EPiC Series in Computing

Volume 38, 2016, Pages 70–74

Proceedings of the 1st and 2nd Vampire Workshops

The Challenges of Evaluating a New Feature in Vampire *

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

[1] University of Manchester, Manchester, UK
[2] Chalmers University of Technology, Gothenburg, Sweden
[3] EasyChair

Abstract

This paper was originally called *Playing with AVATAR* and was meant to present and explore experimental results examining the usage of a new feature in Vampire: the AVATAR (Advanced Vampire Architecture for Theories and Resolution) [9] approach to splitting. The results and analysis from this original study have been extended and published elsewhere [6] and instead we use this space to briefly discuss the issues that arise when attempting to evaluate a new feature in Vampire. We suggest four different approaches to evaluating new features.

1 Introduction

The standard design cycle within a theorem prover such as Vampire is (i) identify a new problem, (ii) develop new theoretical and practical methods to solve that problem, (iii) implement these in the theorem prover, and (iv) evaluate how well you did at solving the new problem. As might be expected, findings from (iv) will usually cause one to go back to (iii), (ii) or even (i) and change things. Obviously, from this view it is important that our evaluation methods give a clear picture of how well the new implementation performs. However, as we discuss later, this is not always straightforward.

The context of this discussion is the AVATAR approach to splitting, a new feature in Vampire that considerably improved its performance. A detailed discussion of this approach and experimental results can be found in [6]. We do not repeat these findings here but will consider one topic covered in the original talk related to this work that did not appear in this recent paper: *how should we evaluate such systems?* This paper can therefore be thought of as *How to Play with AVATAR.*

2 The New Feature

The context of this work was a project [6] to explore experimental results examining the usage of AVATAR (Advanced Vampire Architecture for Theories and Resolution) [9]. Various options and extensions were investigated within the context of problems taken from the CASC

*This work was supported by EPSRC. Andrei Voronkov is supported by the Wallenberg Foundation.

competition (http://www.cs.miami.edu/tptp/CASC/24/). We studied an implementation of AVATAR within the first-order theorem prover Vampire [5].

AVATAR is a new architecture for first-order resolution and superposition theorem provers that places a SAT (or SMT) solver at the centre of the theorem prover to direct exploration of the search space. This architecture was introduced as a new technique for *splitting* clauses. Splitting addresses the issue of a rapidly growing search-space containing multi-literal and heavy clauses by splitting the search space $S \cup \{C_1 \vee C_2\}$ into $S \cup \{C_1\}$ and $S \cup \{C_2\}$ for variable disjoint C_1 and C_2.

The AVATAR framework carries out splitting in the following way. If a new clause can be split into components with disjoint sets of variables then a SAT clause is constructed by replacing each component (consistently) by a propositional literal. A (partial) model of these SAT clauses is then used to select which components should be asserted in the first-order prover. Clauses derived from these asserted clauses keep track of this dependency and then if the empty clause is produced with dependencies this information is added as a clause and the model recomputed. At any point a previously asserted clause may be unasserted and it, and any clauses produced from it, should be (temporarily) removed. Therefore, the relation between first-order solver and SAT solver is complex and needs to be explored experimentally.

Previous work [4] explored various options for splitting (leading to 481 different strategies), aiming to understand how different options affected performance of the prover. These results were partially extended for AVATAR in [9] but some questions remained unanswered. The aim of the project was to extend the experimental analysis of splitting in AVATAR.

The original project examined the efficiency of three variations of AVATAR:

1. **Adding different components to the SAT solver.** We can choose to add information about nonsplittable clauses to the SAT solver. If these clauses have dependencies, or may become components later, this may give the SAT solver more useful information when constructing a model. Additionally we tried adding a feature that delays adding clauses to the SAT solver if they can be made trivially true i.e. contain a new variable. This new feature was not reported in [6] as it had minimal impact.
2. **Minimising the produced model.** We can minimise the model in various ways i.e. construct a model only of the split clauses, or a minimal model of all clauses. There is then the decision of whether to eagerly remove asserted clauses that are no longer in a partial model.
3. **Varying how the SAT solver constructs a model.** We considered three SAT solvers – a home-grown solver, MiniSAT [2], and lingeling [1]. The results for lingeling were not reported in [6].

