Playing with AVATAR

Giles Reger, Martin Suda and Andrei Voronkov *

University of Manchester, Manchester, UK

Abstract. Modern first-order resolution and superposition theorem provers use saturation algorithms to search for a refutation in clauses derivable from the input clauses. On hard problems, this search space often grows rapidly and performance degrades especially fast when long and heavy clauses are generated. One approach that has proved successful in taming the search space is *splitting* where clauses are split into components with disjoint variables and the components are asserted in turn. This reduces the length and weight of clauses in the search space at the cost of keeping track of splitting decisions.
This paper considers the new AVATAR (Advanced Vampire Architecture for Theories And Resolution) approach to splitting which places a SAT (or SMT) solver at the centre of the theorem prover and uses it to direct the exploration of the search space. Using such an approach also allows the propositional part of the search space to be dealt with outside of the first-order prover.
AVATAR has proved very successful, especially for problems coming from applications such as program verification and program analysis as these commonly contain clauses suitable for splitting. However, AVATAR is still a new idea and there is much left to understand. This paper presents an in-depth exploration of this new architecture, introducing new, highly experimental, options that allow us to vary the operation and interaction of the various components. It then extensively evaluates these new options, using the TPTP library, to gain an insight into which of these options are essential and how AVATAR can be optimally applied.

1 Introduction

AVATAR [9] is a new architecture for first-order resolution and superposition theorem provers that places a SAT (or SMT) solver at the centre of the theorem prover to direct exploration of the search space. Certain options control this exploration and this paper describes these options in detail and extensively evaluates how they impact proof-search with the aim of highlighting those parameters that lead to a) more problems being solved, and b) problems being solved more efficiently.

Modern first-order resolution and superposition provers use saturation algorithms, i.e., they attempt to construct a saturated set of all clauses derivable from an initial set. A common issue is a rapidly growing search space containing multi-literal and heavy clauses. A multi-literal clause is one with many literals and a heavy clause is one with many symbol occurrences. Processing such clauses is expensive and typically leads to less of the search space being explored in a given time.

One solution is to throw away such clauses that will probably not be used within the time-limit [6]; however, this destroys completeness as we can no longer saturate the

* Partially supported by the EPSRC grant "Reasoning for Verification and Security"

set. Another approach is *splitting*. The idea behind splitting is to take a search space $S \cup \{C_1 \vee C_2\}$ and split it into $S \cup \{C_1\}$ and $S \cup \{C_2\}$, for variable-disjoint C_1 and C_2. The benefit is that in each search space the potentially long and heavy clause $C_1 \vee C_2$ is replaced by one of the shorter and lighter clauses C_1 or C_2. Each search space can be saturated separately. If a refutation is found in both then the original search space is unsatisfiable, but if one is saturated without a refutation then the original search space is satisfiable.

To perform splitting it is necessary to make *splitting decisions*, i.e. assert one component of a clause, and potentially *backtrack* these decisions. Different splitting approaches have been considered in the past. In splitting with backtracking (as seen in SPASS [10]) this is done via a (conceptual) splitting tree where a splitting decision is made and we explore one half of the search space and then backtrack (undo the decision) before exploring the second half. In splitting without backtracking [4], when splitting a clause $C_1 \vee \ldots \vee C_n$ each component C_i is named by a fresh propositional variable p_i and the whole clause is split into clauses $(C_1 \vee \neg p_1), \ldots, (C_{n-1} \vee \neg p_{n-1})$ and $(C_n \vee p_1 \vee \ldots \vee p_{n-1})$. This approach is admittedly easier to implement than splitting with backtracking, but the presence of propositional variables sometimes prevents the prover from performing reductions, which may lead to weaker performance [3].

The AVATAR architecture uses a SAT solver to make splitting decisions. As explained later, the SAT solver is passed information about new clauses and produces a model representing valid branches of the (conceptual) splitting tree. The first-order prover can then assert these components and attempts to find a contradiction, which is then passed back to the SAT solver to prune the search space.

AVATAR proved highly successful in previous work evaluating it against alternative splitting mechanisms [3]. Introducing the architecture helped to solve 421 problems previously unsolvable by Vampire [5] or any other prover. However, its full power, and the best way to use it, is not yet fully understood. Certain architectural choices were based on (informed) intuition and not evaluated experimentally; the aim of this paper is to understand these choices and use this understanding to improve AVATAR.

The paper begins with a description of AVATAR's implementation in Vampire (Sections 2 to 4). It then introduces and explains new variations to the AVATAR architecture (Sec. 5) that will allow us to better understand the interaction between different parts of the architecture. These variations themselves represent a contribution to the understanding of how AVATAR can be organised.

