simplification rule

$$\frac{s \simeq t \qquad \cancel{L[s\theta] \vee D}}{L[t\theta] \vee D}$$

where

1. $s\theta \succ t\theta$, and
2. $L[s\theta] \vee D \succ s\theta \simeq t\theta$. ■

Note that the second side condition $L[s\theta] \vee D \succ s\theta \simeq t\theta$ is often omitted in discussions of demodulation, but it is indeed necessary to ensure redundancy of the main premise.

Example 5. Consider the clauses $C_1 = f(f(x)) \simeq f(x)$ and $C_2 = P(f(f(c))) \vee Q(d)$. Using demodulation and the substitution $\theta = \{x \mapsto c\}$, C_2 is simplified into the clause $C_3 = P(f(c)) \vee Q(d)$:

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

The terms highlighted in blue indicate where the rewriting is performed. ■

3.2 Subsumption

Definition 11. Given clauses C and D, we say C *subsumes* D if there is some substitution θ such that $C\theta \subseteq_M D$. *Subsumption* [KV13] is the deletion rule which removes subsumed clauses D from the search space. ■

Example 6. The clause $C = P(x) \vee Q(f(x))$ subsumes the clause

$$D = P(f(c)) \vee P(g(c)) \vee Q(f(c)) \vee Q(f(g(c))) \vee R(y),$$

as witnessed by the substitution $\{x \mapsto g(c)\}$ which maps the literals of C to the literals of D highlighted in red. ■

Although checking whether a clause C subsumes another clause D is NP-complete, subsumption is very important for proof search. Even though the subsumption check often accounts for a large part of the total solver runtime (20 % and more is common), this investment pays off in many cases. Efficient algorithms and an optimized implementation are required to use subsumption effectively in practice.

3.3 Subsumption Demodulation

Subsumption resolution is based on the idea of re-using the subsumption check to find simplifying applications of binary resolution. In similar spirit, we define subsumption demodulation to generalize demodulation to non-unit equalities.

Definition 12. *Subsumption demodulation* is the inference rule

$$\frac{s \simeq t \vee C \qquad \cancel{L[s\theta] \vee C\theta \vee D}}{L[t\theta] \vee C\theta \vee D} \tag{3.1}$$

where

1. $s\theta \succ t\theta$, and
2. $L[s\theta] \vee C\theta \vee D \succ s\theta \simeq t\theta \vee C\theta$.

The equality $s \simeq t$ in the side premise is called the *rewriting equality*. ■

Detecting possible applications of subsumption demodulation involves selecting one equality of the side clause as rewriting equality and matching each of the remaining literals, denoted C in (3.1), to some literal in the main clause, indicated by $C\theta$ in (3.1). Note that this matching allows any instantiation of C to $C\theta$ via substitution θ, but we do *not* unify the two clauses, cf. Example 9 in the sequel. Furthermore, we need to find a term in the main premise outside $C\theta$ that can be rewritten by the rewriting equality.

We will now give some examples to illustrate subsumption demodulation. As in the earlier examples, we will indicate the rewritten terms in blue color and the subsumed literals (i.e., $C\theta$) in red color.

Example 7. Consider the two clauses

$$C_1 = f(g(x)) \simeq g(x) \vee Q(x) \vee R(y) \text{ and}$$
$$C_2 = P(f(g(c))) \vee Q(c) \vee Q(d) \vee R(f(g(d))),$$

and recall that equality literals are smaller than any non-equality literals in our literal ordering.

Then subsumption demodulation with the substitution $\theta = \{x \mapsto c, y \mapsto f(g(d))\}$ simplifies C_2 into $C_3 = P(g(c)) \vee Q(c) \vee Q(d) \vee R(f(g(d)))$:

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

The first side condition of subsumption demodulation, here $f(g(c)) \succ g(c)$, holds because of the subterm property of simplification orderings. The second side condition is satisfied due to $P(f(g(c))) \succ f(g(c) \simeq g(c)$, which is due to our assumption on the literal ordering.

Note that there is no instance of subsumption demodulation that rewrites the term $f(g(d))$ in C_2 using side premise C_1, even though we can find a match for the subsumption part using the substitution $\{x \mapsto d, y \mapsto f(g(d))\}$:

$$P(f(g(c))) \vee Q(c) \vee Q(d) \vee R(f(g(d))).$$

The rewriting may only occur in unmatched literals, here $P(f(g(c)))$ and $Q(c)$. ■

In Example 7, the rewriting equality is $f(g(x)) \simeq g(x)$. Due to the subterm property of simplification orderings, this equality is already oriented before applying any substitutions. We say the equality is *pre-oriented*. Note that in general, rewriting equalities are not pre-oriented.

Example 8. Consider the clause $C_1 = f(g(x)) \simeq g(y) \vee Q(x) \vee R(y)$. Only the first literal $f(g(x)) \simeq g(y)$ is a positive equality and as such eligible as rewriting equality. As the two terms $f(g(x))$ and $g(y)$ are incomparable due to occurrences of different variables, this equality is not pre-oriented.

Let us now examine how subsumption demodulation can be applied with side premise C_1 and different main premises.

- Consider the clause $C_2 = P(f(g(c))) \vee Q(c) \vee R(c)$ as main premise. The highlighted matching induces the substitution $\{x \mapsto c, y \mapsto c\}$. Under this substitution, the equality becomes $f(g(c)) \simeq g(c)$ and is thus oriented, enabling an application of subsumption demodulation.
- A different substitution can lead to rewriting in the other direction, illustrating why the rewriting equality at hand cannot be pre-oriented. This situation occurs, for instance, with the main premise $C_3 = P(g(f(g(c)))) \vee Q(c) \vee R(f(g(c)))$ and the substitution $\{x \mapsto c, y \mapsto f(g(c))\}$.

- On the other hand, using the clause $C_3 = P(f(g(c))) \vee Q(c) \vee R(z)$ as main premise leads to the substitution $\{x \mapsto c, y \mapsto z\}$. Here, the equality becomes $f(g(c)) \simeq g(z)$, which is not oriented. Even though we can find matching terms for rewriting, no application of subsumption demodulation is possible in this case. ■

Note that the substitution appearing in subsumption demodulation can only be used to instantiate the side premise, but not for unification with the main premise, as the following example illustrates.

Example 9. Consider the clauses

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

The only possible match for $Q(x,y)$ is $Q(c,d)$, inducing the substitution $\{x \mapsto c, y \mapsto d\}$. Similarly, matching $R(x,y)$ to its only possible match $R(c,z)$ induces $\{x \mapsto c, y \mapsto z\}$, which is incompatible to the previous substitution due to conflicting values of y. Because of this, subsumption demodulation is not applicable with premises C_1 and C_2.

In fact, subsumption demodulation would still be sound if we allowed unification (because we could view unification as preceding applications of an "instantiation rule" to both premises before applying subsumption demodulation, and instantiation is obviously sound). However, it would not be a simplification rule any more. ■

Note further that each literal in the side premise (except the rewriting equality) must be matched to a *different* occurrence of some literal in the main premise. For example, there is no instance of subsumption demodulation with side premise $f(x) \simeq x \vee Q(x) \vee Q(y)$ and main premise $P(f(c)) \vee Q(c)$, while one with main premise $P(f(c)) \vee Q(c) \vee Q(c)$ exists (however, the latter one would usually be simplified by duplicate literal elimination first).

3.4 Correctness of Subsumption Demodulation

A basic requirement of inference rules in saturation-based theorem proving is that they are *sound*. A rule is sound if its premises logically entail the conclusion of the rule.

Theorem 2. *Subsumption demodulation is sound.*

Proof. Let $\mathcal{M}$ be a model of the premises $s \simeq t \vee C$ and $L[s\theta] \vee C\theta \vee D$. We show that $\mathcal{M}$ also satisfies the conclusion $L[t\theta] \vee C\theta \vee D$.

In the case that $\mathcal{M}$ satisfies $C\theta \vee D$, the conclusion holds trivially in $\mathcal{M}$.

