First-Order Theorem Proving and VAMPIRE[⋆]

Laura Kovács[1] and Andrei Voronkov[2]

[1] Chalmers University of Technology
[2] The University of Manchester

Abstract. In this paper we give a short introduction in first-order theorem proving and the use of the theorem prover VAMPIRE. We discuss the superposition calculus and explain the key concepts of saturation and redundancy elimination, present saturation algorithms and preprocessing, and demonstrate how these concepts are implemented in VAMPIRE. Further, we also cover more recent topics and features of VAMPIRE designed for advanced applications, including satisfiability checking, theory reasoning, interpolation, consequence elimination, and program analysis.

1 Introduction

VAMPIRE is an automatic theorem prover for first-order logic. It was used in a number of academic and industrial projects. This paper describes the current version (2.6, revision 1692) of VAMPIRE. The first version of VAMPIRE was implemented in 1993, it was then rewritten several times. The implementation of the current version started in 2009. It is written in C++ and comprises about 152,000 SLOC. It was mainly implemented by Andrei Voronkov and Krystof Hoder. Many of the more recent developments and ideas were contributed by Laura Kovács. Finally, recent work on SAT solving and bound propagation is due to Ioan Dragan.

We start with an overview of some distinctive features of VAMPIRE.

- VAMPIRE is *very fast*. For example, it has been the winner of the world cup in first-order theorem proving CASC [32, 34] twenty seven times, see Table 1, including two titles in the last competition held in 2012.
- VAMPIRE runs on all common platforms (Linux, Windows and MacOS) and can be downloaded from `http://vprover.org/`.
- VAMPIRE can be used in a *very simple way* by inexperienced users.
- VAMPIRE implements a unique *limited resource strategy* that allows one to find proofs quickly when the time is limited. It is especially efficient for short time limits which makes it indispensable for use as an assistant to interactive provers or verification systems.
- VAMPIRE implements *symbol elimination*, which allows one to automatically discover first-order program properties, including quantified ones. VAMPIRE is thus the first theorem prover that can be used not only for proving, but also for generating program properties automatically.

[⋆] This research is partially supported by the FWF projects S11410-N23 and T425-N23, and the WWTF PROSEED grant ICT C-050. This work was partially done while the first author was affiliated with the TU Vienna.

Table 1 Vampire's trophies.

CASC	year	FOF	CNF	LTB
CASC-16, Trento	1999		✔	
CASC-17, Pittsburgh	2000	✔		
CASC-JC, Sienna	2001		✔	
CASC-18, Copenhagen	2002	✔	✔	
CASC-19, Miami	2003	✔	✔	
CASC-J2, Cork	2004	✔	✔	
CASC-20, Tallinn	2005	✔	✔	
CASC-J3, Seattle	2006	✔	✔	
CASC-21, Bremen	2007	✔	✔	
CASC-J4, Sydney	2008	✔	✔	
CASC-22, Montreal	2009	✔	✔	✔
CASC-J5, Edinburgh	2010	✔	✔	✔
CASC-23, Wroclaw	2011	✔		✔
CASC-J6, Manchester	2012	✔		✔
total		12	11	4

Notes:

- In CASC-16 Vampire came second after E-SETHEO, but E-SETHEO was retrospectively disqualified after the competition when one of its components E was found unsound.
- In 2000–2001 Vampire used FLOTTER [35] implemented at MPI Informatik to transform formulas into clausal form; since 2002 clausal form transformation was handled by Vampire itself.
- In 2010–2012 the clausifier of Vampire was used by iProver, that won the EPR division of CASC.

- Vampire can produce *local proofs* [13, 19] in first-order logic with or without theories and extract interpolants [5] from them. Moreover, Vampire can *minimise interpolants* using various measures, such as the total number of symbols or quantifiers [11].
- Vampire has a special mode for working with *very large knowledge bases* and can *answer queries* to them.
- Vampire can prove theorems in combinations of first-order logic and *theories*, such as integer arithmetic. It implements several *theory functions* on integers, real numbers, arrays, and strings. This makes Vampire useful for reasoning with theories and quantifiers.
- Vampire is fully compliant with the first-order part of the *TPTP syntax* [31, 33] used by nearly all first-order theorem provers. It understands sorts and arithmetic. It was the first ever first-order theorem prover to implement the TPTP if-then-else and let-in formula and term constructors useful for program analysis.
- Vampire supports several other *input syntaxes*, including the SMTLib syntax [2]. To perform program analysis, it also can read programs written in C.
- Vampire can analyse *C programs with loops* and generate loop invariants using symbol elimination [18].
- Vampire can produce *detailed proofs*. Moreover, it was the first ever theorem prover that produced proofs for *first-order* (as opposed to clausal form) derivations.
- Vampire implements *many options* to help a user to control proof-search.
- Vampire has a special *consequence elimination mode* that can be used to quickly remove from a set of formulas some formulas implied by other formulas in this set.
- Vampire can run *several proof attempts in parallel* on a multi-core processor.
- Vampire has a *liberal licence*, see [20].

Overview of the Paper

The rest of this paper is organised as follows. Sections 2-5 describe the underlining principles of first-order theorem proving and address various issues that are only implemented in VAMPIRE. Sections 6-11 present new and unconventional applications of first-order theorem proving implemented in VAMPIRE. Many features described below are implemented only in VAMPIRE.

- Section 2 (Mini-Tutorial) contains a brief tutorial explaining how to use VAMPIRE. This simple tutorial illustrates the most common use of VAMPIRE on an example, and is enough for inexperienced users to get started with it.
- To understand how VAMPIRE and other superposition theorem provers search for a proof, in Section 3 (Proof-Search by Saturation) we introduce the superposition inference system and the concept of saturation.
- To implement a superposition inference system one needs a saturation algorithm exploiting a powerful concept of *redundancy*. In Section 4 (Redundancy Elimination) we introduce this concept and explain how it is used in saturation algorithms. We also describe the three saturation algorithms used of VAMPIRE.
- The superposition inference system is operating on sets of clauses, that is formulas of a special form. If the input problem is given by arbitrary first-order formulas, the preprocessing steps of VAMPIRE are first applied as described in Section 5 (Preprocessing). Preprocessing is also applied to sets of clauses.
- VAMPIRE can be used in program analysis, in particular for loop invariant generation and interpolation. The common theme of these applications is the symbol elimination method. Section 6 (Coloured Proofs, Interpolation, and Symbol Elimination) explains symbol elimination and its use in VAMPIRE.
- In addition to standard first-order reasoning, VAMPIRE understands sorts, including built-in sorts integers, rationals, reals and arrays. The use of sorts and reasoning with theories in VAMPIRE is described in Section 7 (Sorts and Theories).
- In addition to checking unsatisfiability, VAMPIRE can also check satisfiability of a first-order formula using three different methods. These methods are overviewed in Section 8 (Satisfiability Checking. Finding Finite Models).
- The proof-search, input, and output in VAMPIRE can be controlled by a number of options and modes. One of the most efficient theorem proving modes of VAMPIRE, called the CASC mode, uses the strategies used in last CASC competitions. VAMPIRE's options and the CASC mode are presented in Section 9 (VAMPIRE Options and the CASC Mode).
- VAMPIRE implements extensions of the TPTP syntax, including the TPTP if-then-else and let-in formula and term constructors useful for program analysis. The use of these constructs is presented in Section 10 (Advanced TPTP Syntax).
- Proving theorems is not the only way to use VAMPIRE. One can also use it for consequence finding, program analysis, linear arithmetic reasoning, clausification, grounding and some other purposes. These advanced features are described in Section 11 (Cookies).

Fig. 1 A TPTP representation of a simple group theory problem.

```
%---- 1 * x = 1
fof(left_identity,axiom,
    ! [X] : mult(e,X) = X).
%---- i(x) * x = 1
fof(left_inverse,axiom,
    ! [X] : mult(inverse(X),X) = e).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,
    ! [X,Y,Z] : mult(mult(X,Y),Z) = mult(X,mult(Y,Z))).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
    ! [X] :  mult(X,X) = e).
%---- prove x * y = y * x
fof(commutativity,conjecture,
    ! [X] : mult(X,Y) = mult(Y,X)).
```

2 Mini-Tutorial

In this section we describe a simple way of using VAMPIRE for proving formulas. Using VAMPIRE is very easy. All one needs is to write the formula as a problem in the TPTP syntax [31, 33] and run VAMPIRE on this problem. VAMPIRE is completely automatic. That is, once you started a proof attempt, it can only be interrupted by terminating VAMPIRE.

2.1 A Simple Example

Consider the following example from a group theory textbook: if all elements in a group have order 2, then the group is commutative. We can write down this problem in first-order logic using the language and the axioms of group theory, as follows:

$$
\begin{array}{ll}
 & \forall x(1 \cdot x = x) \\
\text{Axioms (of group theory):} & \forall x(x^{-1} \cdot x = 1) \\
 & \forall x \forall y \forall z((x \cdot y) \cdot z = x \cdot (y \cdot z)) \\
 & \\
\text{Assumptions:} & \forall x(x \cdot x = 1) \\
\hline
\text{Conjecture:} & \forall x \forall y(x \cdot y = y \cdot x)
\end{array}
$$

The problem stated above contains three axioms, one assumption, and a conjecture. The axioms above can be used in any group theory problem. However, the assumption and conjecture are specific to the example we study; they respectively express the order property (assumption) and the commutative property (conjecture).

The next step is to write this first-order problem in the TPTP syntax. TPTP is a Prolog-like syntax understood by all modern first-order theorem provers. A representation of our example in the TPTP syntax is shown in Figure 1. We should save this problem to a file, for example, `group.tptp`, and run VAMPIRE using the command::

Fig. 2 Correspondence between the first-order logic and TPTP notations

first-order logic	TPTP
$\bot, \top$	`$false`, `$true`
$\neg F$	`~F`
$F_1 \wedge \ldots \wedge F_n$	`F1 & ... & Fn`
$F_1 \vee \ldots \vee F_n$	`F1 | ... | Fn`
$F_1 \rightarrow F_n$	`F1 => Fn`
$F_1 \leftrightarrow F_n$	`F1 <=> Fn`
$(\forall x_1) \ldots (\forall x_n) F$	`![X1,...,Xn]:F`
$(\exists x_1) \ldots (\exists x_n) F$	`?[X1,...,Xn]:F`

```
vampire group.tptp
```

Let us consider the TPTP representation of Figure 1 in some detail. The TPTP syntax has *comments*. Any text beginning with the `%` symbol is considered a comment. For example, the line `%---- 1 * x = 1` is a comment. Comments are intended only as an additional information for human users and will be ignored by VAMPIRE.

