AI Communications 15 (2002) 91–110
IOS Press

The design and implementation of VAMPIRE *

Alexandre Riazanov and Andrei Voronkov
Computer Science Department, University of Manchester, Oxford Road, Manchester, M13 9PL, UK
E-mail: {riazanov,voronkov}@cs.man.ac.uk

Abstract. In this article we describe VAMPIRE: a high-performance theorem prover for first-order logic. As our description is mostly targeted to the developers of such systems and specialists in automated reasoning, it focuses on the design of the system and some key implementation features. We also analyze the performance of the prover at CASC-JC.

Keywords: Resolution theorem proving, VAMPIRE

1. Introduction

Since the invention of resolution and paramodulation, theoretical research in the area of resolution-based theorem proving has achieved a remarkable progress in constructing inference systems based on various refinements of these calculi. The developed theory is believed to have a significant, but mostly unrealized potential for such applications as formal hardware and software development, computer algebra, and assisting human mathematicians. One of the main obstacles to creating a practically valuable technology on the base of the theory is our lack of knowledge of efficient implementation techniques. Attempts of implementing efficient theorem provers constitute the main source of such knowledge. This article describes such an attempt.

VAMPIRE is a system for proving theorems in first-order logic with or without equality. It was originally developed in 1993–1995 (only for logic without equality) as an experimental vehicle for testing resolution and the inverse method. The work on VAMPIRE has been discontinued in 1995 and resumed in 1998 when the first author took over the implementation. Then VAMPIRE developed in a resolution-based theorem prover, this time with equality. In the beginning of 2001 a (very primitive) clausifier and preprocessor have been implemented with the purpose of treating arbitrary first-order formulas rather than clauses. From 2001, by VAMPIRE we understand the system consisting of the resolution- and superposition-based subsystem, the preprocessor with clausifier and scripts gluing them together.

*Both authors are supported by grants from EPSRC.

The resolution-based subsystem is the main component of VAMPIRE. We now refer to it as the *kernel*. Most of this article is dedicated to the description of the kernel, its functionality and implementation. The preprocessor and clausifier subsystem is still in its infancy and has no established name.

Between 1998 and 2001 the VAMPIRE development was mainly driven by CASC. As a consequence, our main design decisions have been in the direction of the speed of the proof-search process. The user-oriented features did not receive enough attention. From 2001 VAMPIRE is developing into a user-friendly tool. We expect that in the next two years our efforts will be equally split among efficient implementation, development of new features, user-oriented extensions, and applications.

This article is mainly oriented towards developers of provers, including novices. However, any interested reader can benefit from it. Some parts of this article do not require any special knowledge from the reader. On the contrary, some parts are very specialized and can only be properly understood by implementors of similar systems. We try to make all the necessary references to help the reader.

The main material is divided into three levels. The top level (Section 2) is a very general declaration of functionality of the system in terms of used calculi and proof search mechanisms. At the next level (Section 3) we show how the functionality is mapped into the VAMPIRE design. The most technical Sections 4 and 5 present some implementation recipes that can be useful for implementors of other systems. Section 6 describes VAMPIRE as a blackbox for the user. Section 7 discusses ongoing work on VAMPIRE. Section 8 provides a summary of features implemented in VAMPIRE

and classifies them from the viewpoints of usefulness and implementation costs. Finally, in Section 9 we explain the performance of VAMPIRE in CASC-JC.

2. Functionality of Vampire

Here we give an overview of some features of VAMPIRE related to its functionality, namely the calculi used in it, including the splitting rule, simplification orders, simplification rules, clause and literal selection, and reasoning with limited time.

2.1. Calculi

VAMPIRE is a resolution- and paramodulation-based theorem prover. It implements several versions of binary resolution and superposition with literal selection for logics with and without equality (see, e.g., [2,19]). In addition, VAMPIRE implements hyperresolution, but only for logic without equality. The calculi implemented in VAMPIRE are quite standard, except for orders (Section 2.4), simplification rules (Section 2.5), selection functions (Section 2.6), and splitting (Section 2.3) described below.

2.2. Proof search by saturation

If we are interested in finding a refutation of a clause set in a calculus based on resolution and paramodulation, we can do this by saturating this set with all possible inferences in the chosen calculus. At each step of the saturation process we select an inference in the calculus, that can be done between some clauses in the current *clause store*. The result, if any, of this inference, is added to the clause store. Inferences that add clauses to the store will be called *generating*. The generating inferences are interleaved with *simplifications* which either remove clauses from the store or replace them by 'simpler' clauses. Finding an empty clause at some step indicates that the initial clause set is unsatisfiable. We are interested not only in testing unsatisfiability of the set, but also in finding a proof that can be later checked by a simple program or used in other ways. For this purpose with every clause we keep information about how and from which clauses it has been derived.

If the initial set is unsatisfiable, we are guaranteed to find the refutation eventually, provided that the calculus is refutationally complete and selection of the inferences is *fair*, i.e., any non-redundant inference has to be eventually selected. As in many other systems, inference selection in VAMPIRE is implemented via clause selection. By selecting a clause from the store we enable the inferences possible between this clause and other clauses that have been selected so far. The only clause selection scheme of VAMPIRE is based on the *age-weight ratio* (called pick-given ratio in some other provers). According to this scheme, clauses are kept in two priority queues: the age and the weight priority queues. The age priority queue prefers older clauses to the younger ones, the weight priority queue prefers clauses of smaller weights. The age is identified by the order of appearance of the clauses, the weight is the total number of symbols in the clause. The age-weight ratio is a non-negative integer. If the ratio is k, then out of every $k+1$ clauses, k clauses are selected from the weight queue and one from the age queue. The age-weight ratio can be specified by the user.

In VAMPIRE saturation is combined with a splitting rule briefly described in Section 2.3. The concrete implementation of the clause store and the general architecture of VAMPIRE are outlined in Section 3.

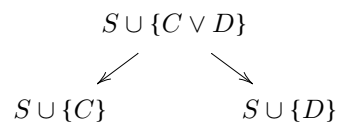
2.3. Splitting

VAMPIRE implements the so-called *splitting without backtracking* described in [23,26].[1] We refer the reader to these papers and also [4] for a thorough discussion of various versions of this rule.

In its simplest form, the splitting rule is formulated as follows:

$$\frac{S \cup \{C \vee D\}}{S \cup \{C \vee p, D \vee \neg p\}},$$

where S is a set of clauses (the current clause store), the clauses C and D have no common variables, and p is a new (not occurring elsewhere) propositional symbol. This rule simulates splitting of the clause store into two new stores, one containing C and another containing D, which is essentially a restricted version of the β-rule in semantic tableaux:



[1] Harald Ganzinger pointed out to us that some versions of splitting without backtracking were implemented in the experimental system Saturate [9].

The new propositional symbol p can be considered as a label of the left branch of this search tree, while $\neg p$ is a label of the right branch. The literals p and $\neg p$ are called the *splitting branch literals*. There also exists another interpretation of this splitting rule via *naming*. We consider p as a 'name' of the formula $\neg\forall(C)$. In [23,26] we have shown how various versions of this rule (based on extensions of the existing literal selection function) can simulate parallel and sequential branch exploration. Once introduced, the name p can be reused: if generated is another clause $C \vee D'$, in which the variables of C and D' are disjoint, we can simply replace C by p instead of introducing another name.

In VAMPIRE v2.0 there is a rigid limit of 1024 on the total number of predicates and functions in the signature. Thus we cannot introduce more than 1000 new predicates needed for splitting. In the forthcoming release, the rigid limit on the signature size will be extended to 65536, but there will be an option allowing the user to limit the number of new symbols introduced during the proof search.

Although the use of splitting results in degradation of performance on the average,[2] there exist many problems which VAMPIRE can solve in reasonable time only with splitting.

2.4. Simplification orders

The current version of VAMPIRE implements only one simplification order on terms: a non-recursive version of the Knuth-Bendix order [11]. We will first define this order, denoted by $\succ$, on ground terms. Let $\mathcal{F}$ denote the set of all function symbols of our signature. The order $\succ$ is parametrized by two parameters: a *weight function* w from $\mathcal{F}$ to the set of positive[3] integers, and a *precedence relation*, that is a total order $\gg$ on the set $\mathcal{F}$ of function symbols. We define the *weight* $|s|$ of a ground term s as the sum of the weights of all symbols in it. The precedence relation $\gg$ on the function symbols induces an order, denoted by $\ggg$, on the set of all ground terms of $\mathcal{F}$ defined as follows. Let $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ be ground terms, where $m, n \geqslant 0$. Then we have $s \ggg t$ if

1. $f \gg g$; or
2. $f = g$ (and hence $m = n$) and, for some i, $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and $s_{i-1} \ggg t_{i-1}$.

If we regard terms as strings, then it is easy to see that $\ggg$ is the lexicographic extension of $\gg$.

Now, for any two ground terms s, t we have $s \succ t$ if

1. $|s| > |t|$; or
2. $|s| = |t|$ and $s \ggg t$.

This definition is similar to that of the standard Knuth-Bendix order on ground terms, except that we do not recursively descend to the subterms of s, t. Due to this fact, we call this modification of the Knuth-Bendix order non-recursive, although, depending on the used representation of terms, recursion may still be needed to implement the order $\ggg$.

