# Incremental Solving with Vampire*

Giles Reger[1] and Martin Suda[2]

[1] University of Manchester, Manchester, UK
[2] TU Wien, Vienna, Austria

**Abstract**

Both SMT and SAT solvers can be used incrementally. This can be useful in lots of applications. Indeed, internally, Vampire uses both Minisat and Z3 incrementally. In this paper, we explore how VAMPIRE could be used incrementally. There are two forms of incremental solving. The first is where a solver is provided with formulas from a problem one by one with consistency being checked at certain points. The second more general form is where stack operations are used to create different solving contexts. We explore both ideas and show how they can be achieved within VAMPIRE. We argue that the second approach may be more suited to VAMPIRE as it allows for the incremental solving of unsatisfiable problems (whereas the first assumes a series of satisfiable problems) and the use of different solving contexts allows VAMPIRE to make use of incomplete proof search strategies. For the first approach, it will be necessary to restrict preprocessing steps to ensure completeness when additional formulas are added. For the second approach, we can make use of clauses labelled with assertions and take advantage of AVATAR to keep track of the stack information.

## 1 Introduction

In this paper we explore how VAMPIRE [2] can be made to process problems *incrementally*. This is a feature available in SMT solvers and is often utilised in applications such as program analysis where there is a general encoding of a problem that is queried in different ways. As a health warning: this paper describes ideas not implementation i.e. many things discussed have not (yet) been implemented.

Before we continue let us, at a high level, describe what we mean by incremental proving. Clearly it means that a problem is processed in increments i.e. some part is given and processed and then another part. However, this can be presented in two ways. Firstly, one can add new assertions to a growing problem and check the consistency of a growing problem. This has the drawback that as soon as the problem becomes inconsistent there is nothing else that can be done. The more general approach is to also be able to remove parts of the problem. Often this is done via a *stack of assertions* that can be pushed and popped, but in theory there should be nothing preventing the arbitrary removal of previous assertions. Clearly, this second more general case subsumes the first but we separate the two as the first is 'easier' as it does not require any backtracking.

Before we discuss how VAMPIRE can be modified to process problems incrementally we should understand why this is a non-trivial thing to do. Firstly, let us consider why such an approach is usually

*Martin Suda was supported by ERC Starting Grant 2014 SYMCAR 639270 and the Austrian research projects FWF S11403-N23 and S11409-N23.

relatively straightforward for SMT solvers. CDCL(T)-style SMT solvers are model-based i.e. they attempt to build a model of the current problem, which can be thought of as finding a theory-consistent 'branch' of the problem's boolean representation. This is inherently incremental in nature; a model is build incrementally by considering more of the given problem. Therefore, adding more information incrementally is a natural extension of this process. Dealing with backtracking adds complications but still fits relatively nicely into this framework. Now let us contrast this to saturation-based theorem proving as performed in VAMPIRE. Now the task is to search the entire search space for an inconsistency, and only when we have explored the full search space and found no inconsistency can we report consistency. Simply, where model-based SMT solving is designed for satisfiability checking, saturation-based ATPs are designed for validity (unsatisfiability) checking and the incremental setting typically assumes a growing consistent problem. Furthermore, we know that finite saturations may not exist in general and that in many cases saturation does not mean that the input problem is satisfiable i.e. in the presence of theories (which is the typical case in the usages of incremental solving mentioned previously) or when using effective but incomplete strategies. This does not mean that we should give up but we should bear this in mind as we continue and attempt to focus on what we are good at e.g. unsatisfiability checking. We return to this thought later.

The reminder of this paper will be organised as follows:

- In Section 2 we explore the different kinds of incremental settings we may wish to consider;
- Sections 3 to 6 consider these different settings and outline our proposed solutions for them;
- Sections 7 and 8 consider some VAMPIRE-specific issues related to incremental solving.

Section 9 concludes. It should be stressed that this paper is an early look at these ideas and many of the suggestions discussed here are not yet fully implemented or integrated into VAMPIRE.