Otherwise, $\mathcal{M} \models L[s\theta]$ holds because of the second premise. Furthermore, we know $\mathcal{M} \not\models C\theta$. By instantiation we can derive $\mathcal{M} \models s\theta \simeq t\theta \vee C\theta$ from the first premise (because free variables are implicitly universally quantified), and thus $\mathcal{M} \models s\theta \simeq t\theta$ must hold. With Lemma 1 we infer $\mathcal{M} \models L[t\theta]$. So the conclusion is true in $\mathcal{M}$ also in the second case. □

Theorem 3. *Subsumption demodulation is a simplification rule.*

Proof. We have to show that the main premise is redundant given the side premise and conclusion. To this end, we show that

1. $s \simeq t \vee C, L[t\theta] \vee C\theta \vee D \models L[s\theta] \vee C\theta \vee D$,
2. $L[s\theta] \vee C\theta \vee D \succ L[t\theta] \vee C\theta \vee D$, and
3. $L[s\theta] \vee C\theta \vee D \succ s\theta \simeq t\theta \vee C\theta$.

These three statements together imply (lifted) redundancy because the clause ordering is stable under substitutions.

The proof of the first part is analogous to the proof of Theorem 2 (with s and t swapped, which is irrelevant because equality is symmetric).

For the second part, we can infer $L[s\theta] \succ L[t\theta]$ from side condition 1 by monotonicity of the simplification ordering $\succ$. By Lemma 2 this is equivalent to the statement we want to prove.

The third statement is given by side condition 2. □

3.5 Refining the Redundancy Condition

Side condition 2 of subsumption demodulation is important to ensure redundancy of the main premise after the inference. However, directly checking the ordering between general clauses is computationally expensive, and not necessary in this case. The following result shows that we simply need to cover the equality by some larger umatched literal, because we can cancel $C\theta$ when checking the second side condition:

Lemma 3. $L[s\theta] \vee C\theta \vee D \succ s\theta \simeq t\theta \vee C\theta$ *if and only if* $L[s\theta] \vee D \succ s\theta \simeq t\theta$.

Proof. Recall that our clause ordering $\succ$ is the multiset extension of some literal ordering. As such, Lemma 2 allows us to cancel $C\theta$ from both sides of the comparison. □

This means redundancy in subsumption demodulation can be checked similarly as in demodulation.

3.6 Impact on Saturation

Example 10. Assume the search space contains, among others, the clauses

$$D_1 = a \simeq b \vee e \simeq f \vee P \text{ and}$$
$$D_2 = b \simeq c \vee d \simeq e \vee P,$$

where a, b, c, d, e, and f are constant symbols. Furthermore, assume the term ordering satisfies $f \succ e \succ d \succ c \succ b \succ a$.

If we now derive the new clause $C = b \simeq c \vee e \simeq f \vee P \vee Q$, there are two possible applications of subsumption demodulation to simplify C using either D_1 or D_2:

- Using D_1, we choose $a \simeq b$ as the rewriting equality, and C is simplified into $C_1 = a \simeq c \vee e \simeq f \vee P \vee Q$.
- Using D_2, we choose $d \simeq e$ as the rewriting equality, and C is simplified into $C_2 = b \simeq c \vee d \simeq f \vee P \vee Q$.

Note that subsumption demodulation is not applicable to C_1 nor C_2 using either of D_1 or D_2, meaning that each of the two applications blocks the other one. ■

Corollary 1. *Subsumption demodulation is not confluent.*

Note that while this result is unfortunate from a theoretical viewpoint, confluence is not necessary for simplification rules. It is ultimately not clear how this property affects saturation in practice, although it may make proof search less stable (if a proof can be found for one branch but not the other).

Besides avoiding search space blow-up, there is another reason why we set up subsumption demodulation as a simplification rule instead of a generating rule. In general, having stronger simplification rules in the inference system means we have a higher chance of achieving saturation for satisfiable instances, because we are able to delete more clauses. Even more importantly, we can not only delete more clauses but also avoid generating clauses that are derivable from the deleted ones. If we view the inference process as a tree, it means we cut off a whole (possibly infinite) branch.

Indeed, our experiments, as described in Section 5.1, confirm that we are able to prove more satisfiable instances with forward subsumption demodulation enabled.

CHAPTER 4

Implementation

We have implemented *forward subsumption demodulation* (FSD), i.e., the forward direction of subsumption demodulation, in the state-of-the-art superposition-based theorem prover VAMPIRE. The source code is available at https://github.com/vprover/vampire/tree/fsd.

In this chapter, we first explain the existing implementation of forward subsumption in VAMPIRE and then describe the extension to forward subsumption demodulation.

4.1 Subsumption and Multiliteral Matching

The implementation of forward subsumption in VAMPIRE is separated into subsumption by unit clauses and subsumption by non-unit clauses. The actual implementation of forward subsumption also takes care of extra bookkeeping for forward subsumption resolution, but we will not discuss this further as it is not relevant to this thesis.

Subsumption by unit clauses is done first because very efficient checks can be exploited in this special case. For our purposes, however, we are interested in the general implementation of subsumption by non-unit clauses. To check whether the given clause D is subsumed, VAMPIRE searches for a previously derived clause C such that $C\theta \subseteq_M D$ for some substitution θ. This part is implemented in two stages.

4.1.1 Subsumption Index

First, VAMPIRE searches for candidate clauses that might subsume D. Concretely, for each literal of the given clause, all clauses containing a generalization of this literal are retrieved from the *subsumption index*.

Term (or literal) indexes in automated theorem proving are data structures similar to associative arrays where data, usually clauses, are indexed by terms (or literals).

The purpose of these data structures is to allow efficient lookups modulo unification, instantiation, or generalization.

The subsumption index in Vampire stores the candidate clauses for subsumption. Each clause is indexed by exactly one of its literals. In principle, any literal of the clause can be chosen. In order to reduce the number of retrieved candidates as much as possible, however, the literal is chosen such that it maximizes a certain heuristic.

The heuristic counts the number of symbols and subtracts the number of distinct variables, which is equal to the number of non-variable symbols plus the number of *repeated* variable occurrences. Intuitively, this value is the number of symbols that induce constraints for matching, because variables only induce constraints for instantiation on their repeated occurrences. Maximizing this value means the most restrictive literal is chosen, which in turn reduces the number of retrieved candidate clauses.

Since the subsumption index is not a perfect index (i.e., it may retrieve non-subsuming clauses), additional checks on the retrieved clauses have to be performed.

4.1.2 Multi-literal matching

Given a candidate clause C, Vampire tries to find a substitution θ that transforms C into a submultiset of the given clause D. This process is called *multi-literal matching*, and is implemented in Vampire in a component called the MLMatcher.

Note that multi-literal matching is an NP-complete problem. Intuitively, this is because the literal correspondences between the two clauses are not obvious in general. The MLMatcher implements backtracking to cover all possible matchings, with numerous optimizations to prune infeasible branches early. Because subsumption is important and takes a large fraction of the solver runtime, the MLMatcher is a heavily optimized component of Vampire.

The input to the MLMatcher is essentially an array of *base literals*, along with one array of *alternatives* per base literal that contains the possible instances that the corresponding base literal may be matched to. The output is a single boolean value, which is `true` if and only if all base literals can be instantiated to one of their alternatives using a uniform substitution. There are some additional technicalities to ensure that $C\theta$ is a sub*multi*set of D and not merely a subset, which we omit here.

Example 11. Consider the clauses

$$\begin{aligned} C &= P(x) \lor Q(x), \\ D &= P(c) \lor P(d) \lor Q(d) \lor Q(f(d)). \end{aligned}$$

We want to check whether C subsumes D.

The input to the MLMatcher consists of the base literal $P(x)$ with alternatives $P(c)$ and $P(d)$, and the base literal $Q(x)$ with alternatives $Q(d)$ and $Q(f(d))$.