We finish by presenting an extensive evaluation of these architectural variations (Sec. 6). It is clear that some variations are more useful than others and we discuss the likely cause of these results. This evaluation also considers how our improved understanding of the AVATAR architecture can be used to construct complementary strategies to solve as many problems as possible with as few strategies. We finish having learned much about this new, highly experimental architecture, but also with a number of further questions that will shape the continued improvement of this approach.

2 AVATAR by Example

Whilst the theory of AVATAR was established in [9], and is reviewed later in this paper, the authors feel that the key ideas behind the approach are best demonstrated via an example.

The general architecture of AVATAR consists of a first-order (FO) prover and a SAT solver. The FO prover stores a set of first-order clauses, performs first-order reasoning using a saturation algorithm and passes some clauses to the SAT solver. The SAT solver keeps a set of propositional clauses and produces a model (or an unsat answer) on request from the FO prover.

For our example we consider the following input clauses:

$$q(b) \quad p(x) \vee r(x,z) \quad \neg q(x) \vee \neg s(x) \quad \neg p(x) \vee \neg q(y) \quad s(z) \vee \neg r(x,z) \vee \neg q(w)$$

We check which of these clauses can be split into *components*, i.e. sub-clauses with pairwise disjoint sets of variables. The first three clauses cannot be split and are added directly to the FO prover. The last two clauses can be split into components. Each component is given a unique propositional name. To do this naming in a consistent way we use a *component index*, as seen below. This, for example, ensures that $\neg q(y)$ and $\neg q(w)$ are associated with the same propositional symbol. This results in two propositional *split clauses* representing the first-order clauses.

The theory of splitting tells us we can assert one component and then the other after we find a refutation with the first. We are going to use the SAT solver to make splitting decisions so we pass the representations of the splittable clauses to the SAT solver, but do not yet add any of their respective components to the FO prover.

The state of the FO prover and the SAT solver are shown below, where we write $C \leftarrow A$ to indicate that clause C depends on a (possibly empty) set of assertions (splitting decisions) A:

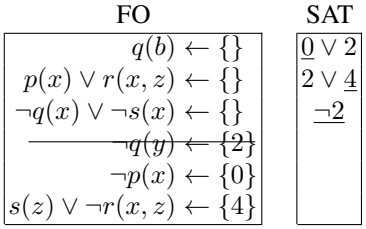
Component Index	FO	SAT
$0 \mapsto \neg p(x)$	$q(b) \leftarrow \{\}$	$0 \vee 2$
$2 \mapsto \neg q(y)$	$p(x) \vee r(x,z) \leftarrow \{\}$	$2 \vee 4$
$4 \mapsto s(z) \vee \neg r(x,z)$	$\neg q(x) \vee \neg s(x) \leftarrow \{\}$	

The FO prover now requests a model. The SAT solver can assign all variables to true, but let us assume the model is minimized into a *partial* model in which only 2 is true and the values of the remaining variables are undefined. Notice that both split clauses in the SAT solver are satisfied by any total extension of this partial model.

Based on the model we assert $\neg q(y)$, the component corresponding to 2. When adding $\neg q(y)$ the FO prover performs a reduction and $\neg q(x) \vee \neg s(x)$ is subsumed. However, this subsumption is conditional on the assertion 2.

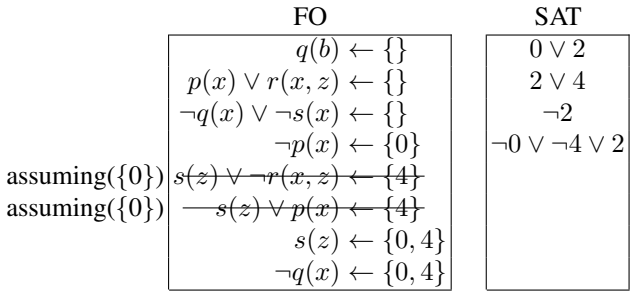
	FO	SAT
	$q(b) \leftarrow \{\}$	$0 \vee \underline{2}$
	$p(x) \vee r(x,z) \leftarrow \{\}$	$\underline{2} \vee 4$
assuming($\{2\}$)	~~$\neg q(x) \vee \neg s(x) \leftarrow \{\}$~~	
	$\neg q(y) \leftarrow \{2\}$	

The FO prover then performs resolution between $q(b)$ and $\neg q(y) \leftarrow \{2\}$ to get the clause $\bot \leftarrow \{2\}$. A corresponding *contradiction clause* $2 \rightarrow \bot = \neg 2$ is then added to the SAT solver and a new model is computed. This time the $\neg 2$ condition forces the SAT solver to construct a model containing 0, 4 and $\neg 2$. As the assertion under which $\neg q(x) \vee \neg s(x)$ was reduced no longer holds, the deletion of this clause is now undone.