# 2 The Incremental Setting

Before we discuss necessary changes to proof search we first consider how incremental problems will/could be described to VAMPIRE. We begin by sketching various plausible settings and then discuss how these compare to the reality of SMT-LIB and TPTP. We are implicitly describing an ideal language but stop short of describing a concrete syntax and semantics.

## 2.1 A Problem Ordered by Queries

The first setting we consider is where there is an *order* placed on a problem by the inclusion of multiple *query* commands. A query command tells the solver to check the consistency of the formulas asserted so far. Therefore, query commands introduce an implicit ordering, or another way of looking at it would be to consider a single problem with $n$ queries (or query blocks, see below) to represent $n$ different problems. This setting requires two kinds of statements in the language:

*Addition Statements.* These are statements that add new things to the problem. Typically this would involve declarations or definitions of new sorts and symbols and the assertion of new formulas. Often there is an ordering constraint that symbols must be defined or declared before they are used.

*Query Statements.* These are statements (or more appropriately, commands) that query some property of the problem defined so far. Answering such a query would typically involve performing some reasoning/solving. A query might ask for a simple result, e.g. `check-sat` would ask if the problem is satisfiable, or it may ask to extract something, e.g. `get-proof` which would require reasoning to

produce something. This second kind of query may be dependent on reasoning producing a certain result; in this case, if the prover fails to produce the required result, the query would need to return a relevant null value. One could imagine a series of queries grouped together in a single *block* requiring a single reasoning process and then extracting multiple pieces of information from this reasoning process. Whether a solver recognises this and does the clever thing is separate from what the queries mean.

One issue to consider is whether progress is allowed if a query *fails* (for some notion of a query failing). The easiest assumption to make is that queries have no impact on the rest of the problem i.e. some output is given to the user and they can use it as they wish. Of course, it is worth noting at this point that we are describing incremental problems as if they are always static objects. In reality, incremental problems are most likely created interactively where the later part of a problem is dependent on the result of some earlier queries. However, this fact does not mean that the problem format need to be aware of this.

## 2.2 Conjectures as Queries

VAMPIRE and other first-order theorem provers often make use of this information to guide proof search, therefore we would like a language that allowed identification of conjectures or goals.

The previous setting can be extended to view conjectures as special queries that ask for the result of the query after adding the given formula to the problem seen so far. Importantly, once that query has been answered the given formula is not considered part of the problem. As before, one might group conjecture statements together such that the efficient solver would be able to perform a single reasoning process to answer them.

Multiple conjectures in a row with different formulas can be considered *unordered* as they are mutually exclusive. This could be an important property for solvers. For example, consider the axiom

$$a > 0 \quad \wedge \quad b > 0 \quad \wedge \quad c > 0$$

with the two conjectures

$$a^3 + b^3 \neq c^3 \qquad and \qquad a \geq b \wedge a \geq c$$

If the two conjectures were mutually exclusive, we could tackle them in either order. As one is significantly easier than the other (the first is an instance of Fermat's Last Theorem), forcing an arbitrary order on them would seem counterproductive.

However, care should be taken when potentially *reordering* queries. The language would need to provide a mechanism for forcing an ordering if this were required by the context.

## 2.3 Supporting Backtracking with a Stack

Adding conjectures as special queries always represents a limited form of backtracking as a conjectured formula is added and then removed. But we would want to support a more sophisticated form of backtracking. The standard approach here is to use *stack* to organise the backtracking process.

The idea is that there is a conceptual stack that the addition statements (assertions etc) are added to. To be more precise, such statements are added to the current *stack frame*. This stack frame begins at level one and then `push` and `pop` commands increase and decrease the stack level, either producing a new stack frame or removing the current one. Removing a stack frame (via `pop`) has the effect of undoing all additions in it, including symbol declarations (although one could add a mode to make these global across stack frames so that symbol declarations can easily be reused).