Let us call an order $>$ on terms of a signature $\mathcal{F}$ *complete* if for all terms s, t, $s > t$ if and only if $s\theta > t\theta$ for every substitution θ grounding for s, t. Every order $>$ on ground terms can be extended to a complete order $>'$ on non-ground terms which is the greatest among all complete orders coinciding with $>$ on ground terms by simply defining $s >' t$ if and only if $s\theta > t\theta$ for every substitution θ grounding for s, t. In the case of non-recursive Knuth-Bendix order $\succ$ the corresponding greatest order $\succ'$ on non-ground terms may be expensive to implement, since comparing terms w.r.t. $\succ'$ requires solving ordering constraints for $\succ$ (see [12,16,17]). It is not known whether solvability of ordering constraints for $\succ$ is decidable, so we define $\succ$ on non-ground terms as an approximation to $\succ'$, so that $s \succ t$ implies $s\theta \succ t\theta$ for all substitutions θ, but the reverse may, in general, not hold.

To define $\succ$ on non-ground terms, we generalize the notion of weight and the order $\ggg$ on non-ground terms.

We generalize the notion of term weight in the following way. Denote by $\mathcal{V}$ the set of all variables used in terms. We call a *weight expression* a linear polynomial over $\mathcal{V}$ with integer coefficients. For every variable or a function symbol q and a term t denote by $C_q(t)$ the number of occurrences of q in a term t. The *weight* of a term t is now defined as the following weight expression:

$$\sum_{f \in \mathcal{F}} C_f(t) \cdot w(f) + \sum_{x \in \mathcal{V}} C_x(t) \cdot x.$$

For example, if $w(f) = w(a) = 1$, then the weight of $f(f(a, x_1), f(x_1, x_2))$ is $2 \cdot x_1 + x_2 + 4$. Note that if s is ground, then the weight of s is the constant

[2] As judged by extensive runs over the whole TPTP library ([31]). An exception is a class of problems in the SYN category where splitting gives a drastic improvement. It is possible that for new classes of problems splitting will improve the performance.

[3] This requirement is imposed to simplify the definition. It can be relaxed, so that the weight of some function symbols may be zero.

function $|s|$. We will denote the weight of an arbitrary term s by $|s|$, by the previous sentence this does not conflict with our previous notation $|s|$ for ground terms.

Let us now introduce an order $>_w$ and a binary relation $\geqslant_w$ on weight functions such that $|s| >_w |t|$ (respectively, $|s| \geqslant_w |t|$) guarantees that for all substitutions θ grounding for s, t we have $|s\theta| > |t\theta|$ (respectively, $|s\theta| \geqslant |t\theta|$). The definition is as follows. Let $p_1 = \alpha_0 + \alpha_1 v_1 + \cdots + \alpha_n v_n$ and $p_2 = \beta_0 + \beta_1 v_1 + \cdots + \beta_n v_n$ be two weight functions, i.e., linear polynomials over $\mathcal{V}$ (some of the α_i's and β_j's may be 0). Denote by μ the smallest weight of a constant in the signature $\mathcal{F}$. Then $p_1 \geqslant_w p_2$ if

1. For all $i = 1, \ldots, n$ we have $\alpha_i \geqslant \beta_i$;
2. $\alpha_0 + \mu(\alpha_1 + \ldots + \alpha_n) \geqslant \beta_0 + \mu(\beta_1 + \ldots + \beta_n)$.

The definition of $p_1 >_w p_2$ is obtained from the definition of $p_1 \geqslant_w p_2$ be replacing the second condition above by

$2'$. $\alpha_0 + \mu(\alpha_1 + \cdots + \alpha_n) > \beta_0 + \mu(\beta_1 + \cdots + \beta_n)$.

It is not hard to argue that $|s| >_w |t|$ (respectively, $|s| \geqslant_w |t|$) implies that for all substitutions θ we have $|s\theta| >_w |t\theta|$ (respectively, $|s\theta| \geqslant_w |t\theta|$) and that on ground terms the relations $|s| >_w |t|$ and $|s| \geqslant_w |t|$ coincide with $|s| > |t|$ and $|s| \geqslant |t|$, respectively.

Let us now generalize the order $\ggg$ to non-ground terms. The definition is nearly identical to the one for ground terms. Let s, t be terms.

1. If at least one of s, t is a variable than $s \ggg t$ does not hold.
2. Let $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ be non-variable terms, where $m, n \geqslant 0$. Then we have $s \ggg t$ if
   (a) $f \gg g$; or
   (b) $f = g$ (and hence $m = n$) and, for some i, $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and $s_{i-1} \ggg t_{i-1}$.

It is not hard to argue that on ground terms the new definition of $\ggg$ coincides with the one introduced before.

The definition of the non-recursive Knuth-Bendix ordering $\succ$ for non-ground terms is similar to its definition for ground terms. We let $s \succ t$ if and only if

1. $|s| >_w |t|$; or
2. $|s| \geqslant_w |t|$ and $s \ggg t$.

One can prove that $\succ$ is a *reduction* order (see, e.g., [19] for a definition) total on ground terms, and therefore a *simplification* order.

In practice, VAMPIRE uses only one weight function such that $w(f) = 1$ for all symbols f. In conjunction with this weight function, we can use arbitrary total order $\gg$ on the signature $\mathcal{F}$ as the precedence relation. VAMPIRE offers a choice of three precedence relations:

1. Symbols with the larger arity are greater (inspired by E [29]);
2. Symbols with the lower arity are greater;
3. The order of symbols is identified by the order of their appearance in the input.

When we compare atoms rather than terms, there are two ways of comparing them. Consider two atoms $P(s_1, \ldots, s_m)$ and $Q(t_1, \ldots, t_n)$. We assume that the precedence relation $\gg$ and the weight function w are also defined on predicate symbols. We can either

1. Treat atoms as terms and hence compare them as terms; or
2. Let $P(s_1, \ldots, s_m) \succ Q(t_1, \ldots, t_n)$ whenever $P \gg Q$, and if $P = Q$ then compare the atoms $P(s_1, \ldots, s_m)$ as $Q(t_1, \ldots, t_n)$ as terms, i.e., as if P were a function symbol.

The non-recursive version of the Knuth-Bendix order used in VAMPIRE allows for an efficient comparison of terms and is convenient for specializing checks of ordering constraints, see Section 5.3. But it also has severe disadvantages, for example it cannot orient the associativity axiom $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

We are going to implement the standard Knuth-Bendix order [11] and the lexicographic path order (see [8]) in the near future. Since both orders are in fact families of orders with parameters, there should also be a way for the user to specify these parameters, and thus identify the desired instance of these orders.

2.5. *Simplification rules*

In addition to deletion of subsumed clauses and simple tautologies, VAMPIRE implements the following simplification rules.

In addition to deletion of subsumed clauses and simple tautologies, VAMPIRE implements the following simplification rules.

Demodulation, also known as *simplification by unit equalities* or *rewriting*.

$$\frac{S \cup \{s \simeq t,\ C[s\sigma]\}}{S \cup \{s \simeq t,\ C[t\sigma]\}}, \quad \text{if } s\sigma \succ t\sigma.$$

The clause $C[s\sigma]$ is removed from the search space after this inference and replaced by a simpler clause $C[s\sigma]$.[4] The user can choose between using only *pre-ordered* unit equalities $s \simeq t$, i.e., such that $s \succ t$, or using *post-ordered* ones, i.e., such that $s \not\succ t$ but $s\sigma \succ t\sigma$.

Branch demodulation. This rule is applicable when splitting is used.

$$\frac{S \cup \{s \simeq t \vee D,\ C[s\sigma] \vee D'\}}{S \cup \{s \simeq t \vee D,\ C[t\sigma] \vee D'\}}$$

where $s\sigma \succ t\sigma$, $D \subseteq D'$, and both D, D' consist of splitting branch literals only. For branch demodulation, the user can also choose to use either pre-ordered or post-ordered equalities as for ordinary demodulation. Experiments described in [23,26] show that branch demodulation is very efficient on non-unit problems with equality. Branch demodulation is similar to the rule of *contextual rewriting*, see, e.g., [36].

Subsumption resolution.

$$\frac{S \cup \{A \vee C,\ \neg A\sigma \vee C\sigma \vee D\}}{S \cup \{A \vee C,\ C\sigma \vee D\}} \quad \text{or}$$

$$\frac{S \cup \{\neg A \vee C,\ A\sigma \vee C\sigma \vee D\}}{S \cup \{\neg A \vee C,\ C\sigma \vee D\}}.$$

This rule can be implemented via subsumption. Indeed, the instance of this rule on the left is applicable if and only if $A \vee C$ subsumes $A\sigma \vee C\sigma \vee D$ with the substitution σ, and similar for the rule on the right.

Subsumption resolution is very helpful, in particular, because it can radically change the order of clause selection. It results in removal of one or more literals in a clause. A shorter clause has a higher chance to be selected and contribute to a derivation of the empty clause. As an example, for the TPTP problem PUZ017-1 solved by VAMPIRE in 0.4 seconds, the last inference consists of 31 applications of forward subsumption resolution. This problem is hard for other provers, for example, by the year 2000 besides VAMPIRE only SPASS ([35]) could solve it, but in 127 seconds.

[4] Unrestricted rewriting together with the use of the general redundancy criteria (a clause which follows from smaller ones can be removed) is incomplete (Robert Nieuwenhuis, private communication). Completeness of calculi which only use the standard simplification rules, such as unit rewriting and subsumption, has not been well-investigated. The current version of VAMPIRE is not intended for checking satisfiability of sets of clauses, so currently we do not pay much attention to completeness.

2.6. Literal selection

The calculi used in VAMPIRE are parametrized by *literal selection* that tells what literals in a multi-literal clause can participate in resolutions and paramodulations. VAMPIRE has several literal selection functions outlined below. The versions before 2001 had five selection functions implementing negative selection.[5] Of them, two selection functions try to select the maximal literals (like in ordered resolution) but may change the selection if there is more than one maximal literal. Three other selection functions prefer negative literals. After making extensive experiments with these functions we have found them inadequate for a number of benchmark suites. For example, most of these functions simply simulate positive hyperresolution, and therefore are not goal-oriented at all. The five selection functions had an especially poor behaviour on Horn problems on which E ([29]) behaved excellently. When we studied the proofs generated by E on some Horn problems, it became clear that our selection functions could not generate such proofs.