The FO prover then performs resolution between $s(z) \vee \neg r(x,z) \leftarrow \{4\}$ and $p(x) \vee r(x,z)$ to produce $s(z) \vee p(x) \leftarrow \{4\}$, which is then immediately (conditionally) replaced by $s(z) \leftarrow \{0,4\}$ after performing subsumption resolution with $\neg p(x) \leftarrow \{0\}$. This new clause replaces $s(z) \vee \neg r(x,z) \leftarrow \{4\}$ conditioned on the assertion 0. Finally, the FO prover performs a resolution step between $s(z) \leftarrow \{0,4\}$ and $\neg q(x) \vee \neg s(x)$ to get $\neg q(x) \leftarrow \{0,4\}$.

As $\neg q(x)$ is a known component (up to variable renaming), we can add $0 \wedge 4 \rightarrow 2$ to the SAT solver as the clause $\neg 0 \vee \neg 4 \vee 2$. Now the SAT solver can no longer produce a model and so the input problem is shown unsatisfiable and the prover terminates.



3 Proof Attempts in Vampire

In this section we give the relevant background on how proof attempts are carried out in the automated theorem prover Vampire [5]. The next section will show how AVATAR-style splitting fits into this process.

Saturation Algorithms. Superposition provers use *saturation algorithms with redundancy elimination*. These work with a search space consisting of a set of clauses and use a collection of generating, simplifying and deleting inferences to explore this space. The theoretical basis of saturation algorithms is the notion of *redundancy* given, e.g., in [1]. Both simplifying and deletion inferences in saturation algorithms are designed in such a way that they only remove redundant clauses.

All saturation algorithms implemented in Vampire belong to the family of *given clause algorithms*, which achieve completeness via a fair *clause selection* process that prevents the indefinite skipping of old clauses. These algorithms typically divide clauses into three sets, *unprocessed*, *passive* and *active*, and follow a simple *saturation loop*:

1. Add non-redundant *unprocessed* clauses to *passive*. Redundancy is checked by attempting to *forward simplify* the new clause using processed clauses.

2. Remove processed clauses made redundant by new clauses i.e. *backward simplify* existing clauses using the new clauses.
3. Select a given clause from *passive*, move it to *active* and perform all generating inferences between the given clause and all other active clauses, adding generated clauses to *unprocessed*.

Vampire implements three saturation algorithms:

1. Otter uses both passive and active clauses for simplifications.
2. Limited Resource Strategy (LRS) [6] extends Otter with a heuristic that discards clauses that are unlikely to be used with the current resources i.e. time and memory. This strategy is incomplete but also generally the most effective at proving unsatisfiability.
3. Discount uses only active clauses for simplifications.

Inferences. The inferences applied by the saturation algorithm are of three different kinds:

- *Generating inferences* derive new clauses that can be immediately simplified and/or deleted by other kinds of inferences. For example, binary resolution and superposition.
- *Simplifying inferences* replace one clause by another simpler clause. For example, demodulation (rewriting by ordered unit equalities) and subsumption resolution (a variant of binary resolution whose conclusion subsumes one of the premises).
- *Deleting inferences* delete clauses, typically when they become redundant. For example, subsumption and tautology deletion.

CASC mode. Finally, there is a special competition mode that Vampire can be run in (using `--mode casc`) that attempts a sequence of strategies, chosen based on structural characteristics of the given problem. This is motivated by two observations, firstly that whilst some strategies perform very well on average there is no silver bullet that can solve all problems, and secondly that most solvable problems have a strategy that can solve that problem quickly.

4 Introducing splitting

As we have previously explained, the search space explored by saturation algorithms can quickly become very large and populated with heavy and long clauses. The technique of splitting, where each component of a clause is asserted in turn, can be used to reduce the search space and improve the prover's performance. This section shows how AVATAR implements this splitting process – a full technical description is given in [9].

Splitting the search space. The general splitting idea can be illustrated by a conceptual splitting tree that is explored during the proof attempt. Every generated clause which can be split is represented by a node and each branch represents a sequence of splitting decisions. When a branch has been found inconsistent *backtracking* occurs and the search moves on to explore a different branch. It can be informative to consider the splitting performed by AVATAR in terms of this splitting tree.

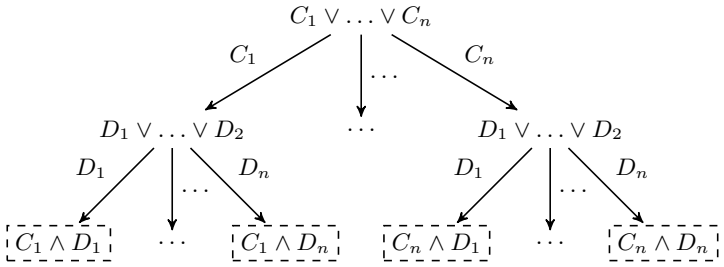


Fig. 1. The conceptual splitting tree

Fig. 1 illustrates this splitting tree using clauses $C_1 \vee \ldots \vee C_n$ and $D_1 \vee \ldots \vee D_n$. This tree grows dynamically as further clauses are added to the search space. If every branch contains a contradiction then the problem is unsatisfiable.