Note that the first choice of matching $P(x)$ to $P(c)$ induces the substitution $\{x \mapsto c\}$ which is incompatible with all possible choices for $Q(x)$. The MLMatcher must backtrack and match $P(x)$ to the next alternative $P(d)$, inducing the substitution $\{x \mapsto d\}$ which allows matching $Q(x)$ to $Q(d)$. We found a match, which means C indeed subsumes D. ■

4.2 Reference Implementation of FSD (“FSD v1”)

In a first attempt, we implemented forward subsumption demodulation in a mostly straightforward way based on forward subsumption. The goal of this implementation was to quickly obtain a proof of concept implementation that is easy to understand and audit. This implies keeping changes to highly optimized core algorithms in Vampire minimal.

The algorithm proceeds according to the following steps:

1. Retrieve candidate clauses (i.e., potential side clauses) from an appropriate index like in forward subsumption.
2. Select a positive equality in the candidate clause as rewriting equality.
3. Construct input to MLMatcher, excluding the selected equality.
4. When a match has been found, retrieve the substitution and the matched literals from the MLMatcher.
5. Find a term in the non-matched part of the given clause that matches a term of the rewriting equality. The substitution may be extended in this step.
6. Check whether the rewriting equality is oriented after substitution.
7. Check whether redundancy holds. Due to Lemma 3, the same optimizations as in demodulation apply.
8. Build the simplified clause and return it.

If one of these steps fails, the algorithm will backtrack to earlier steps and try alternative choices, as long as any remain.

Note that the substitution θ is built in two stages: first we get a partial substitution from the MLMatcher, then it (possibly) gets extended when matching terms of the rewriting equality. The following example illustrates this situation.

Example 12. Consider the following FSD inference:

$$\frac{f(x,y) \simeq y \lor Q(x) \qquad P(f(c,d)) \lor Q(c)}{P(d) \lor Q(c)}$$

We first select the only possible rewriting equality $f(x, y) \simeq y$. Querying the MLMatcher then reveals the (partial) substitution $\theta' = \{x \mapsto c\}$. We arrive at the final substitution $\theta = \{x \mapsto c, y \mapsto d\}$ only when we match the left-hand side $f(c, y)$ of the (partially substituted) rewriting equality to the rewritable term $f(c, d)$.

Note that if there is a variable in the side premise that is not bound even in the final substitution (e.g., if we replace the equality in the inference above by $f(x, y) \simeq z$), this variable can only occur in the right-hand side of the rewriting equality. In this case, no subsumption demodulation is possible because the rewriting equality will not be oriented after the final substitution. ■

4.2.1 Index adjustments

The original subsumption index is not a good fit for subsumption demodulation because the rewriting equality does not participate in the multi-literal matching. We might lose potential FSD applications when a positive equality is chosen for indexing.

Example 13. Consider the clause $C = f(f(x)) \simeq f(x) \vee P(x)$. The heuristic assigns a value of 4 to the equality literal and a value of 1 to the other literal, thus C is indexed by $f(f(x)) \simeq f(x)$.

There is an FSD application with side premise C and main premise $D = P(c) \vee Q(f(f(c)))$ resulting in the conclusion $P(c) \vee Q(f(c))$.

However, using the original subsumption index we cannot retrieve C as candidate clause with the given clause D. ■

To address this issue, we added a new index called the *forward subsumption demodulation index* (FSD index). The basic idea of the FSD index is the same as the subsumption index, however, it takes into account not only the “best” literal (best according to the heuristic) but also the “second best”.

If the best literal in a clause C is a positive equality but the second best is not, C is indexed by the second best literal, and vice versa. If both the best and second best literal are positive equalities, C is indexed by both of them.

Furthermore, because the index is now exclusively used by FSD, the index only needs to keep track of clauses that contain at least one positive equality.

Another possibility to solve this issue would be to adjust the subsumption index by adding additional entries in the problematic case, i.e., additionally index a clause by its second best literal when the best literal is a positive equality. However, this might negatively impact forward subsumption and runs contrary to our goal of minimally modifying existing VAMPIRE code. Moreover, the impact of adding an additional index is small from a performance perspective, because index maintenance in VAMPIRE is very efficient.

4.2.2 MLMatcher modifications

In order to support the algorithm described in Section 4.2, the MLMatcher had to be extended with functions to retrieve information about the concrete matching. In particular, the algorithm requires the substitution and the set of matched literals. This data was already stored for internal bookkeeping purposes but was not accessible from outside. However, no substantial modifications of the MLMatcher were necessary.

4.3 Optimized Implementation of FSD (“FSD v2”)

We have adapted the implementation to interleave selection of the rewriting equality with multi-literal matching, which reduces the overhead of calling the MLMatcher multiple times. This required extensive changes to the MLMatcher, but also enabled some additional smaller optimizations in the matching algorithm. We refer to the modified variant as MLMatcher2.

The modified algorithm proceeds as follows:

1. Retrieve candidate clauses (i.e., potential side clauses) from the FSD index.
2. Construct input to MLMatcher2. This input now includes all positive equalities and depends only on the candidate clause.
3. When a successful match has been found, retrieve the selected rewriting equality and the current substitution.
4. Find a term in the non-matched part of the given clause that matches a term of the rewriting equality. The substitution may be extended in this step.
5. Check whether the rewriting equality is oriented after substitution.
6. Check whether redundancy holds. Due to Lemma 3, the same optimizations as in demodulation apply.
7. Build the simplified clause and return it.

As before, if one of these steps fails, the algorithm will backtrack to earlier steps and try alternative choices, as long as any remain.

4.3.1 Extended Multiliteral Matching

Instead of first selecting a rewriting equality and then performing multi-literal matching on the remaining literals, the MLMatcher2 allows interleaving both actions. Conceptually, for each base literal that is a positive equality, the MLMatcher2 implicitly adds an additional alternative that means “this base literal is selected as rewriting equality”.

At most one of these implicit alternatives may be chosen at any time. As the rewriting equality currently does not participate in building the substitution, the MLMatcher2 takes care to ignore the implicit alternatives when checking compatibility of partial substitutions. We have ideas how to extend the MLMatcher2 to also select a concrete rewritable term along with the rewriting equality, however, this requires another substantial implementation effort which we leave for future work.

Note that, in principle, the MLMatcher2 is also able to report subsumption if all literals can be matched to instances without using the implicit alternatives. Support for this result has been implemented in FSD v2, but in the current setting it does not occur because forward subsumption is still performed separately by the existing implementation. In future work, we want to study how much can be gained by merging the implementations of forward subsumption and FSD.

4.4 Possibility of Multiple Matches

In general, a multi-literal match problem may have more than one solution. Moreover, solutions concerning FSD may differ in which equality has been selected as rewriting equality, cf. Example 10. We give further examples where backtracking occurs.

Example 14. Consider the two clauses

$$\begin{aligned} C_1 &= f(f(x)) \simeq f(x) \lor Q(x) \text{ and} \\ C_2 &= P(f(f(d))) \lor Q(c) \lor Q(d), \end{aligned}$$

where C_1 is intended to be the side premise and C_2 the main premise. There is only one choice for the rewriting equality, and two choices of literals to match to $Q(x)$.

Assume the algorithm first returns the solution matching $Q(x)$ to $Q(c)$, inducing the substitution $\{x \mapsto c\}$. Since there is no term $f(f(c))$ in the main premise, FSD cannot be applied and the algorithm needs to backtrack.

The algorithm will then find the next solution by matching $Q(x)$ to $Q(d)$ with the substitution $\{x \mapsto d\}$. Now there is a suitable term in C_2, leading to the simplified clause $P(f(d)) \lor Q(c) \lor Q(d)$ that is returned as result. ■

Example 15. Consider the two clauses

$$\begin{aligned} C_1 &= P(x_1) \lor P(x_2) \lor \cdots \lor P(x_n) \lor f(x_1, \ldots, x_n) = x_1 \text{ and} \\ C_2 &= P(c_1) \lor P(c_2) \lor \cdots \lor P(c_n) \lor Q(f(d_1, \ldots, d_n)), \end{aligned}$$

for $n > 0$.

