EPiC Series in Computing

Volume 41, 2016, Pages 39–52

GCAI 2016. 2nd Global Conference on Artificial Intelligence


# AVATAR Modulo Theories

Nikolaj Bjøner[1], Giles Reger[2], Martin Suda[3], and Andrei Voronkov[2,4,5]

[1] Microsoft Research, Redmond, USA
[2] University of Manchester, Manchester, UK
[3] Institute for Information Systems, Vienna University of Technology, Austria
[4] Chalmers University of Technology, Gothenburg, Sweden
[5] EasyChair

**Abstract**

This paper introduces a new technique for reasoning with quantifiers and theories. Traditionally, first-order theorem provers (ATPs) are well suited to reasoning with first-order problems containing many quantifiers and satisfiability modulo theories (SMT) solvers are well suited to reasoning with first-order problems in ground theories such as arithmetic. A recent development in first-order theorem proving has been the AVATAR architecture which uses a SAT solver to guide proof search based on a propositional abstraction of the first-order clause space. The approach turns a single proof search into a sequence of proof searches on (much) smaller sub-problems. This work extends the AVATAR approach to use an SMT solver in place of a SAT solver, with the effect that the first-order solver only needs to consider ground-theory-consistent sub-problems. The new architecture has been implemented using the Vampire theorem prover and Z3 SMT solver. Our experimental results, and the results of recent theorem proving competitions, show that such a combination can be highly effective.

## 1 Introduction

Many applications of automated deduction, such as program analysis and verification, require efficient reasoning with quantifiers and theories. Such problems can be viewed as existing on a scale from theory-light / quantifier-heavy to theory-heavy / quantifier-light. Automated theorem provers (ATPs) are successful at the former end of the scale, largely via the inclusion of *theory axioms*. SMT (Satisfiability Modulo Theories) solvers are successful at the latter end of the scale via heuristic instantiation techniques such as E-matching [15, 11] and model-based quantifier instantiation [15, 11], followed by applying decision procedures to sets of ground clauses. This work aims to combine the two successes to provide a technique that not only covers both previous approaches, but complements them by targeting the 'middle end' of the scale where we have non-trivial usage of both quantifiers and theories.

The new approach utilises the AVATAR architecture [34] for saturation-based theorem proving. The idea behind AVATAR is to use a SAT solver to guide proof search by making decisions over a propositional abstraction of the clause search space and iteratively selecting a sub-problem for the saturation-based theorem prover to tackle. This approach is lifted to be *modulo theories* by replacing the SAT solver by an SMT solver, ensuring that the sub-problem is theory-consistent in the ground part. The result is that the ATP and SMT solver deal with the parts of the problem to which they are best suited.

We have implemented this approach using the first-order ATP VAMPIRE [19] and the SMT solver Z3 [13]. This combination employs VAMPIRE for non-ground reasoning via the superposition calculus

and Z3 for ground reasoning over theories of uninterpreted functions, arithmetic and arrays. This paper describes how the transition from a SAT based solver to an SMT based solver influences design choices in AVATAR and we summarize experimental findings with the resulting system.

The rest of this paper is organised as follows. Section 2 gives a brief overview of the VAMPIRE system. Section 3 reviews the AVATAR architecture and extends it to reason modulo theories. Section 4 discusses how the extension is realised using the Z3 SMT solver. Section 5 presents a comparative evaluation. Section 6 gives related work. Section 7 concludes.

## 2 Overview of Vampire

VAMPIRE [19] is an automated first-order theorem prover. For the last 15 years it has led the first-order theorem division of the CASC competition [32]. Its main mode of operation implements saturation with the *ordered binary resolution* calculus [1] and, for handling equality, the *superposition* calculus [22]. This is what we briefly describe below. It also implements instance-based and finite-model finding techniques that we do not discuss here as they are not relevant to AVATAR. Nevertheless, they may be utilised during the proof attempts reported in Section 5. We review relevant aspects of the VAMPIRE system here before discussing the AVATAR architecture in detail in the next section.

**General Design.** VAMPIRE implements a given-clause algorithm which operates on *first-order clauses*, therefore VAMPIRE implements preprocessing and clausification techniques that translate general *first-order formulas* into this form. Preprocessing is important as it can dramatically alter characteristics of the problem that effect proof search. VAMPIRE handles input in both TPTP and SMT-LIB formats.

The given-clause algorithm works with an inference system $\mathbb{I}$[1] as follows. Initially, all clauses are added to a *passive* set and a loop is executed until a contradiction is found or the passive set becomes empty. On each iteration of the loop a *given clause* is *selected* from passive and added to an *active set*. Then, all generating inferences of $\mathbb{I}$ are performed between the given clause and clauses in the active set. Generated clauses are added to passive and the loop repeated. If at any point the empty clause is derived then the initial input is unsatisfiable. If the passive set is emptied, the proof search strategy was complete and no contradiction is found then the initial input is satisfiable. Note that VAMPIRE can make effective use of many incomplete strategies that remove heavy clauses from the search space [27] or perform arbitrary literal selection [17].

For proof search to be effective it is necessary to remove *redundant* clauses from the search space. VAMPIRE implements redundancy elimination techniques such as tautology deletion, condensation, subsumption and global subsumption.