Our first step towards diversifying selection functions was to introduce for each selection function a dual one. The dual function uses positive selection instead of the negative one (except for equality literals). This means that one should first change comparison of literals so that each positive non-equality literals $P(t_1, \ldots, t_n)$ is greater than the respective negative literal $\neg P(t_1, \ldots, t_n)$, then take the definition of negative selection and replace positive non-equality literals by the negative ones and vice versa. As a result, one can select either

1. A positive non-equality literal; or
2. A negative equality literal; or
3. A negative non-equality literal or a positive equality literal, but in this case all maximal literals must be selected.

Positive selection tends to simulate negative hyperresolution, and is therefore more goal-oriented. After introducing positive selection, we could solve a large num-

[5] Our definition of literal selection is different from that of [2] and is in the style of [5]. In [2] only negative literals can be selected. If no negative literal is selected, an inference is performed on a maximal positive literal. In Vampire, as well as in [5], at least one literal is selected in every non-empty clause, and the generating inferences are only applied to selected literals. Nevertheless, we speak about negative selection to emphasize that a negative literal can always be selected, while in the absence of negative selected literals all maximal positive ones must be selected.

ber of problems in the TPTP library that we could not solve before (and even some problems not previously solved by any other prover).

Both negative selection and positive selection are complete. Our next step was to introduce several incomplete selection functions. For example, one of these selection functions simply selects the heaviest literal in the clause. In some cases this incompleteness results in VAMPIRE's being unable to find proofs for some relatively simple problems. At the same time incomplete selection functions often make more cunning selections resulting in shorter proofs. Currently, VAMPIRE has 23 selection functions, of which 13 are complete. The list of all selection functions will be included in VAMPIRE's reference manual.

2.7. Reasoning with limited time

One of the most powerful features of VAMPIRE is the *limited resource strategy* (LRS) which is intended for efficient reasoning when a time limit for solving a particular problem is imposed. The main idea of this strategy is the following. The system tries to identify which clauses have no chance to be processed by the time limit at all and discards these clauses. Such clauses are called *unreachable*. Note that the notion of unreachable clause is fundamentally different from the notion of a redundant one: redundant clauses are those that can be discarded without compromising completeness at all, the notion of unreachable clauses makes sense only in the context of reasoning with limited resources. Both persistent and newly generated clauses can be identified as unreachable and discarded.

When the system starts a proof search on a problem, it is also given a time limit as an argument. The system keeps track of statistics on the use of resources during the proof search. The main resource measure is the average time spent by processing each clause selected for inferences. From time to time the system tries to estimate how the proof search statistics would develop towards the time limit and, based on this estimation, identify unreachable clauses. The limited resource strategy is described in detail in [27,28].

2.8. Negative equality splitting

If s is a ground term, a clause $s \neq t \vee C$ can be replaced by the following two clauses: $p(s)$ and $\neg p(t) \vee C$, where p is a new predicate symbol. This allows us to process s and the rest of the clause separately which leads, apart from other things, to the following addition-instead-of-multiplication effect. Suppose we are trying to resolve $s \neq t$ by paramodulating into s and t. For simplicity, we assume that C is empty, all paramodulations are done from unit clauses, and none of the considered paramodulations is a demodulation. Suppose that paramodulations into s produce m different versions, $s_1, \ldots, s_m$, of the term s. Likewise, suppose that paramodulations into t produce n different versions, $t_1, \ldots, t_n$, of the term t. Paramodulations into s are done independently of those into t, so, without *negative equality splitting* we would have to keep $m{\cdot}n$ clauses $s_i \neq t_j$, where $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$. Negative equality splitting allows us to keep only $m{+}n$ clauses, namely $p(s_i)$ and $\neg p(t_j)$. Moreover, having done negative equality splitting for the term s, we can replace any other clause of the form $s \neq t' \vee C'$ by $\neg p(t') \vee C'$.

3. Outline of architecture

The theoretical notion of saturation discussed in the previous section is too abstract to be immediately implemented. When designing a prover one needs a more elaborate scheme of the saturation process that clarifies how the search state is represented, how the available inferences are identified and how the next performed inference is chosen between them. In this section we show how these questions are addressed in the architecture of VAMPIRE. We outline the structure of VAMPIRE's kernel, describe functionality of its components and show how they work together in one procedure.

3.1. Given-clause algorithm

The first successful attempt to devise a practical procedure for saturation was made in OTTER [15] and some of its predecessors [13]. The main idea behind the proposed scheme, called the *given-clause algorithm*, is to implement selection of inferences by selecting one of the participating clauses. In order to illustrate this idea it is enough to speak only about generating inferences, so the algorithm presented in Fig. 1 performs no simplification steps.

The set of all clauses constituting the saturation state is divided into two sets. Clauses from the first set are called *active* and available for generating inferences. Clauses in the second set are called *passive* and only available for simplifications. These clauses are waiting their turn to become active. At every iteration of the algorithm one of the passive clauses is selected by the

```
var active, passive: sets of clauses
var current, new : clause
active := ∅
passive := set of input clauses
while passive ≠ ∅ do
  current := select(passive)
  passive := passive − {current}
  active := active ∪ {current}
  new := infer(current, active)
  if new contains empty clause
    then return provable
  passive := passive ∪ new
od
return unprovable
```

Fig. 1. Given-clause algorithm.

selection function *select*. After being added to *active*, the selected clause *current* is submitted to the inference engine represented by the function *infer* in the algorithm. This function constructs all clauses that can be inferred from *current* and any other active clauses. Generating an empty clause indicates refutability of the input clause set. Otherwise, we add the generated clauses to *passive* and proceed with selecting the next clause.

3.2. Saturation with simplification: the OTTER algorithm

Our goal now is to show how simplifications are integrated into the given-clause algorithm. Figure 2 shows schematically how saturation with simplification is done in VAMPIRE's kernel. In this picture, the boxes denote operations performed on clauses. The rounded boxes denote sets of clauses. The thin ones correspond to the sets that typically contain very few clauses, while the thick ones correspond to the sets that can grow to a substantial size. The arrows reflect the information flow for different operations. The thin solid arrows show the movements of clauses between sets and from operations to the sets. A dashed arrow from a set to an operation says that the operation depends on the clauses from the set. A broad arrow from an operation to a set indicates that the operation modifies the set by removing or replacing some clauses. Since the used scheme is a slightly modified version of the main loop used in the OTTER prover [15], we call it the OTTER algorithm.

When the component that implements generating inferences receives the selected clause, it starts deriving consequences of this clause and clauses from *active*. The newly generated clauses are placed into the variable *new*. Every new nonempty clause is now subject to redundancy tests and simplifications. For some of them only the clause itself is needed, others involve clauses from the current database, i.e., *active* ∪ *passive*. All the steps are performed until the clause either cannot be simplified any more with the available simplification rules or has been identified as redundant. This phase of the saturation procedure is called *forward simplification* and corresponds to the arrows labeled by 5 in Fig. 2.

The new clause which is completely simplified by the clauses from *active* ∪ *passive* and is nonredundant w.r.t. these clauses, is identified as retained and added to the set *retained*. Now it can itself be used to simplify the clauses in the current database *active* ∪ *passive*. This phase of the procedure is called *backward simplification*. In the current version of VAMPIRE it consists of *backward subsumption* and *backward demodulation*. During backward subsumption we try to identify and delete all clauses in *active* ∪ *passive* that are subsumed by *retained*. If *retained* contains a unit equality, the backward demodulation procedure uses it as a rewrite rule to simplify clauses from *active* ∪ *passive*. The simplified versions of these clauses are passed to the variable *new* and processed in the same way as those produced by generating inferences. Different procedures can be used to realize the presented scheme. Our graphical description, unlike the more traditional one used Fig. 1, allows us to show the essence of the algorithm without losing generality.

3.3. DISCOUNT saturation algorithm

One of the main design goals behind the OTTER algorithm is to keep the current database of clauses always completely reduced w.r.t. the available simplification techniques. This comes at a certain cost. Typically the set of passive clauses grows quite quickly and at some point the time spent on search for applicable simplifications involving passive clauses begins to dominate the time spent on other operations. Thus, giving higher priority to generating inferences may be beneficial for solving some problems.

In order to realize this idea in VAMPIRE we implemented another modification of the given-clause algorithm depicted in Fig. 3. We call this procedure *the DISCOUNT algorithm* since it originates from the DISCOUNT prover [1,7]. Unlike in the OTTER algorithm, the passive clauses are no longer used for simplifications. Once a new clause has been generated it

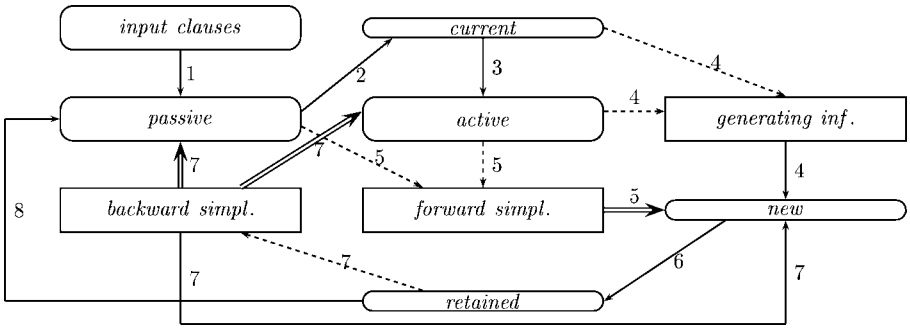


Fig. 2. OTTER algorithm.