There are $n!$ possibilities to match the literals $P(x_1), \ldots, P(x_n)$ to $P(c_1), \ldots, P(c_n)$, one for each permutation of the n literals. The algorithm will examine all $n!$ matches, failing to find a rewritable term for any of the matches.

Note that even if rewriting were possible (e.g., if we add the literal $Q(f(c_1, \ldots, c_n))$ to C_2), it may still happen that the correct match is found last in which case all $n!$ matches will be considered. ■

While extremely bad cases such as described in Example 15 are rare in practice, they do occur, as shown by the experimental results presented in Section 5.2. However, those results also show that most applications of FSD are found within the early matches.

This observation leads to a simple practical approach to avoid situations as in Example 15, which is to place an upper limit on the number of matches that are examined, and give up if this upper limit is reached. We have added such an upper limit to our implementation. The limit is user-configurable through the option `-fsdmm`, whose usage is described in Section 4.5.

The implementation of FSD is "complete" in the sense that it always finds an FSD application if one exists (under slight restrictions, cf. Section 4.6), provided that the upper limit of matches is disabled.

4.5 Overview of New Options

This section gives a brief explanation of the new Vampire options that control the FSD implementations.

- `-fsd`, short for `--forward_subsumption_demodulation`

  Value: `off`, `v1`, or `v2`

  Default value: `off`

  The option `-fsd` determines which FSD implementation is active, if any.

- `-fsdmm`, short for `--forward_subsumption_demodulation_max_matches`

  Value: a non-negative integer

  Default value: `1`

  The option `-fsdmm` sets the upper limit on the number of matches to examine for each candidate side clause as discussed in Section 4.4. The value `0` is special and means that no upper limit should be enforced.

Currently, all FSD-related options are still marked as being experimental. To see these options in the built-in help text of Vampire, one should use the command `vampire --show_options on --show_experimental_options on`.

4.6 Limitations

There are certain instances of FSD that our implementation does not detect, because the algorithm only finds substitutions that are directly induced by matching.

Example 16. Assume that $f \gg g$ in the symbol precedence, and consider instances of FSD with side premise $C_1 = f(x) \simeq g(y) \vee Q(x)$ and main premise $C_2 = P(f(c)) \vee Q(c)$.

Our implementation finds the substitution $\theta = \{x \mapsto c\}$. Since the rewriting equality is unoriented under θ, no inference is performed.

However, an FSD inference is indeed possible with the substitution $\sigma = \{x \mapsto c, y \mapsto c\}$, because $f(c) \succ g(c)$ due to our assumption on symbol precedence.

The algorithm does not find this application of FSD, because the matching algorithms find the most general substitution and there is no direct reason to substitute anything for y. It is not obvious how to "guess" substitutions that lead to oriented equalities in cases such as above. Note that this issue is not specific to subsumption demodulation, but already appears for demodulation. In fact, Vampire will not even add clauses like $f(x) \simeq g(y)$ to the demodulation index.

Finally, we argue that this situation is not relevant in practice. This issue occurs when the clause contains a variable that only appears in one side of the rewriting equality and nowhere else in the clause. Because clauses are universally quantified, such equalities are far too general for most useful problem encodings. ■

CHAPTER 5

Experiments

5.1 Evaluation of Forward Subsumption Demodulation

We have evaluated the implementation of forward subsumption demodulation (FSD) on benchmark problems from the TPTP problem library version 7.2.0[1] [Sut17] and from the SMT-COMP 2019 Single Query Track[2] [BDdM+13]. In this chapter, we mainly focus on the evaluation of FSD v2 as described in Section 4.3.

The evaluation compares the version of Vampire with FSD (branch `fsd`[3]) against the standard version of Vampire (branch `master`[4]) as of July 2019. The experiments were conducted on the StarExec cluster [SST14]. For both branches, Vampire has been compiled with the command `make vampire_z3_rel_static`. To analyse the results, we have used the relational database SQLite and various Haskell libraries (cassava, sqlite-simple, ...). The plots have been generated with the Haskell library Chart.

The following options were given to Vampire:

- Portfolio mode (`--mode portfolio`)
- Timeout of 300 seconds (`-t 300`)
- Collect and print statistics (`-stat full` and `-tstat on`)
- Do no print proofs (`--proof off`)

  The reason for this option is to avoid exceeding the storage quota on StarExec.

[1] Website of the TPTP: http://www.tptp.org/
[2] Website of SMT-COMP 2019: https://smt-comp.github.io/2019/
[3] https://github.com/vprover/vampire/tree/fsd
[4] https://github.com/vprover/vampire/tree/master

- For SMT-COMP benchmarks only, set the appropriate schedule for the portfolio mode (`--schedule smtcomp`)
- Configurations with FSD enabled are named after the scheme fsdv$x$-mm$y$, where x is the version of FSD and y is the maximum number of multi-literal matches to examine. To enable these settings in portfolio mode (which does not know about FSD yet), we have used the option `--forced_options fsd=v`x`:fsdmm=`y.

We have used the portfolio mode for the main experiments because it is the most useful mode in practice. However, the portfolio mode is tweaked to deliver the best results for the default set of Vampire options. Enabling FSD via `--forced_options` instructs Vampire to enable FSD in all strategies of the portfolio mode, which is not optimal. Unfortunately, re-training the portfolio mode to take into account FSD is a big effort and relies on closed-source software not availabe to the author.

Because of these caveats, it is to be expected that the overall results are skewed in favour of standard Vampire (i.e., configuration `master`). Nonetheless, some problems are uniquely solved by configurations with FSD enabled, even including some problems that have not been solved before by any other automatic theorem provers (according to the TPTP solutions database).

5.1.1 Results for the TPTP v7.2.0

The TPTP v7.2.0 contains a total of 22,026 problems, of which 17,573 problems can be parsed (and worked on) by Vampire as of July 2019.

The overall number of solved problems is given in Table 5.1 and is mostly as expected. Note that while the FSD configurations solve less UNSAT instances overall, they can solve more SAT instances even though the portfolio mode is not optimized for FSD at all.

Also for UNSAT instances, FSD can solve some problems that `master` cannot solve. This can be seen in Table 5.2, which shows the number of problems uniquely solved by each configuration.

Notably, among the unique problems are some that have not been solved by automated theorem provers before, according to the TPTP solutions library as of September 2019. There are four unsatisfiable problems that have been solved by a configuration with FSD

Configuration	UNSAT	SAT	Unknown	Total
fsdv1-mm1	12,049	1,185	4,339	17,573
fsdv2-mm1	12,014	1,186	4,373	17,573
fsdv2-mm3	12,017	1,181	4,375	17,573
fsdv2-mm0	11,995	1,184	4,394	17,573
master	12,189	1,178	4,206	17,573

Table 5.1: Number of solved problems from the TPTP v7.2.0

	UNSAT		SAT	
Configuration	Unique FSD	Unique master	Unique FSD	Unique master
fsdv2-mm1	37	212	10	2
fsdv2-mm3	39	211	8	5
fsdv2-mm0	35	229	10	4
fsdv2 combined	47	172	12	0
fsdv1-mm1	39	179	10	3

Table 5.2: Number of uniquely solved problems from the TPTP v7.2.0

enabled and that do not yet have a solution registered on TPTP, namely `SEU054+1`, `SWC263+1`, `SWC199+1`, and `GRP634+3`. As for satisfiable problems, configurations with FSD enabled were able to solve the problems `SWV549-1.010`, `SWV531-1.007`, `SWV953-1`. For the former two, no solution is registered on TPTP, and for the latter one, a solution by E 2.4 was only registered in July 2019.

Note that some of these problems had been partially solved before, e.g., the three satisfiable problems have been shown to be finitely unsatisfiable by Infinox. However, this is no complete solution of the problem as it merely states that no finite model of the formula exists.