Attempting to explore this tree explicitly can be expensive for a number of reasons. Firstly, if clauses share components (i.e. C_i is a variant of D_j) this sharing is not captured by the splitting tree. Secondly, the exploration of the splitting tree is rigid and is difficult to alter based on newly learned information about the components involved. And lastly, information discovered on one branch cannot be easily transferred to a different branch. As we see below, AVATAR implicitly explores this splitting tree by translating the information about splitting components into constraints for a SAT solver and uses the produced model to make component assertions.

The Architecture Fig. 2 illustrates the AVATAR architecture. There are three main parts: the first-order (FO) prover, the SAT solver and the Splitting Interface. The FO prover deals with *clauses with assertions* of the form $D \leftarrow A$ where D is a first-order clause and A is a finite set of propositional variables representing asserted components.

The Splitting Interface manages a mapping between first-order components C and the propositional variable $[C]$ naming that component. The *variant index* ensures that two components C_1 and C_2 are mapped to the same propositional variable if they are equal up to variable renaming, order of literals, and symmetry of equality. This mapping also ensures that the negation of a ground component is translated to the negation of the corresponding propositional variable, i.e. $[\neg C] = \neg[C]$ for every ground component C.[1]

For each component, the Splitting Interface also maintains a *record* which stores:

1. *children* of the component, i.e., clauses that are derived from the component,
2. clauses that were *reduced* by a clause depending on this component.

See below for an explanation of these sets of clauses.

Lastly, to avoid asserting previously asserted components, the Splitting Interface keeps track of the current model previously obtained from the SAT solver. The following sections will explain the communication between the three parts.

[1] This useful optimization is not derictly available for non-ground components. Negating a non-ground component would require skolemization and is not considered in this paper.

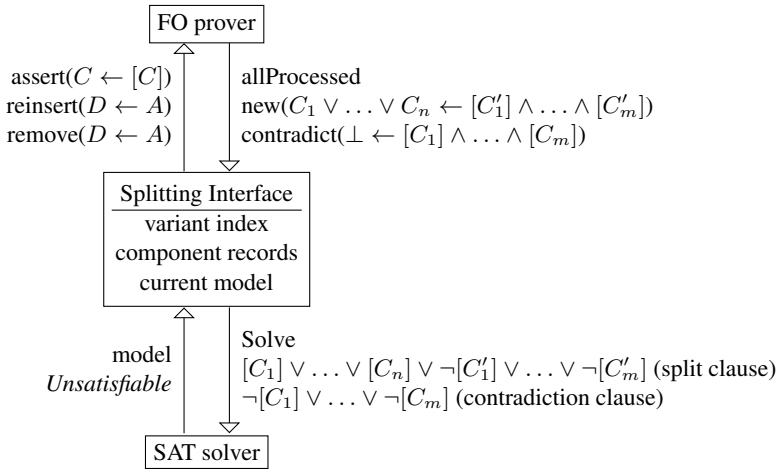


Fig. 2. The AVATAR architecture

Dealing with assertions in the FO prover. As we said above, the FO prover is updated to deal with *clauses with assertions*. This affects the way that inferences are carried out in the prover. Firstly, to ensure that assertions are properly propagated, any generating inference of the form

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

is replaced by the inference

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

and $(D \leftarrow A_1 \cup \ldots \cup A_k)$ is added to the *children* of each component in $A_1 \cup \ldots \cup A_k$ in the component records kept by the Splitting Interface.

Simplifying inferences of the form

$$\frac{D_1 \quad \cdots \quad D_{m-1} \quad \cancel{D_m}}{D} \ .$$

previously meant that D is a logical consequence of $D_1, \ldots, D_m$ and D makes D_m redundant. This is replaced by

$$\frac{(D_1 \leftarrow A_1) \quad \cdots \quad (D_{m-1} \leftarrow A_{m-1}) \quad \cancel{(D_m \leftarrow A_m)}}{(D \leftarrow A)} \ ,$$

where $A = A_1 \cup \ldots \cup A_m$, distinguishing the following two cases.

1. If $A = A_m$ then $(D_m \leftarrow A_m)$ can be deleted as all the other clauses are based on the same or a weaker set of assertions. It should also be deleted from the *children* of components in A_m.

2. Otherwise, $(D_m \leftarrow A_m)$ can only be *conditionally deleted* as there exists a branch of the splitting tree where the deleted clause is valid but at least one of the side conditions $(D_i \leftarrow A_i)$ is not. To conditionally delete a clause we remove it from the FO prover and add it to the *reduced* set for each component in $A \setminus A_m$.