The axiom $\forall x(1 \cdot x = x)$ appears in the input as `fof(left_identity, axiom, ! [X] : mult(e,X) = X).` The keyword `fof` means “first-order formula”. One can use the keyword `tff` (“typed first-order formula”) instead, see Section 7. VAMPIRE considers `fof` and `tff` as synonyms[3]. The word `left_identity` is chosen to denote the *name* of this axiom. The user can choose any other name. Names of input formulas are ignored by VAMPIRE when it searches for a proof but they can be used in the proof output.

The variable x is written as *capital* `X`. TPTP uses the Prolog convention for variables: variable names start with upper-case letters. This means that, for example, in the formula `mult(e,x) = x`, the symbol `x` will be a considered a constant.

The universal quantifier $\forall x$ is written as `! [X]`. Note that the use of `![x] : mult(e,x) = x)` will result in a syntax error, since `x` is not a valid variable name. Unlike the Prolog syntax, the TPTP syntax does not allow one to use operators. Thus, one cannot write a more elegant `e * X` instead of `mult(e,X)`. The only exception is the equality (=) and the inequality symbols (`!=`), which must be written as operators, for example, in `mult(e,X) = X`. The correspondence between the first-order logic and TPTP notation is summarised in Figure 2.

2.2 Proof by Refutation

VAMPIRE tries to prove the conjecture of Figure 1 by adding the negation of the conjecture to the axioms and the assumptions and checking if the the resulting set of formulas is unsatisfiable. If it is, then the conjecture is a logical consequence of the axioms and the assumptions. A proof of unsatisfiability of a negation of formula is sometimes called

[3] Until recently, `fof(...)` syntax in TPTP was a special case of `tff(...)`. It seems that now there is a difference between the two syntaxes in the treatment of integers, which we hope will be removed in the next versions of the TPTP language.

Fig. 3 VAMPIRE's refutation

```
Refutation found. Thanks to Tanya!
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

a *refutation* of this formula, so such proofs are commonly referred to as proofs by refutation.

Figure 3 shows a (slightly modified) refutation found by VAMPIRE for our simple group theory example. Let us analyse this refutation, since it contains many concepts discussed further in this paper.

Proof outputs by VAMPIRE are dags, whose nodes are labelled by formulas. Every formula is assigned a unique number, in our example they are numbered `1` to `203`. Numbers are assigned to formulas during the proof search, in the order in which they are generated. The proof consists of *inferences*. Each inference infers a formula, called the *conclusion* of this inference, from a set of formulas, called the *premises* of the inference. For example, formula `87` is inferred from formulas `71` and `27`. We can also say that formulas `71` and `27` are *parents* of formula `87`. Some formulas (including formulas read from the input file) have no parents. In this example these are formulas `1–5`. VAMPIRE sometimes packs long chains of proof steps into a single inference, resulting in clauses with many parents, sometimes over a hundred. The dag proof of Figure 3 is rooted at formula `203`: `$false`. To see that the proof is not a tree consider, for example, formula `11`: it is used to infer three different formulas (`16`, `18` and `20`).

Each formula (or inference) in the proof is obtained using one or more *inference rules*. They are shown in brackets, together with parent numbers. Some examples of inference rules in this proof are `superposition`, `inequality splitting` and `skolemisation`. All together, VAMPIRE implements 79 inference rules.

There are several kinds of inference rules. Some inferences, marked as `input`, introduce input formulas. There are many inference rules related to preprocessing input formulas, for example `ennf transformation` and `cnf transformation`. Preprocessing is discussion in Section 5. The input formulas are finally converted to *clauses*, after which VAMPIRE tries to check unsatisfiability of the resulting set of clauses using the *resolution and superposition inference system* discussed in Section 3. The superposition calculus rules can generally be divided in two parts: *generating* and *simplifying* ones. This distinction will be made more clear when we later discuss *saturation* and *redundancy elimination* in Section 4. Though the most complex part of proof search is the use of the resolution and superposition inference system, preprocessing is also very important, especially when the input contains deep formulas or a large number of formulas.

Formulas and clauses having free variables are considered implicitly universally quantified. Normally, the conclusion of an inference is a logical consequence of its premises, though in general this is not the case. Notable exception are inference rules that introduce new symbols: in our example these are `skolemisation` and `inequality splitting`. Though not preserving logical consequence, inference rules used by VAMPIRE guarantee soundness, which means that an inference cannot change a satisfiable set of formulas into an unsatisfiable one.

One can see that the proof of Figure 3 is a refutation: the top line explicitly mentions that a refutation is found, the proof derives the false formula `$false` in inference `203` and inference `6` negates the input conjecture.

Finally, the proof output of VAMPIRE contains only a subset of all generated formulas. The refutation of Figure 3 contains 26 formulas, while 203 formulas were generated. It is not unusual that very short proofs require many generated formulas and require a lot of time to find.

Besides the refutation, the output produced by VAMPIRE contains *statistics* about the proof attempt. These statistics include the overall running time, used memory, and the termination reason (for example, `refutation found`). In addition, it contains information about the number of various kinds of clause and inferences. These statistics are printed even when no refutation is found. An example is shown in Figure 4.

If one is only interested in provability but not in proofs, one can disable the refutation output by using the option `-- proof off`, though normally this does not result in considerable improvements in either time or memory.

In some cases, VAMPIRE might report `Satisfiability detected`. Remember that VAMPIRE tries to find a *refutation*: given a conjecture F, he tries to establish (un)satisfiability of its negation $\neg F$. Hence, when a conjecture$F$ is given, “Satisfiability” refers not to the satisfiability of F but to the satisfiability of $\neg F$.

2.3 Using Time and Memory Limit Options

For very hard problems it is possible that VAMPIRE will run for a very long time without finding a refutation or detecting satisfiability. As a rule of thumb, one should *always run Vampire with a time limit*. To this end, one should use the parameter `-t` or `--time_limit`, specifying how much time (in seconds) it is allowed to spend for proof search. For example, `vampire -t 5 group.tptp` calls VAMPIRE on the input file `group.tptp` with the time limit of 5 seconds. If a refutation cannot be

Fig. 4 Statistics output by Vampire

```
Version: Vampire 2.6 (revision 1692)
Termination reason: Refutation

Active clauses: 14
Passive clauses: 35
Generated clauses: 194
Final active clauses: 8
Final passive clauses: 11
Input formulas: 5
Initial clauses: 6

Split inequalities: 1

Fw subsumption resolutions: 1
Fw demodulations: 68
Bw demodulations: 14

Forward subsumptions: 65
Backward subsumptions: 1
Fw demodulations to eq. taut.: 20
Bw demodulations to eq. taut.: 1

