Experimenting with SAT solvers in Vampire *

Armin Biere[1], Ioan Dragan[2], Laura Kovács[2,3], and Andrei Voronkov[4]

[1] Johannes Kepler University, Linz, Austria
[2] Vienna University of Technology, Vienna, Austria
[3] Chalmers University of Technology, Gothenburg, Sweden
[4] The University of Manchester, Manchester, UK

Abstract. Recently, a new reasoning framework, called AVATAR, integrating first-order theorem proving with SAT solving has been proposed. In this paper, we experimentally analyze the behavior of various SAT solvers within first-order proving. For doing so, we first integrate the Lingeling SAT solver within the first-order theorem prover Vampire and compare the behavior of such an integration with Vampire using a less efficient SAT solver. Interestingly, our experiments on first-order problems show that using the best SAT solvers within AVATAR does not always give best performance. There are some problems that could be solved only by using a less efficient SAT solver than Lingeling. However, the integration of Lingeling with Vampire turned out to be the best when it came to solving most of the hard problems.

1 Introduction

This paper aims to experimentally analyze and improve the performance of the first-order theorem prover Vampire [8] on dealing with problems that contain propositional variables and also other clauses that can be splitted. The recently introduced AVATAR framework [15], proposes a way of integrating a SAT solver in the framework of an automatic theorem prover. The main task that a SAT solver has in this framework is to help the theorem prover in splitting clauses. Although initial results obtained by using this framework in Vampire proved to be really efficient, it is unclear whether efficiency of AVATAR depends on the efficiency of the used SAT solver. In this paper we address this problem using various SAT solvers and experimentally evaluate AVATAR as follows. We first integrate the Lingeling [3] SAT solver inside Vampire and compare its behavior against a less efficient SAT solver already implemented in Vampire. Our experiments on a large number of problems show significantly different results when using Lingeling and/or the default SAT solver of Vampire for splitting clauses in AVATAR.

Splitting clauses is a well studied problem in the community of automated theorem provers. First, the method introduced in the SPASS [16] theorem prover tries to do splitting and uses backtracking to recover from a bad split. Another way of dealing with splittable clauses was introduced in Vampire [10] and takes care of splitting without backtracking. Both ways of splitting on a clause are highly optimized for these theorem provers. Implementation of these splitting techniques is not trivial and can

* This work was partially supported by Swedish VR grant D0497701 and the Austrian research projects FWF S11410-N23, FWF S11408-N23 and WWTF ICT C-050.

highly influence the overall performance of the prover. Therefore, in [6] the authors are performing an extensive evaluation of different ways of doing splitting and evaluate other methods using BDDs and SAT solvers for clause splitting. Although the use of splitting improves performance these splitting techniques cannot compete against the methods used in SAT solvers on propositional problems or even in SMT solver on ground instances [15].

The problem of dealing with splitting clauses in AVATAR is motivated by the way first-order theorem provers usually work. In general first-order provers make use of three types of inferences: *generating, deleting* and *simplifying* inferences. In practice, using these inferences one can notice a couple of problems. Usually the complexity for implementing different algorithms for the inference rules are dependent in the size (length) of the clauses they operate on. As an example of simplifying inference, subsumption resolution is known to be NP-complete and the algorithms that implement it are exponential in the number of literals in a clause. Another issue arises when we want to use generating inferences. In this case assuming we have two clauses containing l_1 and l_2 literals and we apply resolution on them then the resulting clause will have $l_1 + l_2 - 2$ literals. Now if these clauses are long it means we generate even longer clauses. This also raises the question of storage for these clauses for example by indexing [9] .They are a couple of methods that deal with large clauses, for example limited resource strategy [11] which is also implemented in Vampire. This method will start throwing away clauses that slow down the prover. An alternative would be to use splitting in order to make the clauses shorter and easy to be manipulated by the prover.