**Theory Reasoning in VAMPIRE.** For dealing with linear and non-linear theories of integer and real arithmetic, and the theory of polymorphic arrays (see [18]), VAMPIRE adds theory axioms. Clearly this is an incomplete method in general, but can be effective and allows VAMPIRE to directly use existing techniques. To see which theory axioms are automatically added for a problem, run VAMPIRE in output mode (`--mode output`), which will print the input problem along with any added axioms. This can be turned off using `--theory_axioms off`.

VAMPIRE will evaluate ground theory terms (e.g. evaluate $2 + 2 = 5$ to $\mathit{false}$) and rewrite some equations (e.g. rewrite $2x - 4 = 2$ to $x = 3$). This has the advantage that many theory-equivalent terms will be represented by the same term. It can also avoid the need to apply axioms to show that $1 + 1 = 2$. However, it has the disadvantage that the term $2 + 4 = a$ will never be present and able to unify with a term such as $x + y \neq a$.

[1] This refers to generating inferences only, simplifying inferences are handled separately.

Adding theory axioms may seem to be inefficient. For example, using these axioms superposition provers may derive a large number of theory tautologies. However, in practice it turns out to be surprisingly effective on many problems. For example, in the CASC 25 competition of theorem provers [33] on problems with quantifiers and theories (the TFA division of the competition), VAMPIRE using theory axiomatizations was very close to the top SMT solver CVC4.

Finally, VAMPIRE uses a term ordering that gives high precedence to uninterpreted operations and totally orders interpreted constants. This means that rewritings will try to rewrite terms involving uninterpreted operations into ones containing fewer (or no) uninterpreted operations and to rewrite terms containing numbers into those with smaller numbers, where possible. More concretely, the term ordering orders integers absolutely (e.g.$5 > 0$ and $-5 > 0$) and rationals $\frac{n}{d}$ are ordered absolutely by $n + d$ with $n$ breaking ties.

**Congruence Closure.** For reasoning with the theory of equality and uninterpreted functions (EUF) VAMPIRE implements an extension of the AVATAR architecture that utilises a congruence closure method [20] to check that the SAT model is ground-consistent with respect to EUF. This could be viewed as an SMT solver for this particular theory (see [34]).

# 3 AVATAR Modulo Theories

AVATAR [24, 34] is based on the concept of *splitting*. The idea is that the inconsistency of a set of clauses $S \cup \{C_1 \vee C_2\}$, for variable-disjoint sub-clauses $C_1$ and $C_2$, is equivalent to the inconsistency of both $S \cup \{C_1\}$ and $S \cup \{C_2\}$. A splitting technique will *split* the proof search into two branches, try to separately establish the inconsistency of each and finally combine the results. AVATAR differs from previous techniques such as splitting with backtracking [35] and splitting without backtracking [26] as it uses a SAT solver to make splitting decisions, as described below.

## 3.1 Preliminaries

Our setting is that of many-sorted first-order predicate logic with equality in *clausal form*.

A signature $\Sigma$ is a set of *predicate* and *function* symbols with associated arities. Terms are of the form $f(t_1, \ldots, t_n), c$ or $x$ where $f$ is a *function symbol* of arity $n \geq 1$, $t_1, \ldots, t_n$ are terms, $c$ is a zero arity function symbol (i.e. a constant) and $x$ is a variable. Atoms are of the form $p(t_1, \ldots, t_n), q$ or $t_1 \simeq t_2$ where $p$ is a *predicate symbol* of arity $n$, $t_1, \ldots, t_n$ are terms, $q$ is a zero arity predicate symbol and $\simeq$ is the *equality symbol*.

A *clause* is a disjunction of literals $L_1 \vee \ldots \vee L_n$ for $n \geq 0$. We disregard the order of literals and treat a clause as a multiset. When $n = 0$, we write $\bot$ and speak of the *empty clause*, which is always false. When $n = 1$ a clause is called a unit clause. Variables in clauses are considered to be universally quantified. Standard methods exist to transform an arbitrary first-order formula into clausal form.

Given clause $C \vee D$, the subclause $D$ is a *component* of $C \vee D$ if it is non-empty, *variable-disjoint* with $C$ and minimal (i.e. there is no proper subset of $D$ variable-disjoint with the rest of the clause). For example, given clause $x + 1 = a \vee p(x, y) \vee q(5)$ there are two components $x + 1 = a \vee p(x, y)$ and $q(5)$; as $x + 1 = a$ and $p(x, y)$ share the variable $x$ they cannot be split further.

An *interpretation* $\mathcal{I}$ interprets terms over a given universe and assigns boolean values to atoms. An interpretation is a *model* for a set of clauses if for each clause $L_1 \vee \ldots \vee L_n$ it assigns *true* to at least one literal $L_i$. A *theory* $\mathcal{T}$ constraints the set of viable interpretations by fixing interpretations for some part of the signature. For theory $\mathcal{T}$ we refer to this part of the signature $\Sigma_{\mathcal{T}}$ and call such symbols *interpreted*. For example, the theory of arithmetic may fix the interpretation for $\{+, -, <, >, *\} \in \Sigma_{Arith}$. An interpretation is consistent with a theory if it satisfies that theories constraints and is consistent with a