Forward superposition: 60
Backward superposition: 39
Self superposition: 6

Unique components: 6

Memory used [KB]: 255
Time elapsed: 0.007 s
```

found within the given time limit, Vampire specifies `time limit expired` as the termination reason.

Vampire tries to adjust the proof search pace to the time limit which might lead to strange (but generally pleasant) effects. To illustrate it, suppose that Vampire without a time limit finds a refutation of a problem in 10 seconds. This does not mean that with a time limit of 9 seconds Vampire will be unable to find a refutation. In fact, in most case it will be able to find it. We have examples when it can find proofs when the time limit is set to less than 5% of the time it needs to find a proof without the time limit. A detailed explanation of the magic used can be found in [27].

One can also specify the *memory limit* (in Megabytes) of Vampire by using the parameter `-m` or `--memory_limit`. If Vampire does not have enough memory, it will not give up immediately. Instead, it will try to discard some clauses from the search space and reuse the released memory.

2.4 Limitations

What if Vampire can neither find a refutation nor establish satisfiability for a given time limit? Of course, one can increase the time limit and try again. Theoretically, Vampire is based on a *complete inference system*, that is, if the problem is unsatisfiable, then given enough time and space VAMPIRE will eventually find a refutation. In practice, theorem proving in first-order logic is a very hard problem, so it is unreasonable to expect provers to find a refutation quickly or to find it at all. When VAMPIRE cannot find a refutation, increasing time limit is not the only option to try. VAMPIRE has many other parameters controlling search for a refutation, some of them are mentioned in later sections of this paper.

3 Proof-Search by Saturation

Given a problem, VAMPIRE works as follows:

- *read the problem*;
- determine *proof-search options* to be used for this problem;
- *preprocess the problem*;
- *convert it into conjunctive normal form (cnf)*;
- run a *saturation algorithm* on it;
- report the *result*, maybe including a refutation.

In this section we explain the concept of saturation-based theorem proving and present a simple saturation algorithm. The other parts of the inference process are described in the following sections.

Saturation is the underlying concept behind the proof-search algorithms using the resolution and superposition inference system. In the sequel we will refer to theorem proving using variants of this inference system as simply *superposition theorem proving*. This section explains the main concepts and ideas involved in superposition theorem proving and saturation. A detailed exposition of the theory of superposition can be found in [1, 26]. This section is more technical than the other sections of the paper: we decided to present superposition and saturation in more details, since the concept of saturation is relatively unknown outside of the theorem proving community. Similar algorithms are used in some other areas, but they are not common. For example, Buchberger's algorithm for computing Gröbner basis [4] can be considered as an example of a saturation algorithm.

Given a set S of formulas and an *inference system* $\mathbb{I}$, one can try to *saturate* this set with respect to the inference system, that is, to build a set of formulas that contains S and is closed under inferences in $\mathbb{I}$. Superposition theorem provers perform inferences on formulas of a special form, called *clauses*.

3.1 Basic Notions

We start with an overview of relevant definitions and properties of first-order logic, and fix our notation. We consider the standard first-order predicate logic with equality. We allow all standard boolean connectives and quantifiers in the language. We assume that the language contains the logical constants $\top$ for always true and $\perp$ for always false formulas.

Throughout this paper, we denote terms by l, r, s, t, variables by x, y, z, constants by a, b, c, d, e, function symbols by f, g, h, and predicate symbols by p, q. As usual, an *atom* is a formula of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and $t_1, \ldots, t_n$ are terms. The equality predicate symbol is denoted by $=$. Any atom of the form $s = t$ is called an *equality*. By $s \neq t$ we denote the formula $\neg(s = t)$.

A *literal* is an atom A or its negation $\neg A$. Literals that are atoms are called *positive*, while literals of the form $\neg A$ *negative*. A *clause* is a disjunction of literals $L_1 \vee \ldots \vee L_n$, where $n \geq 0$. When $n = 0$, we will speak of the *empty clause*, denoted by $\square$. The empty clause is always false. If a clause contains a single literal, that is, $n = 1$, it is called a *unit clause*. A unit clause with $=$ is called an *equality literal*. We denote atoms by A, literals by L, clauses by C, D, and formulas by F, G, R, B, possibly with indices.

Let F be a formula with free variables $\bar{x}$, then $\forall F$ (respectively, $\exists F$) denotes the formula $(\forall \bar{x})F$ (respectively, $(\exists \bar{x})F$). A formula is called *closed*, or a *sentence*, if it has no free variables. We call a *symbol* a predicate symbol, a function symbol or a constant. Thus, variables are not symbols. We consider equality $=$ part of the language, that is, equality is not a symbol. A formula or a term is called *ground* if it has no occurrences of variables. A formula is called *universal* (respectively, *existential)* if it has the form $(\forall \bar{x})F$ (respectively, $(\exists \bar{x})F$), where F is quantifier-free. We write $C_1, \ldots, C_n \vdash C$ to denote that the formula $C_1 \wedge \ldots \wedge C_n \rightarrow C$ is a tautology. Note that $C_1, \ldots, C_n, C$ may contain free variables.

A *signature* is any finite set of symbols. The *signature of a formula* F is the set of all symbols occurring in this formula. For example, the signature of $b = g(z)$ is $\{g, b\}$. The *language of a formula* F is the set of all formulas built from the symbols occurring in F, that is formulas whose signatures are subsets of the signature of F.

We call a *theory* any set of closed formulas. If T is a theory, we write $C_1, \ldots, C_n \vdash_T C$ to denote that the formula $C_1 \wedge \ldots \wedge C_1 \rightarrow C$ holds in all models of T. In fact, our notion of theory corresponds to the notion of *axiomatisable theory* in logic. When we work with a theory T, we call symbols occurring in T *interpreted* while all other symbols *uninterpreted*.

We call a *substitution* any expression θ of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $n \geq 0$. An *application* of this substitution to an expression E, denoted by $E\theta$, is the expression obtained from E by the simultaneous replacements of each x_i by t_i. By an expression here we mean a term, atom, literal, or clause. An expression is *ground* if it contains no variables.

We write $E[s]$ to mean an expression E with a particular occurrence of a term s. When we use the notation $E[s]$ and then write $E[t]$, the latter means the expression obtained from $E[s]$ by replacing the distinguished occurrence of s by the term t.

A *unifier* of two expressions E_1 and E_2 is a substitution θ such that $E_1\theta = E_2\theta$. It is known that if two expressions have a unifier, then they have a so-called *most general unifier (mgu)*. For example, consider terms $f(x_1, g(x_1), x_2)$ and $f(y_1, y_2, y_2)$. Some of their unifiers are $\theta_1 = \{y_1 \mapsto x_1, y_2 \mapsto g(x_1), x_2 \mapsto g(x_1)\}$ and $\theta_2 = \{y_1 \mapsto a, y_2 \mapsto g(a), x_2 \mapsto g(a), x_1 \mapsto a\}$, but only θ_1 is most general. There are several algorithms for finding most general unifiers, from linear or almost linear [23] to exponential [28] ones, see [12] for an overview. In a way, VAMPIRE uses none of them since all unification-related operations are implemented using *term indexing* [30].

3.2 Inference Systems and Proofs

An *inference rule* is an n-ary relation on formulas, where $n \geq 0$. The elements of such a relation are called *inferences* and usually written as:

$$\frac{F_1 \quad \dots \quad F_n}{F} \enspace .$$

The formulas $F_1, \dots, F_n$ are called the *premises* of this inference, whereas the formula F is the *conclusion* of the inference. An *inference system* $\mathbb{I}$ is a set of inference rules. An *axiom* of an inference system is any conclusion of an inference with 0 premises. Any inferences with 0 premises and a conclusion F will be written without the bar line, simply as F.

A *derivation* in an inference system $\mathbb{I}$ is a tree built from inferences in $\mathbb{I}$. If the root of this derivation is F, then we say it is a *derivation of* F. A derivation of F is called a *proof* of F if it is finite and all leaves in the derivation are axioms. A formula F is called *provable* in $\mathbb{I}$ if it has a proof. We say that a derivation of F is *from assumptions* $F_1, \dots, F_m$ if the derivation is finite and every leaf in it is either an axiom or one of the formulas $F_1, \dots, F_m$. A formula F is said to be *derivable from* assumptions $F_1, \dots, F_m$ if there exists a derivation of F from $F_1, \dots, F_m$. A *refutation* is a derivation of $\perp$.

Note that a proof is a derivation from the empty set of assumptions. Any derivation from a set of assumptions S can be considered as a derivation from any larger set of assumptions $S' \supseteq S$.

3.3 A Simple Inference System and Completeness

We give now a simple inference system for first-order logic with equality, called the *superposition inference system*. For doing so, we first introduce the notion of a *simplification ordering* on terms, as follows. An ordering $\succ$ on terms is called a *simplification ordering* if it satisfies the following conditions:

1. $\succ$ is *well-founded*, that is there exists no infinite sequence of terms $t_0, t_1, \dots$ such that $t_0 \succ t_1 \succ \dots$.
2. $\succ$ is *monotonic*: if $l \succ r$, then $s[l] \succ s[r]$ for all terms s, l, r.
3. $\succ$ is *stable under substitutions*: if $l \succ r$, then $l\theta \succ r\theta$.
4. $\succ$ has the *subterm property*: if r is a subterm of l and $l \neq r$, then $l \succ r$.

Given a simplification ordering on terms, we can extend it to a simplification ordering on atoms, literals, and even clauses. For details, see [26, 1, 7]. One of the important things to know about simplification orderings is that they formalise a notion of "being simpler" on expressions. For example, for the Knuth-Bendix ordering [14], if a ground term s has fewer symbols than a ground term t, then $t \succ s$.

In addition, to simplification orderings, we need a concept of a *selection function*. A selection function selects in every non-empty clause a non-empty subset of literals. When we deal with a selection function, we will underline selected literals: if we write a clause in the form $\underline{L} \vee C$, it means that L (and maybe some other literals) are selected in $L \vee C$. One example of a selection function sometimes used in superposition theorem provers is the function that selects all maximal literals with respect to the simplification ordering used in the system.

The superposition inference system is, in fact, a family of systems, parametrised by a simplification ordering and a selection function. We assume that a simplification ordering and a selection function are fixed and will now define the *superposition inference system*. This inference system, denoted by Sup, consists of the following rules:

Resolution.

$$\frac{\underline{A} \vee C_1 \quad \underline{\neg A'} \vee C_2}{(C_1 \vee C_2)\theta},$$

where θ is a mgu or A and A'.

Factoring.

$$\frac{\underline{A} \vee \underline{A'} \vee C}{(A \vee C)\theta},$$

where θ is a mgu or A and A'.

Superposition.

$$\frac{\underline{l = r} \vee C_1 \quad \underline{L[s]} \vee C_2}{(L[r] \vee C_1 \vee C_2)\theta} \quad \frac{\underline{l = r} \vee C_1 \quad \underline{t[s] = t'} \vee C_2}{(t[r] = t' \vee C_1 \vee C_2)\theta} \quad \frac{\underline{l = r} \vee C_1 \quad \underline{t[s] \neq t'} \vee C_2}{(t[r] \neq t' \vee C_1 \vee C_2)\theta},$$

where θ is a mgu of l and s, s is not a variable, $r\theta \not\succeq l\theta$, (first rule only) $L[s]$ is not an equality literal, and (second and third rules only) $t'\theta \not\succeq t[s]\theta$.

Equality Resolution.

$$\frac{\underline{s \neq t} \vee C}{C\theta},$$

where θ is a mgu of s and t.

Equality Factoring.

$$\frac{\underline{s = t} \vee \underline{s' = t'} \vee C}{(s = t \vee t \neq t' \vee C)\theta},$$

where θ is an mgu of s and s', $t\theta \not\succeq s\theta$, and $t'\theta \not\succeq t\theta$.

VAMPIRE uses the names of inference rules in its proofs and statistics. For example, the proof of Figure 3 and statistics displayed in Figure 4 use `superposition`. The term `demodulation` used in them also refers to superposition, as we shall see later.

If the selection function is *well-behaved*, that is, it either selects a negative literal or all maximal literals, then the superposition inference system is both *sound* and *refutationally complete*. By soundness we mean that, if the empty clause $\square$ is derivable from a set S of formulas in Sup, then S is unsatisfiable. By (refutational) completeness we mean that if a set S of formulas is unsatisfiable, then $\square$ is derivable from S in Sup.

Defining a sound and complete inference system is however not enough for automatic theorem proving. If we want to use the sound and complete inference system of Sup for finding a refutation, we have to understand how to *organise the search for a refutation in* Sup. One can apply *all possible inferences* to clauses in the search space in a certain order until we derive the empty clause. However, a simple implementation of this idea will hardly result in an efficient theorem prover, because blind applications of all possible inferences will blow up the search space very quickly. Nonetheless, the idea of generating all clauses derivable from S is the key idea of saturation-based theorem proving and can be made very efficient when one exploits a powerful concept of *redundancy* and uses good *saturation algorithms*.

3.4 Saturation