In this paper we study the use of splitting in the new AVATAR framework (see Section 2) for first-order theorem proving, by integrating different SAT solvers into the Vampire automated theorem prover (see Section 3). We evaluate the new approach (see Section 4) on a large set of problems in order to better understand how does the use of an state of the art SAT solver influences the AVATAR framework.

2 Preliminaries

This section overviews the main notions used in the paper, for more details we refer to [8, 15]. In the framework of first-order logic, a *first-order clause* is a disjunction of *literals* of the following form $L_1 \vee \ldots \vee L_n$, where a literal is an atomic formula or the negation of an atomic formula. Usually when we speak about splitting we speak about clauses as being sets of literals. Due to this description of a clause we can safely assume that we do not have duplicate literals. We also assume that predicates, functions are uninterpreted and the language might contain the equality predicate (=).

In a nutshell splitting of clauses starts from the following remark. Suppose that we have a set *S* of first-order clauses and $C_1 \vee C_2$ a clause such that the variables of C_1 and C_2 are disjoint. Then $\forall(C_1 \vee C_2)$ is equivalent to $\forall(C_1) \vee \forall(C_2)$. This transformation implies that the set $S \cup \{C_1 \vee C_2\}$ is unsatisfiable if and only if both $S \cup \{C_1\}$ and $S \cup \{C_2\}$ are unsatisfiable. In practice one can notice the fact that splittable clause usually appear when theorem provers are used in software verification applications.

Let $C_1, \ldots, C_n$ be clauses such that $n \geq 2$ and all the C_i's have pairwise disjoint sets of variables. We can safely say that $SP \stackrel{\text{def}}{=} C_1 \vee \ldots \vee C_n$ is splittable into components $C_1, \ldots, C_n$. We will also say that the set $C_1, \ldots, C_n$ is a splitting of SP. An

example of such a splittable clause can be considered any ground clause that contains multiple literals. One problem that arises in splitting is the fact that there are multiple ways of splitting a clause. But this is not a major issue since we know that there is always a unique splitting such that each component cannot be splitted more. We call this splitting of a clause maximal. Computation of such a splitting proves to always give the maximal number of components of a clause, see [10] for details.

Let us first discuss how the mapping between the first-order problem and the propositional problem is done. In the propositional problem that is sent to the SAT solver we basically keep track of clause components. In order to do that we have to use a mapping [.] from components to propositional literals. The mapping has to satisfy the following properties: 1. $[C]$ is a positive literal if and only if C is either a positive ground literal or a non-ground component; 2. for a negative ground component $\neg C$ we have $[\neg C] = \neg[C]$; 3. $[C_1] = [C_2]$ if and only if C_1 and C_2 are equal up to variable renaming and symmetry of equality. In order to implement this mapping Vampire uses a component index, which maps every component that satisfies the previous conditions into a propositional variable $[C]$. And for each such component C the index checks whether there is already a stored component C‘ that are equal than it returns $[C$‘$]$ as propositional variable. Doing so we ensure that we do not have multiple propositional variables that are mappings of equal components. In case there is no such component stored in the index, than a new propositional variable $[C]$ is introduced and we store the association between C and $[C]$. A model provided by the SAT solver for the propositional problem is considered a component interpretation. Such a model contains only variables of the form $[C]$ or their negations and does not contain in the same time both a variable and its negation. The truth definition of a propositional variable in such an interpretation is standard. With the small difference that in case for a component C neither $[C]$ not $\neg[C]$ belongs to the interpretation, than $[C]$ is considered *undefined*, meaning it is neither true nor false.

In a nutshell AVATAR works as follows. The first-order reasoning part works as usual, using a saturation algorithm [8]. The main difference with respect to a classical approach is the way it treats splittable clauses. In the case that a clause $C_1 \vee C_2 \ldots \vee C_n$ is splittable in $C_1, C_2, \ldots, C_n$ components and the clause passes the retention test it is not added to the set of *passive* clauses. Instead we add a clause $[C_1] \vee [C_2] \vee \ldots \vee [C_n]$ to the SAT solver and check if the problem added to the solver is satisfiable. If the SAT solver returns unsatisfiable, it means that we are done and report it to the first-order reasoning part. In case the problem is satisfiable, we ask the SAT solver to produce a model. This model acts as a component interpretation *I*. If in the interpretation a literal has the form $[C]$ for some component C then we pass to the first-order reasoner the component, where C is used as an assertion. Exception from this rule are those literals of the form $\neg[C]$, where *C* is a non-ground component. This is due to the fact that such a literal does not correspond to any component.