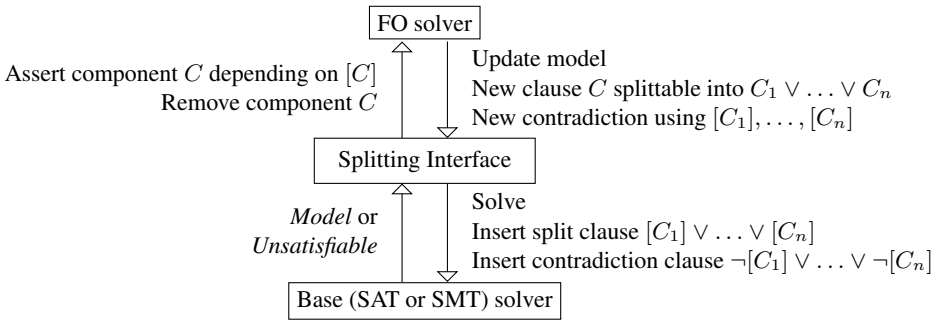


Figure 1: The AVATAR architecture.

*combination* of theories if it satisfies the constraints of each theory. We call such interpretations *theory consistent*. Note that we generally assume the theory of uninterpreted functions.

## 3.2 The Architecture

The AVATAR architecture (see Fig. 1) consist of three parts: the first order solver FO, the base (SAT or SMT) solver, and the intermediate Splitting interface. The steps that each part can take are given in Figure 2. This describes the AVATAR architecture as a set of rewrite rules where one of three configurations $\langle S, N, L, X, M\rangle_1$, $\langle S, L, X, M\rangle_2$, or $\langle S, L, X, M_{old}, M_{new}\rangle_3$ is rewritten by the different parts. The configurations are made up of:

- $S$ the set of first-order clauses with assertions (see below) held by the FO part
- $N$ the new first-order clauses with assertions derived from $S$
- $L$ a set of *locks*, mapping each assertion to a set of clauses with assertions (see below)
- $X$ a set of *abstracted clauses* where the abstraction depends on the base solver
- $M, M_{old}, M_{new}$ models over abstracted components

Below we explain each of the parts and the steps they can take in further detail.

**FO solver.** This deals with *clauses with assertions* (A-clauses) $C \leftarrow A$ where $C$ is a disjunction of literals and $A$ is a conjunction of *assertions*. The assertions of an A-clause capture the sub-problem that it belongs to. Later we will see that assertions denote abstracted components but for now consider them as propositional symbols.

For the purpose of this explanation we assume that the FO solver makes use of a set of *sound* inference rules $\mathbb{I}$ of the form

$$\frac{C_1 \leftarrow A_1 \quad \ldots \quad C_n \leftarrow A_n}{D \leftarrow A_1 \wedge \ldots \wedge A_n}$$

i.e. the assertion of the conclusion is the union of the assertions of the premises. This implies that if $C_2 \leftarrow A_2$ is derived from $C_1 \leftarrow A_1$ then $A_1 \subseteq A_2$. We write $S \vdash_{\mathbb{I}} N$ if the clauses $N$ can be derived from $S$ by the inference rules in $\mathbb{I}$.

As illustrated in Figure 2, the FO solver can make two kinds of step. Firstly, see rule (1), it can derive a set of new clauses that follow from the current clauses using $\mathbb{I}$. Secondly, it can attempt to delete an existing clause if it is *subsumed* by another existing clause.[2] There are two cases to consider here. If

[2] The AVATAR architecture can deal with any standard redundancy elimination rule. To simplify the exposition, we only use subsumption as an example here.

(1) $\langle S, N, L, X, M\rangle_1 \quad \Rightarrow_{FO} \quad \langle S, N \cup N', L, X, M\rangle_1$
where $S \cup N \vdash_{\mathbb{I}} N'$,

(2) $\langle S \cup \{C \leftarrow A\}, N, L, X, M\rangle_1 \quad \Rightarrow_{FO} \quad \langle S, N, L, X, M\rangle_1$
where $C'$ subsumes $C$ for some $C' \leftarrow A' \in S$ and $A' \subseteq A$,

(3) $\langle S \cup \{C \leftarrow A\}, N, L, X, M\rangle_1 \quad \Rightarrow_{FO} \quad \langle S, N, L' = L[A_i \mapsto L(A_i) \cup \{C \leftarrow A\}], X, M\rangle_1$
where $C'$ subsumes $C$ for some $C' \leftarrow A' \in S$ and $A_i \in A' \backslash A$,
and $L'$ is obtained from $L$ by adding $C \leftarrow A$ to $L(A_i)$,

(4) $\langle S, N, L, X, M\rangle_1 \quad \Rightarrow_{SI} \quad \langle S, L, X \cup \{[D_1]_B \vee \ldots \vee [D_k]_B \leftarrow A \mid C \leftarrow A \in N\}, M\rangle_2$
where $D_i$ are the components of $C$; note that it is possible that $C = \bot$,

(5) $\langle S, L, X, M\rangle_2 \quad \Rightarrow_B \quad \bot$
where $\not\models_B X$,

(6) $\langle S, L, X, M_{old}\rangle_2 \quad \Rightarrow_B \quad \langle S, L, X, M_{old}, M_{new}\rangle_3$
where $M_{new} \models_B X$,