Exploring the splitting tree in AVATAR. When a new splittable clause is selected for processing, the FO prover passes this clause to the Splitting Interface instead of attempting to add it to passive. The Splitting Interface then uses the variant index to translate it into a propositional *split clause* and pass this to the SAT solver.

Once unprocessed is empty, the FO prover sends the `allProcessed` message to the Splitting Interface, which sends the `Solve` message to the SAT solver. The SAT solver then either replies with `Unsatisfiable`, indicating that all splitting branches have been explored, or it returns a new model M.

We allow for partial models which we represent by consistent sets of propositional literals of the form $[C]$ or $\neg[C]$. We require that at least one literal of each propositional clause registered by the SAT solver must be satisfied by the model, but some literals may stay undefined.

Given a new model M and old model $\overline{M}$, the Splitting Interface does the following:

1. For each $[C] \in (\overline{M} \setminus M)$, remove component C and all of its children from the FO prover using remove($D \leftarrow A$). Add any clause $D \leftarrow A$ in the component's reduce set such that $A \subseteq M$, using reinsert($D \leftarrow A$).
2. For each $[C] \in (M \setminus \overline{M})$, add component C to the FO prover using assert($C \leftarrow [C]$) and add each of the component's children $D \leftarrow A$ such that $A \subseteq M$, using reinsert($D \leftarrow A$).

Removing the children of removed components is necessary as they rely on assertions that are no longer true. Reinserting a clause that has been reduced with the help of a removed component retracts the now no longer supported deletion of the clause. Reinserting the children of a component means that clauses generated from this component on previous branches are brought into this branch. In both cases, we only reinsert those clauses that have all their assertions true in the new model M.

Split clauses introduce new branches into the conceptual splitting tree, although note that due to the use of the variant index some of these branches may be shared. To prune the splitting tree we need *contradiction clauses*. When the FO prover produces a contradiction with assertions, this contradiction is passed to the Splitting Interface, which performs the translation into a propositional *contradiction clause* and sends this to the SAT solver. This contradiction forces the model to change. Notice that a contradiction clause can cut off many branches of the splitting tree.

5 Varying the architecture

We now consider some of the choices made in the architecture of AVATAR, how we may change these, and what effects these changes may have.

Adding components. When providing information about new clauses to the SAT solver (in the form of component clauses) we need to decide what to do with *non-splittable* clauses, i.e., those that cannot be split into multiple components. We consider two values for this option (named `nonsplittable_components`):

1. `none`: do not add any non-splittable component,
2. `known`: add such a clause if it has previously been introduced as a component.

The example in Sec. 2 uses this second option value when it adds the component clause $\neg 0 \vee \neg 4 \vee 2$ as the clause $\neg q(x)$ is nonsplittable. With the option set to `none`, the FO prover would have performed an additional resolution step to produce the contradiction clause $\neg 0 \vee \neg 4$. By using the known component, we constrained the split tree explored by the SAT solver and thus avoided performing the additional inference.

Constructing a model. Previously, we referred to the SAT solver as just a SAT solver, but different SAT solvers take a different amount of time to construct a model and potentially also construct different models. With the option `sat_solver`, we can vary which SAT solver we use. In this paper we consider our own SAT solver and Minisat (version 2.2) [2] using the default options. As a SAT solver, Minisat is better than Vampire's native solver. Our aim was to understand whether a better SAT solver results in a better overall performance of AVATAR.

Partial models and minimization. While models produced by a standard SAT solver are total, AVATAR may work with partial models in which some literals are undefined, provided each such model makes true at least one literal in every clause. Total models may result in adding unnecessary assertions to the FO prover, for instance, when they set to true multiple literals from a single split clause. This corresponds to the exploration of multiple splitting branches at once, an effort which is often wasted as each of the branches usually needs to be considered separately later on as well.

We can attempt to minimize the total model produced by the SAT solver by dropping literals that are not needed for satisfying any clause and thus to restrict the exploration to a single branch. We use a simple greedy procedure for the minimization, whose result is a partial model, a sub-model of the original one. Minimization is controlled by the option `minimize_model`. Again, we can vary how we choose to do this.

1. `off`: We use the total model.
2. `all`: We minimize with respect to all clauses.
3. `sco`: We minimize with respect to split clauses only.

Note that the `sco` option value is sound, because we always start minimizing from a total model which satisfies all the clauses.

Asserting complements. Another factor to consider is the possibility of treating ground components specially as we are able to assert these negatively to the FO prover. That is, when the SAT solver sets the value of a ground component C to false in the model we can assert $\neg C \leftarrow [\neg C]$ to the FO prover, even if this is not needed to satisfy any split clause. This is controlled by the option `add_complementary`. We are prevented from asserting both C and $\neg C$ at the same time as the mapping from components to propositional variables ensures that $[\neg C] = \neg[C]$ for ground components.

