Implementing Superposition in iProver (System Description)

André Duarte(✉) and Konstantin Korovin(✉)

The University of Manchester, Manchester, UK
{andre.duarte,konstantin.korovin}@manchester.ac.uk

Abstract. iProver is a saturation theorem prover for first-order logic with equality, which is originally based on an instantiation calculus Inst-Gen. In this paper we describe an extension of iProver with the superposition calculus. We have developed a flexible simplification setup that subsumes and generalises common architectures such as DISCOUNT or Otter. This also includes the concept of "immediate simplification", wherein newly derived clauses are more aggressively simplified among themselves, and the concept of "light normalisation", wherein ground unit equations are stored in an interreduced TRS which is then used to simplify new clauses. We have also added support for associative-commutative theories (AC), namely by deletion of AC-joinable clauses, semantic detection of AC axioms, and preprocessing normalisation.

iProver[1] [10] is an automated theorem prover for first-order logic. It is a saturation prover, and is based primarily on the Inst-Gen calculus [7], but also implements resolution and supports running them in combination in an abstraction-refinement framework [9,12]. In this work we detail how iProver was extended with support for the superposition calculus.

Currently, iProver deals with equality axiomatically, which can be inefficient for problems heavy on equality. At the same time, the superposition calculus is a set of complete inference rules specialised for first-order logic with equality. It complements the instantiation calculus since it is effective on problems where instantiation struggles, and vice-versa. We show that running the two calculi in combination yields better results than either plain instantiation or plain superposition.

Rules in a calculus can be classified as "generating", if they derive new clauses, or "simplifying", if some premise gets deleted. While generating rules are the ones necessary for completeness of a calculus, simplification rules are crucial for practical performance. Intuitively, simplification rules are beneficial to taming the growth of the search space as more clauses get generated. However, the computation of those simplifications itself takes time, so being too eager in applying them will also grind the prover to a halt. It is an open problem, what the optimal strategy to balance these conflicting requirements is, and although there is

[1] Available at http://www.cs.man.ac.uk/~korovink/iprover.

N. Peltier and V. Sofronie-Stokkermans (Eds.): IJCAR 2020, LNAI 12167, pp. 388–397, 2020.
https://doi.org/10.1007/978-3-030-51054-1_24

a huge amount of flexibility in how to perform simplifications (see e.g., [21]), most provers are rather restrictive about this. In iProver we developed a flexible simplification setup that subsumes and generalises most common architectures.

Finally, we have also implemented specialised techniques to deal with associative-commutative (AC) theories. These are theories of great interest which arise in several domains [20], and are traditionally problematic for theorem provers to deal with, due to combinatorial explosion and the non-orientability of the AC axioms.

The paper is structured as follows: first, we give a quick overview of the architecture of iProver. Then, we describe the implementation of superposition in iProver, its modifications, the simplification architecture and given clause loop, and AC reasoning rules. For a more in-depth description of basic features of iProver, see [9].

1 Overview

iProver is based on the Inst-Gen calculus, which is based on the following idea. We approximate the first-order problem to a propositional problem, and submit it to a black box SAT solver. It either finds an inconsistency, which is also an inconsistency at the first-order level, or else it returns a model, which guides the instantiation of new clauses whose abstraction witnesses some inconsistency at the ground level. If no such instantiation exists, then the problem is satisfiable.

The SAT solver is also used to implement "global" simplification rules (in the sense that they involve reasoning with the clause set which is shared between different calculi), such as global propositional subsumption [9]. In a nutshell, we submit ground abstractions of clauses in S to a set S_{gr}. If the SAT solver finds that S_{gr} propositionally implies $D\gamma$, with D a strict subset of C and γ an injective substitution of variables to fresh constants, then we can replace $C\theta$, in S, by D.

$$\text{Glob. prop. subs.} \qquad \frac{C\theta}{D}, \quad \begin{array}{l}\text{where } D \subsetneq C \\ \text{and } C \in S,\ S \models S_{gr},\ S_{gr} \models D\gamma\end{array} \tag{1}$$

As mentioned before, iProver can also run other calculi. This is beneficial because (i) some problems are solved easily by one strategy and not by others, and (ii) clauses derived in e.g. resolution can be passed to the instantiation solver to participate in simplifications. For example, clauses derived in all calculi are submitted to a shared global propositional subsumption set, which is in turn used by all calculi to simplify its clauses.