In our context a SAT solver has to expose an incremental behavior. By incremental we mean that the solver receives from time to time new clauses that have to be added at the propositional problem and checks whether the problem is satisfiable upon request from the first-order reasoner. If the problem is satisfiable than all it has to do is to pass back to the first-order reasoner a model (component interpretation) for all the propo-

sitional variables. Otherwise it simply has to return unsatisfiable and communicate the unsatisfiability result to the first-order reasoning part as well.

3 Integration

We now describe how we integrated the Lingeling SAT solver in the framework of Vampire. We also overview the options implemented in order to control the behavior of Lingeling in Vampire. Although Lingeling is used in commercial applications, the source code is publicly available. Also the default license allows Lingeling to be used in non-commercial and academic context. Our main goal after integrating the new solver in Vampire framework was to obtain better performance in the process of solving first-order problems.

In general any SAT solver is designed to accept as its input problems described in the DIMACS format [2]. We have decided to implement an interface that allows us to directly control Lingeling via its API. By using the API one can also control the options for the background SAT solver at run time depending on the strategy being deployed.

In the case when the SAT solver establishes satisfiability of a given problem, we are interested in obtaining a model for the problem. This behavior matches the intended use for the majority of SAT solver and in particular Lingeling. In the case of satisfiability though there are some situations when we would be interested in obtaining similar models. By similar models we mean that in the incremental case, if the current problem is proved to be satisfiable, we add new clauses to the solver and the solver decides that the problem is still satisfiable, we would like to obtain a model that has as few different assignments from the previous model as possible.

For the purpose of our work, we use Lingeling in an incremental manner, but there is still the question of how should we add the clauses to the solver. Incrementality in the context of SAT solving refers to the fact that a SAT solver is expected to be invoked multiple times. Each time it is asked to check satisfiability status of all the available clauses under assumptions that hold only at that specific invocation. The problem to be solved thus grows upon each call to add new clauses to the solver, for details see [5]. In the context of Vampire at some particular point the first-order reasoner can add a set of clauses to the existing problem. In order to add these clauses to the underlying SAT solver we implemented two versions of using Lingeling in the AVATAR architecture of Vampire. The first version, given in Algorithm 1, iterates over the clauses that appear in the original problem and adds them one by one to Lingeling. After we have added the entire set of clauses to the SAT solver we call for satisfiability check. We call this method of adding clauses "almost incremental" since it does not call for satisfiability check after each clause is added. Algorithm 1 is very similar to non-incremental SAT solving at each step when the first-order reasoning part asks for satisfiability check, since the call for satisfiability is done only after all the new clauses are added to the solver (line 6). Overall, the approach is still based on incrementality of the underlying SAT solver, since we keep adding clauses to the initial problem.

Another way of using the underlying solver would be to simulate the pure incremental approach, as presented in Algorithm 2. This approach is similar to the previous

one with the difference that now as soon as a new clause is added to Lingeling we are also calling for a satisfiability check (line 4).

