Instantiation and Pretending to be an SMT Solver with VAMPIRE *

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

[1] University of Manchester, Manchester, UK
[2] TU Wien, Vienna, Austria
[3] Chalmers University of Technology, Gothenburg, Sweden
[4] EasyChair

Abstract

This paper aims to do two things. Firstly, it discusses how the VAMPIRE automatic theorem prover both succeeds and fails at pretending to be an SMT solver. We discuss how Vampire reasons with problems containing both quantification and theories, the limitations this places on what it can do, and the advantages this provides over the standard SMT approach. Secondly, it focuses on the problem of instantiation of quantified formulas and asks whether VAMPIRE needs it (it does) and whether it can directly borrow techniques from SMT solving (maybe).

1 Introduction

VAMPIRE [22] is an automatic theorem prover (ATP) for first-order logic. It is successful at establishing both theorems and non-theorems, winning related tracks in the well-known CASC competition[1] [37]. For the last few years, we have been extending VAMPIRE's theory reasoning abilities and in 2016 VAMPIRE entered SMT-COMP[2] [5] and performed well in a number of divisions.

Whilst there is a long history of extending ATPs with theory reasoning capabilities, this has only achieved limited success and in recent years theory reasoning has been dominated by SMT solvers. To extend satisfiability checking procedures for ground theories to handle quantifiers, the focus has been on *instantiation* techniques that aim to find useful instances of quantified formulas to pass to the ground procedures. Arguably, this may become a bottleneck when the reasoning requires many non-trivial instances. Conversely, ATPs reason directly with universally quantified clauses and make inferences at this level, e.g., $p(x) \vee q(x)$ resolves with $\neg p(f(y))$ to establish that all instances of $q(f(y))$ must hold. This approach performs generally well for pure first-order problems, but is not traditionally suited to reasoning with theories. We conjecture (currently without evidence) that instance-based techniques are more suited to problems with large ground parts and small quantified parts, whilst saturation-based techniques are more suited to problems making heavy use of quantifiers. We are currently exploring this conjecture. The goal of recent developments in VAMPIRE has been to preserve efficient techniques for reasoning with quantifiers whilst adding effective techniques for reasoning with theories.

This paper assumes general knowledge of SMT solving and first-order logic as this paper is written with the SMT community in mind. An introduction to these topics can be found in many of the papers cited in this paper. We begin (Section 2) by describing how VAMPIRE is both similar and different to an SMT solver. We then describe briefly how VAMPIRE currently reasons with quantifiers and theories (Section 3). Finally, we consider the issue of *instantiation*, the primary approach for dealing with quantified formulas in SMT solvers, and consider its relevance to reasoning in VAMPIRE (Section 4).

*Martin Suda and Andrei Voronkov were partially supported by ERC Starting Grant 2014 SYMCAR 639270. Martin Suda was also partially supported by the Austrian research projects FWF S11403-N23 and S11409-N23. Andrei Voronkov was also partially supported by the Wallenberg Academy Fellowship 2014 - TheProSE.

[1] www.cs.miami.edu/~tptp/CASC/
[2] http://smtcomp.sourceforge.net/2016/

2 Pretending to be an SMT Solver

Here we describe the main differences between SMT solvers and an ATP such as VAMPIRE. The following discussion is applicable to provers similar to VAMPIRE, but to save space we will take VAMPIRE as a canonical example of provers similar to VAMPIRE and make a direct comparison between SMT solvers and VAMPIRE. To be more precise, when we refer to SMT solvers we are generally thinking of lazy CDCL(T) solvers [6] that handle quantifiers such as Z3 [13] and CVC4 [4].

2.1 The Main Similarities and Differences

The main similarity between an SMT solver and VAMPIRE is that both accept problems in first-order logic extended with theories. VAMPIRE accepts problems in SMT-LIB syntax [7], including recent additions such as datatypes [21]. The only logic not supported by VAMPIRE is the theory of bitvectors, although this is also under active development.

The main difference between an SMT solver and VAMPIRE is that SMT solvers have *satisfiability-checking* at their core, whilst VAMPIRE has *unsatisfiability-checking* at its core, as it is a refutational prover. By this we mean that SMT solvers attempt to build a model satisfying the input problem, whilst VAMPIRE attempts to find a proof that the input problem is inconsistent. If an SMT solver succeeds it returns sat and if VAMPIRE succeeds it returns unsat. Under certain circumstances, a failure to build a model or find a proof means that no model or proof exists. In that case, an SMT solver would return unsat and VAMPIRE, respectively, would return sat. A third option is that the process runs out of resources before establishing either or the process is *incomplete*, in which case both would return unknown. For SMT solvers, incompleteness comes from ground theories lacking decision procedures and from incomplete instantiation techniques. For VAMPIRE, incompleteness can come from certain proof search heuristics (e.g. selection functions [17]) or from *any* theory that is not finitely axiomatisable. This last point means that VAMPIRE can only answer sat for the theory of uninterpreted functions, or for ground problems where all theory parts are passed to an SMT solver via the AVATAR Modulo Theories (see Section 3.3) technology (in which case this is uninteresting).

