Vampire 1.1
(system description)

Alexandre Riazanov and Andrei Voronkov

Computer Science Department, University of Manchester

Abstract. In this abstract we describe version 1.1 of the theorem prover Vampire. We give a general description and comment on Vampire's original features and differences with the previously described version 0.0.

From the very beginning, the main research principle of Vampire was efficiency. Vampire uses a large number of datastructures for indexing terms and clauses. Efficiency is still the most distinctive feature of Vampire. Due to reimplementation of some algorithms and datastructures, Vampire 1.1 is on the average considerably more efficient than Vampire 0.0.

However, the last year many efforts were invested in flexibility: several new inference and simplification rules were implemented, options for controlling the search space added, and new literal selection schemes designed.

For the remaining time before CADE, we are going to concentrate on adding more flexibility to Vampire, both for experienced and nonexperienced users.

1 General description

Vampire is a completely automatic saturation-based theorem prover for first-order logic with or without equality.

Calculi. Two kinds of calculi are implemented:

1. binary resolution with superposition and negative selection;
2. positive and negative hyperresolution, but only for logic without equality.

Saturation algorithm and splitting. Most of the existing first-order theorem provers implement either the OTTER-style saturation algorithm [5] or the DISCOUNT-style algorithm [1] (see [7] for details). Both saturation algorithms are implemented in SPASS [10]. Vampire implements both algorithms and in addition an original algorithm based on the so-called *limited resource strategy* [7]. The DISCOUNT algorithm has been implemented recently. The main feature of this algorithm is that unused clauses are absolutely passive until they are selected for performing inferences, i.e. they are not eligible for any simplifications.

Until recently, SPASS was the only prover implementing a splitting rule on top of a saturation algorithm. To avoid high cost of implementing the standard splitting, we have implemented *splitting without backtracking*. If the search space contains a clause $C \vee D$, where the variables of C and D are disjoint, we replace this clause by two clauses $C \vee p$ and $D \vee \neg p$, where p is a new propositional symbol. There are several options controlling splitting in Vampire 1.1:

1. *Blocking* and *parallel* splitting are obtained by different modifications of the literal selection function to the clauses containing new predicate symbols. These versions allow us to simulate, to some extent, sequential and parallel case analysis.
2. New literals can be used as *names* of clauses. For example, if we split $C \vee D$ into $C \vee p$ and $D \vee \neg p$, we then consider $\neg p$ as the name of C. The next time we encounter a clause $C \vee D'$ such that the variables of C and D' are disjoint, we simply replace this clause by $D' \vee \neg p$.
3. A simplification rule called *branch rewriting* can be used. This rule is essentially simplification by nonunit equalities of the form $s = t \vee P$, where P only consists of the new literals. Such clause can be used to rewrite a clause $C[s\theta]$ into $C[t\theta]$ under the condition $P \subset C[s\theta]$.

Literal selection. Several literal selection strategies are now available. For example, the user may choose selecting only the maximal literals, or first selecting the maximal literals and then trying to change it by negative selection if it gives (heuristically) better selection. We have also added an option to use *inherited negative selection* used in the prover E [8]: after paramodulating into a literal that was selected by negative selection, the rewritten literal is necessarily selected again.

Orderings. Only one kind of term orderings is implemented in Vampire 1.1: a nonrecursive version of the Knuth-Bendix ordering. For two ground terms s, t we have $s \succ t$ if either the weight of s is greater than the weight of t, or the weights are equal but s is lexicographically greater than t. The lexicographic comparison is done on the words obtained by enumerating the symbols of the terms visited in the left-to-right depth-first traversal. For nonground terms, the ordering is defined in a similar way. Apart from some other pleasant properties, this ordering enables efficient approximation of algorithms for solving ordering constraints.

2 Other original features

Precompiled ordering constraints. In version 0.0 rewriting by a unit equality $s = t$ was allowed only when this equality was ordered, i.e. $s \succ t$. In the new version more simplifications by unit equalities can be done due to introduction of constrained rewrite rules. Now, if an equality $s = t$ is not preordered, we can still rewrite a clause $C[s\theta]$ into $C[t\theta]$ provided that $s\theta \succ t\theta$.

In general, the check $s\theta \succ t\theta$ can be expensive. To avoid this expensive check, we generate constraints which encode the sufficient and necessary conditions for $s\theta \succ t\theta$ to hold. For example, for the commutativity axiom $x \cdot y = y \cdot x$ and the Knuth-Bendix ordering $\succ$ instead of checking $x\theta \cdot y\theta \succ y\theta \cdot x\theta$ we can check a simpler but equivalent condition $x\theta \succ y\theta$. For the ordering used in Vampire, the corresponding condition will be a simple lexicographic comparison of x and y.

