AVATAR: The Architecture for First-Order Theorem Provers

Andrei Voronkov

University of Manchester, Manchester, UK

Abstract. This paper describes a new architecture for first-order resolution and superposition theorem provers called AVATAR (Advanced Vampire Architecture for Theories and Resolution). Its original motivation comes from a problem well-studied in the past — dealing with problems having clauses containing propositional variables and other clauses that can be split into components with disjoint sets of variables. Such clauses are common for problems coming from applications, for example in program verification and program analysis, where many ground literals occur in the problems and even more are generated during the proof-search.

This problem was previously studied by adding various versions of splitting. The addition of splitting resulted in some improvements in performance of theorem provers. However, even with various versions of splitting, the performance of superposition theorem provers is nowhere near SMT solvers on variable-free problems or SAT solvers on propositional problems.

This paper describes a new architecture for superposition theorem provers, where a superposition theorem prover is tightly integrated with a SAT or an SMT solver. Its implementation in our theorem prover Vampire resulted in drastic improvements over all previous implementations of splitting. Over four hundred TPTP problems previously unsolvable by any modern prover, including Vampire itself, have been proved, most of them with short runtimes. Nearly all problems solved with one of 481 variants of splitting previously implemented in Vampire can also be solved with AVATAR.

We also believe that AVATAR is an important step towards efficient reasoning with both quantifiers and theories, which is one of the key areas in modern applications of theorem provers in program analysis and verification.

Definitions of *Avatar* (from various dictionaries):

(Hindu Mythology) the descent of a deity to the earth in an incarnate form or some manifest shape; the incarnation of a god
(Science Fiction) a hybrid creature, composed of human and alien DNA and remotely controlled by the mind of a genetically matched human being
(Automated Reasoning) a first-order theorem prover, which embodies a SAT solver controlling the prover's behaviour

1 Introduction

The work described in this paper started with an attempt to make further improvement in dealing with problems having clauses containing propositional variables and other clauses that can be split into components with disjoint sets of variables. The problem

A. Biere and R. Bloem (Eds.): CAV 2014, LNCS 8559, pp. 696–710, 2014.

of dealing with such clauses started with splitting with backtracking, implemented in Spass [20] and splitting without backtracking [12] implemented in Vampire [9]. A very extensive investigation of various ways of organising splitting in a theorem prover was undertaken in [7], where both kinds of splitting were augmented with various options, including the use of BDDs and SAT solvers. Though the use of splitting results in the improvement of theorem provers performance, the methods used in them cannot compete with the methods used in SAT solvers on propositional problems or methods used in SMT solvers on ground problems with equality.

In first-order theorem proving, theorem provers based on variants of resolution and superposition calculi (in the sequel simply called *superposition provers*) are predominant. This is confirmed by the results of the last CASC competitions[1], see [19] for a description of CASC. The top three theorem provers Vampire [9], E-MaLeS and E [17] are resolution and superposition-based, while the fourth one iProver [8] implements both an instance-based calculus and resolution with superposition.

Superposition theorem provers use *saturation algorithms*. They deal with a search space consisting of clauses. Inferences performed by saturation algoritms are of three different kinds:

1. *Generating inferences* derive news clause from clauses in the search space. These new clauses can then be immediately simplified and/or deleted by other kinds of inference. Examples of generating inferences are binary resolution and superposition.
2. *Simplifying inferences* replace a clause by another clause that is simpler in some strict sense. Examples of simplifying inferences are demodulation (rewriting by ordered unit equalities) and subsumption resolution (binary resolution inference whose conclusion subsumes one of the premises).
3. *Deletion inferences* delete clauses from the search space. Examples of deletion inferences are subsumption and tautology deletion.

On hard problems the search space of superposition provers is often growing rapidly, and simplifications and deletions consume considerable time. Performance of such provers degrades especially fast when they generate many clauses having more than one literal (*multi-literal clauses* for short) and heavy clauses (clauses of large sizes). There are several reasons for this degradation of performance:

1. The complexity of algorithms implementing inference rules depends on the size of clauses. For example, subsumption and subsumption resolution are known to be NP-complete and algorithms implementing them are exponential in the number of literals in clauses.
2. Storing heavy clauses requires more memory. Moreover, every literal in a clause (and sometimes every term occurring in such a literal) are normally added to one or more indexes. Index maintenance requires considerable space and time and operations on these indexes slow down significantly when the indexes become large.
3. Generating inferences applied to heavy clauses usually generate heavy clauses. Generating inferences applied to clauses with many literals usually generate clauses with many literals. For example, resolution applied to two clauses containing n_1 and n_2 literals typically gives a clause with $n_1 + n_2 - 2$ literals.

[1] `http://www.cs.miami.edu/~tptp/CASC/24/`

To deal with multi-literal and heavy clauses, one can simply start discarding them after some time, thus losing completeness as in [14]. Alternatively, one can use *splitting*. There are two kinds of splitting described in the literature: splitting with backtracking (originally introduced in SPASS [20]) or splitting without backtracking (originally introduced in Vampire [13]).