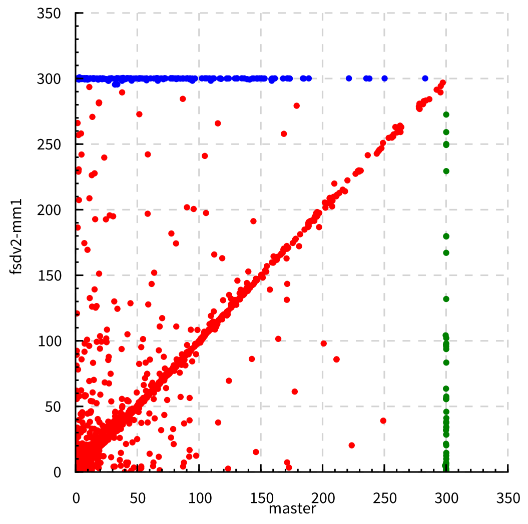
In Figure 5.1, some comparisons of different configurations have been visualized. Figure 5.1d compares runs for different values of option `-fsdmm`. In all plots it can be seen that each configuration can solve some problems that the other cannot solve.

5.1.2 Results for the SMT-COMP 2019 Single Query Track

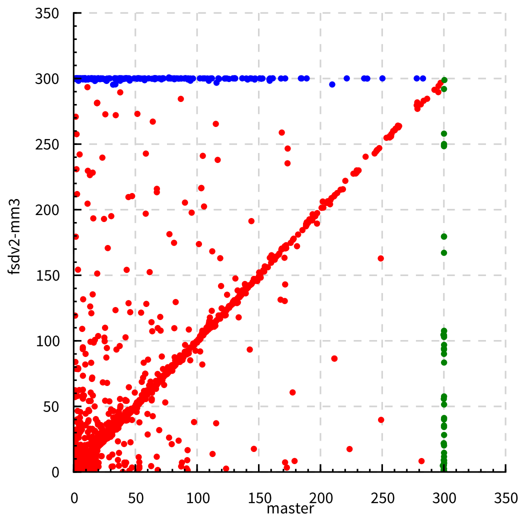
We performed the same experiments for 19,116 benchmark problems selected from the SMT-COMP 2019 Single Query Track. We have chosen the same benchmark problems that Vampire was competing for at the SMT-COMP 2019 competition [RSV+19]. Vampire can work with all SMT-LIB logics except bit vectors, floating points, and strings. Furthermore, the quantifier-free variants have been omitted because Vampire has no special support for ground reasoning.

The overall number of solved problems is given in Table 5.3, and the numbers of uniquely solved problems per configuration are listed in Table 5.4. Table 5.5 gives the overall number of solved problems grouped by SMT-LIB logic. Furthermore, Table 5.6 shows the number of uniquely solved unsatisfiable problems grouped by SMT-LIB logic. The uniquely solved satisfiable problems are all in the logic UF, which is expected because Vampire is refutationally complete only for first-order logic and uses incomplete theory axiomatizations for reasoning in first-order theories.

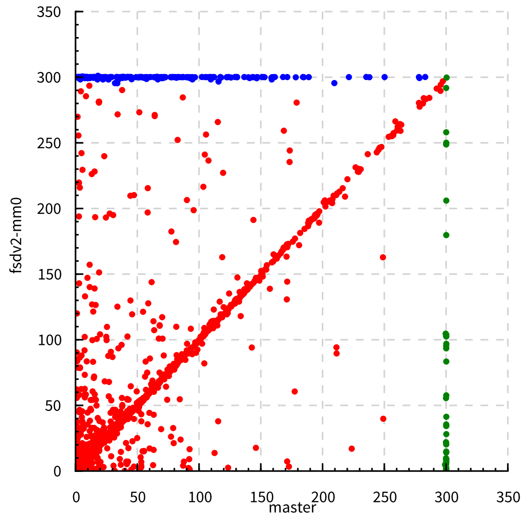
Some comparisons of different configurations have been visualized in Figure 5.2.



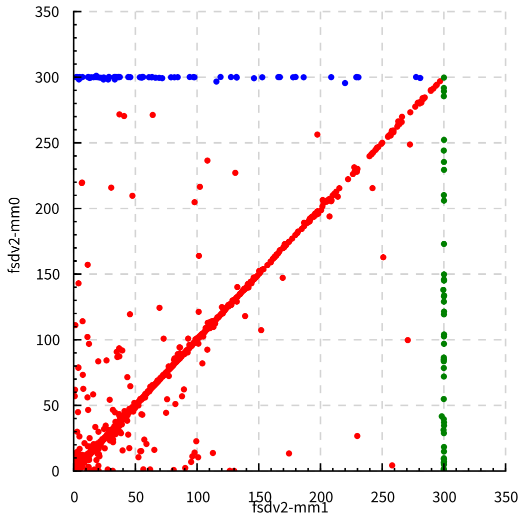
(a) `master` vs. `fsdv2-mm1`



(b) `master` vs. `fsdv2-mm3`



(c) `master` vs. `fsdv2-mm0`



(d) `fsdv2-mm1` vs. `fsdv2-mm0`

Figure 5.1: Comparing solver runtimes of different configurations on TPTP problems. Problems that can be solved by both configurations are indicated in red, while problems that can be solved by only one configuration are indicated in blue or green.

Configuration	UNSAT	SAT	Unknown	Total
fsdv1-mm1	7,792	541	10,783	19,116
fsdv2-mm1	7,785	540	10,791	19,116
fsdv2-mm3	7,799	541	10,776	19,116
fsdv2-mm0	7,788	541	10,787	19,116
master	7,826	539	10,751	19,116

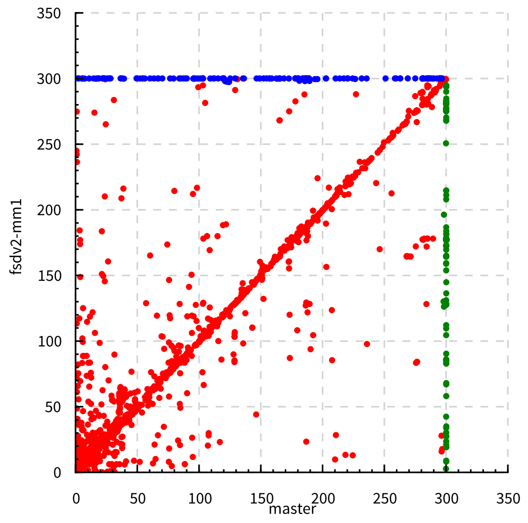
Table 5.3: Number of solved problems from SMT-COMP 2019

	UNSAT		SAT	
Configuration	Unique FSD	Unique master	Unique FSD	Unique master
fsdv2-mm1	127	168	2	1
fsdv2-mm3	138	165	2	0
fsdv2-mm0	130	168	2	0
fsdv2 combined	155	143	2	0
fsdv1-mm1	124	158	2	0

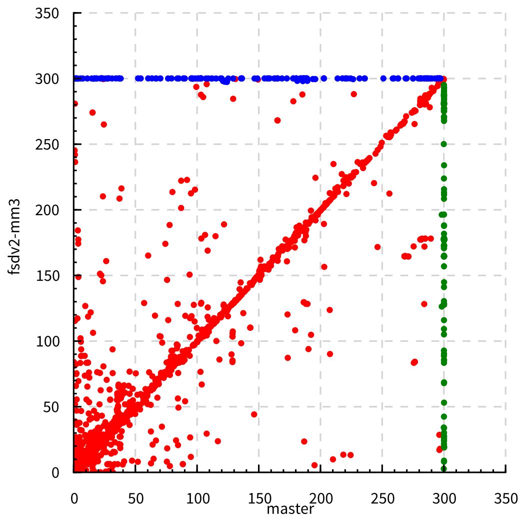
Table 5.4: Number of uniquely solved problems from SMT-COMP 2019

		fsdv1-mm1		fsdv2-mm1		fsdv2-mm3		fsdv2-mm0		master	
Logic	Total	unsat	sat	unsat	sat	unsat	sat	unsat	sat	unsat	sat
ALIA	19	9	–	9	–	9	–	9	–	9	–
AUFDTLIA	275	177	–	177	–	177	–	176	–	178	–
AUFLIA	1,638	1,267	99	1,267	99	1,269	99	1,271	99	1,274	99
AUFLIRA	1,683	1,551	–	1,549	–	1,552	–	1,550	–	1,552	–
AUFNIA	3	–	–	–	–	–	–	–	–	–	–
AUFNIRA	300	56	–	55	–	55	–	54	–	58	–
LIA	300	140	5	140	5	140	5	140	5	140	5
LRA	1,003	177	–	177	–	177	–	177	–	177	–
NIA	11	–	–	–	–	–	–	–	–	–	–
NRA	93	81	–	81	–	81	–	81	–	81	–
UF	2,816	627	437	625	436	623	437	624	437	649	435
UFDT	1,547	340	–	335	–	334	–	332	–	348	–
UFDTLIA	299	65	–	64	–	63	–	63	–	66	–
UFDTNIA	1	1	–	1	–	1	–	1	–	1	–
UFIDL	20	7	–	7	–	7	–	7	–	7	–
UFLIA	2,848	1,398	–	1,383	–	1,380	–	1,377	–	1,393	–
UFLRA	7	2	–	2	–	2	–	2	–	2	–
UFNIA	6,253	1,894	–	1,913	–	1,929	–	1,924	–	1,891	–
All logics	19,116	7,792	541	7,785	540	7,799	541	7,788	541	7,826	539