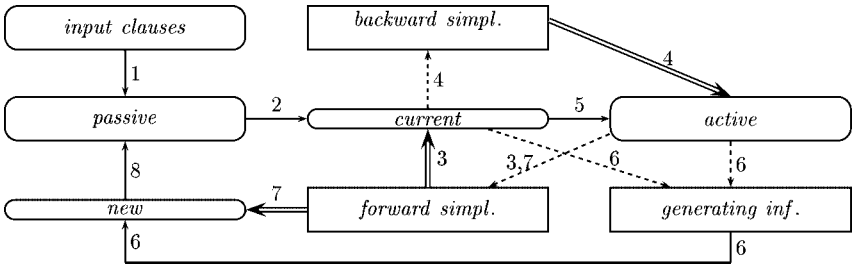


Fig. 3. DISCOUNT algorithm.

can be forward simplified but the simplifications are now allowed to use only the active clauses. We still want to keep *active* completely reduced. For this purpose, when the current clause has been selected from *passive*, it is again subject to forward simplification, i.e., it can be subsumed or rewritten by some active clauses. If we eventually decide to make the clause active, it is used for simplifying *active* before being used in generating inferences. Again, the backward simplification step does not involve any clauses from *passive*.

3.4. Kernel components and their functionality

The description of the used saturation procedures given so far is still too abstract to be immediately realized in a prover. The purpose of this subsection is to show how the abstract procedures are mapped into a practical implementation. We start from describing functionality of the main structural components of VAMPIRE's kernel.

3.4.1. Inference dispatcher

The *inference dispatcher* is the component responsible for performing generating inferences. Passing the selected clause to the inference dispatcher initiates a new iteration of the main loop. The inference dispatcher analyses the clause to detect what types of inferences should be applied according to what literals in the clause are selected. For example, if *current* contains the clause $\underline{p(a,b)} \vee q(c)$ (underlining a literal means that it is selected), the inference dispatcher decides that paramodulation can be done into $p(a,b)$ as well as resolution with this literal.

The number of possible inferences between the current clause and *active* can be quite large. Generating all the consequences at once is not very practical since this would require storing huge sets of temporary clauses. Instead, communicating the newly generated clauses to the main loop is done in a *lazy* manner. The inference dispatcher starts searching for the next possible inference only when it is asked to do so by the main loop. If it fails, it indicates that all possible inferences have been tried with the current clause.

3.4.2. New-clause buffer

The *new-clause buffer* is a structure for temporary storage and processing of the newly generated clauses. When the inference dispatcher or backward simplification infers a new clause, it is assembled in a symbol-by-symbol manner in the buffer. The literals of the new clause are communicated one by one to the buffer. Communicating a literal is done by communicating one by one all its symbols. Along with the collection process we perform certain operations with the clause. First of all, the size and depth of the clause is checked. When the next symbol of the clause is communicated

to the temporary structure, we check that the size and depth of the clause do not exceed some limits. If they do, the new clause is rejected and its collection is immediately aborted. These checks are not postponed until the whole new clause has been assembled, in order to avoid collection of the rest of the clause. This makes sense because the heaviest part of the job of building a new resolvent is actually done inside the inference dispatcher when the literals of the premise clauses are traversed in presence of a substitution. Surprisingly, the time spent on this is not ignorable compared to other operations, which is partially explained by the high efficiency of the implementation for the other operations. For the same reason, after assembling a literal L, the tautology check is immediately performed. If $\neg L$ is one of the previously collected literals or L is of the form $t \simeq t$, the whole new clause is identified as redundant and its collection is immediately aborted.

The main purpose of the buffer is to represent the newly generated clause in a way most suitable for forward simplification. The following procedures use the new-clause buffer as their input: factoring identical literals, tautology detection, forward subsumption, forward subsumption resolution, and forward demodulation.

3.4.3. Term sharing

After being forward simplified, the retained new clause is transformed into a representation suitable for its long-term storage as a passive or active clause. The top level of this representation is simple: a clause is represented as a list of its literals and some additional information, e.g., the clause number and its parents. Literals and terms are represented as *perfectly shared tree-like terms*. This reduces memory consumption and allows one to check syntactic equality between terms quickly, as syntactically identical terms are always represented by the same pointer. The concrete representation is discussed in more detail in Section 5.1.2.

3.4.4. Indexes

The major operations used by our proof-search procedures involve search in large sets for terms or clauses satisfying certain conditions. For example, in order to produce resolvents of the clause $\underline{L} \vee C$ the inference dispatcher must retrieve all selected literals in the active clauses that are unifiable with $\neg L$. Likewise, backward subsumption requires scanning both *passive* and *active* for clauses subsumed by the retained new clause. Special datastructures, called *indexes*, are employed to implement such operations efficiently. Generally, a module implementing an indexed operation comes with methods for the datastructure maintenance, such as insertion of a new indexed clause or term and removal of an obsolete one. The interface for retrieval varies depending on the nature of the indexed operation. Sometimes, given a *query* term or clause, we need to find all the objects in the indexed set that are in a specified *retrieval relation* with the query. This is normally done in a *lazy* way: we feed the query into the retrieval mechanism and then ask it to return the retrieved objects one by one until they are all enumerated. For other operations, such as forward subsumption, finding one relevant object is enough. In VAMPIRE we use specialised indexing datastructures for all retrieval relations, so, speaking about different indexes we mean different datastructures as well as different retrieval relations. Here is the list of the main indexes used in VAMPIRE.

The *unification index* provides fast access to all selected literals and subterms of the active clauses that are unifiable with the query literal or term. It is used to speed up the search for resolution and paramodulation inferences.

The *forward subsumption index* is built on top of the database *active* $\cup$ *passive*. This is a very heavily used index. As soon as a clause is assembled in the new-clause buffer, the retrieval from this index is performed in an attempt to find a clause that subsumes it. The forward demodulation steps are also interleaved with forward subsumption calls: if the clause in the buffer has been rewritten, it is immediately checked for redundancy. Additionally, we use the index to implement forward subsumption resolution. For every literal in the new clause, we try to cut it off by temporarily flipping its polarity and subsuming the modified clause by a clause from *active* $\cup$ *passive*.

The *index for retrieval of generalisations*, or *forward matching*, is the core of the forward demodulation mechanism. The indexed set contains all terms t, such that there is some positive unit equality $t \simeq s$ in *passive* $\cup$ *active* such that $s \not\succeq t$. Given a subterm q of the clause in the new-clause buffer which we are going to rewrite, we are interested in finding its generalisations, i.e., such terms t that $t\theta = q$ for some substitution θ. In the simplest case, when all the indexed positive equalities are ordered, it is enough to find only one of the generalisations. Non-ordered equalities $s \simeq t$ can become ordered either way after the application of θ, so both s and t are stored in the index. Since finding a term in the index does not mean that the corresponding equation becomes ordered, the retrieval should be able to enumerate *all* generalisations of the query.

The *index for backward subsumption* is intended to facilitate fast retrieval of all clauses in *active* ∪ *passive* that are subsumed by the retained new clause.

The *index for instance retrieval*, or *backward matching*, is needed when we want to rewrite the database clauses by the retained new clause. The index represents the set of all non-variable subterms of the passive and active clauses. If $t \simeq s$ is the rewrite rule to be used, the retrieval consists in finding all instances of t in the index, i.e., all terms of the form $t\theta$ where θ is a substitution.

The *index for sharing* supports retrieval of terms that are syntactically equal to the query. In Section 5.1.2 we discuss in sufficient detail how this index works for maintaining perfect sharing.

The *variant index for splitting* is used to check whether a component of a new clause which can be split, is a variant of a previously named component. This index is similar to the code trees index used for forward subsumption.

3.4.5. Buffers for backward simplification

It is very difficult to devise efficient indexing techniques that allow one to control how insertion and removal interfere with lazy retrieval. For this reason, nearly all the indexing techniques with lazy retrieval used in VAMPIRE are designed with the following assumption in mind: retrieval in progress cannot be interrupted by maintenance operations. This introduces some complications in the implementation of saturation.

Suppose that a clause generated by the inference dispatcher has been identified as retained. We then start using it for backward subsumption. Every subsumed clause is redundant, it is no longer available for any operations and should be removed from all the indexes. Unfortunately, we cannot do this straight away because some of the indexes are locked by retrieval. For example, if the subsumed clause is active, we cannot remove it from the unification index since the index is being used by the inference dispatcher. We also cannot remove the clause from the index for backward subsumption since it is being used for retrieval too. In order to solve the problem, we introduce a new set of clauses to the saturation procedure which serves as a buffer for storing the backward subsumed clauses. When another clause is backward subsumed it is simply placed in the buffer. When the inference dispatcher finishes processing the current clause and the last of the retained clauses has been used for backward subsumption, we can remove all the clauses in the buffer from the indexes since the indexes are no longer locked by retrieval.

When it comes to backward demodulation, the situation is even more complex. As with the backward subsumption, we need a buffer to store redundant clauses that have been rewritten. But this is not all. When a retained clause goes to *passive*, it is to be integrated into the indexes for passive clauses. This does not apply to the index for backward demodulation since at the moment it is being used for retrieval. To avoid this problem we need another buffer for storing clauses waiting their turn to be integrated into the index for backward demodulation.

3.5. Splitting without backtracking

If the splitting is turned on and it is decided that a simplified clause in the new-clause buffer is to be retained, we try to apply splitting to this clause. First of all, we check whether the clause is splittable at all and, if so, identify its minimal components. A practical algorithm for solving this problem is described in [26]. If the used version of splitting employs *naming*, at the next step we check which components have already been given names and replace these components by the corresponding propositional literals. If after this step the clause can still be split, some components of the clause are given new names as described in Section 2.3. Negative equality splitting is implemented in a similar manner.