(7) $\langle S, L, X, M_{old}, M_{new}\rangle_3 \quad \Rightarrow_{SI} \quad \langle \mathit{kept} \cup \mathit{new} \cup \mathit{unlocked}, \{\}, L', X, M_{new}\rangle_1$
where

$$\begin{array}{lcl} \mathit{kept} & = & \{C \leftarrow A \in S \mid M_{new} \models_B A\}, \\ \mathit{new} & = & \{C \leftarrow [C] \mid (\ldots \vee [C] \vee \ldots) \leftarrow A \in X \wedge M_{old} \not\models_B [C] \wedge M_{new} \models_B [C]\}, \\ \mathit{unlocked} & = & \{C \leftarrow A \mid C \leftarrow A \in L(A_i) \wedge M_{new} \not\models_B A_i\}, \\ L' & = & \{A_i \mapsto L(A_i) \mid M_{new} \models_B A_i\}. \end{array}$$

Figure 2: The AVATAR rules.

the subsuming clause will be present whenever the subsumed clause is present (i.e. if the assertions of the latter include those of the former) then the deletion is unconditional; see rule (2). Otherwise, as in rule (3), there exist circumstances (where an assertion belonging to the former but not the latter is false) where the subsumption will no longer hold. Therefore, it is necessary to store subsumptions that should be undone. This is done using the set of locks mentioned previously. $L$ associates an assertion with a set of A-clauses that must be reinserted if that assertion becomes false.

**Base solver.** The base solver works with abstracted clauses. As we discuss below, in the case of SAT these abstract clauses are propositional and in the case of SMT they are first-order ground clauses over theories. The base solver uses an abstraction function $[.]_B$ which will be specified later for each base solver we consider. The result of abstraction is a set of abstracted clauses $X$.

The base solver is asked to build a model of these clauses. The rules (5) and (6) in Figure 2 correspond to the two possible outcomes. The base solver can either find the $X$ is unsatisfiable and thus fail to to build a model. As the abstracted clauses follow from the first-order clauses, if the former is unsatisfiable then so are the latter. In this system this is the only method for establishing unsatisfiability. Alternatively the base solver can construct a model, which can then be queried. We write $M \models_B A$ if assertion $A$ (which denotes an abstract component) is true in the model.

**Splitting Interface.** This is the core of the AVATAR architecture and can make two steps. Firstly, it can perform the abstraction of the newly derived clauses for the base solver; see rule (4). Secondly, it can update the set of first-order clauses $S$ using a newly generated model. This second step is complex and deserves further explanation; see also rule (7).

When the model is updated three things need to happen. Firstly, A-clauses whose assertions are not true in the new model need removing. All others remain in the $kept$ set. Secondly, any components which have been newly asserted needed adding, this is the $new$ set. This is the point where we see that assertions denote abstracted components. Lastly, if an A-clause was locked based on an assertion $A_i$ (see above) and that assertion has become false then this means that the A-clause should be unlocked. To preserve the invariant that locked assertions are true in the current model, all assertions that are not true in the new model are also removed from $L$.

This presentation has left out some optional alternatives for ease of explanation. For example, clauses may not be splittable (i.e. if they consist of a single component) and then there is questionable utility in adding them to the abstraction (unless they are ground in the SMT case). See [24] for a further discussion.

**Proof Search.** Proof search begins with the configuration $\langle \emptyset, S, \emptyset, \emptyset, \emptyset \rangle_1$ where $S$ is the problem in clausal normal form, formally a set of A-clauses with empty assertions. This implies, we begin by assuming that these clauses have been newly derived and immediately consider splitting them.

## 3.3 The SAT Abstraction

The definition of a component means it is either (i) non-ground and of arbitrary size, e.g. $p(x)$ or $s(x, y) \vee r(y)$, or (ii) ground and a unit, e.g. $p(f(a))$ or $a > f(1)$.

The SAT abstraction does not differentiate between the two kinds of components. We construct $[.]_{\mathrm{SAT}}$ as an injective mapping from components to propositional variables such that components that are equivalent up to literal reordering, variable renaming, and symmetry of equality are mapped to the same propositional literal. This literal is always positive for non-ground components (i). For the ground components (ii), we maintain that $[\neg L]_{\mathrm{SAT}} = \neg[L]_{\mathrm{SAT}}$. This ensures the FO solver is never asked to work on a problem containing both $\neg L$ and $L$, thus being trivially inconsistent.

**Example 1.** *Consider the clause*

$$C = x > (y + 1) \vee y = 2 \vee a < 5 \vee z > a$$

*the components are* $D_1 = x > (y + 1) \vee y = 2$, $D_2 = a < 5$, *and* $D_3 = z > a$. *Therefore the abstraction* $[C]_{\mathrm{SAT}} = p_1 \vee p_2 \vee p_3$ *where* $p_i$ *are propositional variables not assigned to any other components.*

## 3.4 The SMT Abstraction

The SMT abstraction $[.]_{\mathrm{SMT}}$ works the same way as the SAT one for the non-ground components of kind (i). Ground components of kind (ii), however, are not abstracted i.e. their first-order form is preserved with the following caveats. Given an SMT solver for a theory (or a combination of theories) $\mathcal{T}$ we treat all symbols not in $\Sigma_{\mathcal{T}}$ as uninterpreted for the SMT solver. We therefore assume that $\mathcal{T}$ includes the theory of uninterpreted functions.

**Example 2.** *Let us revisit the above clause* $C$. *The abstraction is now* $[C]_{\mathrm{SMT}} = p_1 \vee (a < 5) \vee p_3$ *i.e. only components of kind (ii) are treated differently.*

Changing the base solver from a SAT solver to an SMT solver only changes what is required of the translation, the rest of the architecture remains the same. The model of the base solver is only queried for the truth of translated components. This means that if a ground component is $a > 5$ and the SMT solver has decided that $a = 10$ in the model, we only learn that $a > 5$ is true and not more.