In this paper we introduce a new way of splitting clauses, driven by a SAT or an SMT solver. This results in a new architecture for first-order theorem proving, which we call AVATAR. We show that the use of AVATAR instead of standard architectures results in a considerable improvement in the performance of theorem provers. Hundreds of problems unsolvable by any prover for years were solved when AVATAR was implemented in Vampire. Moreover, we believe that AVATAR is a significant step towards major improvements in one of the main problems in modern first-order theorem proving: reasoning with both quantifiers and theories.

2 Preliminaries

We assume that the reader is familiar with SAT solving and has some knowledge of first-order theorem provers. A deeper knowledge of superposition theorem proving, as well as SMT solving, is useful, but not necessary, since we give some background material on saturation algorithms implemented in superposition theorem provers.

Recall that a *(first-order) clause* is a disjunction $L_1 \vee \ldots \vee L_n$ of *literals*, where a literal is an atomic formula or a negation of an atomic formula. A literal or clause is *ground* if it contains no occurrences of variables. In the context of splitting we sometimes consider a clause as a set of its literals. In other words, we assume that clauses do not contain multiple occurrences of the same literal and clauses equal up to permutation of literals are considered equal. We assume that all predicates and functions in first-order logic are uninterpreted and that the language may contain (but not necessarily contains) the equality predicate, denoted by $=$. The empty clause is denoted by $\square$.

Unlike SMT solving, clauses containing variables are considered implicitly universally quantified. Suppose that C is a clause with variables $x_1, \ldots, x_k$. Then $\forall C$ will denote the formula $(\forall x_1) \ldots (\forall x_k)C$, also called the *universal closure* of C. In first-order theorem proving the semantics of a clause is its universal closure, so a set of clauses $C_1, \ldots, C_n$ is satisfiable if and only if so is the set of formulas $\forall C_1, \ldots, \forall C_n$. Any clause obtained by applying a substitution to a clause C is called an *instance* of C. If this instance is also a ground clause, it is called a *ground instance* of C. Satisfiability of a set of clauses in first-order predicate logic (in the SMT terminology it is the logic of equality and uninterpreted predicates and functions) is characterised by the Herbrand theorem: a set S of clauses is unsatisfiable if and only if some finite set of ground instances of clauses in S in unsatisfiable.

Our next aim is to explain splitting. In very simple terms, splitting is based on the following idea. Suppose that S is a set of (first-order) clauses and $C_1 \vee C_2$ a clause such that the variables of C_1 and C_2 are disjoint. Then $\forall(C_1 \vee C_2)$ is equivalent to $(\forall C_1) \vee (\forall C_2)$, which implies that the set $S \cup \{C_1 \vee C_2\}$ is unsatisfiable if and only if both $S \cup \{C_1\}$ and $S \cup \{C_2\}$ are unsatisfiable.

Let $C_1, \ldots, C_n$ be clauses such that $n \geq 2$ and all the C_i's have pairwise disjoint sets of variables. Then we say that the clause $D \stackrel{\text{def}}{=} C_1 \vee \ldots \vee C_n$ is *splittable* into *components* $C_1, \ldots, C_n$. We will also say that the set $C_1, \ldots, C_n$ is a *splitting* of D. For example, every ground multi-literal clause is splittable. There may be more than one way to split a clause, however there is always a unique splitting such that each component C_i is non-splittable; we call this splitting *maximal*. It is easy to see that a maximal splitting has the largest number of components and every splitting with the largest number of components is the maximal one. There is a simple algorithm for finding the maximal splitting of a clause [12], which is, essentially, the union-find algorithm.

In the sequel, when we speak about a splitting of a clause we will only consider maximal splittings and only deal with components that are non-splittable. We will denote arbitrary clauses by D and components by C, maybe with indexes.

Splittable clauses appear especially often when theorem provers are used for software verification and static analysis. Problems used in these applications usually have a large number of ground clauses (coming from program analysis) and a small number of non-ground clauses (for example, axiomatisations of memory or objects).

3 Saturation Algorithms

In this section we briefly discuss *saturation algorithms with redundancy elimination* used in superposition theorem provers. Essentially, a saturation algorithm works with a set of clauses S (the current search space) and uses a collection of generating, simplifying and deletion inferences. The theoretical basis of saturation algorithms is the notion of *redundancy* given e.g., in [1]: a clause D is redundant if D is a logical consequence of clauses in the search space, which are strictly smaller than D w.r.t. a simplification ordering $\succ$ on clauses. Both simplifying and deletion inferences in saturation algorithms are designed in such a way that they only remove redundant clauses.

There is more than one saturation algorithm. For illustration we will use the *Otter saturation algorithm* [9], though AVATAR works equally well with other saturation algorithms. For an overview of saturation algorithms we refer to [15,9].