4. Implementation of term indexing in VAMPIRE

The speed of processing clauses in a saturation-based prover largely depends on how fast are the operations employing search in large spaces of terms and clauses. The major bottlenecks are the simplification operations, such as forward and backward subsumption and demodulation. Since the very beginning, implementing such operations with very efficient specialised indexing techniques has been one of the cornerstones of the VAMPIRE design. In Section 3.4.4 we discussed functionality of the main indexes. Now we are going to expose some internals of the used indexing techniques.

4.1. Indexing with compilation

The most important indexing schemes in VAMPIRE use, in one way or another, the idea of *compilation* [32]. In very general terms this idea can be described as follows.

Suppose that some piece of complex data, such as a term or a clause, is frequently used as a part of the input of some complex procedure. For example, it can be one of the terms for which we want to establish unifiability, or one of the clause arguments of the subsumption check. The procedures we are interested in normally involve traversing the corresponding datastructure and checking various conditions on its components. Instead of using a general algorithm, we can *specialise* it for this particular term or clause. The specialization can exploit specific properties of the object in order to optimize its behaviour. Typically, it can avoid many tests that the general procedure would perform, since the results of this tests are known in advance. Also, it can rearrange the order in which some operations are performed, sometimes this allows us to reduce the number of costly operations at the cost of performing more of the inexpensive ones. The specialised version of the procedure is represented as a code of some *abstract machine* which is then interpreted every time we need to call the procedure.

In the context of term indexing, the idea of compilation can be applied in two ways. The first one is to compile the query, i.e., to specialize the retrieval procedure every time a new query is submitted. A less obvious approach is to compile the index instead. The latter use of compilation is discussed in more detail in the next subsection.

4.2. Code trees for retrieval of generalisations and forward subsumption

Code trees were introduced in [32]. The main idea is as follows. Suppose we have a procedure *match*(s, t) that checks whether the term t is a generalisation of s, i.e., $t\theta = s$ for some substitution θ. When t is fixed, we *compile* it to get a specialised version *match*$_t(s)$ of the matching test written as a sequence of instructions for some abstract machine. Now, imagine that we do this for all terms contained in our index for instance retrieval. The retrieval can be implemented by sequentially interpreting all these codes on a given query. Normally many of such code sequences have long coinciding prefixes. This suggests that interpreting them should be shared. In order to do this, we combine the code sequences in a tree-like datastructure which is essentially a program for some abstract machine. In addition to the retrieval of generalisations, a composition of such compiled indexes is used in VAMPIRE to implement a more complex and costly operation: forward subsumption with multiliteral clauses.

Concrete procedures for matching, and hence their compilation, depend on how traversal of the argument terms is performed. Therefore, it is important how the terms are represented. The original version of the code trees in [32] works with the tree-like queries. Later we realized that *flatterm* representation [3] of queries is better suited for the purposes of forward simplification. The modified version of code trees is described in [25] together with another important optimization based on rearranging the order of some instructions in the codes.

4.3. Path-indexing with database joins

Implementing backward demodulation requires fast retrieval of instances. VAMPIRE uses for this a technique based on *path-indexing* [21,30] (see also [10,14]). A *path in a term* is a sequence of function symbols interleaved with integer numbers corresponding to arguments of the functions. For example, the term $f(g(X), h(a))$ contains the following paths: $f.1$, $f.2$, $f.1.g$, $f.1.g.1$, $f.2.h$, $f.2.h.1$ and $f.2.h.1.a$. Every position in a term can be identified by a path leading to the subterm in this position. For example, in the term $f(g(X), h(a))$ the subterms X and a are identified by the paths $f.1.g.1$ and $f.2.h.1$.

Path-indexing for retrieval of instances is based on the following fact: every instance $t\theta$ of a term t always contains all paths that are present in t. The standard path-indexing associates with every term path the list of indexed terms containing this path. Given a query term q, the retrieval operation returns the intersection of the term lists associated with the longest paths in q. This set contains all the indexed instances of q, but it can also contain some extra terms. For example, if $q = f(X, X)$, the term $f(a, b)$ can be retrieved, although it is not an instance of q.

In general, if the retrieval operation finds those and only those terms that are identified by the retrieval relation, we say that the indexing technique *provides perfect filtering*. The standard path index obviously does not. The indexing technique used in VAMPIRE [27] achieves perfect filtering by the following changes to the standard path-indexing. The example above hints that perfectness can be achieved by adding some equality checks on subterms of the indexed terms. Although

$f(a, b)$ contains all the paths of $q = f(X, X)$, the terms a and b are in the positions corresponding to occurrences of the same variable X in q. In an instance of q such terms must be equal. To realize this idea, together with the indexed terms associated with the paths, we store their subterms that can be checked for equality upon retrieval. Now, the objects stored with the paths are relations in the same sense as in relational databases and the retrieval process can be described as computing a relational database query. The query is built of joins that correspond to the equality checks on indexed terms, needed to find the intersection, and equality checks of the subterms corresponding to occurrences of the same variable in the query term. In order to speed up the computation of the joins, the relations associated with paths are represented by skip-lists [20].

Although fast retrieval of instances is interesting enough by itself, the idea of using joins allows us to go much further. In VAMPIRE this technique is also used for implementing retrieval with perfect filtering for a much more complex task: backward subsumption with multiliteral clauses. This is done by adding to the path relations another attribute for storing the number of a literal and introducing joins on this attribute.

4.4. Other indexes in VAMPIRE

Various versions of *discrimination trees* [10,14] are used for other indexed operations in VAMPIRE. For this purpose we use a very simple version of *perfect discrimination trees* (see [10] for a definition). They combine fast retrieval of terms syntactically equal to the query, with fast maintenance operations and moderate memory consumption. The inference dispatcher uses a version of discrimination trees specialised for unification. The main used optimization consists of compiling the query term in order to minimise the cost of cycle detection in the substitutions.

5. Nuts and bolts

As any serious scientific engineering problem, implementing an efficient prover requires solving many smaller technical subproblems. It does not take to be a rocket scientist to get these bolts and nuts right, but learning a few such things from other people's experience can speed up the development and help to avoid reinventing the wheel. For the benefit of a reader who is as obsessed with implementation as ourselves, in this section we share a few such recipes.

5.1. Representation of terms and clauses

When a theorem prover is designed, one of the basic decisions to be made is how the central objects, terms and clauses, are going to be represented. Different operations employed in saturation may require different representation of the data.

5.1.1. Clauses for long-term storage

A clause stored as passive or active is represented by a structure that, apart from a list of literals, contains clause number, weight, background and a number of flags. The background field refers to the parent clauses and the inference rules used to produce the clause. This information allows for a fast extraction of the proof when a problem has been solved. The main purpose of the flags is to keep record of what clause sets contain the clause and what indexes it is integrated into. One flag indicates existence of children of the clause. This information is mostly needed when we want to destroy the clause since then it has to be removed from all sets and indexes. If a clause with children is destroyed, we cannot destroy the structure itself since the clause is referred to by the children's background fields and may be eventually used in the found proof.

5.1.2. Perfectly shared tree-like terms

The literals and their subterms in the stored passive and active clauses are represented in the following way. A term is identified by a pointer to a structure. If the term is compound, one of the elements of the structure contains the top function symbol of the term and the other elements are cells containing pointers to the top-level subterms. Structures representing constants and variables do not contain the subterm cells. To illustrate this we depict in Fig. 4 the representation of the term $f(g(a), h(X))$ and $f(h(X), b)$.

Syntactically identical terms, like the first argument $h(X)$ in $f(h(X), b)$ and the second one in $f(g(a), h(X))$ in Fig. 4, are represented by isomorphic structures. By combining isomorphic structures into a single structure we obtain DAG-like structures shown

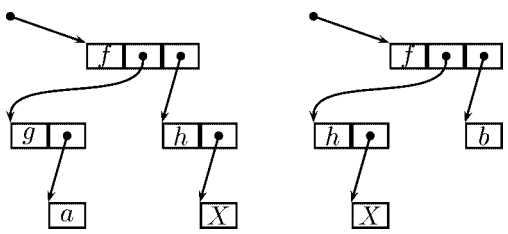


Fig. 4. Tree-like representation of $f(g(a), h(X))$ and $f(h(X), b)$.

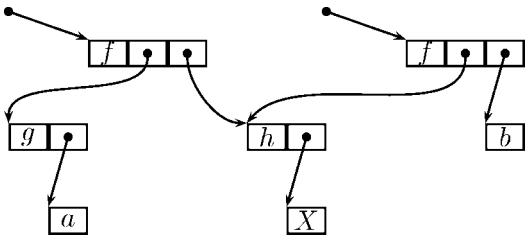


Fig. 5. Shared representation of $f(g(a), h(X))$ and $f(h(X), b)$.

in Fig. 5. Sharing used in VAMPIRE is perfect, i.e., all structures that can be shared, are shared. Such structures can be referred to as *perfectly shared tree-like terms*.

In order to transform a literal from the new-clause buffer into its perfectly shared representation, every subterm t of the literal is examined. If the index for sharing contains a term which is syntactically identical to t, the pointer to the structure representing t is obtained. If this is not the case, we have to create the corresponding structure and integrate t together with the corresponding pointer into the index. Every term in the sharing index is accompanied by a counter of references to it, i.e., the number of cells in other terms and clauses containing the pointer to the structure. When, for some reason, we decide that a database clause is no longer needed, the counters corresponding to its subterms are decremented. If the counter for a term t becomes zero, it indicates that the term can be removed from the index and the corresponding structure can be destroyed to release memory.