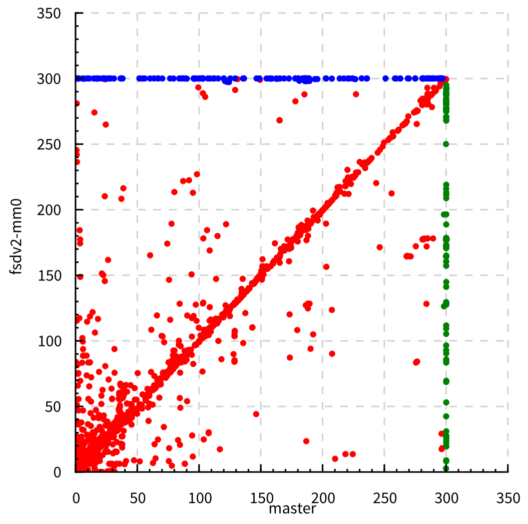
Table 5.5: Number of solved problems from SMT-COMP 2019 by logic



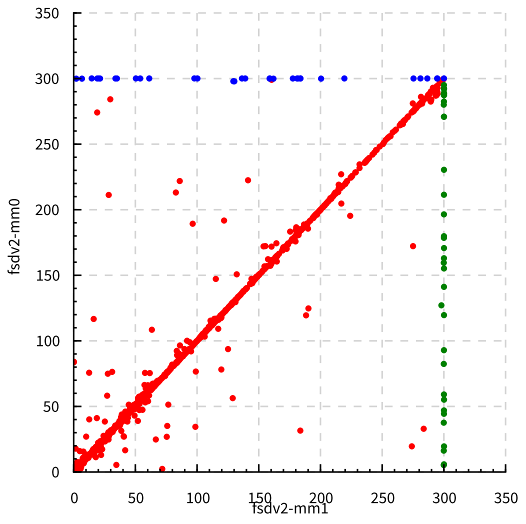
(a) `master` vs. `fsdv2-mm1`



(b) `master` vs. `fsdv2-mm3`



(c) `master` vs. `fsdv2-mm0`



(d) `fsdv2-mm1` vs. `fsdv2-mm0`

Figure 5.2: Comparing solver runtimes of different configurations on problems from the SMT-COMP 2019 Single Query Track. Problems that can be solved by both configurations are indicated in red, while problems that can be solved by only one configuration are indicated in blue or green.

	Unique UNSAT: FSD/master				
Logic	fsdv2-mm1	fsdv2-mm3	fsdv2-mm0	fsdv2 combined	fsdv1-mm1
ALIA	–/–	–/–	–/–	–/–	–/–
AUFDTLIA	–/1	–/1	–/2	–/1	–/1
AUFLIA	–/7	2/7	2/5	2/5	–/7
AUFLIRA	2/5	2/2	1/3	3/2	2/3
AUFNIA	–/–	–/–	–/–	–/–	–/–
AUFNIRA	1/4	1/4	–/4	1/4	1/3
LIA	–/–	–/–	–/–	–/–	–/–
LRA	1/1	1/1	1/1	1/1	1/1
NIA	–/–	–/–	–/–	–/–	–/–
NRA	–/–	–/–	–/–	–/–	–/–
UF	9/33	7/33	7/32	9/30	8/30
UFDT	2/15	2/16	2/18	2/14	4/12
UFDTLIA	–/2	–/3	–/3	–/2	–/1
UFDTNIA	–/–	–/–	–/–	–/–	–/–
UFIDL	–/–	–/–	–/–	–/–	–/–
UFLIA	22/32	21/34	19/35	25/23	25/20
UFLRA	–/–	–/–	–/–	–/–	–/–
UFNIA	90/68	102/64	98/65	112/61	83/80
All logics	127/168	138/165	130/168	155/143	124/158

Table 5.6: Number of uniquely solved UNSAT problems from SMT-COMP 2019 by logic

5.2 On the Number of Multi-Literal Matches

We have performed a separate run on the same problem set as in Section 5.1 with additional runtime statistics enabled. In particular, we were interested to learn more about useful values for the option `-fsdmm`. To this end, we have logged for each successful FSD inference how many multi-literal matches were examined. The aggregated results are given in Table 5.7 for the problems from the TPTP and in Table 5.8 for the problems from SMT-COMP 2019.

Most FSD inferences are found already with the first multi-literal match, which suggests a rather small value of `-fsdmm` might be a good default. For example a value of 5 for TPTP problems and a value of 8 for SMT-COMP problems covers 99 % of the possible FSD inferences. However, the best value for this option likely depends strongly on the concrete problem, as comparing the results of configurations `fsdv2-mm1` and `fsdv2-mm0` does not give a clear preference (cf. Figures 5.1d and 5.2d).

On the three problems `GEO309+1.p`, `GEO313+1.p`, and `GEO322+1.p` from the TPTP we have observed certain FSD inferences where as many as 327,164 matches had to be examined. Manual inspection has shown that the situation in these problems is similar to what we describe in Example 15.

Matches examined	FSD inferences	Cumulative percentage
1	215,203,323	90.8636 %
2	13,942,080	96.7502 %
3	3,202,922	98.1026 %
4	1,076,591	98.5571 %
5	1,204,196	99.0656 %
6	370,723	99.2221 %
7	219,229	99.3147 %
8	112,557	99.3622 %
9	128,549	99.4165 %
10	167,584	99.4872 %
11–20	607,796	99.7439 %
21–100	550,484	99.9763 %
101–1,000	53,964	99.9991 %
1,000–100,000	2,167	100.0000 %
≥100,001	34	100.0000 %

Table 5.7: Number of FSD inferences by number of matches examined (TPTP)

Matches examined	FSD inferences	Cumulative percentage
1	955,515,096	79.8259 %
2	85,329,923	86.9545 %
3	48,650,079	91.0188 %
4	34,469,685	93.8985 %
5	27,187,786	96.1698 %
6	17,854,951	97.6615 %
7	12,347,235	98.6930 %
8	6,076,305	99.2006 %
9	3,836,938	99.5212 %
10	2,143,043	99.7002 %
11–20	3,419,842	99.9859 %
21–100	163,828	99.9996 %
101–1,000	4,523	100.0000 %
1,000–100,000	336	100.0000 %
≥100,001	0	100.0000 %

Table 5.8: Number of FSD inferences by number of matches examined (SMT-COMP)

5.3 Soundness Tests

To increase confidence in the correctness of the implementation of FSD v2, we have performed *soundness tests*. This means checking for all successful FSD inferences that the conclusion does indeed follow from the two premises. Of course, to avoid circular reasoning, this proof must be done without using FSD.

To enable the soundness tests, we extended the implementation of FSD v2 with *inference logging*: all successful inferences are printed on standard output in the TPTP format.