When to do splitting. Previously we described the splitting process occurring at clause introduction, i.e., when we attempt to move it into passive. Alternatively we can consider splitting a clause at *activation*. This is controlled by the option `split_at_activation`. This delays the expense of splitting but also delays the benefits of additional information being passed to the SAT solver. For example, a subset of passive clauses may already be propositionally unsatisfiable, but we will not discover this until all clauses in this subset become activated.

Table 1. The Vampire options of interest.

Option	Short name	Considered values (default)
saturation_algorithm	sa	lrs, discount
sat_solver	sas	vampire, minisat
nonsplittable_components	ssnc	known, none
minimize_model	smm	sco, off, all
add_complementary	ssac	ground, none
split_at_activation	sac	off, on
delete_deactivated	sdd	on, off
handle_zero_implied	shzi	off, on

To delete or to deactivate. In the previous presentation, clauses that are deactivated due to switching the splitting branch are reasserted when they become valid again. Remembering these clauses may cost us a lot of memory. Moreover, some of these clauses may never need to be reasserted, if they depend on a partial branch which will not be visited anymore. With the option `delete_deactivated`, we delete these clauses instead and later only reassert the respective component clause $C \leftarrow [C]$, which is sufficient for completeness. The downside is that we may need to recompute some of these delete clauses if a particular partial branch is revisited.

Currently even deactivated clauses are removed from the term indexing structures used for efficient inferences. Providing an option to preserve deactivated clauses in these structures remains further work, and may prove beneficial as deleting and inserting clauses into indexing structures can become very costly.

Clearing the assertions. SAT solvers typically perform DPLL splitting and may, at some point, derive that a propositional literal must be true in any possible model. These are called zero-implied literals as their truth value is decided at the zeroth level. This information can be used to remove the corresponding assertions from clauses in the FO prover as these are redundant. This can reduce the number of conditional clause deletions as any deletions conditional only on zero-implied components can be considered unconditional. This option is controlled by `handle_zero_implied`.

Summary. Table 1 describes the Vampire options we consider in this work, i.e., those we will vary in experiments later. All other options will be fixed at their default value. Note that some options are experimental and may not be included in future releases of Vampire.

6 Experiments

In this section we experimentally evaluate the impact of the different variations of the AVATAR architecture on the performance of theorem prover Vampire.

Designing experiments. The aim of these experiments is to evaluate how effective the different architectural variations are. To do that we need to understand what we mean by effective. The existence of the CASC portfolio mode is a testament to the fact that there is no best strategy. In fact, the value of a strategy is difficult to understand. Some strategies perform very well on average but cannot solve problems solvable by other strategies. The motivation behind CASC mode is that a collection of strategies, each

Table 2. Best and worst strategies with respect to the number of problems solved, option values that define them, the number of problems solved by the 10% worst and best strategies in union, respectively, and the respective proportional representation of the option values in these strategies.

	worst	worst 10%	best	best 10%
problems solved	796	1149	1103	1223
`saturation_algorithm`	`lrs`	61%	`discount`	100%
`sat_solver`	`vampire`	100%	`minisat`	63%
`nonsplittable_components`	`none`	79%	`known`	47%
`minimize_model`	`off`	63%	`all`	42%
`add_complementary`	`ground`	53%	`ground`	100%
`split_at_activation`	`off`	100%	`on`	100%
`delete_deactivated`	`off`	55%	`on`	53%
`handle_zero_implied`	`on`	50%	`off`	50%

of which may be bad on average, can easily outperform a collection of strategies, each of which is good on average. However, within a collection of strategies we need those that can solve many problems as time limits do not usually allow for running too many strategies. Therefore, the aim of these experiments is to identify those options that allow us to solve previously unsolved problems as well as those options that help us solve the most problems.

Experimental setup. For our benchmarks we use TPTP [7] problems containing non-unit clauses with a rating of 0.5 or higher. The TPTP rating [8] is the percentage of (eligible) provers that cannot solve a problem, thus, for example, a rating of 0.5 means that half of (eligible) provers can solve the problem and a rating of 1 means that the problem cannot be solved by any of these provers. However, the rating evaluation does not use every mode of each prover, so it is possible that a prover used to generate ratings can solve a problem of rating 1 using a different mode. We only include problems in our experiment that we know are solvable by some prover, e.g., Vampire. This led to the selection of 3823 problems.[2]

For the experiments, we took all combinations of options discussed in Sec. 5. This cartesian product (cube) gives us 384 strategies and represents almost 1.5 million experiments in total. We ran experiments with a time limit of 10 seconds[3], meaning that our results reasonably reflect the utility of the strategy when placed within a cocktail of other strategies. We used the default values for all options not explicitly stated.