This setting can be straightforwardly combined with the above setting of problems ordered by queries. We now have a new notion of mutually exclusive parts of a problem when we have a `pop`

followed by a `push` as the stack frames before and after this will have the same previous context but will be unrelated (unless a notion of global symbol declarations is being used). This is another case where care might be needed if the solver starts reordering stack frames.

## 2.4 Generalising the Stack

The stack-based approach described previously is a nice model that most likely suits most settings. But it is not the most general model. One may wish to remove an arbitrary formula and then later add the same formula back, making it clear that this is the same formula.

The general approach would be to explicitly *label* assertions with *contexts* and then activate and deactivate contexts for each query. The labelling may involve opening and closing a particular context or adding an explicit context to a formula. For maximal generality, one would want contexts to nest, so that they can be grouped together.

This setting could easily emulate the stack-based setting by each push being the start of a new context and a pop being the end of the most recently created surviving context.

## 2.5 Relation to TPTP and SMT-LIB

The above describes what we would like. Here we describe how far what we have is away from this.

**TPTP.** This language [4] has no notion of incremental problems. There is a notion of multiple conjectures but the semantics is that all conjectures should hold i.e. they are taken in conjunction, whereas in our current setting the assumption is that each conjecture is a separate query. Adding the extra commands described above would be quite straightforward, and we use this in some examples later in this paper. One could straightforwardly add `push` and `pop` as commands, use the current `conjecture` label as a shortcut for a `check-validity` query with a conjecture formula, and introduce a family of additional queries in the standard Prolog-like language. One could also introduce special commands for opening and closing contexts.

**SMT-LIB.** This language [1] already supports incremental problems. The separation of statements is somewhat different. They have the addition statements as described above but then they have a `check-sat` command that runs the solver and a set of separate queries, such as `get-model`. For these queries to be used they use the notion of an *execution mode* to determine when certain commands are allowed. Assertions etc can be made in an *assert mode* and a `check-sat` command will transfer to either a *sat mode* or *unsat mode* where certain queries are allowed (e.g. `get-model` is only allowed in *sat mode*). Adding further assertions or calling `push`/`pop` then returns the solver to the *assert mode*.

SMT-LIB also uses an assertion stack exactly as described above. However, it does not have a concept of a *goal* or *conjecture*. One can emulate this to a certain extent using the assertion stack but the conjectured formula would still be stated (negatively) as an assertion and the solver would not know it is a goal to be potentially treated differently in proof search.

## 2.6 When Queries Should be Answered

The above few sections have discussed how we might present incremental problems so that the different problems that they contain are clearly defined, whilst separating this notion from how a solver might be expected to handle such incremental problems. In SMT-LIB the assumption is that a `check-sat` command is answered before the rest of the problem is processed. This fits well with an 'online' usage of a solver as one might expect the following commands to be dependent on the result of the query.

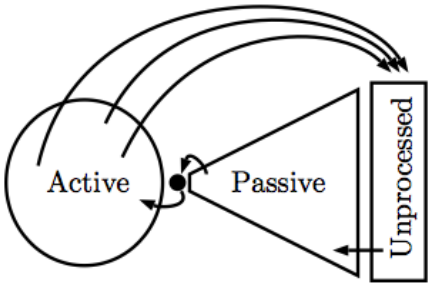


Figure 1: The saturation looop.

However, we have (so far) not placed any restrictions on when queries should be answered and have hinted that we want to allow solvers to address multiple conjectures concurrently.

The question is therefore, how to incorporate the notion of online interaction in a static problem description such that it does not matter whether the problem is being received online or viewed statically. Our proposal is to introduce *synchronous* and *asynchronous* versions of each query and an explicit `wait` command that indicates that the problem should not be read further until all queries have been answered. Note that throughout this paper there is an implicit assumption that single conjectures are synchronous but blocks of conjectures are asynchronous followed by a single `wait` command.