A key feature of SMT solvers is the ability to produce *models*. As explained further below, VAMPIRE finds unsatisfiability by deriving a contradiction from the input problem through the iterative generation of consequences of that problem. This approach is well-suited to producing *proofs* of unsatisfiability, but does not in general produce *models*. Whilst VAMPIRE does support finite model building [30], this is (currently) restricted to the theory of uninterpreted functions only.

Another frequently used feature of SMT solvers is their ability to be used *incrementally*, due to their basis in the backtracking search of CDCL. VAMPIRE does not share this quality as proof search is inherently non-incremental and cumbersome to backtrack. We give further explanation for why below.

Similar to many SMT solvers, VAMPIRE is a portfolio solver. It implements many different techniques and when solving a problem it may use tens to hundreds of different strategies in a time-sliced fashion. Along with the below saturation-based proof search method, VAMPIRE also implements the InstGen calculus [19] and a finite-model building through reduction to SAT [30]. In addition, VAMPIRE can be run in a parallel mode where strategies are distributed over a given number of cores.

2.2 Saturation-Based Proof Search

To illustrate the differences outlined above we will briefly describe saturation-based proof search. Usually, VAMPIRE deals with the separate notions of *axioms* and *conjecture*,[3] but SMT-LIB does not have

[3] Intuitively, axioms formalise the domain of interest and the conjecture is the claim that should logically follow from the axioms. Conjecture is, therefore, negated before we start the search for a contradiction.

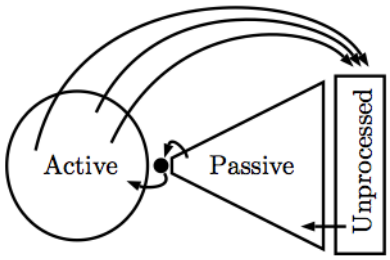

Figure 1: Illustrating the Given Clause Algorithm.

these notions and we assume an input (conjunctive) set of formulas. This set of formulas is clausified to produce a set of clauses S. This set is then *saturated* with respect to some inference system $\mathcal{I}$ meaning that for every inference from $\mathcal{I}$ with premises in S the conclusion of the inference is also added to S. If the saturated set S contains a contradiction then the initial formulas are unsatisfiable. Otherwise, if $\mathcal{I}$ is a complete inference system and, importantly, the requirements for this completeness have been preserved, then the initial formulas are satisfiable. Finite saturation may not be possible and many heuristics are employed to make finding a contradiction more likely.

The standard approach to saturation is the *given-clause algorithm* illustrated in Figure 1. The idea is to have a set of active clauses with the invariant that all inferences between active clauses have been performed and a set of passive clauses waiting to be activated. The algorithm then iteratively selects a *given clause* from passive and performs all necessary inferences to add it to active. This process of clause selection is important for good performance (and fairness) but not discussed here.

VAMPIRE uses resolution and superposition as its inference system $\mathcal{I}$ [1, 25]. A key feature of this calculus is the use of *literal selection* and *orderings* to restrict the application of inference rules, thus restricting the growth of the clause sets. There are well-known conditions on literal selection that preserve completeness. But VAMPIRE also uses incomplete approaches, which often turn out to be better practically [17]. Another very important concept related to saturation is the notion of *redundancy*. The idea is that some clauses in S are *redundant* in the sense that they can be safely removed from S without compromising completeness. The notion of saturation then becomes saturation-up-to-redundancy [1, 25]. An important redundancy check is *subsumption*. A clause A subsumes B if some subclause of B is an instance of A, in which case B can be safely removed from the search space as doing so does not change the possible models of the search space S. Later we will see how this hinders instantiation.

It is important to note that the process of going from input formulas to clauses is (i) important for efficiency, but also (ii) typically not equivalence preserving. Whilst the generation of clauses from formulas typically preserves models on the initial signature [31], many useful preprocessing steps remove part of the input. For example, the removal of unused definitions [16] removes the definition of a symbol if it is unused in the problem, or partially removes it if the symbol is used with a single polarity.