The advantage of using an SMT solver as a base solver is that all models will be theory-consistent in their ground part. This should reduce the number of models that need to be considered in order to find a contradiction. However, this is not always the case as selecting a different model can cause proof search to take a significantly different direction, missing useful clauses that may otherwise have been quickly derived.

Finally, we note that the SMT solver can select more than one component in a clause. In some cases this may be necessary i.e. to satisfy all clauses. Previous work [24] considers the minimisation of models to (greedily) use as few components as possible.

### 3.5 Examples

We give a brief artificial example that demonstrates the advantage of using an SMT solver as a base solver. Consider the following problem over three integer constants.

$$(a_0 > 1) \wedge (a_0 > 0 \Rightarrow a_1 > a_0) \wedge (a_1 > a_0 \Rightarrow a_2 > a_1) \wedge (\forall x)(p(x) \Leftrightarrow x > a_2) \wedge p(4)$$

The ground part of the problem forces $1 < a_0 < a_1 < a_2$, meaning that $3 < a_2$. The non-ground part defines a predicate $p$ to be true when its input is greater than $a_2$. The ground assertion $p(4)$ then leads to the inconsistency $3 < a_2 \wedge a_2 < 4$. This is difficult to solve when using the SAT abstraction but straightforward when using the SMT abstraction as the SMT solver can rule out all but one model.

However, the above example is simply solved by the MBQI technique of Z3 [15, 11]. Indeed, small problems where this combination outperforms SMT-based methods are difficult to find as we are targeting problems with non-trivial quantifier usage and non-trivial theory usage. However, it can be the case that small problems are solved much faster using this approach.

The problem NRA/keymaera/vsl.proof-node2228 from SMT-LIB, for example, consists of a single assertion over 10 constants and a single variable. However, it takes Z3 159.75 seconds to solve this problem whilst VAMPIRE finds a solution in 0.169 seconds. Interestingly, the proof consists of some non-trivial theory reasoning with theory axioms followed by a single call to Z3 which finds the resulting ground abstraction unsatisfiable.

## 4 Implementation

In this section we describe how we implemented the AVATAR modulo theories approach using the Z3 SMT solver.

### 4.1 Overview of Z3

Z3 [13] is a Satisfiability Modulo Theories (SMT) solver [21] developed at Microsoft research. We focus on Z3 as a black box theory solver, as that is how it is integrated into VAMPIRE, rather than considering its architecture.

**Theories.** Z3 can handle various theories but the ones we use here correspond to those supported by VAMPIRE i.e. theories of linear and non-linear real and integer arithmetic, uninterpreted functions and extensional arrays. Z3 can decide ground problems involving combinations of linear arithmetic,

uninterpreted functions and extensional arrays. However, non-linear integer arithmetic is undecidable and in this case Z3 may return *unknown* as its verdict.

**Quantifiers.** Whilst Z3 has support for quantifiers they are not currently used in this work as the underlying hypothesis is that methods such as superposition and resolution are more effective at dealing with quantified formulas. There may be scope in the future to consider situations where Z3's E-matching mechanisms could be utilised to produce useful instantiations. Earlier versions of Z3 contained an integration of superposition and theory reasoning [12] and the approach we take here revisits this direction through the lens of AVATAR.

**Incremental API.** We make use of the C++ API provided for Z3 that allows for the creation of solver objects that can incrementally be passed formulas and asked if the current set of formulas is consistent. If the formulas are consistent the solver can be queried for a model, which can be used to evaluate terms in the signature of the original problem.

## 4.2 Implementing the Abstraction.

The implementation of $[.]_{\mathrm{Z3}}$ is straightforward. Predicates are represented as boolean functions, interpreted operations are mapped to their counterparts, numbers are translated directly, new sorts are created where necessary and uninterpreted functions are mapped to uninterpreted functions. Some interpreted functions from the TPTP arithmetic theory needed careful translation to the corresponding functions in Z3. For example, the $round(t) function with the semantics that the number is rounded to the nearest integral (with even numbers breaking ties) must be translated as

$$\mathsf{ite}(t > (t + 1/2), t + 1, \mathsf{ite}(t == 1/2, \mathsf{ite}(\mathsf{mod}(t, 2) == 0, t, t + 1), t))$$

where ite is a native conditional operator in the SMT language. Furthermore, we chose to add functions such as quotient_t and remainder_t as uninterpreted and include additional axioms. For example,

$$b \neq 0 \rightarrow ((b * \mathtt{quotient\_t}(a, b)) + \mathtt{remainder\_t}(a, b)) = a$$

Non-ground components are translated abstracted by boolean variables as in the SAT abstraction.

**Models.** For each component we record the associated Z3 expression. Then, when a model is computed, we evaluate this expression in the model. For example, if the clause $a > 5 \vee b > 5$ is added to Z3 and Z3 builds a model where $a = 1$ and $b = 10$ we will evaluate the expressions $a > 5$ and $b > 5$ when deciding whether this components should be asserted.