# 3 Handling an Ordered Problem Without Conjectures

In this section we consider the first setting describe above i.e. where a problem is ordered by queries. For the sake of simplicity, let us assume that all queries are of the form `check-sat`. The first observation is that if a query ever returns the result that the problem is unsatisfiable then all future queries will return the same result. This means that in this setting proof search should stop on the first refutation. Below we describe a natural way to handle such a setting. We ignore the option of starting again whenever we see a new query as we want proof search to be incremental.

Handling a problem of the given form can be viewed as a quite natural extension of the standard *given-clause-algorithm saturation-loop* as illustrated in Figure 1. This loop involves three sets of clauses and on each step (i) all clauses in *unprocessed* that pass a *retention test* are placed in *passive*, and (ii) a clause is selected from *passive* and all inferences between it and all clauses in *active* are performed with the result being placed in *unprocessed*. If *passive* becomes empty then the clauses are *saturated* which, under certain conditions, means that they must be consistent. The reason that this setting can be viewed as an extension is that we can view the process as restricting clause selection such that it only selects from those clauses that have already arrived.

Practically, this approach would consist of the following steps:

1. Receive new formulas up to a `check-sat` query
2. Preprocess formulas and add resulting clauses to *unprocessed*
3. Do solving until passive is empty or an inconsistency is found
4. Report result of query
5. If passive is empty goto 1 else terminate

There would be no reason to not start preprocessing and reasoning with formulas before a `check-sat` query is given, but then one should check that a query has been given before reporting a result (i.e. if passive becomes empty before a `check-sat` query is given).

Another variation would be to restrict the number of steps taken on step 3 i.e. to allow one to 'give up' on a query. This could be encapsulated in a special query that gave a limit on steps, or a time limit, for that query. This would be particularly important in our setting where a clause set may not have a finite saturation.

It should be noted at this stage that for this approach to be effective the proof search would need to be *complete* so that a saturation actually implied satisfiability. However, this does exclude the possibility of using incomplete heuristics during proof search. How to avoid incompleteness and the wider impact of incremental proof search on heuristics is discussed in Section 8.

However, this restriction (to a single complete proof search strategy) can be viewed as overly restrictive. One of the underlying principles of VAMPIRE development is that it is usually better to run many short complementary heuristically-guided strategies than one single long-running strategy. The question is then how to do this in the incremental setting.

The proposed solution would be to periodically *fork* new short-lived proof search attempts utilising heuristic strategies. Such forked processes could run in parallel to the main process and would (from our experience) have a better chance of finding inconsistencies, although it would not be necessary to run them in parallel as the main reason to fork a process is to create a (copy-on-write) copy of the current memory space that can be updated by proof search and then discarded. VAMPIRE already uses this method to implement its portfolio mode – the given problem is parsed and then a process forked for each proof search strategy in the portfolio.

# 4 Handling an Ordered Problem With Conjectures

We now consider the case when a problem contains conjecture queries. Such queries must be considered independently i.e. at least conceptually as different proof search attempts. Here we discuss two ways in which this could be achieved.

## 4.1 The Forking Solution Again

Previously we mentioned that processes could be forked to apply separate strategies to the same growing problem. The same idea can be applied here directly. On each conjecture, one or more processes would be forked to attempt to answer this conjecture query. This framework would also support the idea discussed earlier that a number of conjectures grouped together may be attempted concurrently as they are mutually exclusive. This approach keeps the previous advantage that multiple heuristic strategies can be used, but has one main disadvantage which is that there is no sharing of reasoning progress as forked processes do not learn what about what happened in other processes. This can be handled by the next proposed solution.

## 4.2 Labelled Conjectures.

The idea here is to use a single proof search to reason about multiple conjectures. Here there are two settings (which are not mutually exclusive):

**Setting A**. The incremental problem with occasional queries. In this setting a number of formulas are asserted and then a conjecture is made about the formulas inserted so far and then this is repeated i.e. with extra formulas and a new conjecture.

**Setting B**. The fixed problem with lots of queries. In this setting the problem is (mostly) described in one go but then there are a lot of conjectures made about this problem.