Commutativity optimization. Special treatment of commutative functors is implemented. Now, every subterm of a generated clause is normalized with respect to commutativity of certain functions. If f is commutative, the term $f(t_1, t_2)$ is replaced by $f(t_2, t_1)$ provided that t_2 is lexicographically smaller than t_1. Due to our choice of the term ordering, this normalization does not violate completeness as it can be interpreted as rewriting using the commutativity law with an ordering constraint. Doing the normalization explicitly spares us the necessity of using a general algorithm for solving the constraint.

Negative equality splitting. This rule may be applied on the preprocessing phase and allows us to simulate an algorithm used in Waldmeister [4]. If the term s is ground, the 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. Apart from other things, this may lead to the following addition-instead-of-multiplication effect. Suppose that s can be rewritten by paramodulation in m different ways resulting in the terms $s_1, \ldots, s_m$, and rewriting of t produces n versions $t_1, \ldots, t_n$. Without *negative equality splitting* we would have to keep $m \cdot n$ clauses $s_i \neq t_j \vee C, 1 \leq i \leq m, 1 \leq j \leq n$, while negative equality splitting allows us to keep only $m + n$ clauses, namely $p(s_i)$ and $\neg p(t_j) \vee C$.

Subsumption resolution. One of the simplification techniques that is heavily used in Vampire, and to which it owes a great part of its power, is *subsumption resolution* [2]. A clause $\neg A \vee C_1$ can be replaced by C_1 if among the kept clauses there is a clause C_2 subsuming $A \vee C_1$.

New algorithm for backward subsumption. One of the major improvements is a new indexing method for backward subsumption. Vampire 0.0 used an algorithm based on discrimination trees. This algorithm proved to be extremely slow on many problems. A new algorithm based on path indexing [9] and database joins was designed. The first implementation of this algorithm was inefficient, but a couple of optimizations turned it into an extremely efficient one. The first key decision was to replace the set of (clause,literal,term) tuples in the leaves of the path index by a more suitable data structure called *skip lists* [6]. The second key decision was to change the order of joins depending on the sizes of the sets. In addition, we added an optimization for handling symmetric predicates.

3 Future developments

In the future we will develop Vampire using the principles mentioned in the introduction: efficiency and flexibility, with the emphasis on the latter.

Efficiency. We will continue experiments with indexing techniques and new algorithms and datastructures for the most important problems. In particular, we will try to improve our indexing techniques for treating symbols with special properties, such as commutativity of functions and symmetry of predicates. Another expected change is a reimplementation of indexing for unification.

Flexibility. So far Vampire was very difficult to use. First, many options were inherited from rather old versions and have become obsolete. The user's manual was primitive. We are planning to make Vampire more user-friendly so it can be used for two kinds of applications: (i) interactive use by experienced users who can understand various proof-control options and use Vampire to prove hard theorems; (ii) as a built-in subsystem of automated proof-assistants, interactive provers, and verification systems. These will require enhancement of Vampire by various features, such as new selection strategies, new simplification orderings, built-in AC etc. We are going to provide interface to interactive provers and automatic proof-assistants, such as HOL and Isabelle.

Preprocessing. Currently, preprocessing of clauses is part of Vampire. We are going to implement preprocessing as a separate program that analyzes input, simplifies it, and calls Vampire with suitable options. A new clausifier will be implemented.

Other. We are going to reimplement memory management in Vampire since currently there are situations when Vampire will request memory from the system behind the specified limit. We are planning to implement *stratified resolution* [3].

4 Availability

Vampire is available free of charge. The authors will be glad to provide the newest versions of the system with necessary assistance to any interested party. The system can be run under Linux and Solaris. A temporarily unsupported port is also available for the Win32 platforms. The latest information about Vampire is available at `http://www.cs.man.ac.uk/fm/vampire/`.

References

1. J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: a system for distributed equational deduction. In J. Hsiang, editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA-95)*, volume 914 of *Lecture Notes in Computer Science*, pages 397–402, Kaiserslautern, 1995.
2. L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science and MIT Press, 2001. To appear.
3. A. Degtyarev and A. Voronkov. Stratified resolution. In D. McAllester, editor, *17th International Conference on Automated Deduction (CADE-17)*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 365–384, Pittsburgh, 2000. Springer Verlag.
4. Th. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. Waldmeister: High-performance equational deduction. *Journal of Automated Reasoning*, 18(2):265–270, 1997.
5. W.W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, January 1994.

6. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.
7. A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. Preprint CSPP-7, Department of Computer Science, University of Manchester, October 2000.
8. S. Schulz. System abstract: E 0.3. In H. Ganzinger, editor, *Automated Deduction—CADE-16. 16th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, pages 297–301, Trento, Italy, July 1999.
9. M. Stickel. The path indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1989.
10. C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topic. System description: SPASS version 1.0.0. In H. Ganzinger, editor, *Automated Deduction—CADE-16. 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 378–382, Trento, Italy, July 1999.