Experiments were run on the StarExec[4] cluster, using 160 nodes. The nodes used contain a Intel Xeon 2.4GHz processor. Experiments used Vampire's default memory limit of 3GB; this memory limit was only reached in rare cases ($<$0.05%).

Time spent in the SAT solver. The experiments show that time spent in the SAT solver does not generally dominate. On average, 9.6% of the time was spent in the SAT solver.

[2] A list of the selected problems, the executable of our prover as well as the results of the experiment are available from `http://vprover.org`.

[3] Note that previous experiments [3] used longer time limits.

[4] `https://www.starexec.org`

In 8.8% of the experiments, calls to the SAT solver took more than 50% of the time and in 0.5% of the experiments calls to the SAT solver took more than 90% of the time[5].

Best and worst strategies. In total, 1444 problems (38% of all problems) were solved by at least one of the considered strategies, of these 328 were of rating 0.8 or higher. Table 2 shows the performance of the worst and best strategy with respect to the number of solved problems and the values of options that define them. We can see that the best strategy only solves 1103 problems which amounts to about 76% of all the problems solved. The table also shows the performance of two meta-strategies, one consisting of the union of the 10% worst and the other of the 10% best strategies, and, in the lower part, the percentage of the 10% worst and best strategies which use the same value for a particular option as the ultimate worst and best strategy, respectively.[6]

Perhaps the most surprising observation is that `lrs` does not appear at all amongst the 10% of the best strategies. We suspect that LRS, which was not adapted to AVATAR, misinterprets the remaining amount of resources available for proving, because it does not take into account the part of the split tree that still needs to explored. Attempting to confirm this hypothesis is one possible direction for future work.

Another interesting fact is that both the worst and the best strategy employ the value `ground` for the `add_complementary` option. This option value is definitely useful (all the best strategies use it), but may have some shortcomings, because it is also used by the majority of the worst strategies.

When interpreting the results for `minimize_model`, one should keep in mind that this option has three possible values and so the result of 42% for `all` with the best strategies is significant. On the other hand, Table 2 indicates that the effect of `delete_deactivated` and, especially, of `handle_zero_implied` is close to random.

Importance of particular options. To better determine the importance of individual options, we put the number of problems solved with a particular value of an option into Table 3. On a per option basis, the table also shows (in parenthesis) the number of problems solved only by a strategy using a particular value of the option and not by any strategy using any of the other values. This means the value is necessary for solving these problems.

An option is important if it has at least two values each necessary for solving many problems. This perspective implies that `saturation_algorithm` is the most important option in our experiment and `split_at_activation` the most important one for AVATAR per se. When focusing on individual values, we can see that `minisat` helps to solve more problems than `vampire`, that the value `ground` should be preferred over `none` for `add_complementary`, and that it perhaps does not pay off to keep the value `sco` for `minimize_model`.

Conditional projections. Having collected the data for all the possible combinations of option values one can also ask questions such as what would Table 3 look like if we focused only on strategies where a particular option is fixed to a certain value. This

[5] Only runs which took at least then one second to complete are considered here.

[6] A different statistic, not shown in the table, is the performance of strategies at the 10% mark from each end of the sorted order (quantiles), which were 865 and 1072, respectively.

Table 3. Number of problems (uniquely) solved with a particular option value.

saturation_algorithm		add_complementary	
lrs	1287 (142)	none	1372 (20)
discount	1302 (157)	ground	1424 (72)
sat_solver		**split_at_activation**	
vampire	1375 (38)	off	1345 (83)
minisat	1406 (69)	on	1361 (99)
nonsplittable_components		**delete_deactivated**	
none	1416 (27)	off	1427 (31)
known	1417 (28)	on	1413 (17)
minimize_model		**handle_zero_implied**	
off	1402 (15)	off	1428 (16)
all	1412 (17)	on	1428 (16)
sco	1401 (11)		

allows us to distinguish generally good values of options from those that are only good under certain conditions.

For example, we observed that while with `discount` we could solve 39 more problems when `split_at_activation` was turned `on`, this did not happen for `lrs`, where we could solve 1208 problems with `split_at_activation` `off`, but only 1202 problems with the option `on`. This is most like related to the fact that LRS uses clauses in passive for simplifications and therefore benefits from these clauses being already split.

Also, both the `lrs` and the `vampire` perspective significantly favour the value `known` over `none` for `add_complementary`, while in Table 3 these two values seem to behave similarly. In the former conditional projection, `none` solves only 1242 while `known` 1266, in the latter, `none` solves 1333 and `known` 1347. This phenomenon seems to be quite difficult to explain and should be further explored.

Greedy problem coverage. Next we consider how the strategies could be greedily ordered to cover all problems solved, i.e., we attempt to produce a (greedy) CASC portfolio mode. We require 61 strategies in total to cover all problems, with the last 32 strategies only contributing one additional problem each.