5.1.3. New-clause buffer and flatterms

The shared representation of stored clauses is mostly aimed at minimizing memory consumption. Representation of the clauses assembled and processed in the new-clause buffer is intended to solve different tasks. The representation should enable very fast assembling of the new clause in the buffer. It is also critical for the representation to support fast traversal of subterms of the new clause, since traversal operations are heavily used in forward simplification. The main part of the new-clause buffer is a list of literals represented as *flatterms*. In the *array-based* flatterm representation adopted in VAMPIRE, a term t is represented as an array. Every cell of the array contains a symbol from the term and a pointer to the end of the subterm starting with this symbol. For example, the term $f(x, g(a))$ is represented by the structure depicted in Fig. 6. Left-to-right traversal of such terms, used in many operations, is performed by following the pointers in the cells or incrementing a pointer to a cell. Unlike traversal of tree-like terms, it does not involve recursion or explicit use of a stack.

5.1.4. Passive clauses in the DISCOUNT algorithm

When the DISCOUNT algorithm is used as the saturation procedure, slightly different requirements are imposed on the representation of passive clauses. The set of passive clauses changes much quicker than in the OTTER algorithm. The main design goal now is to provide fast ways of adding a new clause to *passive* and removing several heaviest clauses from it. The representation used in VAMPIRE addresses this problem by using *packed* representation of clauses. The whole clause with all its elements is allocated in one chunk of memory of sufficient size. The literals are represented simply as arrays of their symbols. This allows for a very fast collection of a new passive clause. Destroying a clause is also very quick, we simply release the occupied chunk of memory. In the next subsection we discuss in more detail a memory management scheme suitable for such a clause representation.

5.2. Memory management

Intensive use of dynamic datastructures, such as stored clauses and various indexes, requires efficient memory management. Allocation of new pieces of data and recycling of obsolete ones are the main operations that should be supported. For these purposes VAMPIRE uses the standard *free-list* allocation discipline. A big chunk of memory is requested from the operating system and the allocation requests from different datastructures are satisfied by slicing the chunk into pieces of requested sizes. When a part of a datastructure is destroyed, these pieces are returned to the memory manager for future reuse. The pieces are placed into special lists, called *free-lists*. Every such list contains pieces of one particular size. When later the memory manager is asked to allocate a piece of memory of a certain size, it first checks if the free-list corresponding to this size contains a piece that can be used for allocation.

5.2.1. Buffered allocation

The free-list allocation discipline serves well the basic needs of memory management. However, solving hard problems requires exploration of large search spaces which is reflected by a very fast growth of memory consumption by clause storage and indexes. The nature of the main operations performed with the datastructures requires them to be stored in the main memory of the computer. The available amount of

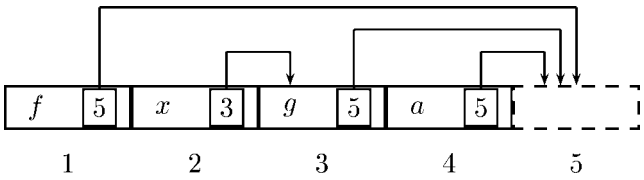


Fig. 6. Flatterm structure for $f(x, g(a))$.

main memory is always restricted by a certain limit. To be practical, a saturation procedure must employ special mechanisms to be able to work within such a limit.

Suppose we are running the OTTER algorithm and at some point the memory manager is unable to fulfill an allocation request since all appropriate free-lists are empty and the limit on the memory that can be requested from the system is exhausted. In this situation VAMPIRE tries to obtain free memory by recycling the heaviest of the passive clauses. There is a serious obstacle for doing so. Passive clauses are integrated in various indexes and must be removed from them upon destruction. This cannot be done immediately since at the moment of performing the operation that requested allocation some of the indexes can be used by retrieval or maintenance operations. Theoretically it is possible to roll back these operations, do recycling and resume the interrupted operations again. Another option is to design all the indexes with a possibility of interrupting some operations by deletion. In practice, it is very difficult to devise efficient indexing techniques with such properties, so VAMPIRE adopts a different scheme called *buffered allocation*.

Suppose the memory limit is 100 Mb. A fraction of the available memory, for example 10 Mb, is designated as a buffer. The system can only use some amount of memory above the 90 Mb of the working space temporarily, it must be reimbursed as soon as we can remove the heaviest passive clauses from the indexes without interfering with other operations. For better precision, we would like to reimburse the buffer deficit by deallocating pieces of memory of the same sizes and in the same numbers as those pieces temporarily borrowed from the buffer. In the extreme case, we destroy as many passive clauses as needed to do so. This often leads to the following undesirable effect. If the size of a piece of memory taken from the buffer is rare, it will take many or even all passive clauses to be destroyed before we reimburse this piece. So, in practice we impose a certain limit on how much memory can be returned to the memory manager when reimbursing the buffer deficit.

5.2.2. *Page-queue allocation for passive clauses in the DISCOUNT algorithm*

The free-list allocation discipline works well when the sizes of the allocated memory pieces do not vary too much. If they do, a piece of memory placed in a free-list has a good chance not to be reused. If, additionally, the allocated objects are large, the memory loss can be too big to be ignored. We encountered this problem when implementing storage of passive clauses for the DISCOUNT algorithm. In this procedure a passive clause is placed in a single chunk of memory. These pieces are quite big and vary in size.

To cope with this problem we adopted the following scheme for storing passive clauses. Instead of allocating every clause in a separate piece of memory, we use a discipline based on a notion of *page*. Pages are pieces of memory of the same fixed size, therefore, their allocation and reuse cause no problem. The page size is big enough to allocate several passive clauses. The structure representing the set of passive clauses, consists of several *page-queues* that are double-linked lists of pages. For every value of clause weight, there is a page-queue for allocating clauses of this weight. Notice that the weight of a clause need not be related in any way to the clause size. Clause selection decides which page-queue should be used and reads the first clause from the top page of the queue. The corresponding piece of memory is marked as freed in the page. When all clauses in the page have been read, the page is removed from the queue and returned to the memory manager. Getting rid of the heaviest clauses, needed to free space for other data, is even simpler: we recycle a whole bunch of clauses by removing the last page from the queue corresponding to the biggest weight.

5.3. *Checking ordering constraints*

The superposition calculus used in VAMPIRE restricts application of paramodulation by imposing ordering conditions on the participating terms. When paramodulating from the equality literal $s \simeq t$ with a substitution θ, we have to check that $t\theta \not\succeq s\theta$. Similar restrictions are imposed on paramodulation into an

equality literal. Demodulation requires even stronger restrictions on the used substitution. When $s \simeq t$ is used to rewrite some term $s\theta$ into $t\theta$, we must check that $s\theta \succ t\theta$ holds. If the equality is pre-ordered, then this condition is guaranteed. But very often the condition can only be checked when the substitution θ is known, for example produced by a retrieval from an index. In this subsection we briefly show how checking such constraints on substitutions is done in VAMPIRE.

5.3.1. Specialization of comparison

Suppose that we want to check the following condition: given two terms s and t, and a substitution θ, does $s\theta \succ t\theta$ hold? When this check is done repeatedly for the same terms s and t (but maybe different substitutions) we can specialize the check by producing a condition on θ which is simpler than verifying $s\theta \succ t\theta$ straightforwardly. To give the reader a flavor of how the specialization is done, we consider an example. A thourough description of this technique will be given in [22].

Let

$$s = f(x_1, f(x_1, f(x_2, f(x_3, x_4)))) \text{ and}$$
$$t = g(x_4).$$

We assume that $w(f) = 1$, $w(g) = 10$, the minimal weight of a constant in the signature is 1, and $f \gg g$. First, note that $s\theta \lll t\theta$ always holds. Therefore, $s\theta \succ t\theta$ if and only if

1. $|s\theta| >_w |t\theta|$; or
2. $|s\theta| \geqslant_w |t\theta|$.

Since the second condition implies the first, $s\theta \succ t\theta$ is equivalent to $|s\theta| \geqslant_w |t\theta|$, that is

$$|f(x_1, f(x_1, f(x_2, f(x_3, x_4))))\theta| \geqslant_w |g(x_4)\theta|. \quad (1)$$

By using the definition of a weight, we can rewrite (1) into

$$4 \cdot w(f) + 2 \cdot |x_1\theta| + |x_2\theta| + |x_3\theta| + |x_4\theta| \geqslant_w w(g) + |x_4\theta|. \quad (2)$$

By evaluating the weight function w on f and g and using elementary properties of $|\ldots|$ and $\geqslant$ we can rewrite (2) into

$$2 \cdot |x_1\theta| + |x_2\theta| + |x_3\theta| \geqslant_w 6. \quad (3)$$

Checking this condition is easier than checking the original $|s\theta| \geqslant_w |t\theta|$, mainly because it does not require traversing the terms $s\theta$ and $t\theta$. Moreover, we do not have to consider the substitution for the variable x_4. Additionally, the condition (3) can be checked incrementally. First, we compute $2 \cdot |x_1\theta|$. If it happens that $2 \cdot |x_1\theta| \geqslant_w 3$, we can safely conclude that $|s\theta| \geqslant_w |t\theta|$ without computing $|x_2\theta|$ and $|x_3\theta|$. Otherwise, at the next step we compute $|x_2\theta|$ and add it to $2 \cdot |x_1\theta|$. Again, if $2 \cdot |x_1\theta| + |x_2\theta| \geqslant_w 4$, we can stop the procedure and conclude that $|s\theta| \geqslant_w |t\theta|$.