We can also see a mixture of both settings but we separate them as they relate to slightly different parts of this proposed solution. In both cases we want to (i) be able to work on multiple conjectures at once, sharing shared parts of the proof search, whilst (ii) be able to report the results of each conjecture query separately with separate proof objects. A further goal is to preserve the goal-directed proof-search heuristics that VAMPIRE may use.

The proposed solution is to work with *labelled clauses* and introduce a label for each conjecture. In this particular case, we will restrict clauses to having no label or one label. Inferences will preserve labels and be restricted by them: if parents disagree on labels then the inference is blocked, and children inherent the labels of the parents. Similarly, labelled clauses may only reduce clauses with the same label and may only conditionally reduce clauses without a label. These restrictions mean that two conjecture clauses and their children will never mix during proof search. In this setting, when an empty clause is derived then either it has a conjecture label and a proof for this conjecture can be reconstructed and reported, or it has no conjecture label and the general problem has been shown to be inconsistent. Figure 2 gives a very simplistic example of how the proof search for two conjectures may share common reasoning steps.

By separating goal and non-goal clauses, this proposed solution also directly handles the issue that goal-directed proof search may treat goal clauses and their children specially. However, there will still be some 'silent' interaction. For example, VAMPIRE may prioritise goal clauses in clause selection but when there are multiple goal clauses from different conjectures we would need to consider whether we want to order them in any way.

Now there is a question as to what should happen when a conjecture is proved. In this case, all clauses labelled with that conjecture's label may be safely removed from the search space. This would be important as we would not want to waste any effort or resources on a conjecture that has already been solved. We now consider the two settings described above.

**Setting A.** In this setting we might have a long-running proof search and may generate many irrelevant clauses in the search for a proof of a given conjecture. It may be fruitful, in this setting, to consider *learning* which parts of the search space were most useful when solving earlier conjectures. One approach would be to simply remove from active any clauses not used in earlier proofs. This would be incomplete and to regain completeness one would need to track and restore any clauses that were reduced by these removed clauses during earlier proof search. Other approaches may be considered for learning from previous proof attempts. Indeed, this is an active area of research by others.

**Setting B.** In this setting we need to consider how to organise multiple proof search attempts. At the most basic level we should ask how many conjectures should be considered concurrently. It could be that some conjectures have very short proofs and are handled quickly, whilst others do not and consume much of the proving time. One approach would be to introduce a queue of conjectures waiting to be solved and to vary the *weight boost* given to goal clauses over time so that goals are given priority early on in their proof search but less so over time.

So far we have assumed that conjectures are mutually exclusive. However, we may have a setting where a conjecture is reused later. For example, we may prove a conjecture $C$ and then be asked to prove that $C \rightarrow D$. We do not propose how we could handle such scenarios but leave this point here.

```
fof(p,axiom,![X] : p(X)).}
fof(q,axiom,![Y] : ~p(Y) | q(Y)).}
fof(a,conjecture,q(a)).
fof(a,conjecture,![Z]: q(Z)).
```

| | | | |
|---|---|---|---|
| 1. | | $\rightarrow p(X)$ | |
| 2. | | $\rightarrow \neg p(Y) \vee q(Y)$ | |
| 3. | 1 | $\rightarrow \neg q(a)$ | |
| 4. | 2 | $\rightarrow \neg q(sk)$ | |
| 5. | | $\rightarrow q(Y)$ | $(1,2)$ |
| 6. | 1 | $\rightarrow \neg p(a)$ | $(2,3)$ |
| 7. | 1 | $\rightarrow \bot$ | $(3,5)$ |
| 8. | 1 | $\rightarrow \neg p(sk)$ | $(2,4)$ |
| 9. | 2 | $\rightarrow \bot$ | $(4,5)$ |

Figure 2: A brief illustration of how the labelled clause solution would work for multiple conjectures.

# 5 Handling a Problem with a Stack

We now consider how VAMPIRE could be extended to the setting where a stack of assertion levels is used to organise the incremental problem.