Schematically, the high-level architecture of iProver is summarised in Fig. 1. We can view it as a modular architecture where each calculus (Inst-Gen, resolution, superposition) runs its own saturation loop, and can (i) query external SAT and SMT solvers, and (ii) submit clauses to, and retrieve clauses from, the 'Exchange' module. The instantiation and resolution modules are discussed in [9]. Here we will focus on the superposition calculus.

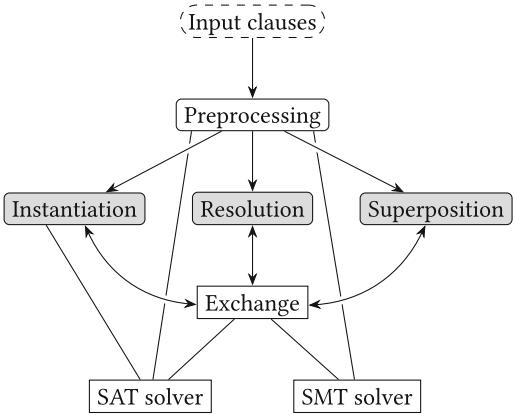

Fig. 1. Architecture of iProver.

2 Extension with Superposition

The superposition inference system consists of the following rules [15]:[2]

$$\text{Superposition} \qquad \frac{l \approx r \vee C \quad t[s] \dot{\approx} u \vee D}{(t[s \mapsto r] \dot{\approx} u \vee C \vee D)\theta} \tag{2}$$

where $\theta = \mathrm{mgu}(l, s)$, $l\theta \not\preceq r\theta$, $t\theta \not\preceq u\theta$, and s not a variable,

$$\text{Eq. Resolution} \qquad \frac{l \not\approx r \vee C}{C\theta} \quad \text{where } \theta = \mathrm{mgu}(l, r), \tag{3}$$

$$\text{Eq. Factoring} \qquad \frac{l \approx r \vee l' \approx r' \vee C}{(l \approx r \vee r \not\approx r' \vee C)\theta} \quad \begin{array}{l}\text{where } \theta = \mathrm{mgu}(l, l'),\\ l\theta \not\preceq r\theta \text{ and } r\theta \not\preceq r'\theta.\end{array} \tag{4}$$

We assume that $\prec$ is a simplification ordering. Non-equality predicates $P(t)$ are encoded as $P(t) \approx \top$. The calculus can easily be generalised to the many-sorted case, which iProver uses even in untyped problems, since it can perform subtype inference during preprocessing. Superposition is sound and refutationally complete for first-order logic with equality (see [2,15]) and implemented in a number of state-of-the-art theorem provers: Vampire [11], E [19] and SPASS [22]. Currently, iProver uses non-perfect discrimination trees to find unification candidates efficiently [8,16]. For the literal selection, to ensure completeness, we must select either a negative literal, or all maximal literals. In iProver we use a variant of the Knuth-Bendix ordering which prioritises non-equational literals.

[2] '$\dot{\approx}$' means '$\approx$' or '$\not\approx$'; also rules are to be read modulo flipping the equalities.

Simultaneous Superposition. In (2), by $t[s]$ and $t[s \mapsto r]$ we can mean resp. "a distinguished occurrence of s as a subterm of t" and "replacing that subterm at that position by r. We call the variant *simultaneous superposition* where we mean instead "replacing all occurrences of s in t by r". This variant is still refutationally complete [3]. In cases where there are several occurrences of the same term, as in $f(\underbrace{s, s, \ldots, s}_{n \text{ times}})$, this avoids producing $2^n - 1$ intermediate clauses with $f(r, s, \ldots, s)$, $f(s, r, \ldots, s)$, $f(r, r, \ldots, s)$, etc., instead producing only $f(r, r, r, \ldots, r)$. In iProver, we implement this variant of superposition.

2.1 Simplifications

Apart from the generating inferences, necessary for completeness, we can add *simplification inferences*. In our implementation, we use the following rules: tautology deletion, syntactic equality resolution, subsumption, subset subsumption, subsumption resolution, demodulation and global subsumption [9,11,17,22].