In order to be able to use any of the previous ways of integrating Lingeling in Vampire one has to be careful when adding clauses to Lingeling. Internally Lingeling tries to apply preprocessing on the problem and during preprocessing a subset of variables could be eliminated. This can lead to some problems since we eliminate a subset variables during the preprocessing and in some future step we might add some of them back to the solver. The issue that arises here is the fact that performing these operations can lead to unsoundness of the splitting solution generated by the first-order reasoner. In order to avoid the issue of not allowing the solver to eliminate variables while performing the preprocessing steps, Lingeling relies on the notion of *frozen* literals [3]. One can see a frozen literal as a literal that is marked as being important and not allowing the preprocessor to eliminate it during preprocessing steps. Using freezing of literals we are ensured that although preprocessing steps are done, it will inhibit the elimination of marked variables. In our case it actually means that one has to freeze all the literals that appear in the initial problem and also all the literals that are due to be added. The process of freezing literals is done on the fly when new clauses are added to the solver. In order to do be efficient and not freeze multiple times the same literal we keep a list of previously added and frozen literals.

```
1:  Input: a set of clauses to be added
2:  while not all clauses added do
3:      Add clause to Lingeling
4:      Keep track of the added clause
5:  end while
6:  Call SAT procedure
7:  if UNSATISFIABLE found then
8:      Report Unsatisfiability
9:  else
10:     Return a model
11: end if
```

Fig. 1. *“Almost” incremental* version of Lingeling in Vampire

```
1:  Input: a set of clauses to be added
2:  while not all clauses added do
3:      Add clause to Lingeling
4:      Call SAT procedure
5:      Keep track of the added clause
6:      if UNSATISFIABLE found then
7:          Report Unsatisfiability
8:      else
9:          Return a model
10:     end if
11: end while
```

Fig. 2. *Incremental* version of Lingeling in Vampire

Although the freezing of all literals proves to be a suitable solution of enforcing Lingeling not to eliminate some variables during the preprocessing steps, this also limits the power of the preprocessing implemented in the solver. One improvement could be to develop a methodology that would allow “predicting” which literals are not going to be used later on and allow the SAT solver to eliminate them if necessary.

3.1 Integrating and using Lingeling in Vampire

In order to run Vampire[5] with Lingeling as a background SAT solver one has to use from command line the following option:

```
--sat_solver lingeling
```

[5] Vampire with all the features presented in this paper can be downloaded from vprover.org

By default when one enables the use of Lingeling as a background SAT solver, the solver is used as presented in Algorithm 1. This means that we add first all the clauses to the SAT solver and only then call for satisfiability check.

In case one wants to use Lingeling in Vampire as presented in Algorithm 2 the following option needs to be used

$$--\texttt{sat_lingeling_incremental}\ [\texttt{on/off}]$$

This enables the incremental use of Lingeling as presented in the algorithm. By default this option is set to **off**.

We are also interested in generating similar models when we use incrementally the underlying solver. In order to control this behavior, one should use the option:

$$--\texttt{sat_similar_models}\ [\texttt{on/off}]$$

By default this option is set to **off**. As for the previous options activating similar model generation has effect only in the case where Lingeling is used as background solver. In the following we present the results obtained by running Vampire with combinations of these options.

4 SAT experiments in Vampire

Currently, there are all together 5 different combinations of values for the new options controlling the use of SAT solvers in Vampire. In order to benchmark these strategies we used problems coming from the TPTP [13] library. The experiments where run on the InfraGrid infrastructure of West University of Timisoara [1]. The infrastructure contains 100 Intel Quad Core processors, each one with dedicated 10GB of RAM. All the experiments presented in this paper are run with a time limit of 60 seconds and with memory limit of 2GB.

4.1 Benchmarks and experiments

As a first set of problems we have considered the 300 problems from the first-order division of the CASC 2013 competition see [14]. Besides these problems we also used 6637 problems from the TPTP library. These 6637 problems are a subset of the TPTP library that have ranking greater than 0.2 and less than 1. Ranking 0.2 means that 80% of the state of the art automatic theorem provers can solve this problem, while ranking 1 means that no state of the art automated theorem prover can solve the problem.

Generally using a cocktail of strategies on a single problem proves to behave always better in first-order automated theorem proving. For this purpose we have decided to evaluate our approaches of using SAT solving in the AVATAR framework of Vampire both using a mixture of options and also using the default options implemented in Vampire.