Let us consider what incrementality would look like in this setting. After saturating the set of clauses, a new set of clauses would be added, and proof search would continue. However, this new set of clauses may break the conditions under which optimising preprocessing steps are applied, making proof search incomplete. An easy solution would be to disable such preprocessing steps, as is done in incremental SMT. Additionally, care should be taken if the signature is extended, as parameters to proof search, such as term orderings, are dependent on the signature. Another important element of incremental proof search is the ability to *backtrack* previously asserted formulas. This would require the removal of clauses that may have caused the reduction of other clauses, requiring heavy machinery for tracking and undoing such steps (as done in splitting with backtracking [18]). Whilst solutions to

Table 1: Results for ground SMT-LIB problems; results for solvers that are not VAMPIRE were taken from SMT-COMP 2016. Time limit 1800s. (- means no results; unique numbers in parentheses).

	CVC4	Yices	mathsat	smtinterpol	veriT	VAMPIRE	z3
QF_ALIA	79	**80**	**80**	**80**	16	79	79
QF_ANIA	**8 (1)**	-	-	-	-	7	7
QF_AUFLIA	**516**	**516**	**516**	**516**	15	511	**516**
QF_AUFNIA	**15**	-	-	-	-	**15**	**15**
QF_AX	**279**	**279**	**279**	**279**	-	**279**	**279**
QF_IDL	726 (2)	727 (2)	-	691	707	485	**751 (23)**
QF_LIA	2,783	2,740	27,67	**2,800** (1)	936	950	2,680 **(13)**
QF_LIRA	**5**	**5**	-	2	-	**5**	4
QF_LRA	**625**	620	619	604	609	595	598
QF_NIA	145	**296**	-	-	-	295 **(2)**	**296**
QF_NIRA	**2 (1)**	1	-	-	-	1	1
QF_NRA	2,694	4,847 **(31)**	-	-	-	4,325 (22)	**4,849** (5)
QF_RDL	111	**113 (1)**	-	104	111	98	111
QF_UF	4,090	**4,100 (1)**	4,021	4,019	4,098	4,025	4,024
QF_UFIDL	306	**333**	-	316	305	171	**333**
QF_UFLIA	**195**	**195**	**195**	**195**	193	**195**	**195**
QF_UFLRA	**853**	**853**	**853**	**853**	**853**	842	**853**
QF_UFNIA	**7**	**7**	-	-	-	**7**	**7**
QF_UFNRA	**18**	**18**	-	-	-	**18**	**18**
Total	13457 (4)	**15,730** (35)	9330	10,459 (1)	7843	12,903 (24)	15,616 **(41)**

these issues exist, it is not clear that they would be practical and they are not implemented in VAMPIRE.

2.3 Some Results

We briefly explore the similarities and differences with some benchmark results.

Quantifier-Free Problems. Our first set of results is for problems that VAMPIRE is unlikely to perform well on: quantifier-free problems. Table 1 uses the results from SMT-COMP 2016 and compares these to results from a VAMPIRE run on the same problems and platform (e.g. StarExec). We only considered problems known to be unsatisfiable, as VAMPIRE cannot show satisfiability for problems with theories. In general, VAMPIRE did not perform as well as most other SMT solvers, although it solved 24 problems uniquely, mostly in non-linear real arithmetic. The conclusion is that VAMPIRE should not be selected as the system of choice for solving quantifier-free problems. The results for QF_UF show that this conclusion is not restricted to problems involving arithmetic, although the difference here is small.

Quantifier Problems. Our previous study [27] compared VAMPIRE and three well-known SMT solvers (CVC4, veriT, and Z3) and we refer the reader to this publication for those results. In this evaluation, which considered problems that were unsatisfiable or whose results were unknown, VAMPIRE solved the most problems overall (by 0.2%) and the most for the logics AUFLIRA, AUFNIRA, UFIDL, and UFNIA. The improvement over other SMT solvers is not very large, but there were unique solutions suggesting a complementary approach.

$$
\begin{array}{ccc}
x + (y + z) = (x + y) + z & x + 0 = x & x + y = y + x \\
-(x + y) = (-x + -y) & - - x = x & x + (-x) = 0 \\
x * 0 = 0 & x * (y * z) = (x * y) * z & x * 1 = x \\
x * y = y * x & (x * y) + (x * z) = x * (y + z) & \neg(x < y) \vee \neg(y < z) \vee \neg(x < z) \\
x < y \vee y < x \vee x = y & \neg(x < y) \vee \neg(y < x + 1) & \neg(x < y) \vee x + z < y + z \\
\neg(x < x) & x < y \vee y < x + 1 \; (for\ ints) & x = 0 \vee (y * x)/x = y \; (for\ reals)
\end{array}
$$

Figure 2: Arithmetic theory axioms currently added by VAMPIRE based on the problem signature.