Light Normalisation. In addition, we introduce the following rule:

$$\text{Light Normalisation} \qquad \frac{R \quad C[l]}{C[l \mapsto r]}, \quad \text{where } l \to r \in R \tag{5}$$

where R is a set of interreduced wrt. (5) oriented rewrite rules. It can be seen as a restricted case of the demodulation rule, but it is advantageous to formulate this separately because it may be implemented much more efficiently than demodulation, by simply looking up terms in a hashtable, rather than having to do matching with variable instantiations.

A *light normalisation index* consists of (i) a hashtable that indexes rewrite rules in R by their left-hand sides for forward light normalisations and (ii) a map of all subterms in R for keeping R interreduced. When we derive a unit equality, we first normalise it wrt. R by recursively replacing each subterm by its normal form wrt. R. Then, if the simplified equality is orientable (wrt. $\prec$) we use it to normalise rules in R add it to R. If there is a conflict between two rules $t \to s$ and $t \to u$ where $t \succ s \succ u$, we keep the rules $t \to u$ and $s \to u$. If s and u are incomparable wrt. $\succ$ we keep one of the rules in R. Since R is only used for simplifications this choice does not affect the completeness. In general, we can restrict which orientable equations we add to R (e.g. only ground ones, or small in size).

2.2 Given Clause Algorithm

In a standard given clause loop, the clause set is split into an active set, where inferences among the clauses have been performed, and a passive set, of clauses waiting to participate in inferences. Clauses are initially added to the passive, then in each iteration one given clause is picked from the passive set, added to

the active set, and all inferences between given and active are performed. Newly derived clauses are pushed into the passive. The loop finishes when all clauses have been moved to the active set, meaning the initial set is satisfiable, or when a contradiction is derived.

Immediate Simplification Set. Next, we introduce the idea of *immediate simplification*. The intuition is as follows. Clauses that are derived in each loop are "related" to each other. It may be beneficial to keep the set of immediate conclusions inter-simplified. Also, throughout the execution of the program the set of generated clauses in each loop remains small compared to the set of passive or active clauses. Therefore, we can get away with applying more expensive rules that we do not necessarily want to apply on the set of all clauses (e.g. only "light" simplifications between newly derived clauses and passive clauses, but more expensive "full" simplifications among newly derived clauses). Finally, during this process, it is possible that the given clause itself becomes redundant (e.g. subsumed by one of its children). If this happens, we can add *only* the clauses responsible for making it redundant to the passive set, then remove the given clause from the active set, and throw away the rest of the iteration's newly generated clauses, abort the rest of the iteration, and proceed to the next given clause. In some problems, a significant number of iterations may be aborted, which means that fewer new clauses are added to the passive queue, and that we avoid the work of computing those inferences. This can be seen as a variant of orphan elimination [17]. Even when the given clause is not eliminated it is often beneficial to extensively inter-simplify immediate descendants of the given clause.

Simplification Setup. How all these simplifications are performed can greatly impact the performance of the solver, so care is needed, and tuning this part of the solver can pay off significantly. There is a significant amount of choice in how to perform simplifications. We can choose which simplifications to perform, and at what times, and with respect to which clauses. Additionally, some of these simplifications require auxiliary data structures (here referred to generally as "indices") to be done efficiently, and some indices support several simplification rules. Therefore we also need to choose which clauses to add to which indices at which stages.

For example, Otter-style loops [14] perform simplifications on clauses before adding them to the passive set. The problem with this is that the passive set is often orders of magnitude larger than the active set, therefore performance will degrade significantly as this set grows, and the system will spend most of its time performing simplifications on clauses that may not even end up being used. On the other hand, DISCOUNT-style loops [4] perform simplifications only with clauses that have been added to the active set. This has the benefit of reducing the time spent in simplifications, at the cost of potentially missing many valuable simplifications wrt. passive clauses. It is not clear where the "sweet spot" is, in terms of these setups.