We reiterate this context here as it serves to illustrate the complexities that a new feature can bring. The project evaluated 11 options, some of which had as many as four possible values. The work in [6] only discusses a subset of these, partly due to space reasons and partly because some combinations produces no interesting results. However, for these uninteresting results we still needed to perform evaluation.

3 The Evaluation Issue

Automated theorem provers such as Vampire [5] typically have many options for organising proof search. They also typically implement *portfolio modes* that combine multiple strategies and select appropriate schedules of strategies based on certain characteristics of the input problem. In Vampire this mode is called *CASC mode* after the competition.

This large number of options can make evaluation very difficult as the search space for the strategies is very large[1]. The CASC mode from Vampire 3.0 makes use of 47 different (still valid) options, many of which have multiple values (some are continuous). If we stick only to values selected in the CASC mode we have 493,748,224 possible combinations (some of which will not be valid). The problem library TPTP [8] has 16,004 FOF and CNF problems in v6.0.0. Giving one minute per experiment would take 1,500 millennia per option value, making 144,000 millennia to compare all strategies on all problems. To finish in time for the talk presenting this paper in July 2014 would require the experiments to begin at the end of the Jurassic period[2]. It is apparent that evaluation needs to be more directed.

A typical evaluation scenario is that we have developed a new option (or 11 new options in this AVATAR case) and now want to determine if the effort was worthwhile. There are two settings for measuring 'improvement' that we can consider:

1. Finding new strategies that solve new problems
2. Improving old strategies i.e. solving solvable problems faster

In the both settings we are comparing against all previous existing strategies as these determine what could be solved previously and how quickly. The first setting usually applies more to brand new options that might implement a new inference or method for organising search, whilst the second setting is more applicable to small improvements of existing techniques where we might not expect new problems to be solved but hope that problems are solved faster[3]. Both settings focus on whether the option would be used for useful theorem proving in practice.

4 Evaluation Methods

We present and discuss four different possible methods when it comes to evaluation in this setting. We are concerned with two main factors. Firstly, how *expensive* it is (we cannot go back to the Jurassic period) and secondly how *accurate* it is i.e. to what extent we can trust a result saying an option (or one of its values) is good or bad.

4.1 Method 1: Doing the 'Cube'

This is the easiest experiment to run and is generally the most widely accepted; but we argue that it is probably the least useful. We note that this is the approach taken in [6].

The idea is that to generally compare different values for an option we need to systematically run through the same experiments for each value. The massive search space described above requires us to select a subset of options or problems. Selecting a subset of the option space means that we may miss the best strategies. Selecting a subset of the problem space means that we may miss some easy or hard problems. Given the size of the search space it is likely it will be necessary to do both.

This method is called doing the 'Cube' as we form a n-dimensional 'cube' of results when varying n options. Once we have this cube we can query it in different ways. For example, we can ask which vertex (single strategy) solves the most problems, solves the most problems uniquely, or solves problems the fastest. We can also look at subcubes i.e. projections of the cube that group together all strategies with a particular value for an option (or multiple

[1] This is an understatement.

[2] Assuming only single-core machines existed in the Jurassic period.

[3] Although often (especially with LRS [7]) proving things faster can alter proof search and lead to new problems being solved.

options). However, whilst this can highlight dependencies between options e.g. if option A has value a then option B does better with value b. They can also be counter-intuitive when general trends in the main results no longer hold in projections.

This method has a fixed cost given by the size of the cube. Adding new options, particularly those with many values, can be expensive and often a single baseline can be picked and then each option compared to this baseline. This effectively creates many smaller cubes that do not cover the full space but can still produce results that allow one to compare values for an option.