2.4 A Summary

SMT solvers and VAMPIRE are different mainly due to their underlying proof search approach, and thus the features available (i.e. limited model-building for VAMPIRE). They are similar as they apply to the same problems and, for a large number of problems, perform similarly. However, results show that, mostly likely due to this fundamental difference in approach, they are likely to complement each other.

3 Theory Reasoning in Vampire

The previous section explained how VAMPIRE is different from an SMT solver. So how does it perform theory reasoning? This section aims at addressing this question.

3.1 Evaluation and Theory Axioms

One of the earliest practical saturation-based approaches for reasoning with theories was SPASS+T [26]. They utilised two ideas that we adopted in VAMPIRE: simplification of theory terms, and introduction of theory axioms. This approach for theory reasoning has been implemented in VAMPIRE since 2010.

Evaluation. In VAMPIRE, this is implemented as an *immediate simplification*, i.e., whenever we generate a new term we eagerly evaluate it. Currently, evaluation is restricted to arithmetic terms only (we do not perform evaluation on arrays) and consists of three kinds of simplification:

- *Normalisation*: Literals containing inequalities are rewritten in terms of $<$ only, e.g., $t_1 \geq t_2 \rightsquigarrow \neg(t_1 < t_2)$. Subtraction is rewritten in terms of addition of negative numbers and some specialised arithmetic operations, such as rounding functions coming from the TPTP language[4], are rewritten in terms of standard operators.

- *Evaluation*: Purely interpreted subterms are evaluated with respect to the standard model of arithmetic, e.g., $f(1 + 2) \rightsquigarrow f(3)$ and $f(x + 0) \rightsquigarrow f(x)$. Purely interpreted literals are treated similarly, e.g., $1 + 2 < 4 \rightsquigarrow true$.

- *Balancing*: Literals containing a single uninterpreted term or a variable may be 'balanced' by bringing interpreted constants to one side to allow further evaluations, e.g. $4 = 2 \times (x + 1) \rightsquigarrow (4 \text{ div } 2) - 1 = x \rightsquigarrow x = 1$ and $-2 \times a > 2 \rightsquigarrow a < 1$. Care is taken to avoid division by zero and integer division is only applied when there is no non-trivial remainder.

[4] A standard input language for first-order theorem provers [36].

Theory Axioms. In VAMPIRE, the signature of the input problem is analysed to select which theory axioms to add. A theory axiom is a statement true in some relevant theory. The theory axioms VAMPIRE might add for the theory of arithmetic are given in Figure 2.[5] It is important to note that we only need to consider $<$ because, as described above, VAMPIRE normalises input formulas by rewriting all inequalities in terms of $<$. VAMPIRE also adds a finite axiomatisation of arrays [20] if they appear in the signature, and relevant axioms for term algebras (datatypes) [21].

One issue with reasoning with theory axioms is that they can be explosive, i.e., they combine with each other, making the search space grow quickly. Recent work in VAMPIRE has explored heuristic techniques for restricting this effect [28]. So far, VAMPIRE has not made any separate efforts to handle particularly harmful associativity and commutativity axioms, as has been done in other solvers [35].

Lastly, the term ordering gives high precedence to uninterpreted operations and totally orders interpreted constants. The result is that the uninterpreted parts of clauses tend to be targeted by the superposition calculus before the interpreted part.

3.2 Unification with Abstraction

A recently implemented feature [32] in VAMPIRE is a specialised unification that handles interpreted terms separately from uninterpreted terms.[6] The general idea is to avoid having to perform *abstraction* eagerly. Take, for example, the clauses

$$r(14y) \qquad \neg r(x^2 + 49) \vee p(x),$$

which cannot unify without first being abstracted to

$$r(u) \vee u \not\simeq 14y \qquad \neg r(v) \vee v \not\simeq x^2 + 49 \vee p(x)$$

allowing resolution to produce

$$u \not\simeq 14y \vee u \not\simeq x^2 + 49 \vee p(x).$$

The unification with abstraction approach will allow the first two clauses to unify directly under the constraint $14y \neq x^2 + 49$. This directly captures the condition under which the terms can unify and is something to which theory reasoning can be directly applied. We note that this rule directly resolves the issue of losing proofs via eager evaluation, i.e. where $p(1 + 3)$ is eagerly evaluated to $p(4)$, missing the chance to resolve with $\neg p(x + 3)$.

This feature is implemented by extending the substitution tree indexing structures used in VAMPIRE to perform unification.