It is possible, for example, to choose to apply only "cheap" simplifications to the full active + passive set (e.g. subset subsumption, and light normalisation), and use more expensive ones only on the small active set (e.g. full subsumption and demodulation). In Listing 1.1 we describe the 'iProver-Sup' given-clause saturation loop for superposition. A *simplification set* consists of a collection of indices, each of which supports one or more simplification rules. In our given clause loop we have four such sets: S_{passive}, to which we add the clauses added to the passive set, S_{active}, for the clauses in the active set, S_{immed} for newly derived clauses (this set is cleared at the end of every given-clause iteration, and the non-redundant clauses added to the passive queue), and S_{input} for preprocessing input clauses. Each set supports the following operations: `add`, which adds a clause to all indices in a set S, and `simplify`, which simplifies via some rules R wrt. a set S. These are called at several points in the loop (see Listing 1.1), and the user can configure which indices/rules are involved in each operation. When `simplify`ing, some rules *forward simplify* the clause wrt. the existing set, and others *backward simplify* the clauses in the set wrt. the new clause.

The simplification setup is specified by the rules to apply at each stage (R_x). and the indices to which to add at each stage (S_x). These can be specified by the user via command-line options. S_x are lists of indices from {`Subsumption`, `SubSetSubsumption`, `FwDemod`, `BwDemod`, `LightNorm`, `PropSubsSet`}. R_x are lists of rules from {`EqResSimp`, `TautologyElim`, `EqTautologyElim`, `TrivRules`, `FwPropSubs`, `FwSubsumption`, `FwSubsumptionStrict`, `FwSubsumptionRes`, `FwDemod`, `FwLightNorm`, `FwLightNormDemodLoop`, `ACJoinability`, `BwSubsumption`, `BwSubsumptionRes`, `BwDemod`}. Their usage is documented in the command-line help. The default options are presented in Table 1.

Currently iProver uses non-perfect discrimination trees for implementing backward and forward demodulation [8,16], feature vector indices for subsumption [18], tries for subset subsumption [8], and MiniSat [6] for global subsumption.

Generally, when during immediate simplification a parent clause of a newly derived clause is made redundant, we can remove all the children of that clause from the immediate set (and thus avoid adding them to the passive queues), except for the ones which caused it to be redundant. Currently, we restrict this feature to the given clause rather than to all the parent clauses, therefore, this simplifies to checking whether the given clause is made redundant in S_{immed}, and if so abort the loop, add only the clauses that made it redundant to the passive, and remove the given clause.

2.3 AC Reasoning

If a problem contains associativity and commutativity axioms,

$$f(x, f(y, z)) = f(f(x, y), z)\,, \qquad f(x, y) = f(y, x)\,, \tag{6}$$

then f is said to be AC.

Table 1. Default simplification options

S_{passive}	`SubsetSubsumption`, `PropSubs`
S_{active}	`Subsumption`, `LightNorm`, `FwDemod`, `BwDemod`
S_{immed}	`SubsetSubsumption`, `Subsumption`, `LightNorm`, `FwDemod`, `BwDemod`
S_{input}	`SubsetSubsumption`, `Subsumption`, `LightNorm`, `FwDemod`, `BwDemod`
R_{passive}	`TrivRules`, `ACJoinability`, `FwLightNormDemod`, `FwSubsumption`
R_{active}	`TrivRules`, `FwPropSubs`, `FwLightNormDemod`, `FwSubsumption`, `FwSubsumptionRes`, `BwDemod`
R_{immed}	`TrivRules`, `FwLightNormDemod`, `FwSubsumption`, `FwSubsumptionRes`, `BwDemod`, `BwSubsumption`
R_{input}	`TrivRules`, `FwLightNormDemod`, `FwSubsumption`, `FwSubsumptionRes`, `BwDemod`, `BwSubsumption`, `BwSubsumptionRes`

AC axioms are particularly problematic in theorem proving, because they are non-orientable, which means they can generate permutations of arguments of AC functions. This leads to combinatorial explosion in the number of clauses. In particular, they will combine with each other to produce an exponential number of instances.