**Underspecified Operations.** The previous approach does not work for underspecified functions such as division and modulo by zero. For example, if the unit clause $5/c = 2 \vee c = 0$ is added to Z3 and Z3 builds a model where $c = 0$ then asking Z3 to evaluate $5/c = 2$ will result in *unknown*. Therefore, we introduce fresh names for expressions possibly containing underspecified operations, i.e. we would add $p = (5/c = 2)$ as an additional clause and $p$ as a unit clause. Then we can query the model for the value of $p$, which must be true or false.

## 4.3 Incompleteness

In the original implementation of AVATAR using a SAT solver, either a model of the SAT clauses is found and splitting can occur, or it is not, which implies that the original clauses are unsatisfiable. In

this new organisation with an SMT solver there is a third option: the SMT solver fails to find a model or a contradiction. This can happen when the ground problem is from an undecidable theory. For example, consider a problem that has the formula

$$(a > 0) \wedge (b > 0) \wedge (c > 0) \wedge (a * a * a) + (b * b * b) = (c * c * c)$$

as an axiom, where $a, b, c$ are integer constants, i.e. we ask the SMT solver to solve Fermat's last theorem.

Although an SMT solver may support undecidable theories, such as non-linear integer arithmetic, in a limited way, we cannot expect the solver to recognise (in)consistency of every statement in the logic. The above axiom is an example of a inconsistent formula the status of which is hard to recognise for obvious reasons.

In the standard AVATAR theory no progress can be made without a model. Therefore, if no model or refutation can be found, VAMPIRE will give up on proof search for this reason. This is clearly undesirable.

To deal with this potential issue, VAMPIRE implements an option that runs a SAT solver alongside the SMT solver (–sat_fallback_for_smt). All clauses that would have been passed to the SAT solver in the original AVATAR framework would be given to this fallback solver. However, the SAT solver is not used until the SMT solver returns Unknown. In this case a model is taken from the SAT solver and proof search proceeds. The next time a model is required the SMT solver is asked again, because further generated clauses could have caused the undecidability to be resolved.

In our experiments described in the next section this fallback mechanism is used rarely. Only four problems from TPTP required the mechanism, using between 1 and 65 fallbacks.

# 5 Comparative Evaluation

We compare VAMPIRE to other first-order ATP systems and SMT solvers on problems from the TPTP library [31] and SMT-LIB library [4]. Evaluation was carried out on the StarExec cluster[3].

## 5.1 Evaluated Solvers

We compare VAMPIRE to competitive versions of other solvers.

**Other ATP systems.** We consider the following other ATP systems taken from the CASC repository on StarExec [30]:

- Beagle (0.9.22) [5] implements hierarchic superposition calculus with weak abstraction [6].
- Princess (20150706) [28, 29] implements a free-variable tableau calculus for linear integer arithmetic with some extensions for non-linear arithmetic.
- SPASS+T (2.2.22) [23, 36] is a combination of saturation-based theorem prover SPASS [35] and an SMT solver.
- ZenonArith (0.1.0) [9] and ZenonModulo (0.4.1) [14] extend the tableaux-based Zenon [7] to linear arithmetic and deduction modulo [16], respectively.
- Zipperposition (0.4) [10] is a superposition-based prover implementing an extension of the calculus for integer linear arithmetic.

[3] https://www.starexec.org

Table 1: Results for TPTP problems (- means unsuitable for the solver).

| Division | Size | Easy | Beagle | CVC4 | Princess | SPASS+T | VAMPIRE | Z3 | ZenonArith | ZenonModulo | Zipperposition |
|---|---|---|---|---|---|---|---|---|---|---|---|
| IL | 461 | 213 | 361 | 355 | 359 | 358 | **426** | 305 | 149 | 101 | 281 |
| IN | 173 | 45 | 97 | 141 | 129 | 73 | **145** | 109 | 61 | 43 | 114 |
| QL | 121 | 34 | 120 | **121** | **121** | 118 | 120 | 58 | 116 | 81 | - |
| QN | 38 | 5 | **37** | **37** | 35 | **37** | **37** | 17 | 36 | 25 | 3 |
| RL | 116 | 66 | **115** | **115** | **115** | **115** | 114 | 114 | 112 | 78 | - |
| RN | 39 | 6 | **39** | 36 | 34 | 37 | 37 | 38 | 37 | 25 | - |
| IRL+N | 9 | 6 | 8 | 8 | **9** | 5 | **9** | **9** | 0 | 0 | - |
| IQRL | 8 | 0 | 2 | 2 | 2 | 0 | 2 | 0 | 0 | 0 | - |
| IQRN | 3 | 1 | 2 | **3** | 2 | 2 | **3** | 2 | 0 | 0 | - |
| Total | 968 | 376 | 787(1) | 824 | 812 | 745 | **899(37)** | 652(2) | 511 | 353 | 398(3) |

**SMT solvers.** We consider three SMT systems taken from the CASC and SMT-COMP repository on StarExec. Firstly, Z3 as described in Section 4.1 from Microsoft Research (4.4.2), secondly **CVC4** (1.5 for CASC and SMT-COMP 2016) [3], and lastly **veriT** [8] (SMT-COMP 2015).

**Strategy Scheduling.** Both VAMPIRE and **CVC4** makes use of strategy scheduling. In both systems this approach is necessary for full coverage of the problems solved. We are not aware if other systems make use of strategy scheduling.