3.3 AVATAR modulo Theories

A key technique in VAMPIRE is the use of AVATAR to control clause splitting. A full description of how AVATAR works is beyond the scope of this paper, but it is also not essential for this discussion and has been given elsewhere [38, 29]. Instead, we imagine VAMPIRE with AVATAR as an SMT solver with a saturation-based theory solver for the theory of quantified formulas, as illustrated in Figure 3.

[5] We have ignored some less standard operators such as floor and ceiling operators. We also ignore integer division for simplicity of exposition.

[6] Note that the experimental results described previously do not contain this new feature.

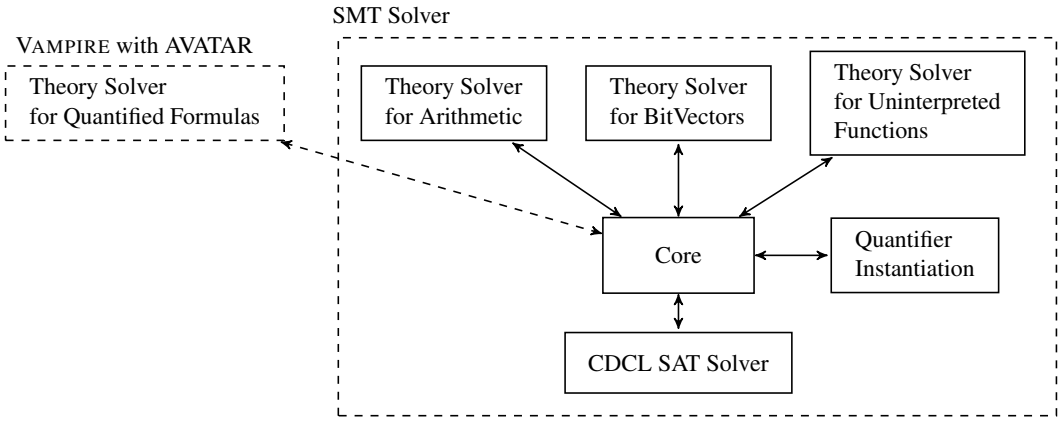

Figure 3: Imagining VAMPIRE with AVATAR as a theory solver.

AVATAR inside VAMPIRE**.** The previously described given-clause algorithm is extended by using a SAT solver to iteratively select sub-problems of the input problem to find inconsistencies within. To do this, the initial clause set, and all subsequent clauses, are propositionally abstracted and the propositional abstraction is given to a SAT solver. This is similar to how the SAT solver is utilised in SMT solvers but is applied to the full problem, not just to the ground part. However, to preserve soundness it is necessary to only break clauses down into variable-disjoint sub-clauses. For example, the following clause

$$\underbrace{s(x) \vee \neg r(x,z)}_{share\ x\ and\ z} \vee \underbrace{\neg q(w)}_{is\ disjoint}$$

is split into two sub-clauses, each propositionally abstracted. If a set of sub-clauses with propositional names $a_1, \ldots, a_n$ are shown to be inconsistent then the propositional clause $\neg a_1 \vee \ldots \vee \neg a_n$ is *learned* (passed to the SAT solver) and a new sub-problem is selected by rerunning the SAT solver.

Replacing SAT by SMT. The relevance to theory reasoning is when the SAT solver is replaced by an SMT solver [27]. The only difference is that where previously sub-clauses were translated as propositional symbols, now only non-ground subclauses are so translated whilst ground (necessarily unit) sub-clauses are mapped directly as themselves. In this case, the subproblem being explored by the first-order saturation-based proof search is guaranteed to be theory-consistent in the ground part.

VAMPIRE **as a Theory Solver.** Conversely, the above can be viewed as adding VAMPIRE as a theory solver for quantified formulas which is invoked when a theory-consistent model for the abstraction is constructed. It is still necessary to propositionally abstract the quantified clauses in the SAT solver, otherwise the clause-splitting advantage of AVATAR would be lost. Indeed, this is one of the main differences between this approach and the work of DPLL(Γ) [12], which also aimed to combine a superposition prover with an SMT solver. Another difference, is that DPLL(Γ) tightly interleaves superposition and SMT steps, resulting in a less flexible method of changing the current sub-problem.

Towards a Tighter Integration? Whilst there appears to be a correspondence between the architecture of an SMT solver and that of AVATAR, we have not explored this further. Indeed, within AVATAR the SAT or SMT solver is used as a black box and, so far, there has been no effort to move towards a tighter integration.