Listing 1.1: iProver-Sup given-clause loop algorithm

```
S_input = ∅
for c in input_clauses:
  simplify(c wrt S_input via R_input)
  add(c to S_input)
add_to_passive_queue(S_input)

S_active = S_passive = ∅
loop:
  S_immed = ∅
  given = pop_from_passive_queue()
  simplify(given wrt S_active ∪ S_passive via R_active)
  add_to_active_set(given)
  add(given to S_active)
  add(given to S_immed)
  for c in generating inferences between given and active:
    simplify(c wrt S_immed via R_immed)
    if given was eliminated in S_immed by clauses U:
      add(U to S_passive)
      remove(given from S_active)
      continue
    simplify(c wrt S_active ∪ S_passive via R_passive)
    add(c to S_immed)
  add_to_passive_queue(S_immed)
  add(S_immed to S_passive)
```

AC problems are ubiquitous and appear in a variety of domains [20]. Although theoretical developments behind AC reasoning have a long history, AC support in most theorem provers is limited due to implementation complexity and is mainly restricted to unit equality problems. We extended some of the techniques to be applicable to the general clausal case, see Theorem 1 below, and implemented them in iProver.

AC Preprocessing. During preprocessing we can transform the input problem into any equisatisfiable form. We can normalise AC terms, by e.g. collecting nested AC subterms into a flat list, sorting wrt. some total extension of the term ordering, and making them right-associative.

Deletion of Joinable Equations. A *rewrite system* is a set of rules $l \to r$, such that, if $l \to r$, then for any substitution σ, $l\sigma \to r\sigma$, and for any term u, $u[l] \to u[l \mapsto r]$. By abuse of notation we can also use unorientable equalities $l \leftrightarrow r$, in which case they stand for the set of its orientable instances, $\{l\sigma \to r\sigma \mid l\sigma \succ r\sigma\} \cup \{r\sigma \to l\sigma \mid r\sigma \succ l\sigma\}$. Two terms s and t are *joinable* wrt. a rewrite system R (written $s \downarrow_R t$) if $s \xrightarrow{\star} c \xleftarrow{\star} t$, where '$\to$' denotes a rewrite step with a rule in R and '$\xrightarrow{\star}$' its reflexive-transitive closure. Two terms are *ground joinable* ($s \Downarrow_R t$) if all its ground instances are joinable. Two terms are *strongly ground joinable* (written $s \Downarrow_{\triangleright R} t$) if, for all $s' = s\sigma$, $t' = t\sigma$ ground instances of s, t resp., with $s' \succeq t'$, either s' is t' or else $s' \xrightarrow{l \to r \in R} u' \downarrow_R t'$ where either $l \prec s'$ or l is s' but not $u' \succ t'$ (see [1,13]).

Theorem 1. *If $s \Downarrow_{\triangleright R} t$, then $s \approx t \vee C$ is redundant wrt. R. If $s \Downarrow_R t$ then $s \not\approx t \vee C$ is redundant wrt. $R \cup \{C\}$.*

Theorem 1 was shown in the context of unit equality reasoning in [1]; we extended this theorem to general clauses and provided a different proof [5].

This abstract theorem can be used for AC reasoning, provided we have a criterion to test $l \Downarrow_{\triangleright} r$. We use the following criterion [1]. Let R_{AC} be

$$f(x, y) \leftrightarrow f(y, x)\,, \tag{7a}$$

$$f(f(x, y), z) \to f(x, f(y, z))\,, \tag{7b}$$

$$f(x, f(y, z)) \leftrightarrow f(y, f(x, z))\,, \tag{7c}$$

Unless $l \stackrel{?}{\approx} r$ is an instance of R_{AC} or can be simplified by an equation in R_{AC}, $l =_{\mathrm{AC}} r$ implies $l \Downarrow_{\triangleright R_{\mathrm{AC}}} r$, which means we can use Theorem 1 to simplify/delete clauses wrt. R_{AC}. This is a cheap test for strong ground joinability to apply in practice, since in order to check whether $s =_{\mathrm{AC}} t$ we can simply treat nested applications of f as a flat-list, and then sort wrt. some total order on terms (see above discussion on AC normalisation).

Semantic Detection of Axioms. Some problems are AC even though the input does not contain the axioms explicitly. We say that a problem S is AC if $S \models \text{AC}$. The usual syntactic detection checks if $\text{AC} \in S$. But we wish also to detect AC problems even when this is not the case.