CASC competition problems. We evaluated Vampire using all the new SAT features and kept all other options with their default values, from now on we will call this version of Vampire *default mode*. Also we evaluated the mode where we launch a cocktail of options (strategies) with small time limits and try to solve the problem, called the *casc mode*. A summary of the results obtained by running these strategies can be found in Table 1 and Table 2. All tables presented in this paper follow the same structure: the

Table 1. Results of running Vampire with default values for parameters on the 300 CASC problems.

Strategy	Vamp	L	L S	L I	L I S
Average Time	3.4747	3.0483	4.2159	2.6728	3.8490
# of solved instances	142	146	156	143	144
# different	2	12	16	10	11

first row presents the abbreviations for all the used strategies, the second row presents the average time used by each of the strategies in order for solving the problems. Here we take into account only the time spent on the problems that can be solved using a particular strategy. The third row presents the total number of problems solved by each strategy. The last row presents the number of different problems. By different problems we mean problems that could be solved either by Vampire with the default SAT solver and not solved by any of the strategies involving Lingeling and the problems that can be solved only by at least one strategy that involves Lingeling but cannot be solved by Vampire using the default SAT solver. The abbreviations that appear in the header

Table 2. Results of running Vampire using a cocktail of strategies on the 300 CASC problems.

Strategy	Vamp	L	L S	L I	L I S
Average Time	3.4679	3.0615	4.2701	2.8139	3.7852
# of solved instances	230	233	240	232	232
# different	1	8	13	8	7

of each table stand for the following: *vamp* stands for Vampire using the default SAT solver, *L* stands for Vampire using Lingeling as background SAT solver, in an "almost" incremental way, *L S* similar to *L* but turning the generation of similar models on the SAT solver side on, *L I* stands for Vampire using Lingeling as background SAT solver in pure incremental way and *L I S* is similar to *L I* but with the change that it turns similar model generation on the SAT solver side.

Table 1 reports on our experiments using the default mode of Vampire on the 300 CASC problems. Among these 300 problems, 23 problems can be solved only by either Vampire using some variations of Lingeling as background SAT solver or by Vampire using the default SAT solver. Table 2 shows our results obtained by running Vampire in casc mode on the 300 CASC problems. Among these 300 problems there are 18

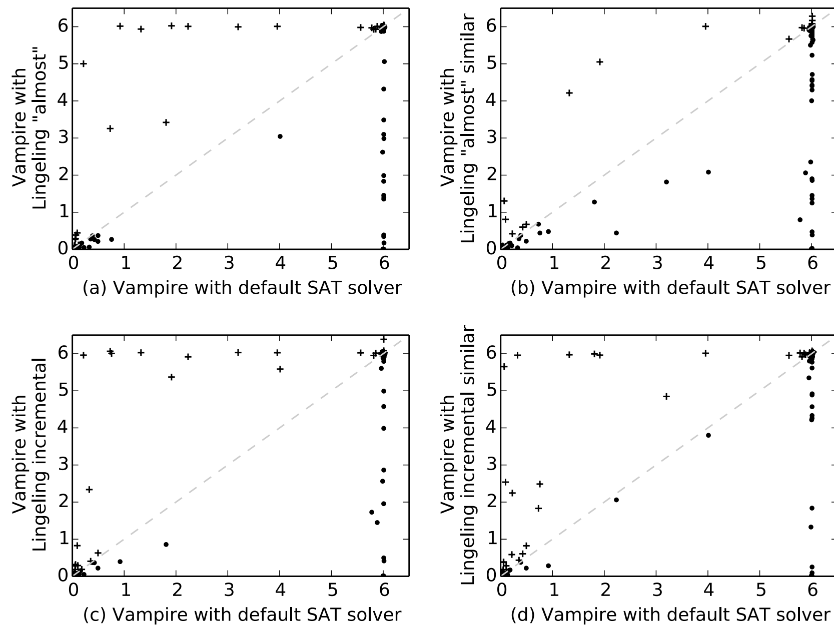

Fig. 3. Comparison of performance between Vampire with the default SAT solver and different Lingeling strategies run in *default* mode. Default SAT solver is compared against: (a) Lingeling "almost" incremental, (b) Lingeling "almost" incremental and similar models, (c) Lingeling incremental and (d) Lingeling incremental and similar models

problems that can be solved only by either Vampire using some variation of Lingeling as background SAT solver or by Vampire using the default SAT solver.