## 5.1 The Forking Solution Again

Again, we can use forked processes to solve this issue. This time we would fork a new process every time the stack is `push`ed and terminate this process on a `pop`. Each parent would wait for its child and there would be at most one child 'active' at any time and this active child would handle proof search, possibly with multiple conjectures, using the techniques described above. The disadvantage exists; reasoning from one stack frame would not be visible after that stack frame is popped.

## 5.2 Labelled Stack Frames

Again we turn to labelled clauses to separate reasoning from different stack frames. As before, each stack frame is given its own label and clauses are initially labelled with the frame at which they are introduced. But here, clauses with different labels are allowed to mix arbitrarily (again, labels are preserved i.e. the labels of a derived clause are the union of the labels of its parents). And again, reductions must be modulo the labels i.e. clauses can only reduce clauses at the same level of below, or reductions may be conditional and potentially backtracked later.

Figure 3 illustrates the potential advantage of this approach. On the left of the example is a small incremental problem and on the right is the sequence of labelled clauses that might be derived. In this example, a clause derived whilst establishing a conjecture in the inner solving context (the unit clause $q(Y)$) is then used to establish the conjecture later on i.e. we reused some of the earlier proof search.

When a stack frame is popped, all clauses labelled by this stack frame should be removed from proof search as there is no notion of revisiting a stack frame (see the next section). This may also mean that some reductions should be backtracked.

# 6 Handling a Problem with Contexts

Finally, we consider the case where instead of a global stack of sets of asserted formulas, each formula is labelled by a context and these contexts can be activated and deactivated. The word 'labelled' in the previous should immediately suggest our proposed solution in this case. Again, we propose the use of

```
fof(p,axiom,![X] : p(X)).}
fof(q,axiom,![Y] : ~p(Y) | q(Y)).}
push().
fof(a,conjecture,q(a)).
pop().
fof(a,conjecture,![Z]: q(Z)).
```

1. $0 \rightarrow p(X)$
2. $0 \rightarrow \neg p(Y) \vee q(Y)$
3. $1 \rightarrow \neg q(a)$
4. $0 \rightarrow q(Y)$ $(1,2)$
5. $0 \wedge 1 \rightarrow \neg p(a)$ $(2,3)$
6. $0 \wedge 1 \rightarrow \bot$ $(3,4)$
7. $0 \rightarrow \neg q(sk)$
8. $0 \rightarrow \neg p(sk)$ $(2,7)$
9. $0 \rightarrow \bot$ $(4,7)$

Figure 3: A brief illustration of how the labelled clause solution would work for stack frames.

labelled clauses (in the vein of the previous section). Notice that this scenario can be viewed as the stack-based one where stack frames may be revisited at any time.

There would be various ways to activate and deactivate contexts. If the labelled clauses were handled by the AVATAR architecture (see below) then this would be a simple case of asking the SAT solver to produce a model under a certain set of assumptions. Otherwise, it would involve prioritising the active labels in clause selection and only reporting refutations belonging to the current activated label set.

# 7 Labelled Clauses and AVATAR

We have discussed various utilisations of labelled clauses above. VAMPIRE already has a mechanism, called AVATAR [5, 3] for handling labelled clauses using a SAT solver to organise which labels take part in the current proof search. Any new approach making use of labelled clauses can also make use of this architecture. However, it would be necessary to clearly separate the different kinds of labels as they have distinct semantics.

The main difference between the labels used here (for conjectures and stack frames) and the labels used in AVATAR is that in AVATAR the assumption is that a label may be 'revisited' often, whilst we would not revisit a conjecture or stack frame. This means that there will be some point where all clauses with a particular label become unreachable and should be removed. Currently, removing clauses related to a single label may be achieved by asserting that label to be false (then the SAT solver will never add these clauses to proof search). However, it may be preferable to remove these clauses from the SAT solver also (if the search space gets large) and this may present some technical challenges.