The overall method is as follows:

1. Run Vampire with FSD v2 and inference logging on all TPTP problems and save the output.
2. Using a simple script, convert each inference in the Vampire output into its own problem file in TPTP format
3. Check each of the resulting problems with Vampire (branch `master`, without any changes relating to FSD). An earlier run was also performed partially using the theorem prover E.

Even though a relatively short timeout of 30 seconds was used in Step 1, more than 3 million inferences had to be checked.

Most of the checks completed quickly in less than a second using Vampire's portfolio mode with a timeout of 5 seconds. However, some checks were harder. From limited manual inspection, we suppose the reason is that the implementation of FSD rewrites all matching terms in the simplified literal at once (i.e., it actually performs more than one inference at once). Whenever more than one term has been replaced, the corresponding inference check instance was much more difficult for the prover to solve. These cases have been solved in a second pass using Vampire's portfolio mode with a timeout of 300 seconds.

In the end, all of the logged FSD inferences have been proved correct.

The inference logging and the soundness tests have been performed on a standard desktop computer. Because Vampire is essentially a single-threaded program, we have used GNU Parallel [Tan18] to run multiple instances in parallel and thus take advantage of additional CPU cores. GNU Parallel also allows the user to interrupt large batch jobs to continue them later, which is very convenient for such tasks.

5.4 Redundancy Tests

Complementing the soundness tests, we also added a post-check to the implementation of FSD v2 which asserts that all successful FSD inferences are indeed simplifying inferences, i.e., that the main premise is larger than the conclusion and the (instantiated) side premise (cf. the proof of Theorem 3).

The clause comparison has been implemented according to a characterization of the multiset extension that is well-suited for implementation [BN98, Lemma 2.5.6].

The redundancy tests have been performed under similar circumstances as the soundness tests. No violations of redundancy have been discovered during these runs.

5.5 Summary

We briefly summarize the experimental results.

Most instances of FSD can be found using only a low number of multi-literal matches, suggesting that placing a limit on the amount of matches to examine might be beneficial. However, experimental results are unclear on this question.

Altogether, Vampire with FSD v2 was able to solve 59 unique problems from the TPTP v7.2.0 and 157 unique problems from the SMT-COMP 2019 Single Query Track that the default version did not solve[5]. Moreover, our implementation was able to solve 6 problems from the TPTP that have not been solved before by another automated theorem prover, according to the TPTP solutions database.

Nevertheless, further investigation is necessary to understand an optimal (portfolio) combination of FSD (v1 and/or v2) with other Vampire options.

[5]Configuration “fsdv2 combined”, taking results for “unsat” and “sat” together.

CHAPTER 6

Related Work

Deductive verification of computer programs is an axiomatic verification approach based on Hoare Logic [Hoa69]. The primitives in Hoare Logic are triples of the form $\{F\}P\{G\}$, where F and G are formulas expressing assertions on the program state, and P is a program. Such a triple expresses, e.g., a *partial correctness claim*: whenever the precondition F holds at the beginning of an execution of P and P terminates, then the postcondition G holds at the end of P. A correctness proof in Hoare Logic follows from a sequence of Hoare proof rules, based on the syntax of the program, building triples for complex programs from smaller components that have been proved previously.

The paradigm of deductive verification forms the basis of reasoning-based verification and is used in the following way. First, syntactic Hoare proof rules are applied in so-called *predicate transformers* on the pre- and post-conditions to obtain a set of formulas, the *verification conditions*. Second, the verification conditions are passed to an automated theorem prover to check their validity. The syntactic rules are set up in such a way that the validity of all verification conditions implies the truth of the correctness claim.

Deductive verification has been used in practice. Dafny [Lei10] is a verification-aware programming language with built-in support for annotating code with pre- and post-conditions, as well as loop invariants. During compilation, Dafny generates the verification conditions and checks them using an external SMT solver. Frama-C [KKP$^+$15] is a framework for static analysis of C code that has multiple plugins for deductive verification. Why3 [FP13] provides a specification and programming language, and relies on multiple automated and interactive theorem provers to discharge its verification conditions.

Model checking [CE82, QS82] is a successful verification method where the program to be verified is transformed into a state transition system. The desired properties of the system are specified as formulas in temporal logic, and the model checker checks whether these formulas are true on the transition system at hand. Whenever the model checker fails to prove a temporal formula, it will construct a counterexample trace, which provides significant help for practitioners working with these tools.

Modern model checkers employ sophisticated techniques to keep the transition system as small as possible [CHV18]. A key feature is abstraction, which allows the model checker to handle very large (even infinite) state transition systems by merging sets of concrete states into single abstract states. Abstraction may lead to spurious counterexamples, i.e., traces that violate the specification only in the abstracted system but do not correspond to an execution in the real system. The most advanced model checkers use counterexample-guided abstraction refinement [CGJ+00], a method that detects spurious counterexamples and automatically refines the abstraction until either the formula is verified or a true counterexample is found.

Examples for classical model checkers that have been widely used in practice are CBMC [CKL04] for verification of C programs, and SPIN [Hol97] for verification of concurrent processes. The SLAM [BR02, BLR11] project is a very successful application of model checking to static verification of hardware drivers in the operating system Windows.

Both model checking and deductive verification, as well as other verification methods, ultimately require some variety of theorem proving, meaning that automation of theorem proving is a prerequisite for automation of program verification.

One of the most prominent approaches to automated theorem proving is propositional satisfiability checking based on the well-known algorithm DPLL [DP60, DLL62]. Modern SAT solvers achieve high performance on structured problem instances due to techniques such as conflict-driven clause learning [MSS99] and specialized heuristics [MMZ+01]. Most automated theorem provers, not just SAT solvers, rely on some variation of these algorithms to solve certain sub-tasks. For example, Vampire employs a SAT solver for clause splitting [Vor14].

DPLL has later been extended to certain fragments of first-order logic. This extension is known as satisfiability modulo theories (SMT) solving [NOT05, BT18]. In particular, current SMT solvers have good support for theories such as linear arithmetic, bit vectors, and arrays, whereas support for quantification is still limited and fragile. Examples for current SMT solvers are Z3 [DMB08] and CVC4 [BCD+11]. SPACER [KGC16] is an SMT-based model checking approach.

Most state-of-the-art automated theorem provers for first-order logic with equality are based on saturation algorithms (such as Otter [McC94], Discount [DKS97], and LRS [RV03]) and the superposition calculus [BG94, BG98, NR01]. Superposition is an extension of the resolution calculus [Rob65, BG01] with the goal of working more efficiently with equalities. Essential ingredients for efficient implementations of superposition are simplification orderings [KB83] and redundancy, both of which we introduce in the preliminaries.

In first-order theorem proving, the current situation is the opposite compared to SMT solving. State-of-the-art superposition-based provers have good support for quantification and quantifier alternation, as well as equality and uninterpreted functions/predicates. However, they encounter difficulties as soon as certain theories are used, with natural numbers and integer arithmetic being especially problematic.

While superposition is the most prominent approach to first-order reasoning, there are alternatives such as the Inst-Gen calculus [GK03, Kor13], which utilizes instantiation to approximate the first-order reasoning problem by a sequence of propositional problems.

Examples for current superposition-based provers are Vampire [KV13], E [Sch13] and SPASS [WDF+09]. iProver [Kor08] is based on the Inst-Gen calculus. The prover included in the KeY project [ABB+16] uses the sequent calculus and offers a combination of automatic and interactive theorem proving. Concerning program verification, the KeY project includes support for verification of Java programs, and Vampire has been used as prover backend for verification as well [GKR18].

Subsumption demodulation, like superposition, is an instance of the general concept of *conditional rewriting*. Unlike superposition, subsumption demodulation is also a simplification rule. Some other simplification rules that are also instances of conditional rewriting have been discussed in the literature.

Contextual reductive rewriting [BG94] is more general than subsumption demodulation. Consider the side premise $s \simeq t \vee C$, where $s \simeq t$ is the rewriting equality. Instead of requiring that $C\theta$ is a submultiset of the main premise, for contextual reductive rewriting it suffices that $C\theta$ is implied by the main premise (positive and negative literals separately). Since contextual reductive rewriting was developed in the context of equational reasoning (i.e., the only predicate is equality), Bachmair and Ganzinger suggest to check these side conditions by recursively invoking contextual reductive rewriting. To simplify the ordering check, they suggest to only consider side clauses where s is a strictly maximal term.