Figure 3 presents a comparison between Vampire using the default SAT solver and each of the new strategies. The scatter plots present on the x-axis the time spent by Vampire in trying to solve an instance, while on the y-axis the time spent by different strategies on the same instance. In order to have more concise figures we have decided to normalize the time spent in solving by a factor of 10. Doing so one can compare time-wise the performance of each of the strategies. A point appearing on the diagonal of the plot represents the fact that both strategies terminated in the same amount of time. A point appearing on top of the main diagonal represents the fact that Vampire using the default strategy managed to solve that instance faster than Vampire using Lingeling variations. Similar a point below the diagonal represents the fact that Vampire using the new strategy solved the problem faster than Vampire using the default SAT solver.

The plot presents one to one comparison between the strategies and the default strategy. In Figure 3 we present the results obtained by running Vampire in "default" mode but varying the SAT solver as described above. From this figure one can notice the fact that more points appear above the diagonal, meaning that the default values of Vampire are better. We can notice however that there are some problems on which

Vampire with default SAT solver time out while using Lingeling they can be solved in very short time. From these plots one could conclude that taken individually these strategies and compared to the default one, they seem to be have similar behavior as the default one. Nevertheless, if we take them together and compare them to the default strategy we notice the fact that indeed they behave better.

Table 2 and Figure 4 present a similar comparison on the same problems, using the same variations of the underlying SAT solver and the same limits as for the *default* mode.

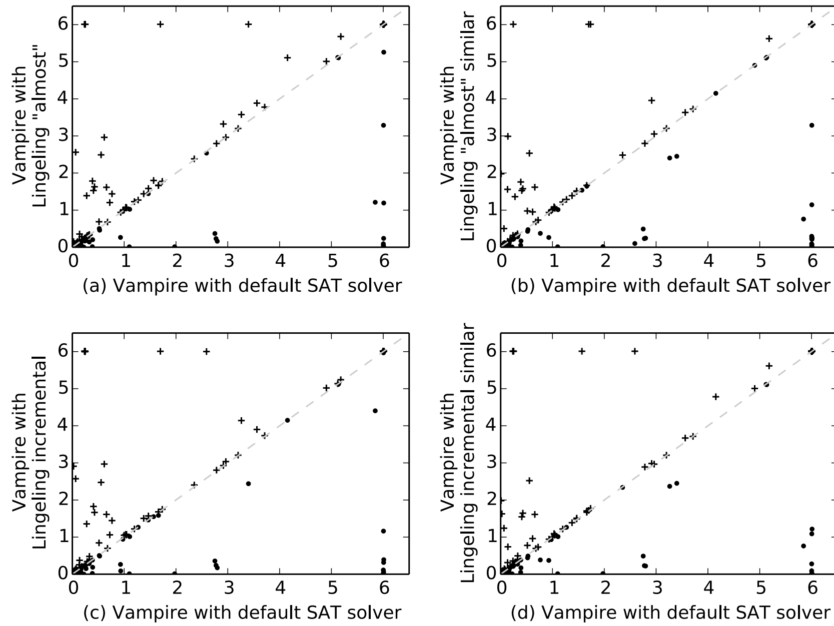

Fig. 4. Comparison of performance between Vampire with the Default SAT solver and different Lingeling strategies run in *casc* mode. Default SAT solver is compared against: (a) Lingeling "almost" incremental, (b) Lingeling "almost" incremental and similar models, (c) Lingeling incremental and (d) Lingeling incremental and similar models

Other TPTP problems. In a similar manner as for the 300 CASC problems we have evaluated our newly added features on a big subset of TPTP problems. The problems that have been selected for test have ranking in the interval [0.2, 1), having the status of either: *Unsatisfiable, Open, Theorem* or *Unknown*.