Another point to consider is that a clause belonging to some labelled proof search may be added with a different label at a later time (i.e. a syntactically identical clause). In this case we could detect this and, instead of adding a new clause, reactive the old clause along with relevant children. This is an idea we have explored within the context of AVATAR (to restore relevant children).

## 7.1 An Experimental Implementation

An experimental implementation of the use of labelled clauses to handle an assertion stack exists but, at time of writing, has not been made public (please contact the first author for details). This implementation introduces a new `--mode incremental` which currently accepts SMT-LIB with multiple `(check-sat)` commands and matching `(push 1)` and `(pop 1)` commands. However, it currently requires the full signature to exist before first `(check-sat)` otherwise the behaviour of VAMPIRE will be undefined and it will most likely crash. As pointed out below, there are still cases where VAMPIRE may not handle some formulas incrementally i.e. if they force the signature to be extended.

On each `(check-sat)` command the new formulas are preprocessed, the resulting clauses are labelled and added to *unprocessed*, and the saturation loop is run until it terminates. If a refutation is found then all `(check-sat)` commands on all affected stack frames will report `unsat` i.e. to make progress the affected stack frames should be popped (and VAMPIRE will ignore any assertions to those stack frames).

To create labels we introduce new propositions, add these to each clauses (i.e. $C$ becomes $\neg p \vee C$, and register each proposition with AVATAR to get a handle on the associated SAT variable. Notice that we are cheating and making use of AVATAR's usage for clause splitting here as a proposition will always be associated with a unique SAT variable. We then ensure that AVATAR is always solving under the assumption of the SAT variables corresponding to the 'active' stack frames (and the assumption that all other stack frames are inactive).

# 8 Pragmatic Issues

Here we briefly consider some pragmatic issues that need to be considered when conducting incremental proof search in a saturation-based solver such as VAMPIRE.

## 8.1 Completeness

Above we mention that completeness should be enforced in certain scenarios. There are two ways in which we typically break completeness for heuristic advantage: blocking inferences (e.g. through incomplete literal selection) and removing clauses from the search space (e.g. by using the limited resource strategy). Another point where clauses (or literals) may be removed is in preprocessing. In general, we would want to avoid *set of support* or *SiNE selection* as both heuristically remove clauses from the input (also note that the information they need to do this effectively is unlikely to be available in the incremental setting). Similarly, preprocessing steps such as *pure literal removal* and *function definition elimination* should be avoided as they, respectively, depend on knowing that a predicate symbol is used with a single polarity or expending all occurrences of a single function symbol in the *whole problem*.

It is worth pointing out at this stage that if we have theories (interpreted symbols) in the input then VAMPIRE cannot be complete. This presents an issue, as most existing incremental benchmarks, and many settings where we may wish to use an incremental mode, make use of theories.

## 8.2 Let's Talk about the Signature

We have just observed that some preprocessing steps should not be performed as they rely on knowing the full problem. Here we discuss a different issue: not knowing the full signature. VAMPIRE makes decisions based on the signature when initialising proof search, but we do not know the full signature when we start solving. In some cases this can be handled quite straightforwardly, in others it is less easy to see what to do. Below we highlight the majority of cases where the issue needs handling:

- Inference rules are selected based on what is needed from the signature. For example, VAMPIRE will not make use of superposition if there is no equality in the problem. Given that we do not know the full signature upfront, this could lead to problems. An easy fix would be to enable all inference rules, which could lead to some extra overhead. A better solution would be to dynamically expand the set of inference rules being considered.

- VAMPIRE uses a term ordering (KBO) as part of the ordered resolution and superposition calculus. This term ordering relies on a symbol precedence but in the incremental setting new symbols can

appear partway through proof search. Theoretically, this should not matter although it would be necessary to place new symbols at the end of the precedence (to avoid reordering) and this could be suboptimal for certain reasons. As a small practical point, VAMPIRE stores term ordering information using an array of symbols which would now need to be expanded during proof search.