A simplified description of the Otter saturation algorithm is shown in Figure 1. The algorithms maintains three sets of clauses:

1. $active$: the set of clauses selected for generating inferences. The algorithm is designed in such a way that all generating inferences among active clauses are applied.
2. $passive$: clauses that are waiting to be activated. The Otter saturation algorithm uses passive clauses for simplifying and deletion inferences.
3. $unprocessed$: clauses that have been generated recently. Unprocessed clauses are waiting in a queue for a *retention test*, which normally includes simplification and deletion inferences applied to these clauses. If a clause C passes the retention test, this clause (or a clause obtained by simplifying C) is added to passive clauses, otherwise it is discarded.

At every step, the algorithm either processes a clause new, picked from unprocessed clauses, or performs generating inferences with the so-called *given clause*, which is the clause most recently added to $active$.

```
input: $init$: set of clauses;
var $active, passive, unprocessed$: set of clauses;
var $given, new$: clause;
$active := \varnothing$;
$unprocessed := init$;
loop
  while $unprocessed \neq \varnothing$
    $new := pop(unprocessed)$;
    if $new = \Box$ then return unsatisfiable;
    if $retained(new)$ then                                        (* retention test *)
      simplify $new$ by clauses in $active \cup passive$ ;         (* forward simplification *)
      if $new = \Box$ then return unsatisfiable;
      if $retained(new)$ then                                      (* another retention test *)
        delete and simplify clauses in $active$ and                (* backward simplification *)
                                     $passive$ using $new$;
        move the simplified clauses to $unprocessed$;
        add $new$ to $passive$
  if $passive = \varnothing$ then return satisfiable or unknown
  $given := select(passive)$;                                      (* clause selection *)
  move $given$ from $passive$ to $active$;
  $unprocessed := forward\_infer(given, active)$;                  (* forward generating inferences *)
  add $backward\_infer(given, active)$ to $unprocessed$;
                                                                   (* backward generating inferences *)
```

Fig. 1. Otter Saturation Algorithm

All operations performed by the saturation algorithm that may take considerable time to execute, are normally implemented using *term indexing*, that is, building a special purpose index data structure that makes the operation faster. For example, all theorem provers with built-in equality reasoning have an index for forward demodulation (rewriting by ordered unit equalities from $active \cup passive$).

4 AVATAR

In this section we describe AVATAR and how it handles splitting. AVATAR consists of two components: a resolution (or resolution and superposition) theorem prover FO and a SAT solver SAT. Later we will consider how an SMT solver can be used in place of SAT. The SAT solver stores propositional clauses, which considered clause components as propositional literals. To consider them as propositional literals, we will use a mapping $[\cdot]$ from components to propositional literals. This mapping satisfies the following properties:

1. $[C]$ is a positive literal if and only if C is either a non-ground component or a positive ground literal;
2. For a negative ground component $\neg C$ we have $[\neg C] = \neg[C]$.
3. $[C_1] = [C_2]$ if and only if C_1 is equal to C_2 up to variable renaming and symmetry of equality.

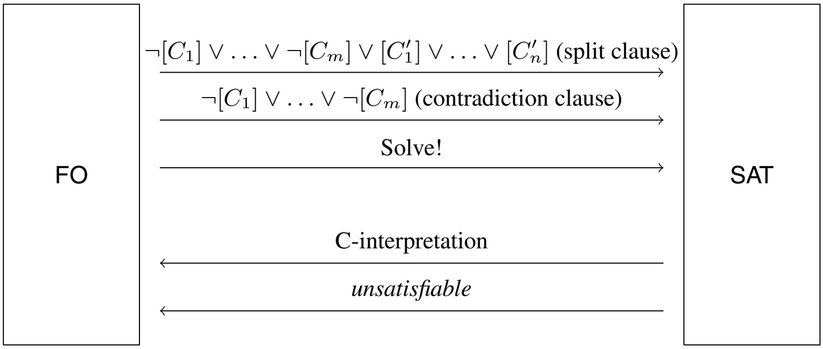
FO
SAT
$\neg[C_1] \vee \ldots \vee \neg[C_m] \vee [C'_1] \vee \ldots \vee [C'_n]$ (split clause)
$\neg[C_1] \vee \ldots \vee \neg[C_m]$ (contradiction clause)
Solve!
C-interpretation
unsatisfiable

Fig. 2. Cooperation between the components of AVATAR

To implement this mapping, Vampire uses a *component index*, which maps every component C that is either positive or non-ground, into $[C]$. For every such component C passed to this index, if C is equal to an already stored component C' up to variable renaming and symmetry of equality, the index returns $[C']$, otherwise it introduces a new propositional variable $[C]$ and stores the association between C and $[C]$. We call a C-interpretation, or a *component interpretation* any set of propositional variables of the form $[C]$ or their negations, which does not contain both a variable and its negation. The definition of a truth of a propositional variable literal in a C-interpretation is standard. If, for a component C, neither $[C]$, not $\neg[C]$ belongs to the interpretation, $[C]$ is considered undefined, that is, neither true nor false.

During the proof search, FO and SAT exchange information. The information exchange is described in Figure 2.