## 5.2 TPTP Problems

Experiments with TPTP problems were run for 5 minutes as this is (usually) the time limit used in the CASC competition. Table 1 presents the results comparing the systems on 968 relevant (not satisfiable) benchmarks taken from the TPTP library. These have been split into categories where I=integer, Q=rational, R=real, L=linear and N=non-linear[4]. We have included all TPTP problems, even those of rating 0.0 (those supposedly solved by all provers). Table 1 reports the total number of and the number of easy problems in each category, the number of solved problems per system, and the total number of problems solved across all categories with unique solutions in brackets.

There are 53 problems unsolved by any system, almost all of which are from the IL category. This is unsurprising as the majority of problems from TPTP are from the IL category. VAMPIRE performs best in the IL and IN categories and solves the most problems overall, being comparable in most other categories. We note that the top two systems (VAMPIRE and CVC4) are relatively close each other in performance. It is interesting to note that even systems such as Zipperposition, which solve relatively few problems overall, solve some problems uniquely.

## 5.3 SMT-LIB Problems

Experiments with SMT-LIB problems were run for 30 minutes as this is (usually) the time limit used in the SMT-COMP competition. Table 2 presents the results comparing systems on all relevant problems

[4]By non-linear we mean non-linear operations such as multiplication, not the theory of non-linear arithmetic. This means that they do not correspond to any SMT-LIB category.

Table 2: Results SMT-LIB problems (- means unsuitable for the solver).

| | Size | CVC4 | VAMPIRE | veriT | Z3 |
|---|---|---|---|---|---|
| ALIA | 41 | **41** | 40 | 27 | **41** |
| AUFLIA | 3 | **3** | 2 | 1 | 2 |
| AUFLIRA | 19,914 | 19,761 **(11)** | **19,777** (9) | 19,259 | 19,751 |
| AUFNIRA | 1,491 | 1,041 | **1,085 (45)** | - | 1,034 (3) |
| LIA | 380 | 86 **(21)** | 65 | **159** | 24 |
| LRA | 605 | **344 (6)** | 331 | 78 | 339 |
| NIA | 8 | 3 | 4 | - | **5 (1)** |
| NRA | 3,813 | 3,735 | 3,802 (4) | - | **3,806 (8)** |
| UFIDL | 74 | 62 | **66 (4)** | 57 | 62 |
| UFLIA | 12,114 | **8,536 (79)** | 8,479 **(151)** | 6,738 | 7,815 (3) |
| UFLRA | 20 | 20 | 20 | 20 | 20 |
| UFNIA | 3,351 | 1,373 (28) | **1,777 (371)** | - | 1,235 (12) |
| Total | 41,814 | 35,390 (145) | **35,448 (584)** | 27,844 | 34,386 (27) |

(i.e. problems containing quantifiers and theories but not using bit-vectors) from the SMT-LIB library that are not known to be satisfiable (problems with unknown status were included). The table uses the divisions of SMT-LIB to report the number of unsat results found in each division.

VAMPIRE performs well in divisions that include *A*rrays or the *U*ninterpreted *F*unctions. Indeed, for many (not all) of the other divisions the SMT solvers implement decision procedures and it would be surprising to see VAMPIRE outperforming these. We note that ATP systems outperforming SMT solvers in the other cases should not be surprising as these are non-ground problems where SMT solvers must perform instantiation to deal with quantifiers. It is not surprising that Z3 has unique solutions as, whilst the AVATAR modulo theories architecture makes use of Z3 it does not utilise its quantifier instantiation procedures.

VAMPIRE solves 584 problems uniquely. We believe this is because VAMPIRE is utilising a fundamentally different approach to solving these problems. Whereas the three SMT solvers are based on the $DPLL(\mathcal{T})$ architecture, VAMPIRE's usage of superposition allows it to derive first-order consequences and ground instances that cannot be derived by SMT solvers.

# 6 Related Work

There are a number of approaches that attempt to combine quantifier and SMT-based theory reasoning.

SMT solvers such as Z3 and CVC4 implement E-matching [15, 11], model based quantifier instantiation [15, 11] and conflict instantiation [25] techniques to deal with quantifiers. These all take the form of instantiation and are generally heuristic, although for some fragments they can form a decision procedure.

The work of $DPLL(\Gamma)$ [12] combines a superposition prover with an SMT solver (Z3) in a similar way to that described in this paper. In this combination the ground literals decided and implied by the SMT solver were used as hypotheses to first-order clauses. Therefore, the main difference with our approach is that (i) we use arbitrary components rather than ground literals; and (ii) we treat the SMT solver as a black box whilst $DPLL(\Gamma)$ interleaves the architectures. The effect of this interleaving is that backtracking in $DPLL(\Gamma)$ is more similar to splitting with backtracking.

SPASS+T [23, 36] is based on the idea of hierarchic superposition [2]. Whilst VAMPIRE relies on the AVATAR architecture to communicate with the SMT solver, in SPASS+T the two systems are

more loosely coupled. SPASS+T simply passes the derived ground formulas to the SMT solver, which may detect inconsistency (of this ground part, and thus implicitly of the whole input formula) using its decision procedures.

# 7 Conclusion and Further Work

We have described an extension to the AVATAR architecture that allows it to reason modulo theories by replacing the SAT solver with an SMT solver. The implementation of the resulting architecture within the VAMPIRE theorem proving using Z3 was described, highlighting practical issues that were overcome. Experimental results demonstrated that this combination can be highly effective, often outperforming both ATP systems and SMT solvers.