- Some parts of the code are specialised for certain contexts. For example, VAMPIRE uses a special term ordering when in the EPR fragment. Given that we cannot know if we are going to continue to be in this fragment when we receive new formulas, we cannot use such specialisations in incremental mode. One workaround would be to declare that formulas will take a certain form upfront and then raise an error if this is violated.
- Many data structures assume a fixed signature. For example, discrimination indexing trees index directly on the signature at the top-level. In most cases this will not be problematic to replace with expanding data structures, but these modifications will need to be done and will probably incur overhead during runtime.
- Certain symbols are treated specially during proof search. For example, we separate those symbols coming from the problem from those introduced in preprocessing by numbering symbols sequentially and remembering the index of the first symbol introduced in preprocessing. Given the new interleaved approach, such simple schemes will need to be made more complicated. This is just one example of where existing assumptions in the code are broken and will need addressing.

These issues can be partially avoided if the full signature is declared before the first call to the solver. However, this is likely to be too restrictive for many contexts. We say partially as preprocessing may add symbols to the signature (e.g. Skolem functions and names for structural transformation and splitting) and preprocessing of new formulas occurs throughout. This means that, unless naming is not performed and no new Skolem functions are required, the signature will definitely be extended during proof search, even if the full signature is declared at the beginning.

# 9 Conclusion

This paper has considered the problem of incremental solving with VAMPIRE and discussed various possible solutions. Throughout there were two main solutions presented: forking new processes and using labelled clauses. At this stage it is worth highlighting their different (dis)advantages. Simply, forking new processes allows one to try multiple different strategies, but it does not allow cooperation between different proof search attempts, which is supported by labelled clauses. One could imagine a hybrid approach where labelled clauses were used but forked processes were used at points to gain the 'portfolio mode' advantage.

Very little has yet been explored practically and the priority research areas to focus on next are:

- Currently, not being able to handle the signature being declared incrementally is a large barrier to the possible applications of these ideas. As describe above, most of the issues are engineering problems that should be straightforward to solve.
- Whilst we have implemented an experimental version of the labelled clause approach for stacks, it is likely that the forking approach would be more successful in many cases due to the ability to employ a portfolio of strategies. This approach should be implemented so that it can be compared. We should also consider possible combinations of the two approaches. Orthogonally, we should implement the labelled clause approach for multiple conjectures. In both cases of the labelled clauses, we should consider whether it is appropriate to integrate these into AVATAR as first-class labels.

- So far, only proof-of-concept experiments have been performed. Further experiments are required and these should focus on benchmarks involving multiple unsat queries (in SMT-LIB most queries are expected to be sat, but given the theory content VAMPIRE cannot answer such queries).
- Finally, to enable users to utilise any developments that come out of this research, we should implement an API to allow users to interact directly with VAMPIRE in an incremental fashion.

Other ideas that were generated whilst writing this paper include the following (lower priority) research directions:

- This framework would allow us to offer support for *solving under assumptions* explicitly as it provides exactly what we need to code it up implicitly. The idea would be to allow clauses to be labelled and then make a single conjecture under an assumption of a subset of those labels. A typical key feature of this approach is to also provide a (minimal) subset of these labels required for the conjecture to hold.
- When proving with multiple conjectures there may be a priority relationship between them. We could extend our existing idea [**?**] on using a variant of set-of-support to *throttle* proof search related to particular low-priority conjectures.
- This paper has only considered saturation-based techniques. It would also be interesting to consider whether our approach to finite model building [**?**] can easily be extended to an incremental setting.

Finally, we reiterate that this is an *ideas paper* where we describe a potential problem area and how solutions within VAMPIRE could be implemented. Little has been implemented yet but the authors have plans to do so.

# References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[2] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35, 2013.

[3] Giles Reger, Martin Suda, and Andrei Voronkov. Playing with AVATAR. In P. Amy Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 399–415, Cham, 2015. Springer International Publishing.

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

[5] Andrei Voronkov. AVATAR: The architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer International Publishing, 2014.