3.4 Other Approaches Taken by ATP Systems

There are some theory reasoning approaches taken by other systems similar to VAMPIRE that are not implemented in VAMPIRE. Perhaps the most prominent of these is Hierarchic Superposition (HS) [2], a generalisation of the superposition calculus for black-box style theory reasoning. This approach separates the theory and non-theory part of the problem and introduces a conceptual hierarchy between uninterpreted reasoning (with the calculus) and theory reasoning (delegated to a theory solver) by making pure theory terms smaller than everything else. HS guarantees refutational completeness under certain conditions including *sufficient completeness*, which essentially requires that every ground non-theory term of theory sort should be provably equal to a pure theory term. This is rather restrictive, for example, the clauses $p(x)$ and $\neg p(f(c))$ cannot be resolved if the return sort of function f is a theory sort as the required substitution is not *simple* [9]. The strategy of *weak abstractions* introduced by Baumgartner and Waldmann [9] partially addresses the downsides of the original approach. One successful implementation of this is in the Beagle [8] theorem prover.

Another prominent approach is to extend the calculus with specialised inference rules. This is what we do for reasoning with terms of boolean sort [20], datatypes [21], and the extensionality rule for arrays [15]. Other work has explored alternative inference rules. For example, the work of SPASS+T [26] introduces the following integer ordering expansion rule

$$\frac{C \vee s \leq t}{C \vee s = t \vee s \leq t - 1 \qquad C \vee s = t \vee s + 1 \leq t}$$

aimed at encoding the above integer ordering axioms, thus avoiding their explosive combination. Another practical work that extends the superposition calculus with additional rules is the Zipperposition tool [11] which adds extra inference rules to deal with integer arithmetic.

4 Instantiation: the Same but Different

This section focuses on the issue of instantiation (missing from the previous section). We separate this issue as it is the key method by which SMT solvers handle quantified formulas and we want to understand if these ideas can help saturation-based solvers.

4.1 Does VAMPIRE Need Instantiation?

To see why VAMPIRE can benefit from instantiation, consider the first-order clause

$$14x \not\simeq x^2 + 49 \vee p(x)$$

for which there is a single integer value for x that makes the first literal false with respect to the underlying theory of arithmetic, namely $x = 7$. However, if we apply standard superposition rules to the original clause and a sufficiently rich axiomatisation of arithmetic (e.g. the one we gave previously), we will most likely end up with a very large number of logical consequences and never generate $p(7)$, or run out of space before generating it. Indeed, VAMPIRE cannot find a refutation of $14x \not\simeq x^2 + 49$ in reasonable time using the techniques described in the previous section.

4.2 Instantiation for SMT Solvers

Here we briefly overview the main instantiation techniques used in SMT solving. This is largely taken from a recent review of the topic [33] and this text should be referenced for full details.

The Quantifier Instantiation Module. Modern SMT solvers based on the CDCL(T) architecture (Figure 3) separate the input into ground and non-ground parts. To handle non-ground parts they first establish a satisfiable ground model and then ask the quantifier instantiation module to suggest instances of the non-ground formulas. Below we outline the three main approaches to quantifier instantiation. They all follow a similar pattern: to take $\forall \mathbf{x} P(\mathbf{x})$ and find some terms $\mathbf{t}$ to produce an instance $P(\mathbf{t})$.

E-Matching. The general idea is to use *patterns* to match against existing terms to produce possible instances. Pattern selection is a complex topic [23], but a simple choice is to use sub-terms of the formula being instantiated. For example, given the non-ground formula $P(x) \vee R(x)$ the sub-term $P(x)$ could be used as a pattern to match against an existing ground term $P(a)$ to produce instance $P(a) \vee R(a)$. This typically makes use of the congruence closure data structures (E-graph) available for the theory of uninterpreted functions, so if this implied that $a = f(b)$ then the instance $P(f(b)) \vee R(f(b))$ could also be created in the above example. A key point is that patterns are usually restricted to applications of uninterpreted functions and instantiation is with existing terms only.

Conflict-Based Instantiation. An issue with E-matching is that it is unguided, it is not clear whether the created instances are helpful. In conflict-based instantiation the idea is to produce instances that are in conflict with the current model, forcing it to change. Such instances are a special case of those found by E-matching and the difficulty is not in checking if an instance is conflicting, but in finding conflicting instances efficiently [34]. Finding conflicting instances in the presence of interpreted symbols is additionally challenging due to the need to check entailment modulo the relevant theory.