In a nutshell, AVATAR works as follows. The superposition prover FO works as usual, using a saturation algorithm. The difference is in the treatment of splittable clauses. If there is a splittable clause $C_1 \vee \ldots \vee C_n$ with components $C_1, \ldots, C_n$, which passed the retention test, it is not added to $\mathit{passive}$. Instead, $[C_1] \vee \ldots \vee [C_n]$ it is passed to the SAT solver. The SAT solver adds the new clause to existing clauses and checks all clauses for satisfiability. If it is unsatisfiable, we are done. Otherwise, it computes a C-interpretation I, which is a model of all clauses stored in it. For each literal in the interpretation, if this literal has a form $[C]$ for some component C, the component C is passed to FO where it is used as an *assertion*. The exception are literals of the form $\neg[C]$, where C is a non-ground component, since such a literal does not correspond to any component.

To explain the cooperation in more detail, we should modify the superposition calculus to deal with these assertions. The description is similar, but not the same as in splitting with backtracking.

An *assertion* is a finite set of components. A *clause with assertions*, or simply an *A-clause* is a pair, consisting of a clause D and an assertion A. Such a clause with assertions will be denoted by $(D \leftarrow A)$, or simply D when the assertion A is empty. We will denote assertions by A and A-clauses as F. An A-clause $(D \leftarrow C_1, \ldots, C_m)$ is logically equivalent to $\forall D \vee \neg\forall C_1 \vee \ldots \vee \neg\forall C_m$ (or, equivalently, to $\forall C_1 \wedge \ldots \wedge \forall C_m \rightarrow \forall D$). A standard clause D can be considered as an A-clause with the empty set of assertions. We will extend the notation $[\cdot]$ to assertions: for an assertion $A = \{C_1, \ldots, C_m\}$, we define $[A] = \{[C_1], \ldots, [C_m]\}$.

We call an A-clause $(D \leftarrow A)$ *splittable* if the clause D is splittable. Likewise, every A-clause of the form $(\square \leftarrow A)$ is called an *empty* A-clause. We can change the superposition calculus (or any other calculus on clauses) to a calculus on clauses with assertions by turning any rule of the superposition calculus

$$\frac{D_1 \quad \cdots \quad D_k}{D}$$

into a set of rules

$$\frac{(D_1 \leftarrow A_1) \quad \cdots \quad (D_k \leftarrow A_k)}{(D \leftarrow A_1 \cup \ldots \cup A_k)}\ ,$$

where $A_1, \ldots, A_k$ are assertions. Later we will explain how the addition of assertions affects simplification and deletion rules.

AVATAR uses A-clauses instead of ordinary clauses. At each time moment, the components used in assertions are those that are computed by the SAT solver as its last model. Since this model changes over time, clauses with assertions can be added and deleted.

We are now ready to describe the AVATAR algorithm. It is defined as a sequence of steps performed by the superposition prover FO and the SAT solver SAT. These steps are interleaved. Each step performed by the superposition prover is followed by a step by the SAT solver and vice versa. After each step performed by FO, some information is passed from it to SAT, as shown in Figure 2. Likewise, after each step performed by SAT, some information is passed from it to FO. These steps are described in detail in the next two sections.

5 The SAT Algorithm

We start with the SAT algorithm since it is simpler that the algorithm employed by FO. Essentially, the SAT solver is behaving like a standard incremental SAT solver. It receives, from time to time, new propositional clauses from FO and checks, upon a "solve" request, satisfiability of the clauses it stores. If they are satisfiable, it passes back to FO a C-interpretation satisfying all the propositional clauses. Otherwise, it returns *unsatisfiable*.

6 The FO Algorithm

In a nutshell, the FO algorithm behaves like a standard saturation algorithm. The main differences are that it operates on A-clauses and that splittable clauses are not stored. Instead, for each splittable or empty A-clause $(C_1 \vee \ldots \vee C_n \leftarrow C'_1, \ldots, C'_m)$, FO passes the propositional clause $[C_1] \vee \ldots \vee [C_n] \vee \neg[C'_1] \vee \ldots \vee \neg[C'_m]$ to SAT.

In reality, the FO algorithm is more sophisticated than standard saturation algorithms because of the way it treats simplified and deleted A-clauses. To illustrate the problem, consider an example. Suppose that we have two clauses D, D' such that D subsumes D'. If D and D' occur in the search space of a standard saturation algorithm, D' will be treated as redundant and can be deleted. In AVATAR, we deal with A-clauses. Suppose that A-clauses $(D \leftarrow A)$ and $(D' \leftarrow A')$ occur in the current search space and D subsumes D'. If $A \subseteq A'$, then $(D' \leftarrow A')$ can still be considered as redundant and deleted. If not, we can only delete it temporarily, since the model computed by the SAT solver can change and make a literal in $[A]$ false, while all the literals in $[A']$ remain true. In this case $(D \leftarrow A)$ will later be removed from the search space and, to preserve completeness, $(D' \leftarrow A')$ must then be undeleted.

For this reason we introduce a special storage for A-clauses that can be temporarily deleted and then undeleted. This storage will be denoted in the saturation algorithm as $locked$. Elements of $locked$ are pairs (F, λ), where F is an A-clause and λ a set of C-literals. If $(F, \lambda) \in locked$, we will informally call λ a *lock* of F. The same A-clause F can occur in $locked$ with different locks.