A set of clauses S is called *saturated with respect to an inference system* $\mathbb{I}$ if, for every inference in $\mathbb{I}$ with premises in S, the conclusion of this inference belongs to S too. When the inference system is clear from the context, in this paper it is always Sup, we simply say "saturated set of clauses". It is clear that for every set of clauses S there exists the smallest saturated set containing S: this set consists of all clauses derivable from S.

From the completeness of Sup we can then conclude the following important property. A set S of clauses is unsatisfiable if and only if the smallest set of clauses containing S and saturated with respect to Sup also contains the empty clause.

To saturate a set of clauses S with respect to an inference system, in particular Sup, we need a *saturation algorithm*. At every step such an algorithm should select an inference, apply this inference to S, and add conclusions of the inferences to the set S. If at some moment the empty clause is obtained, we can conclude that the input set of clauses is unsatisfiable. A good strategy for *inference selection* is crucial for an efficient behaviour of a saturation algorithm. If we want a saturation algorithm to preserve completeness, that is, to guarantee that a saturated set is eventually built, the inference selection strategy must be *fair*: every possible inference must be selected at some step of the algorithm. A saturation algorithm with a fair inference selection strategy is called a *fair saturation algorithm*.

By completeness of Sup, there are three possible scenarios for running a fair saturation algorithm on an input set of clauses S:

1. At some moment the empty clause $\square$ is generated, in this case S is unsatisfiable.
2. Saturation terminates without ever generating $\square$, in this case S is satisfiable.
3. Saturation runs forever, but without generating $\square$. In this case S is satisfiable.

Note that in the third case we do not establish satisfiability of S after a finite amount of time. In reality, in this case, a saturation-based prover will simply run out of resources, that is terminate by time or memory limits, or will be interrupted. Even when a set of clauses is unsatisfiable, termination by a time or memory limit is not unusual. Therefore in practice the third possibility must be replaced by:

3'. Saturation will run *until the system runs out of resources,* but without generating $\square$. In this case it is unknown whether S is unsatisfiable.

4 Redundancy Elimination

However, a straightforward implementation of a fair saturation algorithm will not result in an efficient prover. Such an implementation will not solve some problems rather trivial for modern provers because of the rapid growth of the search space. This is due to two reasons:

1. the superposition inference system has many inferences that can be avoided;
2. some clauses can be removed from the search space without compromising completeness.

In other words,

1. some *inferences* in the superposition system are *redundant;*
2. some *clauses* in the search space are *redundant.*

To have an efficient prover, one needs to exploit a powerful concept of *redundancy* and *saturation up to redundancy*. This section explains the concept of redundancy and how it influences the design and implementation of saturation algorithms.

The modern theory of resolution and superposition [1, 26] deals with inference systems in which clauses can be *deleted* from the search space. Remember that we have a simplification ordering $\succ$, which can also be extended to clauses. There is a general redundancy criterion: given a set of clauses S and a clause $C \in S$, C is *redundant* in S if it is a logical consequence of those clauses in S that are strictly smaller than C w.r.t. $\succ$. However, this general redundancy criterion is undecidable, so theorem provers use some sufficient conditions to recognise redundant clauses. Several specific redundancy criteria based on various sufficient conditions will be defined below.

Tautology deletion. A clause is called a *tautology* if it is a valid formula. Examples of tautologies are clauses of the form $A \vee \neg A \vee C$ and $s = s \vee C$. Since tautologies are implied by the empty set of formulas, they are redundant in every clause set. There are more complex equational tautologies, for example, $a \neq b \vee b \neq c \vee a = c$. Equational tautology checking can be implemented using congruence closure. It is implemented in VAMPIRE and the number of removed tautologies appears in the statistics.

Subsumption. We say that a clause C subsumes a clause D if D can be obtained from C by two operations: application of a substitution θ and adding zero or more literals. In other words, $C\theta$ is a submultiset of D if we consider clauses as multisets of their literals. For example, the clause $C = p(a, x) \vee r(x, b)$ subsumes the clause $D = r(f(y), b) \vee q(y) \vee p(a, f(y))$, since D can be obtained from C by applying the substitution $\{x \mapsto f(y)\}$ and adding the literal $q(y)$. Subsumed clauses are redundant in the following sense: if a clause set S contains two different clauses C and D and C subsumes D, then D is redundant in S. Although subsumption checking is NP-complete, it is a powerful redundancy criterion. It is implemented in VAMPIRE and its use is controlled by options.

The concept of redundancy allows one to remove clauses from the search space. Therefore, an inference process using this concept consists of steps of two kinds:

1. add to the search space a new clause obtained by an inference whose premises belong to the search space;
2. delete a redundant clause from the search space.

Before defining saturation algorithms that exploit the concept of redundancy, we have to define a new notion of inference process and reconsider the notion of fairness. Indeed, for this kind of process we cannot formulate fairness in the same way as before, since an inference enabled at some point of the inference process may be disabled afterwards, if one or more of its parents are deleted as redundant.

To formalise an inference process with clause deletion, we will consider such a process as a sequence $S_0, S_1, \ldots$ of sets of clauses. Intuitively, S_0 is the initial set of clauses and S_i for $i \geq 0$ is the search space at the step i of the process. An *inference process* is any (finite or infinite) sequence of sets of formulas $S_0, S_1, \ldots$, denoted by:

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots \tag{1}$$

A *step* of this process is a pair $S_i \Rightarrow S_{i+1}$.

Let $\mathbb{I}$ be an inference system, for example the superposition inference system Sup. An inference process is called an $\mathbb{I}$-process if each of its steps $S_i \Rightarrow S_{i+1}$ has one of the following two properties:

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

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

   such that $\{C_1, \ldots, C_n\} \subseteq S_i$.
2. $S_{i+1} = S_i - \{C\}$ such that C is redundant in S_i.

In other words, every step of an $\mathbb{I}$-derivation process either adds to the search space a conclusion of an $\mathbb{I}$-inference or deletes from it a redundant clause.

An inference process can delete clauses from the search space. To define fairness we are only interested in clauses that are never deleted. Such clauses are called *persistent*. Formally, a clause C is *persistent* in an inference process (1) if for some step i it belongs to all sets S_j for which $j \geq i$. In other words, a persistent clause occurs in S_i and is not deleted at steps $S_i \Rightarrow S_{i+i} \Rightarrow \ldots$. An inference process (1) is called *fair* if it satisfies the following principle: every possible inference with persistent clauses as premises must be performed at some step.

The superposition inference system Sup has a very strong completeness property formulated below.

THEOREM 1 (COMPLETENESS). Let Sup be the superposition inference system, S_0 be a set of clauses and $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$ be a fair Sup-inference process. Then S_0 is unsatisfiable if and only if some S_i contains the empty clause.

An algorithm of *saturation up to redundancy* is any algorithm that implements inference processes. Naturally, we are interested in *fair saturation algorithms* that guarantee fair behaviour for every initial set of clauses S_0.

4.1 Generating and Simplifying Inferences

Deletion of redundant clauses is desirable since every deletion reduces the search space. If a newly generated clause makes some clauses in the search space redundant, adding such a clause to the search space comes "at no cost", since it will be followed by deletion of other (more complex) clauses. This observation gives rise to an idea of prioritising inferences that make one or more clauses in the search space redundant. Since the general redundancy criterion is undecidable, we cannot in advance say whether an inference will result in a deletion. However, one can try to find "cheap" sufficient conditions for an inference to result in a deletion and try to search for such inferences in an eager way. This is exactly what the modern theorem provers do.

Simplifying inferences. Let S be an inference of the form

$$\frac{C_1 \quad \ldots \quad C_n}{C}\ .$$

We call this inference *simplifying* if at least one of the premises C_i becomes redundant after the addition of the conclusion C to the search space. In this case we also say that we *simplify* the clause C_i into C since this inference can be implemented as the replacement of C_i by C. The reason for the name "simplifying" is due to the fact that C is usually "simpler" than C_i in some strict mathematical sense. In a way, such inferences simplify the search space by replacing clauses by simpler ones. Let us consider below two examples of a simplifying rule. In these examples, and all further examples, we will denote the deleted clause by drawing a line through it, for example $\cancel{B \vee D}$.
Subsumption resolution is one of the following inference rules:

$$\frac{A \vee C \quad \cancel{\neg B \vee D}}{D} \qquad \text{or} \qquad \frac{\neg A \vee C \quad \cancel{B \vee D}}{D}\ ,$$

such that for some substitution θ we have $A\theta \vee C\theta \subseteq B \vee D$, where clauses are considered as sets of literals. The name "subsumption resolution" is due to the fact that the applicability of this inference can be checked in a way similar to subsumption checking.
Demodulation is the following inference rule:

$$\frac{l = r \quad \cancel{C[l\theta]}}{C[r\theta]}$$

where $l\theta \succ r\theta$ and $(l = r)\theta \succ C[l\theta]$. Demodulation is also sometimes called *rewriting by unit equalities*. One can see that demodulation is a special case of superposition where one of the parents is deleted. The difference is that, unlike superposition, demodulation does not have to be applied only to selected literals or only into the larger part of equalities.

Generating inferences. Inferences that are not simplifying are called *generating*: instead of simplifying one of the clauses in the search space, they generate a new clause C.

Many theorem provers, including VAMPIRE, implement the following principle:

apply simplifying inferences eagerly;
apply generating inferences lazily.

This principle influences the design of saturation algorithms in the following way: from time to time provers try to search for simplifying inferences at the expense of delaying generating inferences. More precisely, after generating each new clause C, Vampire tries to apply as many simplifying rules using C as possible. It is often the case that a single simplifying inference gives rise to new simplifying inferences, thus producing long chains of them after a single generating inference.

Deletion rules. Even when simplification rules are in use, deletion of redundant clauses is still useful and performed by Vampire. Vampire has a collection of *deletion rules*, which check whether clauses are redundant due to the presence of other clauses in the search space. Typical deletion rules are subsumption and tautology deletion. Normally, these deletion rules are applied to check if a newly generated clause is redundant (forward deletion rules) and then to check if one of the clauses in the search space becomes redundant after the addition of the new clause to the search space (backward deletion rules). An example of a forward deletion rule is forward demodulation, listed also in the proof output of Figure 3.

4.2 A Simple Saturation Algorithm

We will now present a *simple algorithm for saturation up to redundancy*. This algorithm is not exactly the saturation algorithm of Vampire but it is a good approximation and will help us to identify various parts of the inference process used by Vampire. The algorithm is given in Figure 5. In this algorithm we mark by ✓ parts that are informally explained in the rest of this section.

The algorithm uses three variables: a set kept of so called *kept clauses*, a set *unprocessed* for storing clauses to be processed, and a clause new to represent the currently processed clause. The set $\mathit{unprocessed}$ is needed to make the algorithm search for simplification eagerly: as soon as a clause has been simplified, it is added to the set of unprocessed clauses so that we could check whether it can be simplified further or simplify other clauses in the search space. Moreover, the algorithm is organised in such a way that kept is always *inter-reduced*, that is, all possible simplification rules between clauses in kept have already been applied.

Let us now briefly explain parts of the saturation algorithm marked by ✓.

Retention test. This is the test to decide whether the new clause should be kept or discarded. The retention test applies deletion rules, such as subsumption, to the new clause. In addition, Vampire has other criteria to decide whether a new clause should be kept, for example, it can decide to discard too big clauses, even when this compromises completeness. Vampire has several parameters to specify which deletion rules to apply and which clauses should be discarded. Note that in the last line the algorithm returns either *satisfiable* or *unknown*. It returns *satisfiable* when only redundant clauses were discarded by the retention test. When at least one non-redundant clause was discarded, the algorithm returns *unknown*.

Fig. 5 A Simple Saturation Algorithm