Approximated Contextual Rewriting [WW08] is a refined notion of contextual reductive rewriting that has been implemented in SPASS. A major difference to subsumption demodulation is that in subsumption demodulation the discovery of the substitution is driven by the side conditions whereas in approximated contextual rewriting the side conditions are evaluated by ckecking the validity of certain implications by means of a reduction calculus. This reduction calculus recursively applies another restriction of contextual rewriting called recursive contextual ground rewriting, among other standard reduction rules. While approximated contextual rewriting is more general, the benefit of subsumption demodulation is that it can be well integrated into the existing subsumption machinery.

Non-unit rewriting [Wei01] is similar to subsumption demodulation in that it instantiates the side premise and performs a subsumption-like check. Non-unit rewriting is however missing the second side condition on subsumption demodulation, which ensures that the main premise becomes redundant after the inference.

CHAPTER 7

Conclusion

We introduced the inference rule *subsumption demodulation* to improve support for reasoning with conditional equalities in superposition-based first-order theorem proving, and we proved this rule to be a sound simplification rule.

We implemented the forward direction of subsumption demodulation in the superposition-based theorem prover VAMPIRE. For the first implementation ("FSD v1"), the aim was to re-use existing components of VAMPIRE with minimal changes in order to obtain a mostly straightforward reference implementation. Even then, a new clause index and some minor additions to existing components were necessary. Next, we optimized this implementation to obtain a more efficient version ("FSD v2"), which necessitated significant changes to the multi-literal matching component of a first-order prover.

Since multi-literal matching is NP-complete, even an optimized implementation may turn out to be a bottleneck in certain cases. We discussed such a case by means of an example, which, although rare, our experiments confirmed to indeed occur in practice. A practical workaround is to place an upper limit on the number of matches to examine. Statistics from the experimental runs show that most FSD instances can be found with a low upper limit, although the ideal value strongly depends on the concrete problem.

We evaluated our implementation of FSD on a large number of benchmark problems from the TPTP v7.2.0 problem library and the SMT-COMP 2019 competition. All configurations with FSD were able to solve new problems compared to the standard configuration of VAMPIRE, even though VAMPIRE's portfolio mode is highly tuned to the set of options available in standard VAMPIRE and does not take into account FSD. What is more, FSD was even able to solve some problems that were unsolved before, according to the TPTP solutions database.

7.1 Future Work

However, while we have experimented with different options influencing FSD, the results do not clearly show a best configuration. Tuning FSD options remains a tricky and open problem. Interestingly, even FSD v1 solved some problems that FSD v2 did not solve. Thus, more experiment-driven research is required. Most importantly, the only way to enable FSD in portfolio mode currently is to enable it in all strategies, which is too inefficient. It would be beneficial to train the portfolio mode of Vampire to take into account FSD. An interesting question is whether there is some set of strategies that combines most of the positive effects of the various FSD configurations.

The implementation also contains opportunities for further improvements.

Currently, the implementations of forward subsumption and forward subsumption demodulation are completely separate. This means some work is repeated, especially in negative cases. Consider the case where the FSD algorithm fails to find an instance of FSD for a given clause C and candidate side clause D because no suitable multi-literal match exists. In such a case, already the forward subsumption algorithm will have retrieved the same clause D for the given clause C (excepting some edge cases), and therefore also performed almost the same (failing) multi-literal matching checks.

This duplicated work could be repeated if forward subsumption were merged into the FSD algorithm. Since the implementation of FSD v2 already supports reporting subsumption by non-unit clauses (which at the moment does not happen in practice since the existing implementation of forward subsumption is called first), this task by itself should not be too much effort. However, currently forward subsumption also performs forward subsumption resolution at the same time, so some care must be taken to properly integrate this inference rule as well. Also, the clause indexes of FSD and forward subsumption need to be merged because at the moment there are subtle differences between the two.

Furthermore, as hinted before, more intelligent selection of the rewriting equality might be beneficial. In particular, the matcher should not only select the rewriting equality but at the same time also a concrete term for rewriting, expanding the current substitution accordingly. This change would especially improve situations such as in Example 15.

Bibliography

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification – The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*. Springer International Publishing, 2016.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011.

[BDdM+13] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 6 Years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[BG98] Leo Bachmair and Harald Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In *Automated Deduction: A Basis for Applications*, volume 1, pages 353–397. Kluwer, 1998.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, pages 19–99. Elsevier BV, 2001.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A Decade of Software Model Checking with SLAM. *Commun. ACM*, 54(7):68–76, July 2011.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM.

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. *Handbook of Model Checking*, pages 305–343, 2018.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[CGJ$^+$00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. *Lecture Notes in Computer Science*, pages 154–169, 2000.

[CHV18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. Introduction to Model Checking. *Handbook of Model Checking*, pages 1–26, 2018.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[DKS97] Jörg Denzinger, Martin Kronenburg, and Stephan Schulz. DISCOUNT – A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, Jul 1962.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, Jul 1960.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where Programs Meet Provers. *Lecture Notes in Computer Science*, pages 125–128, 2013.

[GK03] Harald Ganzinger and Konstantin Korovin. New Directions in Instantiation-Based Theorem Proving. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, LICS '03, pages 55–64, Washington, DC, USA, 2003. IEEE Computer Society.

[GKR18] Bernhard Gleiss, Laura Kovács, and Simon Robillard. Loop Analysis by Quantification over Iterations. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 57, pages 381–399. EasyChair, 2018.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct 1969.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[KB83] D. E. Knuth and P. B. Bendix. Simple Word Problems in Universal Algebras. *Automation of Reasoning*, pages 342–376, 1983.

[KGC16] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, Jun 2016.

[KKP$^+$15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.

[Kor08] Konstantin Korovin. iProver – An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*, pages 292–298, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Kor13] Konstantin Korovin. Inst-Gen – A Modular Approach to Instantiation-Based Automated Reasoning. *Lecture Notes in Computer Science*, pages 239–270, 2013.

[KV13] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Lei10] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Löc06] Bernd Löchner. Things to Know when Implementing KBO. *Journal of Automated Reasoning*, 36(4):289–310, Aug 2006.

[McC94] William W. McCune. OTTER 3.0 Reference Manual and Guide. Technical report, Argonne National Laboratory, January 1994.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[MSS99] J. P. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[NOT05] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. *Lecture Notes in Computer Science*, pages 36–50, 2005.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning*. Elsevier BV, 2001.

[PG86] David A. Plaisted and Steven Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, sep 1986.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[Rob65] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, January 1965.

[RSV+19] Giles Reger, Martin Suda, Andrei Voronkov, Evgeny Kotelnikov, Simon Robillard, Laura Kovács, and Martin Riener. Vampire 4.4-SMT System Description. https://smt-comp.github.io/2019/system-descriptions/vampire.pdf, 2019.

[RV03] Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1-2):101–115, Jul 2003.

[Sch13] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[SST14] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A Cross-Community Infrastructure for Logic Solving. *Lecture Notes in Computer Science*, pages 367–373, 2014.

[Sut17] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, Feb 2017.

[Tan18] Ole Tange. *GNU Parallel 2018.* Ole Tange, March 2018.

[Vor14] Andrei Voronkov. AVATAR: The Architecture for First-Order Theorem Provers. *Lecture Notes in Computer Science*, pages 696–710, 2014.

[WDF$^+$09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS Version 3.5. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, pages 140–145, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Wei01] Christoph Weidenbach. Combining Superposition, Sorts and Splitting. In *Handbook of Automated Reasoning*, pages 1967–2013. Elsevier BV, 2001.

[WW08] Christoph Weidenbach and Patrick Wischnewski. Contextual Rewriting in SPASS. In *Proceedings of the First International Workshop on Practical Aspects of Automated Reasoning*, 2008.