We say that a C-interpretation I *unlocks* a pair $((C \leftarrow A), \lambda)$ if

1. all C-literals in $[A]$ are true in I;
2. at least one C-literal in λ is either false or undefined in A.

When a pair (F, λ) is added to $locked$, all of the literals in λ are true in the current model int computed by the SAT solver (this follows from a general invariant of the AVATAR algorithm: for every A-clause $(D \leftarrow A)$ in the search space, each literal in $[A]$ is true in this model). If one of the literals in λ later becomes false or undefined, the A-clause F must be unlocked by removing it from $locked$ and adding it to the set of unprocessed clauses.

The FO algorithm is shown on Figure 3. Its parts that are specific to AVATAR are marked by ✓. Simplifications will be explained separately.

The AVATAR algorithm maintains, in addition to the sets $active$, $passive$, and $unprocessed$, the following collections.

- A C-interpretation int returned by the SAT solver. This interpretation makes the assertions of all stored clauses, apart from locked ones, true. We store this interpretation to maintain locking and unlocking operations. To check which clauses should be locked or unlocked, we compute, at each step, the difference between the current and the previous values of int.
- The set of A-clauses sat_queue waiting to be passed to the SAT solver. We store A-clauses in sat_queue instead of passing them immediately to the SAT solver because changes in the model found by the SAT solver can induce considerable changes in A-clauses and other data structures used by the saturation algorithm, so recomputing this model too often may result in the overall degradation of performance. The only exception is made when an empty A-clause is derived. In this case we recompute the interpretation immediately, since the new model int will make the given clause (and potentially many other stored clauses) locked or even deleted.
- The set $locked$ of locked A-clauses with locks. A-clauses in this set are temporarily deleted, since for some components C in their assertions, $[C]$ can be false or undefined in the current C-interpretation int. However, they can be unlocked later.

```
  input: $init$: set of clauses;
  var $active$, $passive$, $unprocessed$: sets of A-clauses, initially empty;
  var $given$, $new$: A-clauses;
✓ var $sat\_queue$: set of A-clauses, initially empty;
✓ var $locked$: set of pairs (A-clause,lock), initially empty;
✓ var $int$: C-interpretation, initially empty;
  forall $D \in init$
✓   if $D$ is splittable or empty
✓     then move it to $sat\_queue$
      else move it to $unprocessed$
  loop                                                                  (* main loop *)
✓   if $sat\_queue \neq \varnothing$ then
✓     forall A-clauses $(C_1 \vee \ldots \vee C_n \leftarrow C'_1, \ldots, C'_m) \in sat\_queue$
✓       pass the clause $[C_1] \vee \ldots \vee [C_n] \vee \neg[C'_1] \vee \ldots \vee \neg[C'_m]$ to SAT
✓     $sat\_queue := \varnothing$;
✓     send the request "solve" to SAT;
✓     if SAT returns unsatisfiable, then return unsatisfiable;
✓     $int$ := the the C-interpretation returned by SAT
✓     forall pairs $((C \leftarrow A), \lambda) \in locked$ unlocked by $int$
✓       remove this pair from $locked$ and add $(C \leftarrow A)$ it to $unprocessed$;
✓     forall A-clauses $(C \leftarrow A)$ in the set $active$, $passive$ or $unprocessed$ such that $[A] \not\subseteq int$
✓       remove $(C \leftarrow A)$ from this set and add $((C \leftarrow A), \varnothing)$ to $locked$;
✓     forall components $[C] \in int$ such that $(C \leftarrow C) \notin active \cup passive \cup unprocessed$
        add $(C \leftarrow C)$ to $unprocessed$
    forall $new \in unprocessed$
✓     if $new$ is splittable or empty
✓       then add $new$ to $sat\_queue$
        else if $retained(new)$                                         (* retention test *)
✓           then simplify $new$ by clauses in $active \cup passive$ ;   (* forward simplification *)
✓                 if $new$ was added to $unprocessed$
✓                                                                       (* backward simplification *)
✓                 then simplify clauses in $active \cup passive$ by $new$
✓   if $sat\_queue$ is non-empty, then start the main loop again;
    if $passive = \varnothing$ then return satisfiable or unknown;
    $given := select(passive)$;                                         (* clause selection *)
    move $given$ from $passive$ to $active$;
    $unprocessed := forward\_infer(given, active)$;                     (* forward generating inferences *)
    add $backward\_infer(given, active)$ to $unprocessed$;              (* backward generating inferences *)
```

Fig. 3. The FO Algorithm

7 Simplifications

We already gave a hint as to how simplifications are performed, when we discussed the use of locking and treatment of subsumed clauses.