5.3.2. Specialization of lexical comparison

If the condition $|s\theta| \geqslant_w |t\theta|$ has been established, we have to compare the terms $s\theta$ and $t\theta$ lexicographically. The only optimization used in VAMPIRE at the moment for this check is as follows. When the terms s and t are preordered lexicographically, the result of comparing $s\theta$ with $t\theta$ is known a priori. For example, the condition $s \lll t$ guarantees that $s\theta \lll t\theta$ for all θ. In all other cases, we simply check $s\theta \lll t\theta$. However, in the future we may consider stronger specializations intended to minimize the cost of traversing the s- and t-parts of the terms $s\theta$ and $t\theta$.

5.3.3. Implementation

In VAMPIRE all operations involving checks of ordering constraints fall into the following scheme. There is a query term and an indexed database of terms. We perform retrieval of terms related to the query from the database, together with the corresponding substitutions. In these operations, checking the constraints is done in two modes.

In one mode, constraints are associated with the database terms. After each successful retrieval, we have to check the constraint associated with the retrieved term on the corresponding substitution. For example, suppose that an unoriented equation $s \simeq t$ is to be used as a rewrite rule in forward demodulation. The terms s and t are stored in the forward matching index together with the corresponding simplified constraints $s\theta \succ t\theta$ and $t\theta \succ s\theta$. A simplified constraint is represented as a pair of linear polinomials and some value indicating the result of lexical comparison. For example, the simplified form $2 \cdot |x_1\theta| + |x_2\theta| + |x_3\theta| \geqslant_w 6$ of the constraint $f(x_1, f(x_1, f(x_2, f(x_3, x_4))))\theta \succ g(x_4)\theta$ can be represented by the pair $(2 \cdot x_1 + x_2 + x_3, 6)$ together with some value, representing the fact $f(x_1, f(x_1, f(x_2, f(x_3, x_4)))) \lll g(x_4)$.

The linear polynomials are represented as linked lists where each node contains one monomial. Checking the constraint against the corresponding substitu-

instruction	arguments	semantics
Init		$W := 0$
Add	$2, x_1$	$W := W + 2 \cdot \lvert x_1\theta \rvert$
SuccessIfRGreater	3	return success if $W >_w 3$
Add	$1, x_2$	$W := W + \lvert x_2\theta \rvert$
SuccessIfRGreater	4	return success if $W >_w 4$
Add	$1, x_3$	$W := W + \lvert x_3\theta \rvert$
SuccessIfRGreater	5	return success if $W >_w 5$
Fail		return failure

Fig. 7. Intructions for performing constraint-checking.

tion amounts to a straightforward application of the incremental procedure described above. Our main optimization target here is to reduce memory consumption. For this purpose, we employ structure sharing for parts of the lists representing linear polynomials.

In another mode, a constraint may be associated with the query. In this case, the constraint itself remains unchanged during retrieval for one fixed query, i.e., potentially for many substitutions corresponding to different terms retrieved from the database. For example, if the equation $s \simeq t$ is to be used as the rewrite rule in backward demodulation, the retrieval will enumerate all substitutions θ, such that $s\theta$ is in the database, and the constraint $s\theta \succ t\theta$ will be tested for every such θ. In this situation we can afford deeper processing of the constraint, since the associated overhead is amortized and the effect of better specialization multiplies with the number of examined substitutions. Moreover, we are much less constrained in space consumption due to the fact that the specialized constraint need not be stored permanently. This again leads us to the idea of using compilation (cf. Section 4.1).

The compilation process consists of two phases. First, we simplify the given constraint in the same way as it is done above. At the second phase, we take the generic incremental procedure for checking different simplified constraints on different subsitutions, and specialize it for dealing with this particular simplified constraint. The specialized version of the procedure is written as a code for an abstract machine, and then interpreted on different input substitutions. To give the reader a flavor of what the code may look like, in Fig. 7 we show the result of compilation for the constraint $f(x_1, f(x_1, f(x_2, f(x_3, x_4))))\theta \succ g(x_4)\theta$ from the example above.

6. Vampire for the user

The distribution package for VAMPIRE v2.0 consists of two Unix programs: kernel and preprocessor.

The kernel reads a set of input clauses in the TPTP format from one or several files and tries to refute it. Proof search process is configured by setting some command line options. There are options for adjusting the basic calculus, simplification methods, saturation procedure, input and output, etc. The list of all available options with necessary explanations can be found in the simple reference manual that comes with the system. For an unexperienced user the list of about fifty different options may look terrifying, so the reference manual contains a sample file with option settings that are good enough for a fast kick-off. The interactive facilities are minimal: all one can do is to interrupt the kernel run to obtain some statistics on the current search state, such as sizes of the main clause sets, memory usage etc. Another possibility to monitor the proof search process is to make the system print out the newly generated, passive or selected clauses.

If a proof is found, the kernel prints it in a human-readable format. Extracting the proof is very easy since with all persistent clauses we store information about their ancestors and rules used for their inference. Additionally, the proof can be printed in a computer-checkable format. Every step of the proof is written to a special log file as a Prolog fact. This file can be later processed by simple Prolog scripts, in particular for proof checking or transformation. Apart from the proofs, the log files contain a lot of different information about proof attempts, including the proof search configuration and statistics, which is invaluable for analyzing results of many runs of the prover on big test suites.

Starting from version 2.0 VAMPIRE comes with a component that transforms first-order formulas into clausal normal form, performs certain preprocessing of the obtained clause set and, optionally, analyses the result to generate a Unix command line for running the kernel with heuristically chosen options. The clausal normal form algorithm of VAMPIRE was implemented

only recently, and essentially uses the naive translation. We expect it to be improved considerably by the next year. However, a few important optimizations are used. Firstly, the clausifier expands equivalences depending on their polarity. Secondly, the clausifier tries to identify definitions of predicates and expand or remove some of them. This optimization sometimes enables us to generate clausal forms which are considerably smaller than those produced by FLOTTER [35]. After clausification, preprocessing continues on generated clauses, which can result in further optimizations, for example, removal or expansion of function definitions, tautology deletion and some others.

The preprocessor of VAMPIRE takes a problem formulation in the TPTP format, which may containing both formulas and clauses, and produces a set of clauses, again in the TPTP format. As an option, the preprocessor can also return a sequence of strings which can be used as command lines for running the VAMPIRE kernel. This strings contain heuristically chosen parameters for the kernel. This feature enables us to combine the preprocessor with the kernel using scripts, for example written in Perl.

7. Ongoing development and future

The current work on VAMPIRE is going in two directions: (i) making it easier to use and more flexible, and (ii) making it faster.

Flexibility. To make VAMPIRE easier to use and more flexible, a few radical changes are required. The user interface has to be changed completely, and obsolete restrictions removed (for example, the restriction on clauses to have at most 31 literals). Since the user will operate with the preprocessor, the preprocessor will have to implement processing of user-supplied parameters, and changing them into parameters for the kernel. A proof-checker will be implemented as a third component.

To make VAMPIRE more flexible, we are going to diversify literal and clause selection possibilities and implement new simplification orders (first the standard Knuth-Bendix order and then the lexicographic path order).

Speed. We are improving our indexing techniques. This work goes in two directions: (i) improvement of the existing indexing techniques, (ii) research on and implementation of new specialised indexing techniques. For the existing indexing techniques, we expect considerable improvements for clause indexing (especially for NP-complete retrieval conditions, such as subsumption). We are also making experiments on polishing the existing term indexing techniques, where some improvements are still possible. The comparison of performance of different techniques has now become possible due to the recent development of the COMPIT methodology [18].

As for completely new indexing techniques, we are interested in providing specialised methods of term indexing in presence of symbols with special algebraic properties, such as commutativity of functions and symmetry of predicates. Of a special interest for us is indexing modulo AC, considered within the framework of building in AC.

Future. Some of our future plans are outlined in [34]. They include features which could enable the use of VAMPIRE for applications which are beyond the capabilities of the state-of-the-art provers. These are built-in theories (such as AC or arithmetic), intelligent work with definitions, stratified resolution [5,6], work with inductive definitions, finite domain reasoning, and some others.

8. Summary of features

For the reader's convenience, we compiled a list of some design and implementation features into Table 1. We evaluate approximate usefulness and implementation costs of these features.

9. Vampire in CASC-JC

At CASC-JC, VAMPIRE was represented by several versions. Since most of the provers at CASC are extremely finely-tuned to the CASC rules and the set of eligible problems [33], it seems very difficult (if possible at all) to be a winner using a prover which is not tuned at all. So we decided to present a finely-tuned system in the most prestigious MIX division. But we were also interested in seeing how strong a non-tuned version of VAMPIRE is. So in all other categories VAMPIRE was presented by a its non-tuned version. The non-tuned version was also presented in the MIX division. We will refer to the tuned version as VAMPIRE-JC.

In the MIX division VAMPIRE-JC solved the same number of problems as E-SETHEO, but it was also discovered that the competition script did not discover a proof of SYN036-1 found by VAMPIRE. It seems that

Table 1
Summary of features with estimates of usefulness and implementation cost

Feature	Usefulness			Implementation cost		
	moderately useful	very useful	crucial	easy	feasible	hard
Splitting without backtracking (2.3)		•		•		
Naming for splitting (2.3)			•	•[a]		
Parallel exploration of branches in splitting (2.3)	•			•		
Branch demodulation (2.5)		•		•		
Negative equality splitting (2.8)	•			•		
Forward subsumption resolution (2.5)			•	•[b]		
Limited Resource Strategy (2.7)			•		•	
Perfect sharing of terms (5.1.2)			•	•		
Code trees for forward matching and subsumption (4.2)			•			•
Path indexing with database joins (4.3)			•			•
Buffered allocation (5.2.1)			•	•		
Page-queue allocation for passive clauses in the DISCOUNT algorithm (5.2.2)			•	•		
Non-recursive Knuth-Bendix order (2.4)		•		•		
Compiling ordering constraints (5.3)			•		•[c]	

[a] Provided that there is a good technique for variant indexing.
[b] When implemented on the basis of code trees for forward subsumption.
[c] Several versions are needed for different purposes, although each of them alone is relatively easy to implement.