```
   var $kept$, $unprocessed$: sets of clauses;
   var $new$: clause;
   $unprocessed$ := the initial sets of clauses;
   $kept$ := $\emptyset$;
   loop
     while $unprocessed \neq \emptyset$
       $new$ :=$select(unprocessed)$;
       if $new = \Box$ then return unsatisfiable;
✓      if $retained(new)$ then                                        (* retention test *)
✓        simplify $new$ by clauses in $kept$ ;                    (* forward simplification *)
         if $new = \Box$ then return unsatisfiable;
✓        if $retained(new)$ then                                (* another retention test *)
✓          delete and simplify clauses in $kept$ using $new$;    (* backward simplification *)
           move the simplified clauses from $kept$ to $unprocessed$;
           add $new$ to $kept$
     if there exists an inference with premises in $kept$ not selected previously then
✓      select such an inference;                                   (* inference selection *)
✓      add to $unprocessed$ the conclusion of this inference      (* generating inference *)
     else return satisfiable or unknown
```

Forward simplification. In this part of the saturation algorithm, VAMPIRE tries to simplify the new clause by kept clauses. Note that the simplified clause can again become subject to deletion rules, for example it can become simplified to a tautology. For this reason after forward simplification another retention test may be required.

Backward simplification. In this part, VAMPIRE tries to delete or simplify kept clauses by the new clause. If some clauses are simplified, they become candidates for further simplifications and are moved to *unprocessed*.

Inference selection. This part of the saturation algorithm selects an inference that has not previously been selected. Ideally, inference selection should be fair. VAMPIRE

Generating Inference. This part of the saturation algorithm applies the selected inference. The inference generates a new clause, so we call it a generating inference.[4] Most of the generating inferences to be used are determined by VAMPIRE but some are user-controlled.

4.3 Saturation Algorithms of VAMPIRE

To design a saturation algorithm, one has to understand how to achieve fairness and how to organise redundancy elimination. VAMPIRE implements three different saturation algorithms. A description and comparison of these algorithms can be found in [27]. A saturation algorithm in VAMPIRE can be chosen by setting the `--saturation_algo-`

[4] This inference may turn out to be simplifying inference, since the new clause generated by it may sometimes simplify a kept clause.

`rithm` option to one of the values `lrs`, `discount`, and `otter`. The `lrs` saturation algorithm is the default one used by VAMPIRE and refers to the *limited resource strategy algorithm*, `discount` is the *Discount algorithm*, and `otter` is the *Otter saturation algorithm*. In this section we briefly describe these three saturation algorithms.

Given Clause Algorithms. All saturation algorithms implemented in VAMPIRE belong to the family of *given clause algorithms*. These algorithms achieve fairness by implementing inference selection using *clause selection*. At each iteration of the algorithm a clause is selected and all generating inferences are performed between this clause and previously selected clauses. The currently selected clause is called the *given clause* in the terminology of [24].

Clause Selection. Clause selection in VAMPIRE is based on two parameters: the *age* and the *weight* of a clause. VAMPIRE maintains two priority queues, in which older and lighter clauses are respectively prioritised. The clauses are selected from one of the queues using an *age-weight ratio*, that is, a pair of non-negative integers (a, w). If the age-weight ratio is (a, w), then of each $a + w$ selected clauses a clauses will be selected from the age priority queue and w from the weight priority queue. In other words, of each $a + w$ clauses, a oldest and w lightest clauses are selected. The age-weight ratio can be controlled by the parameter `--age_weight_ratio`. For example, `--age_weight_ratio 2:3` means that of each 5 clauses, 2 will be selected by age and 3 by weight.

The *age* is implemented using numbering of clauses. Each kept clause is assigned a unique number. The numbers are assigned in the increasing order, so older clauses have smaller numbers. Each clause is also assigned a *weight*. By default, the weight of a clause is equal to its size, that is, the count of symbols in it, but it can also be controlled by the user, for example, by using the option `--nongoal_weight_coefficient`.

Active and passive clauses. Given clause algorithms distinguish between kept clauses previously selected for inferences and those not previously selected. Only the former clauses participate in generating inferences. For this reason they are called *active*. The kept clauses still waiting to be selected are called *passive*. The interesting question is whether passive clauses should participate in simplifying inferences.

Otter saturation algorithm. The first family of saturation algorithms do use passive clauses for simplifications. Any such saturation algorithm is called an *Otter* saturation algorithm after the theorem prover Otter [24]. The Otter saturation algorithm was designed as a result of research in resolution theorem proving in Argonne National Laboratory, see [22] for an overview.

Retention test. The *retention test* in VAMPIRE mainly consists of deletion rules plus the weight test. The weight test discards clauses whose weight exceeds certain limit. By default, there is no limit on the weight. The weight limit can be controlled by using the option `--max_weight`. In some cases VAMPIRE may change the weight limit. This happens when it is running out of memory or when the limited resource strategy is on. Note that the weight test can discard a non-redundant clause. So when the weight limit was set for at least some time during the proof-search, VAMPIRE will never return "satisfiable".

The growth of the number of kept clauses in the Otter algorithm causes fast deterioration of the processing rate of active clauses. Thus, when a complete procedure based on the Otter algorithm is used, even passive clauses with high selection priority often have to wait indefinitely long before they become selected. In theorem provers based on the Otter algorithm, all solutions to the completeness-versus-efficiency problem are based on the same idea: some non-redundant clauses are discarded from the search space.

Limited resource strategy algorithm. This variation of the Otter saturation algorithm was introduced in one of the early versions of VAMPIRE. It is described in detail in [27]. This strategy is based on the Otter saturation algorithm and can be used only when a time limit is set.

The main idea of the limited resource strategy is the following. The system tries to identify which passive and unprocessed clauses have no chance to be selected by the time limit and discards these clauses. Such clauses will be called *unreachable*. Note that unreachability is fundamentally different from redundancy: redundant clauses are discarded without compromising completeness, while the notion of an unreachable clause makes sense only in the context of reasoning with a time limit.

To identify unreachable clauses, VAMPIRE measures the time spent by processing a given clause. Note that usually this time increases because the sets of active and passive clauses are growing, so generating and simplifying inferences are taking increasingly longer time. From time to time VAMPIRE estimates how the proof search pace will develop towards the time limit and, based on this estimation, identifies potentially unreachable clauses. Limited resource strategy is implemented by adaptive weight limit changes, see [27] for details. It is very powerful and is generally the best saturation algorithm in VAMPIRE.

Discount algorithm. Typically, the number of passive clauses is much larger than the number of active ones. It is not unusual that the number of active clauses is less than 1% of the number of passive ones. As a result, inference pace may become dominated by simplification operations on passive clauses. To solve this problem, one can make passive clauses truly passive by not using them for simplifying inferences. This strategy was first implemented in the theorem prover Discount [6] and is called the Discount algorithm.

Since only a small subset of all clauses is involved in simplifying inferences, processing a new clause is much faster. However, this comes at a price. Simplifying inferences between a passive and a new clause performed by the Otter algorithm may result in a valuable clause, which will not be found by the Discount algorithm. Also, in the Discount algorithm the retention test and simplifications are sometimes performed twice on the same clause: the first time when the new clause is processed and then again when this clause is activated.

Comparison of saturation algorithms. Limited resource strategy is implemented only in VAMPIRE. Thus, VAMPIRE implements all three saturation algorithms. The theorem provers E [29] and Waldmeister [21] only implement the Discount algorithm. In VAMPIRE limited resource strategy gives the best results, closely followed by the Discount algorithm. The Otter algorithm is generally weaker, but it still behaves better on some formulas.

Fig. 6 Inferences in Saturation Algorithms

clauses	Otter and LRS	Discount
active	generating inferences with the given clause; simplifying inferences with new clauses	generating inferences with the given clause; simplifying inferences with new clauses; simplifying inferences with re-activated clauses
passive	simplifying inferences with new clauses	
new	simplifying inferences with active and passive clauses	simplifying inferences with active clauses

To give the reader an idea on the difference between the three saturation algorithms, we show in Figure 6 the roles played by different clauses in the Otter and Discount algorithms.

5 Preprocessing

For many problems, preprocessing them in the right way is the key to solving them. VAMPIRE has a sophisticated preprocessor. An incomplete collection of preprocessing steps is listed below.

1. Select a *relevant* subset of formulas (optional).
2. Add *theory axioms* (optional).
3. *Rectify* the formula.
4. If the formula contains any occurrence of $\top$ or $\bot$, *simplify* the formula.
5. Remove *if-then-else* and *let-in* connectives.
6. *Flatten* the formula.
7. Apply *pure predicate elimination*.
8. Remove *unused predicate definitions* (optional).
9. Convert the formula into *equivalence negation normal form (ennf).*
10. Use a *naming technique* to replace some subformulas by their names.
11. Convert the formula into *negation normal form* (optional).
12. *Skolemise* the formula.
13. Replace *equality axioms*.
14. Determine a *literal ordering* to be used.
15. Transform the formula into its *conjunctive normal form (cnf)*.
16. *Function definition elimination* (optional).
17. Apply *inequality splitting* (optional).
18. Remove *tautologies*.
19. Apply *pure literal elimination* (optional).
20. Remove *clausal definitions* (optional).

Optional steps are controlled by VAMPIRE's options. Since input problems can be very large (for example, several million formulas), VAMPIRE avoids using any algorithm that may cause slow preprocessing. Most algorithms used by the preprocessor run in linear or $O(n \cdot \log n)$ time.

6 Coloured Proofs, Interpolation, and Symbol Elimination

VAMPIRE can be used in program analysis, in particular for *loop invariant generation* and *building interpolants*. Both features have been implemented using our *symbol elimination method*, introduced in [18, 19]. The main ingredient of symbol elimination is the use of *coloured proofs* and *local proofs*. Local proofs are also known as split proofs in [13] and they are implemented using coloured proofs. In this section we introduce interpolation, coloured proofs and symbol elimination and describe how they are implemented in VAMPIRE.

6.1 Interpolation

Let R and B be two closed formulas. Their *interpolant* is any formula I with the following properties:

1. $\vdash R \to I$;
2. $\vdash I \to B$;
3. Every symbol occurring in I also occurs both in R and B.

Note that the existence of an interpolant implies that $\vdash R \to B$, that is, B is a logical consequence of R. The first two properties mean that I is a formula "intermediate" in power between R and B. The third property means that I uses only (function and predicate) symbols occurring in both R and B. For example, suppose that p, q, r are propositional symbols. Then the formulas $p \wedge q$ and $q \vee r$ have an interpolant q.

Craig proved in [5] that every two formulas R and B such that $\vdash R \to B$ have an interpolant. In applications of interpolation in program verification, one is often interested in finding interpolants w.r.t. a theory T, where the provability relation $\vdash$ is replaced by $\vdash_T$, that is validity with respect to T. Interpolation in the presence of a theory T is discussed in some detail in [19].

6.2 Coloured Proofs and Symbol Elimination

We will reformulate the notion of an interpolant by *colouring* symbols and formulas. We assume to have three colour: *red*, *blue* and *grey*. Each symbol (function or predicate) is coloured in exactly one of these colours. Symbols that are coloured in red or blue are called *coloured symbols*. Thus, grey symbols are regarded as uncoloured. Similarly, a formula that contains at least one coloured symbols is called *coloured*; otherwise it is called *grey*. Note that coloured formulas can also contain grey symbols.

Let T be a theory and R and B be formulas such that

- each symbol in R is either red or grey;
- each symbol in B is either blue or grey;
- every formula in T is grey.

We call an *interpolant* of R and B any grey formula I such that $\vdash_T R \to I$ and $\vdash_T I \to B$.

It is easy to see that this definition generalises that of Craig [5]: use the empty theory, colour all symbols occurring in R but not in B in red, all symbols occurring in B but not in R blue, and symbols occurring in both R and B in grey.

When we deal with refutations rather than proofs and have an unsatisfiable set $\{R, B\}$, it is convenient to use a *reverse interpolant* of R and B, which is any grey formula I such that $\vdash_T R \rightarrow I$ and $\{I, B\}$ is unsatisfiable. In applications of interpolation in program verification, see e.g. the pioneering works [25, 13], an interpolant is typically defined as a reverse interpolant. The use of interpolation in hardware and software verification requires deriving (reverse) interpolants from refutations.

A *local derivation* [13, 19] is a derivation in which no inference contains both red and blue symbols. An inference with at least one coloured premise and a grey conclusion is called a *symbol-eliminating inference*. It turns out that one can extract interpolants from local proofs. For example, in [19] we gave an algorithm for extracting a reverse interpolant of R and B from a local refutation of $\{R, B\}$. The extracted reverse interpolant I is a boolean combination of conclusions of symbol-eliminating inferences, and is polynomial in the size of the refutation (represented as a dag). This algorithm is further extended in [11] to compute interpolants which are minimised w.r.t. various measures, such as the total number of symbols or quantifiers.

Coloured proofs can also be used for another interesting application of program verification. Suppose that we have a set Π of formulas in some language L and want to derive logical consequences of these formulas in a subset L_0 of this language. Then we declare the symbols occurring only in $L \setminus L_0$ coloured, say red, and the symbols of L_0 grey; this makes some of the formulas from Π coloured. We then ask VAMPIRE to eliminate red symbols from Π, that is, derive grey consequences of formulas in Π. All these grey consequences will be conclusions of symbol-eliminating inferences. This technique is called *symbol elimination* and was used in our experiments on automatic loop invariant generation [18, 9]. It was the first ever method able to derive loop invariants with quantifier alternations. Our results [18, 19] thus suggest that symbol elimination can be a unifying concept for several applications of theorem provers in program verification.

6.3 Interpolation and Symbol Elimination in VAMPIRE

To make VAMPIRE generate local proofs and compute an interpolant we should be able to assign colours to symbols and define which part of the input belongs to R, B, and the theory T, respectively. We will give an example showing how VAMPIRE implements coloured formulas, local proofs and generation of interpolants from local proofs.

Suppose that q, f, a, b are red symbols and c is a blue symbol, all other symbols are grey. Let R be the formula $q(f(a)) \wedge \neg q(f(b))$ and define B to be $(\exists v.v \neq c)$. Clearly, $\vdash R \rightarrow B$.

Specifying colours. The red and blue colours in Vampire are respectively denoted by `left` and `right`. To specify, for example, that the predicate symbol `q` of arity 1 is red and the constant `c` is blue, we use the following declarations:

```
vampire(symbol,predicate,q,1,left).
vampire(symbol,function,c,0,right).
```

Specifying R and B. Using the TPTP notation, formulas R and B are specified as:

```
vampire(left_formula).           vampire(right_formula).
  fof(R,axiom,q(f(a))&~q(f(b))).   fof(L,conjecture,?[V].(V!=c)).
vampire(end_formula).            vampire(end_formula).
```

Local proofs and interpolants. To make VAMPIRE compute an interpolant I, the following option is set in the VAMPIRE input:

```
vampire(option,show_interpolant,on).
```

If we run VAMPIRE on this input problem, it will search for a local proof of $\vdash R \rightarrow B$. Such a local proof will quickly be found and the interpolant $\neg(\forall x, y)(x = y)$ will appear in the output.

Symbol elimination and invariants. To run VAMPIRE in the symbol elimination mode with the purpose of loop invariant generation, we declare the symbols to be eliminated coloured. For the use of symbol elimination, a single colour, for example `left`, is sufficient. We ask VAMPIRE to run symbol elimination on its coloured input by setting:

```
vampire(option,show_symbol_elimination,on).
```

When this option is on, VAMPIRE will output all conclusions of symbol-eliminating inferences.

7 Sorts and Theories

Standard superposition theorem provers are good in dealing with quantifiers but have essentially no support for theory reasoning. Combining first-order theorem proving and theory reasoning is very hard. For example, some simple fragments of predicate logic with linear arithmetic are already Π^1_1-complete [15]. One relatively simple way of combining first-order logic with theories is adding a first-order axiomatisation of the theory, for some example an (incomplete) axiomatisation of integers.

Adding incomplete axiomatisations is the approach used in [9, 18] and the one we have followed in the development of VAMPIRE. We recently added integers, reals, and arrays as built-in data types in VAMPIRE, and extended VAMPIRE with such data types and theories. For example, one can use integer constants in the input instead of representing them using, for example, zero and the successor function. VAMPIRE implements several standard predicates and functions on integers and reals, including addition, subtraction, multiplication, successor, division, and standard inequality relations such as $\geq$. VAMPIRE also has an axiomatisations of the theory of arrays with the select and store operations.

7.1 Sorts

For using sorts in VAMPIRE, the user should first specify the sort of each symbol. Sort declarations need to be added to the input, by using the `tff` keyword (“typed first-order formula”) of the TPTP syntax, as follows.

Defining sorts. For defining a new sort, say sort `own`, the following needs to be added to the VAMPIRE input:

```
tff(own_type,type,own: $tType).
```

Similarly to the `fof` declarations, the word `own_type` is chosen to denote the name of the declaration and is ignored by VAMPIRE when it searches for a proof. The keyword `type` in this declaration is, unfortunately, both ambiguous and redundant. The only important part of the declaration is `own: $tType`, which declares that `own` is a new sort (type).

Let us now consider an arbitrary constant a and a function f of arity 2. Specifying that a and the both the arguments and the values of f have sort `own` can be done as follows:

```
tff(a_has_type_own,type,a : own).
tff(f_has_type_own,type,f : own * own > own).
```

Pre-defined sorts. The user can also use the following pre-existing sorts of VAMPIRE:

- `$i`: sort of individuals. This is the default sort used in VAMPIRE: if a symbol is not declared, it has this sort;
- `$o`: sort of booleans;
- `$int`: sort of integers;
- `$rat`: sort of rationals;
- `$real`: sort of reals;
- `$array1`: sort of arrays of integers;
- `$array2`: sort of arrays of arrays of integers.

The last two are VAMPIRE-specific, all other sorts belong to the TPTP standard. For example, declaring that p is a unary predicate symbol over integers is written as:

```
tff(p_is_int_predicate,type,p : $int > $o).
```

7.2 Theory Reasoning

Theory reasoning in VAMPIRE is implemented by adding theory axioms to the input problem and using the superposition calculus to prove problems with both quantifiers and theories. In addition to the standard superposition calculus rules, VAMPIRE will evaluate expressions involving theory functions, whenever possible, during the proof search.

This implementation is not just a simple addition to VAMPIRE: we had to add simplification rules for theory terms and formulas, and special kinds of orderings [17]. Since VAMPIRE's users may not know much about combining theory reasoning and first-order logic, VAMPIRE adds the relevant theory axiomatisation automatically. For example, if the input problem contains the standard integer addition function symbol $+$, denoted by `$sum` in VAMPIRE, then VAMPIRE will automatically add an axiomatisation of integer linear arithmetic including axioms for additions.

The user can add her own axioms in addition to those added by Vampire. Moreover, the user can also choose to use her own axiomatisation *instead* of those added by VAMPIRE one by using the option `--theory axioms off`.

For some theories, namely for linear real and rational arithmetic, VAMPIRE also supports DPLL(T) style reasoning instead of using the superposition calculus, see Section 11.3. However, this feature is yet highly experimental.

A partial list of interpreted function and predicate symbols over integers/reals/rationals in VAMPIRE contains the following functions defined by the TPTP standard:

- `$sum`: addition $(x + y)$
- `$product`: multiplication $(x \cdot y)$
- `$difference`: difference $(x - y)$
- `$uminus`: unary minus $(-x)$
- `$to_rat`: conversion to rationals
- `$to_real`: conversion to reals
- `$less`: less than $(x < y)$
- `$lesseq`: less than or equal to $(x \leq y)$
- `$greater`: greater than $(x > y)$
- `$greatereq`: greater than or equal to $(x \geq y)$

Let us consider the formula $(x + y) \geq 0$, where x and y are integer variables. To make VAMPIRE try to prove this formula, one needs to use sort declarations in quantifiers and write down the formula as the typed first-order formula:

```
tff(example,conjecture, ? [X:$int,Y:$int]:
        $greatereq($sum(X,Y),0)).
```

When running VAMPIRE on the formula above, VAMPIRE will automatically load the theory axiomatisation of integers and interpret 0 as the corresponding integer constant.

The quantified variables in the above example are explicitly declared to have the sort `$int`. If the input contains undeclared variables, the TPTP standard requires that they have a predefined sort `$i`. Likewise, undeclared function and predicate symbols are considered symbols over the sort `$i`.

8 Satisfiability Checking. Finding Finite Models

Since 2012, VAMPIRE has become competitive with the best solvers on satisfiable first-order problems. It can check satisfiability of a first-order formula or a set of clauses using three different approaches.

1. By saturation. If a complete strategy is used and VAMPIRE builds a saturated set, then the input set of formulas is satisfiable. Unfortunately, one cannot build a good representation of a model satisfying this set of clauses - the only witness for satisfiability in this case is the saturated set.
2. VAMPIRE implements the instance generation method of Ganzinger and Korovin [8]. It is triggered by using the option `--saturation_algorithm inst_gen`. If this method succeeds, the satisfiability witness will also be the saturated set.

Table 2 Strategies in VAMPIRE

category	strategies	total	best	worst	explanation
EPR	16	560	350	515	effectively propositional (decidable fragment)
UEQ	35	753	198	696	unit equality
HNE	22	536	223	469	CNF, Horn without equality
HEQ	23	436	376	131	CNF, Horn with equality
NNE	25	464	404	243	CNF, non-Horn without equality
NEQ	98	1861	1332	399	CNF, non-Horn with equality
PEQ	30	434	373	14	CNF, equality only, non-unit
FNE	34	1347	1210	585	first-order without equality
FEQ	193	4596	2866	511	first-order with equality

3. The method of building finite models using a translation to EPR formulas introduced in [3]. In this case, if satisfiability is detected, VAMPIRE will output a representation of the found finite model.

Finally, one can use `--mode casc_sat` to treat the input problem using a cocktail of satisfiability-checking strategies.

9 VAMPIRE Options and the CASC Mode

VAMPIRE has many *options* (parameter-value pairs) whose values that can be changed by the user (in the command line). By changing values of the options, the user can control the input, preprocessing, output of proofs, statistics and, most importantly, proof search of VAMPIRE. A collection of options is called a *strategy*. Changing values of some parameters may have a drastic influence on the proof search space. It is not unusual that one strategy results in a refutation found immediately, while for another strategies VAMPIRE cannot find refutation in hours.

When one runs VAMPIRE on a problem, the default strategy of VAMPIRE is used. This strategy was carefully selected to solve a reasonably large number of problems based on the statistics collected by running VAMPIRE on problems from the TPTP library. However, it is important to understand that there is no single best strategy, so for solving hard problems using a single strategy is not a good idea. Table 2 illustrates the power of strategies, based on the results of running VAMPIRE using various strategies on several problem categories. The acronyms for categories use the TPTP convention, for example FEQ means “first-order problems with equality”, their explanation is given in the rightmost column. The table respectively shows the total number of strategies used in experiments, the total number of problems solved by all strategies, and the numbers of problems solved by the best and the worst strategy. For example, the total number of TPTP FEQ problems in this category that VAMPIRE can solve is 4596, while the best strategy for this category can solve only 2866 problems, that is, only about 62% of all problems. The worst strategy solves only 511 FEQ problems, but this strategy is highly incomplete.

There is an infinite number of possible strategies, since some parameters have integer values. Even if we fix a small finite number of values for such parameters, the total

Fig. 7 Running Vampire in the CASC mode on SET014-3

```
Hi Geoff, go and have some cold beer while I am trying to solve
                                        this very hard problem!
% remaining time: 2999 next slice time: 7
dis+2_64_bs=off:cond=fast:drc=off:fsr=off:lcm=reverse:nwc=4:...
Time limit reached!
------------------------------
Termination reason: Time limit
...
------------------------------
% remaining time: 2991 next slice time: 7
dis+10_14_bs=off:cond=fast:drc=off:gs=on:nwc=2.5:nicw=on:sd=...
Refutation not found, incomplete strategy
------------------------------
Termination reason: Refutation not found, incomplete strategy
...
------------------------------
% remaining time: 2991 next slice time: 18
dis+1011_24_cond=fast:drc=off:nwc=10:nicw=on:ptb=off:ssec=of...
Refutation found. Thanks to Tanya!
...
% Success in time 0.816 s
```

number of possible strategies will be huge. Fortunately, Vampire users do not have to understand all the parameters. One can use a special Vampire mode, called the CASC mode and used as `--mode casc`, which mimics the strategies used at the last CASC competition [32]. In this mode, Vampire will treat the input problem with a cocktail of strategies running them sequentially with various time limits.

The CASC mode of Vampire is illustrated in Figure 7 and shows part of the output produced by Vampire in the CASC mode on the TPTP problem `SET014-3`. This problem is very hard: according to the TPTP problems and solutions document only Vampire can solve it. One can see that Vampire ran three different strategies on this problems. Each of the strategies is given as a string showing in a succinct way the options used. For example, the string `dis+2_64_bs=off:cond=fast` means that Vampire was run using the Discount saturation algorithm with literal selection 2 and age-weight ratio 64. Backward subsumption was turned off and a fast condensing algorithm was used. In the figure, we truncated these strings since some of them are very long and removed statistics (output after running each strategy) and proof (output at the end).

The time in the CASC mode output is measured in deciseconds. One can see that Vampire used the first strategy with the time limit of 0.7 seconds, then the second one with the time limit of 0.7 seconds, followed by the third strategy with the time limit of 1.8 seconds. All together it took Vampire 0.816 seconds to find a refutation.

There is also a similar option for checking satisfiability: `--mode casc_sat`. The use of the CASC mode options, together with a time limit, is highly recommended.

Fig. 8 A partial correctness statement and its VAMPIRE input representation.

	% sort declarations tff(1,type,p : $int > $o). tff(2,type,f : $int > $int). tff(3,type,q : $int > $o). tff(4,type,a : $int).

Precondition:
{(∀ X) (p(X) => X
≥ 0)}
{(∀ X) (f(X) > 0)}
{p(a)}
	% precondition
	tff(5,hypothesis,
	  ! [X:$int] : (p(X) => $greatereq(X,0))).
	tff(6,hypothesis,
	  ! [X:$int] : ($greatereq(f(X),0))).
	tff(7,hypothesis,p(a)).

Program:
if (q(a))
{ a := a+1 }
else
{ a := a + f(a)}
	% transition relation
	tff(8,hypothesis,
	  a1 = $ite_t(q(a),
	            $let_tt(a,$sum(a,1),a),
	            $let_tt(a,$sum(a,f(a)),a))).

Postcondition: {a > 0}	% postcondition tff(9,conjecture,$greater(a1,0)).

The total number of parameters in the current version of VAMPIRE is 158, so we cannot describe all of them here. These options are not only related to the proof search. There are options for preprocessing, input and output syntax, output of statistics, various proof search limit, interpolation, SAT solving, program analysis, splitting, literal and clause selection, proof output, syntax of new names, bound propagation, use of theories, and some other options.

10 Advanced TPTP Syntax

VAMPIRE supports let-in and if-then-else constructs, which have recently become a TPTP standard. VAMPIRE was the first ever first-order theorem prover to implement these constructs. Having them makes VAMPIRE better suited for applications in program verification. For example, given a simple program with assignments, composition and if-then-else, one can easily express the transition relation of this programs.

Consider the example shown in Figure 8. The left column displays a simple program together with its pre- and postconditions. Let a1 denote the next state value of a. Using if-then-else and let-in expressions, we can express this next-state value by a simple transformation of the text of the program as follows:

```
a1 = if q(a) then (let a=a+1 in a)
             else (let a=a+f(a) in a)
```

We can then express partial correctness of the program of Figure 8 as a TPTP problem, as follows:

- write down the precondition as a hypothesis in the TPTP syntax;
- write down the next state value of `a` as as a hypothesis in the extended TPTP syntax, by using the if-then-else (`$ite_t`) and let-in (`$let_tt`) constructs;
- write down the postcondition as the conjecture.

The right column of Figure 8 shows this TPTP encoding.

11 Cookies

One can use VAMPIRE not only as a theorem prover, but in several other ways. Some of them are described in this section.

11.1 Consequence Elimination

Given a large set S of formulas, it is very likely that some formulas are consequences of other formulas in the set. To simplify reasoning about and with S, it is desirable to simplify S by removing formulas from S that are implied by other formulas. To address this problem, a new mode, called *the consequence-elimination mode* is now added to VAMPIRE [10]. In this mode, VAMPIRE takes a set S of clauses as an input and tries to find its proper subset S_0 equivalent to S. In the process of computing S_0, VAMPIRE is run with a small time limit. Naturally, one is interested in having S_0 as small as possible. To use VAMPIRE for consequence elimination, one should run:

```
vampire --mode consequence elimination set_S.tptp
```

where `set_S.tptp` contains the set S.

We illustrate consequence elimination on the following set of formulas:

```
fof(ax1, axiom, a => b).
fof(ax2, axiom, b => c).
fof(ax3, axiom, c => a).
fof(c1, claim, a | d).
fof(c2, claim, b | d).
fof(c3, claim, c | d).
```

The word `claim` is a new VAMPIRE keyword for input formulas introduced for the purpose of consequence elimination. VAMPIRE is asked to eliminate, one by one, those claims that are implied by other claims and axioms. The set of non-eliminated claims will then be equivalent to the set of original claims (modulo axioms).

In this example, the axioms imply that formulas `a`, `b` and `c` are pairwise equivalent, hence all claims are equivalent modulo the axioms. VAMPIRE detects that the formula named `c2` is implied by `c1` (using axiom `ax2`), and then that `c1` is implied by `c3` (using axiom `ax3`). Formulas `c1` and `c2` are thus removed from the claims, resulting in the simplified set of claims containing only `c3`, that is, `c | d`.

Fig. 9 Array partition

```
while ($a \leq m$) do
  if $A[a] \geq 0$
    then $B[b]$ := $A[a]$;$b$ := $b + 1$;
    else $C[c]$ := $A[a]$;$c$ := $c + 1$;
  $a$ := $a + 1$;
end do
```

Consequence elimination turns out to be very useful for pruning a set of automatically generated loop invariants. For example, symbol elimination can generate invariants implied by other generated invariants. For this reason, in the program analysis mode of VAMPIRE, described in the next section, symbol elimination is augmented by consequence elimination to derive a minimised set of loop invariants.

11.2 Program Analysis

Starting with 2011, VAMPIRE can be used to parse and analyse software programs written in a subset of C, and generate properties of loops in these programs using the symbol elimination method introduced in [10]. The subset of C consists of scalar integer variables, array variables, arithmetical expressions, assignments, conditionals and loops. Nested loops are yet unsupported. Running VAMPIRE in the program analysis mode can be done by using:

```
vampire --mode program_analysis problem.c
```

where `problem.c` is the C program to be analysed.

We illustrate the program analysis part of VAMPIRE on Figure 9. This program respectively fills the arrays B and C with the non-negative and negative values of the source array A. Figure 10 shows a (slightly modified) partial output of VAMPIRE's program analysis on this program. VAMPIRE first extracts all loops from the input program, ignores nested loops and analyses every non-nested loop separately. In our example, only one loop is found (also shown in Figure 10). The following steps are then performed for analysing each of the loops:

1. *Find all loop variables* and classify them into variables updated by the loop and constant variables. In Figure 10, the program analyser of VAMPIRE detects that variables B, C, a, b, c are updated in the loop, and variables A, m are constants.
2. *Find counters*, that is, updated scalar variables that are only incremented or decremented by constant values. Note that expressions used as array indexes in loops are typically counters. In Figure 10, the variables a, b, c are classified as counters.
3. *Infer properties of counters*. VAMPIRE adds a new constant `'$counter'` denoting the "current value" of the loop counter, and program variables updated in the loop become functions of the loop counter.
   For example, the variable a becomes the unary function $a(X)$, where $a(X)$ denotes the value of a at the loop iteration $X \leq$ `'$counter'`. The value $a(0)$ thus denotes

Fig. 10 Partial output of VAMPIRE's program analysis.