Whilst the cube method is rigorous and perhaps the most satisfying approach it may not be the best approach for improving a theorem prover. In our search for strategies useful for a portfolio mode we make the following observation:

If a strategy can be shown to perform well for some problems, its performance on other problems is unimportant.

That is, if we can use the strategy to solve some problems other strategies cannot solve (or are slow to solve) it does not matter if that strategy cannot solve many other problems (quickly). The following three methods are based on this observation. Importantly, it means that we do not need to compare every strategy on every problem.

4.2 Method 2: Randomly sampling the 'SuperCube'

If we are only interested in strategies that perform better than other strategies then we can randomly search the 'SuperCube' of all options and problems previously described. The idea is to repeat the following steps searching for such 'interesting' strategies:

1. Randomly select a problem and existing strategy
2. Randomly select an option to experiment on (or take a given option)
3. Vary the values for the option and see if the result is *interesting* i.e.
   - If some values solve the problem and some do not
   - If all values solve the problem but some much faster than others

If ran for long enough we can inspect the 'winning' strategies and see which option values are most frequently used. This can give some indication of the comparative usefulness of different values. However, this approach is highly dependent on the selected problems and strategies and the results do not necessarily generalise to unselected problems or strategies. Also, it is not clear what running times are required to achieve reasonable coverage. So whilst this method could be a useful indicator of usefulness it may not be a good evaluation method.

4.3 Method 3: Comparing with the 'Past'

It can be argued that the previous (existing) CASC mode characterises the set of existing strategies that have been show to be useful. In this method the idea is to examine how these strategies would perform if extended with a given option value. This is particularly relevant when adding a new option and was the approach taken in [3] when evaluating the impact of the new *extensionality resolution* inference.

If the results are improved when modifying the existing portfolio mode then it is clear that the modification is useful. However, if the results are not improved no conclusions can be drawn. The portfolio mode was assembled to cover many different problems and contains complementary strategies that may be quite fragile; changing how proof search occurs can

prevent many previously solved problems from being solved. Furthermore, it may be the case that there are trivial strategies not in the portfolio mode that could use the new option to solve many new problems. Therefore, this approach is open to false negatives. This is the cheapest evaluation method as it only requires two 300 second runs per problem.

4.4 Method 4: Building different 'Futures'

Finally, we propose an evaluation method we have not yet tried but suggest is the most promising in establishing the practical usefulness of an option. The idea is to construct two portfolio modes with one mode using a particular option value and the other restricted so it cannot use this value. These modes represent two alternative 'futures' where we have the option value or do not. We can then compare the two modes on a set of problems.

The challenge is then to construct these portfolio modes. This can be achieved through a random search of the strategy space coupled with methods for improving strategies that perform well. Such techniques are already used to construct vampire's CASC portfolio mode. It is clear that this process can be very expensive. The advantage is that we get a tuned portfolio mode

5 Summary

As mentioned at the end of [6], there is still much more work to be done in understanding the interplay between different methods for architectures such as AVATAR, and automated theorem provers in general. Key to this work is the development of pragmatically useful evaluation techniques that reflect the actual benefit of a particular technique or modification.

References

[1] A. Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. In A. Balint, A. Belov, M. Heule, and M. Jrvisalo, editors, *Proceedings of SAT Competition 2013*, 2013.

[2] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

[3] A. Gupta, L. Kovcs, B. Kragl, and A. Voronkov. Extensional crisis and proving identity. In F. Cassez and J.-F. Raskin, editors, *Automated Technology for Verification and Analysis*, volume 8837 of *Lecture Notes in Computer Science*, pages 185–200. Springer International Publishing, 2014.

[4] K. Hoder and A. Voronkov. The 481 ways to split a clause and deal with propositional variables. In M. P. Bonacina, editor, *Automated Deduction CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2013.

[5] L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In N. Sharygina and H. Veith, editors, *CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35, 2013.

[6] G. Reger, M. Suda, and A. Voronkov. Playing with AVATAR. In A. Felty and A. Middeldorp, editors, *25th Internal Conference on Automated Deduction CADE-25*. 2015.

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

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

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