Consider now simplification rules. All simplification rules in Vampire and other superposition provers have the following form:

```
if $new$ is unconditionally deleted by A-clauses in $active \cup passive$
    then do nothing
else if $new$ is conditionally deleted by A-clauses in $active \cup passive$ with a lock $\lambda$
    then add $(new, \lambda)$ to $locked$
else if $new$ is unconditionally simplified by A-clauses in $active \cup passive$ into $new'$
    then add $new'$ to $unprocessed$
else if $new$ is conditionally simplified by A-clauses in $active \cup passive$ into $new'$
      with a lock $\lambda$
    then add $new'$ to $unprocessed$ ;
         add $(new, \lambda)$ to $locked$
```

Fig. 4. Forward Simplification

$$\frac{D_1 \quad \cdots \quad \cancel{D_m}}{D} \; . \tag{1}$$

This means that D is a logical consequence of $D_1, \ldots, D_m$ and addition of D to the search space makes D_m redundant. There are three commonly used simplification rules: subsumption, subsumption resolution, and demodulation (rewriting by unit equalities). A inference on A-clauses corresponding to (1) is

$$\frac{(D_1 \leftarrow A_1) \quad \cdots \quad (D_m \leftarrow A_m)}{(D \leftarrow A)} \; ,$$

where $A = A_1 \cup \ldots \cup A_m$. If $A = A_m$, then $(D_m \leftarrow A_m)$ can be safely deleted. Otherwise, consider the assertion $A' = A - A_m$. At the moment this inference is performed, all literals in $[A]$ are true in the current C-interpretation int. However, there may be a moment in the future, when $[A_m]$ is still true, while some literals in $[A']$ false. In that case $(D_m \leftarrow A_m)$ must be put back in the search space. Thus, we lock $(D_m \leftarrow A_m)$ with the lock A'. Any change to int, which makes a C-literal in $[A']$ false will trigger unlocking of this A-clause.

To define simplifications formally, we introduce new notions. Suppose that a clause C can be simplified into a clause C' using clauses $C_1, \ldots, C_m$. Consider A-clauses $(C \leftarrow A)$ and $F_i = (C_i \leftarrow A_i)$ for $i = 1, \ldots, m$. Define $A' = A_1 \cup \ldots \cup A_m$. If $A' \subseteq A$, then we say that $(C \leftarrow A)$ is *unconditionally simplified by* $F_1, \ldots, F_m$ *into* $(C \leftarrow A)$. If $A' \nsubseteq A$, then we say that $(C \leftarrow A)$ is *conditionally simplified by* $F_1, \ldots, F_m$ *into* $(C \leftarrow A \cup A')$ *with the lock* $A' - A$. In a similar way we can define notions $(C \leftarrow A)$ *is unconditionally deleted by* $F_1, \ldots, F_m$ and $(C \leftarrow A)$ *is conditionally deleted by* $F_1, \ldots, F_m$ *with the lock* $A' - A$.

Forward simplification in AVATAR is shown in Figure 4. Backward simplification is similar and not included in this paper.

To avoid excessive locking and unlocking, it is desirable to have a SAT solver, which tries to return a model similar to the previously returned one. To this end, one can use the following simple rule: if a new A-clause passed to the SAT solver contains a C-literal $[C]$ undefined in the previous C-interpretation, we satisfy this clause by making $[C]$ true. Such A-clauses are common and appear when a new component is found. Using this rule also helps the SAT solver, since it does not have to be run at all when all recently added A-clauses have this property.

When all literals in a new A-clause passed to the SAT solver are false in the current C-interpretation int, this interpretation must change. For example, this always happens when we derive an empty A-clause $(\Box \leftarrow A)$. *Phase saving* in SAT solvers introduced in [10] assigns to a propositional variable a value that was most recently assigned to it. Although we did not make experiments with various strategies in a SAT solver, phase saving seems to be useful for achieving a "small model difference" effect. We also use a data structure allowing one to find locked and unlocked clauses upon changes in the SAT solver model in time linear in the size of the number of found clauses plus the number of variables that changed their values.

8 Term Indexing

When we discuss the use of splitting in superposition theorem provers, it is very important to understand how the use of splitting affects other components of such provers. The efficiency and power of modern superposition theorem provers comes from two techniques: *redundancy elimination* (see [1] for the theory and [14] for the implementation aspects) and *term indexing* [18].

Even when simplifications are used, the search space can quickly grow to hundreds of thousands of clauses. To perform inferences on such a large search space efficiently, theorem provers maintain several indexes storing information about terms and clauses. These indexes make it easier to find candidates for inferences. In some cases inferences can be performed only by using the relevant index, without retrieving clauses used for these inferences. The number of different indexes in theorem provers varies and can be as many as about 10. Frequent insertions and deletions in an index can affect performance of a theorem prover. A typical example is when a theorem prover generates an equality $a = b$ between two constants and uses it to rewrite a into b. For nearly all indexing techniques used in the superposition theorem provers, every term and clause containing a must be removed from all indexes and a new term containing b inserted in them again. Doing this single simplification step on an indexed set with 100,000 clauses can take a very long time.

In AVATAR, clauses can be locked and then unlocked. This happens often when the number of clauses stored by the SAT solver grows and it recomputes its C-model int. Frequent deletions of a clause from all indexes it is stored in, followed by its insertion in these indexes, can be very expensive. There is an alternative to deleting locked clauses and information about them from indexes. If an A-clause is deleted or simplified conditionally, we do not remove it from indexes at all. Instead, we change index retrieval operations. For each successful retrieval operation we check if the result is a locked clause. If it is locked, we ignore the retrieved clause and the corresponding inference. This alternative is not yet implemented in Vampire and requires further experiments.