During preprocessing, we query an SMT solver to find out whether $S \models \text{AC}$. Since SMT solvers only accept ground problems, we need to use a sound approximation of the entailment relation. We do this using an injective substitution mapping variables to fresh constants similar as it is done for global subsumption [9]. This is a sound approximation, since $\phi(\bar{c}) \models \psi(\bar{c}) \Rightarrow \forall x \phi(\bar{x}) \models \forall x \psi(\bar{x})$. In order to make SMT reasoning more efficient we can further restrict reasoning to fast rules like unit propagation or place a limit on the number of backtracks. Apart from this, we also check if the AC axioms (6) get produced at some point during saturation, among binary symbols of sort $\alpha \times \alpha \to \alpha$.

3 Implementation and Experimental Results

We integrated the simultaneous superposition calculus, with the iProver-Sup saturation loop, into iProver and evaluated it over 15 168 first-order problems in TPTP-v7.2.0. The superposition loop can solve 7375 (49%), the instantiation loop (on the previous version of iProver) can solve 7884 (52%), and their combination can solve 8708 (57%). We can see that the combination with superposition and the iProver-Sup simplification setup improved the performance of iProver over the whole TPTP library.

Among problems that were solved by superposition, (excluding trivial problems solved by preprocessing), immediate simplification was used in 71.7 % of problems and light normalisation was used in 64.5 % of problems. AC axioms were detected in 1903 problems.

References

1. Avenhaus, J., Hillenbrand, T., Löchner, B.: On using ground joinable equations in equational theorem proving. J. Symb. Comput. **36**(1–2), 217–233 (2003). https://doi.org/10.1016/S0747-7171(03)00024-5
2. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. Log. Comput. **4**(3), 217–247 (1994). https://doi.org/10.1093/logcom/4.3.217
3. Benanav, D.: Simultaneous paramodulation. In: Stickel, M.E. (ed.) CADE 1990. LNCS, vol. 449, pp. 442–455. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52885-7_106
4. Denzinger, J., Kronenburg, M., Schulz, S.: DISCOUNT - a distributed and learning equational prover. J. Autom. Reasoning **18**(2), 189–198 (1997). https://doi.org/10.1023/A:1005879229581
5. Duarte, A., Korovin, K.: AC Reasoning Revisited (2020, to appear)
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37

7. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: Proceedings of the 18th IEEE Symposium on Logic in Computer Science (LICS 2003), pp. 55–64. IEEE Computer Society Press (2003)
8. Graf, P. (ed.): Term Indexing. LNCS, vol. 1053. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-61040-5. 284 p. ISBN 978-3-540-61040-3
9. Korovin, K.: Inst-Gen – a modular approach to instantiation-based automated reasoning. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics. LNCS, vol. 7797, pp. 239–270. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37651-1_10
10. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_24
11. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
12. Lopez Hernandez, J.C., Korovin, K.: An abstraction-refinement framework for reasoning with large theories. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 663–679. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_43
13. Martin, U., Nipkow, T.: Ordered rewriting and confluence. In: Stickel, M.E. (ed.) CADE 1990. LNCS, vol. 449, pp. 366–380. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52885-7_100
14. McCune, W.: OTTER 3.3 Reference Manual. CoRR cs.SC/0310056 (2003). arXiv: cs.SC/0310056
15. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, pp. 371–443. Elsevier and MIT Press, Cambridge (2001). ISBN 0-444-50813-9
16. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning, vol. 2. Elsevier and MIT Press, Cambridge (2001). ISBN 0-444-50813-9
17. Schulz, S.: E - a brainiac theorem prover. J. AI Commun. **15**(2/3), 111–126 (2002)
18. Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics. LNCS (LNAI), vol. 7788, pp. 45–67. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36675-8_3
19. Schulz, S.: System description: E 1.8. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 735–743. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_49
20. Sutcliffe, G.: The TPTP problem library and associated infrastructure. from CNF to TH0, TPTP v6.4.0. J. Autom. Reasoning **59**(4), 483–502 (2017). https://doi.org/10.1007/s10817-017-9407-7
21. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12166, pp. xx–yy. Springer, Heidelberg (2020)
22. Weidenbach, C., Schmidt, R.A., Hillenbrand, T., Rusev, R., Topic, D.: System description: Spass version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 514–520. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_38