Model-Based Instantiation. The above two approaches are incomplete, i.e., the failure to find a (conflicting) instance does not imply that formula in question is satisfiable. In model-based instantiation [14], the general idea is to construct a candidate model $\mathcal{I}$, interpreting all uninterpreted function symbols, and to check that it is a model of the quantified formulas. To check that a model satisfies a quantified formula φ it is sufficient to check that $\neg\varphi^{\mathcal{I}}$ is unsatisfiable where $\varphi^{\mathcal{I}}$ replaces all function symbols with their interpretations in $\mathcal{I}$. A model for $\neg\varphi^{\mathcal{I}}$ can be used to extract an instance of φ which is in conflict with $\mathcal{I}$. The challenge here is how to construct $\mathcal{I}$. Whilst this approach can detect satisfiability, it is only complete for some restricted fragments (for obvious reasons).

4.3 What can we learn from SMT?

We briefly examine each of the above instantiation approaches and consider whether we could borrow the idea and use it within VAMPIRE, or a similar ATP. Firstly, we note that the focus of instantiation will, in general, be different. In ATP systems, there is a good approach for dealing with quantified formulas containing uninterpreted symbols and it is those containing interpreted symbols that are the challenge. Whilst for SMT solvers instantiation seems to be focussed on the uninterpreted parts, as they typically have reasonable methods for dealing with interpreted symbols, e.g., quantifier elimination [10, 24].

E-Matching. The idea behind this approach is to use existing terms to instantiate quantified formulas. The *Theory Instantiation* inference rule from SPASS+T [26] follows this general scheme. It is given as

$$\frac{C[s] \qquad L[t] \vee D}{(L[t] \vee D)\sigma}$$

where $L[t] \vee D$ is not ground, t is uninterpreted and all symbols above t in L are interpreted, $\sigma = \mathsf{mgu}(t, s)$, and $C[s]$ and $(L[t] \vee D)\sigma$ are ground. Prevosto and Waldmann [26] point out that $(L[t] \vee D)\sigma$

is typically immediately subsumed by $L[t] \vee D$. To handle this, they pass $(L[t] \vee D)\sigma$ directly to their SMT solver and, to allow it to participate in further instantiations, they replace the above rule by the two rules

$$\frac{C[s]}{\mathsf{ground}(s)} \qquad \frac{\mathsf{ground}(s) \quad L[t] \vee D}{\mathsf{ground}(u_1) \ldots \mathsf{ground}(u_n) \quad \mathbf{export}((L[t] \vee D)\sigma)}$$

with similar restrictions to the above, but in addition s is uninterpreted and $u_1, \ldots, u_n$ are the uninterpreted ground terms in $(L[t] \vee D)\sigma$ different from s. The general idea is to use the special predicate ground to store the set of ground terms that are removed by subsumption.

We would like to reiterate the above point about subsumption. Instantiating the clause $x > c \vee x < c$ to produce $0 > c \vee 0 < c$ is unhelpful as the new clause is immediately subsumed and deleted. However, if one of the literals in $(L[t] \vee D)\sigma$ becomes *false* the subsumption no longer holds. For example, instantiating $x > y \vee p(x, y)$ with $\{x \mapsto 0, y \mapsto 1\}$ produces $p(0, 1)$, which is not subsumed.

This leads us to the observation that the instances useful to saturation-based proof search are those that remove literals, which in the limit will lead to the empty clause. In [32] we introduce a theory instantiation rule that explicitly produces instances that remove literals. The general idea is as follows. Given a clause $C[\mathbf{x}] \vee D[\mathbf{x}]$ where $D[\mathbf{x}]$ consists purely of interpreted symbols, a solution σ such that $(\neg D[\mathbf{x}])\sigma$ is T-valid (true in every model of the relevant theory) can be used to produce the instance $C[\mathbf{x}]\sigma$. The problem of finding such a solution is something that an SMT solver handles well.

Conflict-Based Instantiation. In typical saturation-based proof search there is no concept of a model to be in conflict with. But, when AVATAR is being used for clause splitting, there is now a model of the current splitting branch. If we were to take the view that VAMPIRE becomes an extra theory module via AVATAR, then we could directly apply conflict-based instantiation to produce instances inconsistent with the current splitting branch. Conversely, attempting to do this whilst using an SMT solver as a black-box does not seem workable.

If this approach were in place, then the new question would become whether it is better to spend effort finding conflicting instances to refute a splitting branch, or using saturation-based reasoning. The above combination and this question remain further research. Our conjecture is that the two approaches would be complementary and a heuristic combination would be beneficial.

Model-Based Instantiation. Currently, if VAMPIRE saturates a particular splitting branch it is not clear if this is due to the incompleteness of its proof search or the consistency of that branch. The model-based instantiation approach could provide a technique for (i) returning satisfiable in such a circumstance, or (ii) otherwise refuting the current splitting branch and continuing with proof search. Again, the question would then be whether the combination with saturation-based proof search contributes to this general approach.