Table 4 gives the first five strategies in this greedily produced portfolio sequence along with the number of problems each strategy contributes to the portfolio, how many problems that strategy normally solves, and the nominal order in all strategies (with respect to number of problems solved).

We first note that we require both strategies that are good on average and also those that solve problems uniquely. In the sequence of strategies, 72% come from the bottom half of strategies in terms of number of problems solved. It is also interesting to note that some option values, such as `sco` for `minimize_model`, that were previously seen to contribute little, are needed here.

Table 4. Sequence of strategies to greedily cover all solved problems. For space reasons, short names are used for options (see Table 1).

	1	2	3	4	5
Contribution	1103	114	45	31	21
Solves	1103	943	905	948	1081
Nominal order	1	155	283	141	23
`sa`	`discount`	`lrs`	`discount`	`lrs`	`discount`
`sas`	`minisat`	`minisat`	`vampire`	`minisat`	`vampire`
`ssnc`	`known`	`known`	`known`	`known`	`none`
`smm`	`all`	`sco`	`sco`	`off`	`sco`
`ssac`	`ground`	`none`	`ground`	`ground`	`ground`
`sac`	`on`	`off`	`off`	`on`	`on`
`sdd`	`on`	`off`	`on`	`off`	`on`
`shzi`	`off`	`on`	`off`	`on`	`on`

Further lessons learned. One of the interesting lessons learned with these experiments is that the choice of a SAT solver significantly influences the performance of a strategy. This suggests that the queries passed to the SAT solver are by no means easy (as we originally assumed) and that on many problems the solver takes over a considerable part of the required reasoning.

Moreover, efficiently dealing with the incremental nature of the presented queries becomes a relevant factor in AVATAR. When restricting solution times to a maximum of 1 second, `vampire` became the solver of choice for the best strategy with respect number of solved problems. The `vampire` solver was designed with the AVATAR application in mind, and therefore deals well with the incremental usage required. However, as it is not as highly tuned as `minisat`, its performance tails off quickly as the size of the problem increases. This may explain the observed behaviour.

Another aspect influenced by the choice of a SAT solver is the inherent "quality" (from the perspective of AVATAR) of the models it produces. It is clear that the produced model affect how the splitting tree is explored, but not yet clear why one solver may produce 'better' models in general. Further investigations will consider the SAT solver options themselves and how varying these affects the models produced.

7 Conclusion

AVATAR is a new and highly successful architecture. While previously used saturation algorithms, their variations and options have been studied for decades, almost nothing is known about options that can improve AVATAR even further. Likewise, almost nothing is known about the behaviour of various existing options in presence of AVATAR. This is the first paper that both introduces AVATAR specific options and investigates their behaviour. We believe this is the first in many studies by us, and others, exploring this novel architecture.

The usage of a SAT solver to perform splitting operations is a novel idea, which has the potential to change how modern first-order theorem provers explore the clause search space. The architectural variations explored in this paper help us better understand the optimal configuration for this new form of splitting.

We found that the importance of the individual options for solving additional problems varies, and while the important ones should be kept and further explored, removing the ones that seem to have negligible influence on the performance of AVATAR could simplify the implementation and improve its maintainability.

We also discovered that the efficiency of the SAT solver is very important for the overall performance of AVATAR. This is not only in terms of proving time, but also their ability to handle incrementality. We observed cases where performance suffered as a result of insufficient support for incremental usage; this suggests that improving SAT solvers in this respect can improve AVATAR. We have also identified new questions to be answered, for example how the model produced by the SAT solver interacts with the exploration of the splitting tree.

An additional discovery is that the limited resource strategy, thought to be the best strategy within Vampire for showing unsatisfiability, does not interact well with the way in which AVATAR explores the clause space. This suggests that further investigation is required to establish how best to adapt the LRS approach to AVATAR.

Whilst the results from the current architecture are impressive, there is more that can be squeezed from this idea. One major area of interest is replacing the SAT solver with an SMT solver, allowing it reason on the theory level.

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. N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
3. K. Hoder and A. Voronkov. The 481 ways to split a clause and deal with propositional variables. In M. P. Bonacina, editor, *Automated Deduction CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2013.
4. A. Riazanov and A. Voronkov. Splitting without backtracking. In B. Nebel, editor, *17th International Joint Conference on Artificial Intelligence, IJCAI'01*, volume 1, pages 611–617, 2001.
5. A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Commun.*, 15(2,3):91–110, 2002.
6. A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computations*, 36(1-2):101–115, 2003.
7. G. Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.
8. G. Sutcliffe and C. Suttner. Evaluating General Purpose Automated Theorem Proving Systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.
9. A. Voronkov. AVATAR: The architecture for first-order theorem provers. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer International Publishing, 2014.
10. C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.