9 Experiments

For our experiments we reproduced the experiments from [7], where various versions of splitting were considered. In fact, AVATAR and some decisions made in it (such as treatment of locked literals in indexes and addition of negations of ground components) are due to what we learned from experiments [7]. We will not show all results from [7] but only consider the most relevant part, where we compare the performance of different versions of splitting. Note that such comparisons are very hard for the following reasons:

1. One cannot simply compare AVATAR to, say, splitting with backtracking, since the latter can be used with different options, giving very different results.
2. In general, a value of a strategy (a collection of parameter-value pairs) is hard to understand. Some strategies perform very well on the average but cannot solve problems unsolvable by other strategies. Modern theorem provers treat hard problems with a cocktail of strategies. For example, Vampire has a CASC mode [9] doing exactly that. A collection of strategies, each of which is bad on the average, can easily outperform a collection of strategies, each of which is good on the average. On the other hand, having too many strategies is not good, since running all of them may consume a considerable time, so strategies that solve many problems are useful as part of a collection: indeed, theorem provers are normally used with short time limits, so that one cannot afford running too many strategies on a problem.

New strategies are most useful if they solve many new problems, and especially with short running times. In this case they can be used to create more powerful cocktails than those used before.

Our experiments have shown that AVATAR shows outstanding results both in terms of its average performance and in the number of problems that it can solve and that were previously unsolvable by any existing prover.

For benchmarking we used unsatisfiable TPTP problems having non-unit clauses and rating greater than 0.2 and less than 1. Essentially, the rating is the percentage of existing provers that cannot solve a problem. For example, rating greater than 0.2 means that less than 80% of existing theorem provers can solve the problem. Likewise, rating 1 means that the problem cannot be solved by the existing provers. However, the rating evaluation uses a single mode of every prover, so it is possible that the same prover can solve a problem of rating 1 using a different mode. For this reason, we also added problems of rating 1 that are solvable by some version of Vampire. We excluded very large problems since for them it was preprocessing, but not other options, that affects results the most. This resulted in selecting 6892 TPTP problems for our experiments.

To conduct our experiments, we took a Vampire strategy that is believed to be nearly the best in the overall number of solved problems, and generated the 481 variations of this strategy obtained by setting the splitting parameters to all possible values described in [7], In addition, we used a single run of this strategy using AVATAR.

Only 5,273 (about 77% of all problems) were solved by at least one splitting strategy. The results are summarised in Table 1. They show that AVATAR is very robust, resulting in a considerable increase of the number of solved problems over the best strategies using other versions of splitting.

Table 1. Problems solved by each setting of the splitting strategy

splitting	strategies	worst	average	best
off	25	3833	3869	3880
backtracking	64	2538	3889	4381
non-backtracking	416	2489	3595	4126
AVATAR	1	4716	4716	4716

The second series of experiments was run on our cluster of 45 servers. Each server has 16G RAM and 4 cores. We used 3 cores on each server since we observed that using all 4 often results in the operating system putting two instances of Vampire on the same core. This results in 135 instances on Vampire running in parallel. The experiments were run for over 6 months in 2012–2013. The aim of this series of experiments was to solve as many TPTP problems overall as possible; and its results were used to configure Vampire for the last CASC competition CASC-24. Eventually, Vampire with AVATAR was able to solve 421 problems unsolvable by Vampire without AVATAR and by any other prover. To get the results of other provers, we used the file `ProblemAndSolutionStatistics` shipped with TPTP, which records results on every TPTP problem by nearly all theorem provers in the recent history. On the contrary, Vampire using splitting with and without backtracking was able to solve only 17 problems unsolvable by any strategy using AVATAR. Solving over 400 previously unsolvable problems is a remarkable result since such all these problems are very hard. In the past, the implementation of various novelties in Vampire would normally result in solving from a few to about 30 previously unsolvable problems.

The experimental results were so successful that all previously implemented code for handling splitting was completely removed from the latest versions of Vampire, resulting in considerable simplifications in its code and better maintainability.

10 Using an SMT Solver or Other Theory Solvers

Another interesting feature of AVATAR is that for a combination of first-order logic with theories one can use any theory solver instead of a SAT solver. In particular, for problems with equality one can use an SMT solver for logic with equality and uninterpreted functions. Non-ground components are then treated in the SMT solver as before, as propositional variables. Ground components can be used by the SMT solver as theory literals. We added to Vampire a very simple SMT solver for logic with equality and uninterpreted functions. This addition allowed us to solve some TPTP problems previously unsolved by any prover, including Vampire using AVATAR and a SAT solver.

The SAT and the SMT solvers implemented in Vampire are very simple and much weaker than best SAT and SMT solvers. It will be interesting to see how the use of better SAT and SMT solvers affects the performance of AVATAR.