5 Conclusion

We have briefly compared how VAMPIRE performs theory reasoning to the standard architecture of modern SMT solvers and discussed the topic of instantiation. It is clear that the integration of saturation provers with SMT solvers can be further explored, especially with respect to instantiation. If we view AVATAR as a method for adding VAMPIRE as a theory solver to an SMT solver then an important open question remains: when should VAMPIRE be called, and when should existing quantifier instantiation techniques be used instead. Recent work [3] has aimed to unify notions of instantiation for SMT solvers and this work may be relevant to our efforts. Writing this brief overview has forced us to ask some question which, we hope, will now lead to further research on this integration.

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] H. Barbosa, P. Fontaine, and A. Reynolds. Congruence closure with free variables. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, pp. 214–230, 2017.

[4] 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.

[5] C. Barrett, M. Deters, L. M. de Moura, A. Oliveras, and A. Stump. 6 years of SMT-COMP. *J. Autom. Reasoning*, 50(3):243–277, 2013.

[6] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, vol. 4246 of *Lecture Notes in Computer Science*, pp. 512–526. Springer, 2006.

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

[8] P. Baumgartner, J. Bax, and U. Waldmann. Beagle - A hierarchic superposition theorem prover. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pp. 367–377, 2015.

[9] 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.

[10] N. Bjørner. Linear quantifier elimination as an abstract decision procedure. In *Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, pp. 316–330. Springer Berlin Heidelberg, 2010.

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

[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] Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, vol. 5643 of *Lecture Notes in Computer Science*, pp. 306–320. Springer, 2009.

[15] A. Gupta, L. Kovács, B. Kragl, and A. Voronkov. Extensional crisis and proving identity. In *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, pp. 185–200, 2014.

[16] K. Hoder, Z. Khasidashvili, K. Korovin, and A. Voronkov. Preprocessing techniques for first-order clausification. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pp. 44–51, 2012.

[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] K. Hoder and A. Voronkov. The 481 ways to split a clause and deal with propositional variables. In *Automated*

Deduction CADE-24, vol. 7898 of *Lecture Notes in Computer Science*, pp. 450–464. Springer, 2013.

[19] K. Korovin. Inst-Gen – A Modular Approach to Instantiation-Based Automated Reasoning. In *Programming Logics: Essays in Memory of Harald Ganzinger*, pp. 239–270. Springer Berlin Heidelberg, 2013.

[20] 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.

[21] L. Kovács, S. Robillard, and A. Voronkov. Coming to terms with quantified reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pp. 260–270. ACM, 2017.

[22] 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.

[23] K. R. M. Leino and C. Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pp. 361–381, 2016.

[24] D. Monniaux. Quantifier elimination by lazy model enumeration. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pp. 585–599. Springer Berlin Heidelberg, 2010.

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

[26] 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.

[27] G. Reger, N. Bjørner, M. Suda, and A. Voronkov. AVATAR modulo theories. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, vol. 41 of *EPiC Series in Computing*, pp. 39–52. EasyChair, 2016.

[28] G. Reger and M. Suda. Set of support for theory reasoning. In *IWIL-2017*, 2017. To appear.

[29] 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.

[30] G. Reger, M. Suda, and A. Voronkov. Finding finite models in multi-sorted first-order logic. In *Theory and Applications of Satisfiability Testing – SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pp. 323–341. Springer International Publishing, 2016.

[31] G. Reger, M. Suda, and A. Voronkov. New techniques in clausal form generation. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, vol. 41 of *EPiC Series in Computing*, pp. 11–23. EasyChair, 2016.

[32] G. Reger, M. Suda, and A. Voronkov. Unification with abstraction and theory instantiation in saturation-based reasoning. In *IWIL-2017*, 2017. To appear.

[33] A. Reynolds. Conflicts, models and heuristics for quantifier instantiation in SMT. In *Vampire 2016. Proceedings of the 3rd Vampire Workshop*, vol. 44 of *EPiC Series in Computing*, pp. 1–15. EasyChair, 2017.

[34] A. Reynolds, C. Tinelli, and L. de Moura. Finding conflicting instances of quantified formulas in SMT. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pp. 31:195–31:202. FMCAD Inc, 2014.

[35] S. Schulz. System Description: E 1.8. In *Proc. of the 19th LPAR, Stellenbosch*, vol. 8312 of *LNCS*. Springer, 2013.

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

[37] G. Sutcliffe. The 8th IJCAR automated theorem proving system competition - CASC-J8. *AI Commun.*, 29(5):607–619, 2016.

[38] 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.