```
Loops found: 1
Analyzing loop...
-------------------
while (a < m)
{
 if (A[a] >= 0)
 {
    B[b] = A[a];
    b = b + 1;
 }
 else
 {
    C[c] = A[a];
    c = c + 1;
 }
  a = a + 1;
}
-------------------
Variable: B updated
Variable: a updated
Variable: b updated
Variable: m constant
Variable: A constant
Variable: C updated
Variable: c updated
Counter: a
Counter: b
Counter: c
```

```
Collected first-order loop properties
-------------------

27. iter(X0) <=> ($lesseq(0,X0) & $less(X0,'$counter'))

17. a(0) = a0

10. updbb(X0,X2,X3) => bb(X2) = X3

9. updbb(X0,X2,X3) <=>
(let b := b(X0) in (let c := c(X0) in (let a := a(X0) in
(let cc(X1) := cc(X0,X1) in (let bb(X1) := bb(X0,X1) in
($greatereq(aa(a),0) & (aa(a) = X3 & iter(X0) & b = X2)))))))

8. updbb(X0,X2) <=>
(let b := b(X0) in (let c := c(X0) in (let a := a(X0) in
(let cc(X1) := cc(X0,X1) in  (let bb(X1) := bb(X0,X1) in
($greatereq(aa(a),0) & (iter(X0) & b = X2))))))

4.
($greater(X1,X0) & $greater(c(X1),X3) & $greater(X3,c(X0)))
=> ? [X2] : (c(X2) = X3 & $greater(X2,X0) & $greater(X1,X2))

3. $greatereq(X1,X0) => $greatereq(c(X1),c(X0))

1. a(X0) = $sum(a0,X0)
```

the initial value of the program variable a. Since we are interested in loop properties connecting the initial and the current value of the loop variable, we introduce a constant $a0$ denoting the initial value of a (see formula `17`). The predicate $iter(X)$ in formula `27` defines that X is a loop iteration. The properties inferred at this step include formulas `1`, `3` and `4`. For example, formula 3 states that c is a monotonically increasing variable.

4. *Generate update predicates* of updated array variables, for example formula `10`. The update predicate `updbb(X0,X1)` expresses that the array B was updated at loop iteration $X0$ at position $X1$ by value $X2$. The generated properties are in the TPTP syntax, a reason why the capitalised array variables are renamed by VAMPIRE's program analyser. For example, the array B is denoted by *bb*.
5. Derive the formulas corresponding to *the transition relation of the loop* by using let-in and if-then-else constructs. These properties include, for example, formulas `8` and `9`.
6. Generate a symbol elimination task for VAMPIRE by colouring symbols that should not be used in loop invariant. Such symbols are, for example, `'$counter'`, `iter`, and `updbb`.
7. Run VAMPIRE in the consequence elimination mode and output a minimised set of loop invariants. As a result of running VAMPIRE's program analyser on Figure 9, one of the invariants derived by VAMPIRE is the following first-order formula:

```
tff(inv,claim, ![X:$int]:
($greatereq(X,0) & $greater(b,X) => $greatereq(bb(X),0) &
(? [Y:$int]: $greatereq(Y,0) & $greater(a,Y) & aa(Y)=bb(X) )))
```

11.3 Bound Propagation

We have recently extended VAMPIRE with a decision procedure for quantifier-free linear real and rational arithmetic. Our implementation is based on the bound propagation algorithm of [16], BPA in the sequel. When VAMPIRE is run in the BPA mode on a system of linear inequalities, VAMPIRE tries to solve these inequalities by applying the BPA algorithm instead of the superposition calculus. If the system is found to be satisfiable, VAMPIRE outputs a solution. To run VAMPIRE in the BPA mode, one should use:

```
vampire --mode bpa problem.smt
```

where `problem.smt` is a set of linear real and rational inequalities represented in the SMTLIB format [2]. We also implemented various options for changing the input syntax and the representation of real and rational numbers in VAMPIRE; we refer to [36] for details. Our BPA implementation in VAMPIRE is the first step towards combining superposition theorem proving with SMT style reasoning.

11.4 Clausifier

VAMPIRE has a very fast TPTP parser and clausifier. One can use VAMPIRE simply to *clausify formulas*, that is, convert them to clausal normal form. To this end, one should use `--mode clausify`. Note that for classification one can use all VAMPIRE's options for preprocessing. VAMPIRE will output the result in the TPTP syntax, so that the result can be processed by any other prover understanding the TPTP language.

11.5 Grounding

One can use VAMPIRE to *convert an EPR problem to a SAT problem*. This problem will be output in the DIMACS format. To this end one uses the option `--mode grounding`. Note that VAMPIRE will not try to optimise the resulting set of propositional clauses in any way, so this feature is highly experimental.

12 Conclusions

We gave a brief introduction to first-theorem proving and discussed the use and implementation of VAMPIRE. VAMPIRE is fully automatic and can be used in various applications of automated reasoning, including theorem proving, first-order satisfiability checking, finite model finding, and reasoning with both theories and quantifiers. We also described our recent developments of new and unconventional applications of theorem proving in program verification, including interpolation, loop invariant generation, and program analysis.

References

1. L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.
2. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
3. P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing Finite Models by Reduction to Function-Free Clause Logic. *J. of Applied Logic*, 7(1):58–74, 2009.
4. B. Buchberger. An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. *J. of Symbolic Computation*, 41(3-4):475–511, 2006. Phd thesis 1965, University of Innsbruck, Austria.
5. W. Craig. Three uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. of Symbolic Logic*, 22(3):269–285, 1957.
6. J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT — A Distributed and Learning Equational Prover. *J. of Automated Reasoning*, 18(2):189–198, 1997.
7. N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier Science, 2001.
8. H. Ganzinger and K. Korovin. New Directions in Instantiation-Based Theorem Proving. In *Proc. of LICS*, pages 55–64, 2003.
9. K. Hoder, L. Kovács, and A. Voronkov. Case Studies on Invariant Generation Using a Saturation Theorem Prover. In *Proc. of MICAI (1)*, pages 1–15, 2011.
10. K. Hoder, L. Kovács, and A. Voronkov. Invariant Generation in Vampire. In *Proc. of TACAS*, pages 60–64, 2011.
11. K. Hoder, L. Kovács, and A. Voronkov. Playing in the Grey Area of Proofs. In *Proc. of POPL*, pages 259–272, 2012.
12. K. Hoder and A. Voronkov. Comparing Unification Algorithms in First-Order Theorem Proving. In *Proc. of KI*, pages 435–443, 2009.
13. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *Proc. of TACAS*, pages 459–473, 2006.
14. D. Knuth and P. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.
15. K. Korovin and A. Voronkov. Integrating Linear Arithmetic into Superposition Calculus. In *Proc. of CSL*, pages 223–237, 2007.
16. K. Korovin and A. Voronkov. Solving Systems of Linear Inequalities by Bound Propagation. In *Proc. of CADE*, pages 369–383, 2011.
17. L. Kovács, G. Moser, and A. Voronkov. On Transfinite Knuth-Bendix Orders. In *Proc. of CADE*, pages 384–399, 2011.
18. L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE*, pages 470–485, 2009.
19. L. Kovács and A. Voronkov. Interpolation and Symbol Elimination. In *Proc. of CADE*, pages 199–213, 2009.
20. L. Kovács and A. Voronkov. Vampire Web Page - `http://vprover.org`, 2013.
21. B. Löchner and T. Hillenbrand. A Phytography of WALDMEISTER. *AI Commun.*, 15(2-3):127–133, 2002.
22. E. L. Lusk. Controlling Redundancy in Large Search Spaces: Argonne-Style Theorem Proving Through the Years. In *Proc. of LPAR*, pages 96–106, 1992.
23. A. Martelli and U. Montanari. An Efficient Unification Algorithm. *TOPLAS*, 4(2):258–282, 1982.

24. W. W. McCune. OTTER 3.0 Reference Manual and Guide. Technical Report ANL-94/6, Argonne National Laboratory, January 1994.
25. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *Proc. of CAV*, pages 1–13, 2003.
26. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.
27. A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computations*, 36(1-2):101–115, 2003.
28. J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.
29. S. Schulz. E — a Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002.
30. R. Sekar, I.V. Ramakrishnan, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier Science, 2001.
31. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.
32. G. Sutcliffe. The CADE-23 Automated Theorem Proving System Competition - CASC-23. *AI Commun.*, 25(1):49–63, 2012.
33. G. Sutcliffe. The TPTP Problem Library - `http://www.cs.miami.edu/~tptp/`, 2013.
34. Geoff Sutcliffe. The CADE ATP System Competition - `http://www.cs.miami.edu/~tptp/CASC/`, 2013.
35. C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER. Version 0.42. In *Proc. of CADE*, pages 141–145, 1996.
36. `http://www.complang.tuwien.ac.at/ioan/boundPropagation`. Vampire with Bound Propagation.