This is a first step in exploring this novel combination of ATP systems and SMT solvers and there are many avenues for further investigation. For example, we would like to consider whether more useful information could be extracted from the SMT solver. For example, the SMT solver may be able to derive bounds on constants that could be used in first-order reasoning. Currently we do not utilise any quantifier reasoning capabilities of Z3. Another extension to this work would be to explore whether (limited applications of) such techniques could complement the quantifier-reasoning carried out in the first-order part in a useful way and thus help to improve the performance of the combined system.

# References

[1] L. Bachmair and H. Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, vol. I, chapter 2, pp. 19–99. Elsevier Science, 2001.

[2] L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierarchic first-order theories. *Appl. Algebra Eng. Commun. Comput.*, 5:193–212, 1994.

[3] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, number 6806 in Lecture Notes in Computer Science, pp. 171–177. Springer-Verlag, 2011.

[4] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2010.

[5] P. Baumgartner, J. Bax, and U. Waldmann. Beagle a hierarchic superposition theorem prover. In *Automated Deduction - CADE-25*, vol. 9195 of *Lecture Notes in Computer Science*, pp. 367–377. Springer International Publishing, 2015.

[6] P. Baumgartner and U. Waldmann. Hierarchic Superposition With Weak Abstraction. In *Proceedings of the 24th International Conference on Automated Deduction*, number 7898 in Lecture Notes in Artificial Intelligence, pp. 39–57. Springer-Verlag, 2013.

[7] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 4790 in Lecture Notes in Artificial Intelligence, pp. 151–165, 2007.

[8] T. Bouton, D. Caminha B. de Oliveira, D. Déharbe, and P. Fontaine. *veriT: An Open, Trustable and Efficient SMT-Solver*, pp. 151–156. Springer Berlin Heidelberg, 2009.

[9] G. Bury and D. Delahaye. Integrating Simplex with Tableaux. In *Proceedings of the 24th Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, number 9323 in Lecture Notes in Computer Science, pp. 86–101. Springer-Verlag, 2015.

[10] S. Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond*. PhD thesis, INRIA, 2015.

[11] L. M. de Moura and N. Bjørner. Efficient e-matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pp. 183–198, 2007.

[12] L. M. de Moura and N. Bjørner. Engineering DPLL(T) + saturation. In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pp. 475–490, 2008.

[13] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proc. of TACAS*, vol. 4963 of *LNCS*, pp. 337–340, 2008.

[14] D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon modulo: When achilles outruns the tortoise using deduction modulo. In *Logic for Programming, Artificial Intelligence, and Reasoning*, vol. 8312 of *Lecture Notes in Computer Science*, pp. 274–290. Springer Berlin Heidelberg, 2013.

[15] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[16] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72.

[17] K. Hoder, G. Reger, M. Suda, and A. Voronkov. Selecting the selection. In *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 – July 2, 2016, Proceedings*, pp. 313–329. Springer International Publishing, 2016.

[18] E. Kotelnikov, L. Kovács, G. Reger, and A. Voronkov. The vampire and the FOOL. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pp. 37–48, 2016.

[19] L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, vol. 8044 of *Lecture Notes in Computer Science*, pp. 1–35, 2013.

[20] R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007.

[21] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL($T$). *J. ACM*, 53(6):937–977, 2006.

[22] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, vol. I, chapter 7, pp. 371–443. Elsevier Science, 2001.

[23] V. Prevosto and U. Waldmann. SPASS+T. In *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning*, number 192 in CEUR Workshop Proceedings, pp. 19–33, 2006.

[24] G. Reger, M. Suda, and A. Voronkov. Playing with avatar. In *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pp. 399–415. Springer International Publishing, 2015.

[25] A. Reynolds, C. Tinelli, and L. M. de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pp. 195–202, 2014.

[26] A. Riazanov and A. Voronkov. Splitting without backtracking. In *17th International Joint Conference on Artificial Intelligence, IJCAI'01*, vol. 1, pp. 611–617, 2001.

[27] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comput.*, 36(1-2):101–115, 2003.

[28] P. Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings of the 15th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, number 5330 in Lecture Notes in Artificial Intelligence, pp. 274–289. Springer-Verlag, 2008.

[29] P. Rümmer. E-Matching with Free Variables. In *Proceedings of the 18th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, number 7180 in Lecture Notes in Artificial Intelligence, pp. 359–374. Springer-Verlag, 2012.

[30] A. Stump, G. Sutcliffe, and C. Tinelli. StarExec, a cross community logic solving service. https://www.starexec.

org, 2012.

[31] G. Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[32] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[33] G. Sutcliffe and J. Urban. The CADE-25 automated theorem proving system competition - CASC-25. *AI Commun.*, 29(3):423–433, 2015.

[34] A. Voronkov. AVATAR: The architecture for first-order theorem provers. In *Computer Aided Verification*, vol. 8559 of *Lecture Notes in Computer Science*, pp. 696–710. Springer International Publishing, 2014.

[35] C. Weidenbach. Combining superposition, sorts and splitting. In *Handbook of Automated Reasoning*, vol. II, chapter 27, pp. 1965–2013. Elsevier Science, 2001.

[36] S. Zimmer. Intelligent Combination of a First Order Theorem Prover and SMT Procedures. Master's thesis, Saarland University, 2007.