There is an interesting option that can be used for logic with equality and maybe other theories. Instead of passing back to FO a propositional model, an SMT solver can pass some canonical representation of the congruence relation computed by it. Also, the SMT solver can use ground (and maybe also non-ground) unit equalities produced by the superposition prover. We leave these extensions as future work.

11 Related Work

The author believes that proving theorems with both quantifiers and theories is the main problem in modern first-order theorem proving. In particular, it is crucial for applications of theorem provers in program analysis and also in interactive theorem provers. AVATAR offers an architecture different from those proposed in first-order theorem provers able to handle theories, including SPASS+T [2,11], Z3 [5], CVC4 [3], Princess [16] and Beagle [4].

This paper was motivated by our analysis of the results [7], which contains an extensive discussion of splitting in superposition theorem provers. In particular, it uses splitting in various forms and SAT solvers, but not in the way discussed in this paper. Earlier work on splitting includes [20] and [13].

Paper [6] describes a calculus $\mathrm{DPLL}(\Gamma)$ using a superposition prover together with a SAT or an SMT solver (Z3) in a way similar to AVATAR. Ground literals decided and implied by the SAT solver were used as hypotheses to first-order clauses. Our approach is different in several aspects:

1. We use arbitrary components, while $\mathrm{DPLL}(\Gamma)$ uses only ground literals;
2. We consider the SAT solver as a black box producing models, while in $\mathrm{DPLL}(\Gamma)$ the SAT solver and the superposition prover architectures and calculi are mutually dependent. The backjump rule and locking (disabling) first-order clauses in $\mathrm{DPLL}(\Gamma)$ essentially uses decision levels of the SAT solvers. The use of decision levels makes $\mathrm{DPLL}(\Gamma)$ is very similar to splitting with backtracking, though with some improvements due to the use of a SAT solver.

Also, [6] discuss very different benchmarks, where theory reasoning and E-matching are often required to solve problems.

12 Conclusion

We described a new architecture AVATAR for first-order theorem provers. In this architecture, splitting in a theorem prover is driven by a SAT (or an SMT) solver. When the input problem is ground, AVATAR is as efficient as a SAT solver (or an SMT solver for logic with equality). On non-ground problems, an implementation of AVATAR in Vampire outperforms the previous versions of Vampire by a very large margin. In particular, using AVATAR allowed us to solve 421 TPTP problems previously unsolvable by any first-order theorem prover.

We believe that AVATAR will become a standard architecture for future first-order theorem provers and can be especially successful in reasoning with both quantifiers and theories. It turned out to be effective in passing information from a first-order theorem prover to a SAT or an SMT solver. Nonetheless, AVATAR does not solve the reverse problem: passing information from an SMT solver to the first-order prover, which is currently done by other approaches, such as E-matching.

Acknowledgments. We thank Krystof Hoder, who implemented the first version of AVATAR, Giles Reger, Laura Kovács and Martin Suda for discussions and reading preliminary versions of this paper.

References

1. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning. ch. 2, vol. I, pp. 19–99. Elsevier Science (2001)
2. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierarchic first-order theories. Appl. Algebra Eng. Commun. Comput. 5, 193–212 (1994)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
4. Baumgartner, P., Waldmann, U.: Hierarchic Superposition with Weak Abstraction. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 39–57. Springer, Heidelberg (2013)
5. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. de Moura, L., Bjørner, N.S.: Engineering DPLL(T) + saturation. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 475–490. Springer, Heidelberg (2008)
7. Hoder, K., Voronkov, A.: The 481 ways to split a clause and deal with propositional variables. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 450–464. Springer, Heidelberg (2013)
8. Korovin, K.: iProver—an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
9. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013)
10. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
11. Prevosto, V., Waldmann, U.: SPASS+T. In: Proc. of ESCoR, pp. 18–33 (2006)
12. Riazanov, A., Voronkov, A.: Splitting without backtracking. In: Nebel, B. (ed.) 17th International Joint Conference on Artificial Intelligence, IJCAI 2001, vol. 1, pp. 611–617 (2001)
13. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Commun. 15(2,3), 91–110 (2002)
14. Riazanov, A., Voronkov, A.: Limited resource strategy in resolution theorem proving. Journal of Symbolic Computations 36(1-2), 101–115 (2003)
15. Riazanov, A., Voronkov, A.: Limited resource strategy in resolution theorem proving. Journal of Symbolic Computations 36(1-2), 101–115 (2003)
16. Rümmer, P.: E-Matching with Free Variables. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 359–374. Springer, Heidelberg (2012)
17. Schulz, S.: E – a brainiac theorem prover. Journal of AI Communications 15(2-3), 111–126 (2002)
18. Sekar, R., Ramakrishnan, I.V., Voronkov, A.: Term indexing. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning. ch. 26, vol. II, pp. 1853–1964. Elsevier Science (2001)
19. Sutcliffe, G.: TPTP, TSTP, CASC, etc. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 6–22. Springer, Heidelberg (2007)
20. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning. ch. 27, vol. II, pp. 1965–2013. Elsevier Science (2001)