VAMPIRE was interrupted while generating and printing the proof, so when the script terminated VAMPIRE, the proof was already in the buffer. To avoid complications, we proposed to divide the first place between VAMPIRE-JC and E-SETHEO. When we analyzed the results later, we discovered that VAMPIRE also rejected input clauses in four problems from the natural language processing domain (NLP), since they contained clauses with more than 31 literals, due to an obsolete feature of VAMPIRE which will be removed in the next version.

VAMPIRE-JC recognised some particular problem sets characterized by quantitative parameters, like the size of the signature. VAMPIRE only divided the problems in five categories depending on the presence of equality, function symbols, and non-Horn clauses. For example, VAMPIRE tried to solve all Horn problems with equality using the same strategy and set of parameters. We did not have much time for tuning VAMPIRE-JC or for selecting the best parameters in VAMPIRE. We believe that otherwise the performance of both versions in the MIX division could have been much better. For example, among the 27 problems not solved by VAMPIRE-JC at least 12 more could be solved by an appropriate parameter setting.

It is worth noting that both E and E-SETHEO do not produce proofs, so among the provers which produce a proof, VAMPIRE was ahead of all provers except for VAMPIRE-JC.

The behaviour of VAMPIRE in the FOF division could have been better, have we had more time for improving the clausifier. We had a choice between clausifying by FLOTTER, as the last year, and using the native clausifier, but decided to use the native one, though we were aware of the fact that we would most likely lose to E-SETHEO, which uses FLOTTER for clausification. We decided that participating with the native clausifier would give us a better motivation for improving the clausifier after CASC.

In the UEQ category VAMPIRE's results were not very impressive. This is due to several factors: no tuning, absence of good lookahead rules, inappropriate simplification orders, and unintelligent clause selection.

References

[1] J. Avenhaus, J. Denzinger and M. Fuchs, DISCOUNT: a system for distributed equational deduction, in: *Proceedings of the 6th International Conference on Rewriting Techniques and Ap-*

plications (RTA-95), J. Hsiang, ed., Vol. 914 of *Lecture Notes in Computer Science*, Kaiserslautern, 1995, pp. 397–402.

[2] L. Bachmair and H. Ganzinger, Resolution theorem proving, in: *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, eds, Vol. I, Elsevier Science, Ch. 2, 2001, pp. 19–99.

[3] J. Christian, Flatterms, discrimination nets, and fast term rewriting, *Journal of Automated Reasoning* **10**(1) (1993), 95–113.

[4] H. de Nivelle, *Splitting Through new Proposition Symbols*, Vol. 2050 of *Lecture Notes in Artificial Intelligence*, Havana, Cuba, 2001.

[5] A. Degtyarev, R. Nieuwenhuis and A. Voronkov, Stratified resolution, *Journal of Symbolic Computations* (2002), 1–24 (to appear).

[6] A. Degtyarev and A. Voronkov, Stratified resolution, in: *17th International Conference on Automated Deduction (CADE-17)*, D. McAllester, ed., Vol. 1831 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, Pittsburgh, 2000, pp. 365–384.

[7] J. Denzinger, M. Kronenburg and S. Schulz, DISCOUNT – a distributed and learning equational prover, *Journal of Automated Reasoning* **18**(2), (1997), 189–198. *citeseer.nj.nec.com/denzinger96discount.html.

[8] N. Dershowitz and D. Plaisted, Rewriting, in: *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, eds, Vol. I, Elsevier Science, Ch. 9, 2001, pp. 533–608.

[9] H. Ganzinger, R. Nieuwenhuis and P. Nivela, *The Saturate System*, Max-Planck Institute für Informatik, 2001. *http://www.mpi-sb.mpg.de/SATURATE/Saturate.html.

[10] P. Graf, *Term Indexing*, Vol. 1053 of *Lecture Notes in Computer Science*, Springer Verlag, 1996.

[11] D. Knuth and P. Bendix, Simple word problems in universal algebras, in: *Computational Problems in Abstract Algebra*, J. Leech, ed., Pergamon Press, Oxford, 1970, pp. 263–297.

[12] K. Korovin and A. Voronkov, Knuth-bendix ordering constraint solving is NP-complete, in: *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001*, F. Orejas, P.G. Spirakis and J. van Leeuwen, eds, Vol. 2076 of *Lecture Notes in Computer Science*, Springer Verlag, 2001, pp. 979–992.

[13] E.L. Lusk, Controlling redundancy in large search spaces: Argonne-style theorem proving through the years, in: *Logic Programming and Automated Reasoning. International Conference LPAR '92*, A. Voronkov, ed., Vol. 624 of *Lecture Notes in Artificial Intelligence*, St. Petersburg, Russia, 1992, pp. 96–106,

[14] W.W. McCune, Experiments with discrimination-tree indexing and path indexing for term retrieval, *Journal of Automated Reasoning* **9**(2) (1992), 147–167.

[15] W.W. McCune, OTTER 3.0 reference manual and guide, Technical Report ANL-94/6, Argonne National Laboratory, 1994.

[16] P. Narendran, M. Rusinowitch and R. Verma, RPO constraint solving is in NP, in: *Computer Science Logic, 12th International Workshop, CSL '98*, G. Gottlob, E. Grandjean and K. Seyr, eds, Vol. 1584 of *Lecture Notes in Computer Science*, Springer Verlag, 1999, pp. 385–398.

[17] R. Nieuwenhuis, Simple LPO constraint solving methods, *Information Processing Letters* **47** (1993), 65–69.

[18] R. Nieuwenhuis, T. Hillenbrand, A. Riazanov and A. Voronkov, On the evaluation of indexing techniques for theorem proving, in: *First International Joint Conference on Automated Reasoning, IJCAR 2001*, R. Goré, A. Leitsch and T. Nipkow, eds, Vol. 2083 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, 2001, pp. 257–271.

[19] R. Nieuwenhuis and A. Rubio, Paramodulation-based theorem proving, in: *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, eds, Vol. I, Elsevier Science, Ch. 7, 2001, pp. 371–443.

[20] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, *Communications of the ACM* **33**(6) (1990), 668–676.

[21] R. Ramesh, I.V. Ramakrishnan and D.S. Warren, Automata-driven indexing of Prolog clauses, in: *Seventh Annual ACM Symposium on Principles of Programming Languages*, San Francisco, 1990, pp. 281–291.

[22] A. Riazanov, Implementing an efficient theorem prover, PhD thesis, University of Manchester, 2002 (to appear).

[23] A. Riazanov and A. Voronkov, Splitting without backtracking, Preprint CSPP-10, Department of Computer Science, University of Manchester, 2001. *http://www.cs.man.ac.uk/preprints/index.html.

[24] A. Riazanov and A. Voronkov, Limited resource strategy in resolution theorem proving, Preprint CSPP-7, Department of Computer Science, University of Manchester, 2000. *http://www.cs.man.ac.uk/preprints/index.html.

[25] A. Riazanov and A. Voronkov, Partially adaptive code trees, in: *Logics in Artificial Intelligence. European Workshop, JELIA 2000*, M. Ojeda-Aciego, I.P. de Guzmán, G. Brewka and L.M. Pereira, eds, Vol. 1919 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, Málaga, Spain, 2000, pp. 209–223.

[26] A. Riazanov and A. Voronkov, Splitting without backtracking, in: *17th International Joint Conference on Artificial Intelligence, IJCAI '01*, B. Nebel, ed., Vol. 1, 2001, pp. 611–617.

[27] A. Riazanov and A. Voronkov, Implementing backward subsumption and search for instances with path indexing and joins, 2002 (in preparation).

[28] A. Riazanov and A. Voronkov, Limited resource strategy in resolution theorem proving, *Journal of Symbolic Computations*, (2002), to appear.

[29] S. Schulz, System abstract: E 0.61, in: *First International Joint Conference on Automated Reasoning, IJCAR 2001*, R. Goré, A. Leitsch and T. Nipkow, eds, Vol. 2083 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, 2001, pp. 370–375.

[30] M. Stickel, The path indexing method for indexing terms, Technical Report 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1989.

[31] G. Sutcliffe and C. Suttner, The TPTP problem library. tptp v. 2.4.1, Technical report, University of Miami, 2001.

[32] A. Voronkov, The anatomy of Vampire: Implementing bottom-up procedures with code trees, *Journal of Automated Reasoning* **15**(2) (1995), 237–265.

[33] A. Voronkov, CASC 16 1/2, Preprint CSPP-4, Department of Computer Science, University of Manchester, 2000. *http://www.cs.man.ac.uk/preprints/index.html.

[34] A. Voronkov, Algorithms, datastructures, and other issues in efficient automated deduction, in: *First International Joint Conference on Automated Reasoning, IJCAR 2001*, R. Goré, A. Leitsch and T. Nipkow, eds, Vol. 2083 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, 2001, pp. 13–28.

[35] C. Weidenbach, B. Gaede and G. Rock, SPASS & FLOTTER. Version 0.42, in: *Automated Deduction – CADE-13*, M.A. McRobbie and J.K. Slaney, eds, Vol. 1104 of *Lecture Notes in Computer Science*, New Brunswick, NJ, USA, 1996, pp. 141–145.

[36] H. Zhang, Implementing contextual rewriting, in: *Conditional Term Rewriting Systems, Third International Workshop*, M. Rusinowitch and J.L. Rémy, eds, Vol. 656 of *Lecture Notes in Computer Science*, Springer Verlag, 1992, pp. 363–377.