In Table 3 we present the summary of obtained results from running Vampire with all the variations on the set of problems in *default* mode. Table 4 presents the summary of our results obtained by running Vampire in *casc* mode on the same set of problems using the same variations as above described.

Table 3. Results of running Vampire using default values for parameters on the 6.5K problems.

Strategy	Vamp	L	L S	L I	L I S
Average Time	6.0440	5.9982	6.5992	5.6805	6.5025
# of solved instances	2672	2810	2925	2750	2788
# different	104	350	422	328	334

From our experiments we noticed that using Lingeling as a background SAT solver in the "almost" incremental and with the similar model generation turned on proves to perform the best among the newly implemented strategies. This sort of behavior can be due to multiple reasons. First it could be due to the fact that the solver tries to keep the model for as long as possible, due to similar model generation option. Another explanation for best performance can be the fact that using this options we do not call the SAT solver after each clause is added, but rather only after we add all the clauses generated by the first-order reasoning part, hence decreasing the time spent by the SAT solver in solving.

Table 4. Results of running Vampire using a cocktail of strategies on the 6.5K problems.

Strategy	Vamp	L	L S	L I	L I S
Average Time	6.1019	6.0895	6.1139	6.3069	6.0638
# of solved instances	4788	4822	4881	4809	4792
# different	81	212	245	207	194

4.2 Analysis of experimental results

While integrating the new SAT solver inside Vampire and during the experiments we observed some issues that might increase the performance of future SAT solvers inside the Vampire's AVATAR architecture. **(i)** It is not necessary that a state of the art SAT solver, as Lingeling, behaves better inside the AVATAR framework. **(ii)** Integration of new solvers is less complicated than fine tuning the newly integrated solvers in order to match the performance of the default SAT solver, which is hard. **(iii)** Using an external SAT solver just in case the SAT problems are hard enough could be a good trade-off. We discuss these issues below.
(i) First, let us discuss the performance issue. At least in the case of Lingeling upon integration we have noticed that it behaves really nice on some of the problems while on some others it seems to fail. There are a couple of factors that could influence the behavior of such a performant tool. (1) Calling the solver many times decreases its performance. Although the solver is designed to be incremental upon adding new clauses to the problem and call for satisfiability check, it restarts. Now the problem appears when we call many times the solver. For example in the case of "pure" incremental way, we call the solver after each clause is added. Although the speed with which the check is done is incredibly fast, calling it n times makes it n times slower. (2) Due to this behavior in the worst case we have n restarts on the SAT solver side, where n is the number of clauses added to the solver. Both these points showed up in the statistics from

the experiments we have performed. It is not uncommon that the first-order reasoning part will create a problem containing 10K or even 100K clauses. Now even if for one call the SAT solver spends 0.01 seconds it results in a timeout.
(ii) The default SAT solver implemented in Vampire follows the general structure of the MiniSAT [5] SAT solver. This architecture is an instantiation of the Conflict-Driven Clause Learning (CDCL) [12] architecture. Although it is incremental, it deals with incrementality in a different manner. Assuming that we add a clause to the solver, first we check whether we can extend the current model so that we can satisfy also the newly added clause if the clause gets satisfied by the current model we keep the model and add the clause to the database. In case the clause is not directly satisfied by the model but does not contain any variable that is used in the model we try to satisfy the clause by extending the model. If we cannot do that, we do not restart, but rather backtrack to the point where the conflict comes from. A conflict can appear only if variables that are used in the model appear also in the newly added clause. If that is the case we take the lowest decision level among the conflict variables and backtrack to it. From there on we continue the classical SAT procedure and try to find a new model.

Using this approach, the SAT solver brings a couple of advantages for the first-order reasoning part. Due to the fact that we try to keep the model with minimal changes, we do not have to modify the indexing structures so often in the first-order part and also from our experiments we have noted that the actual SAT solving procedure gets called less often than in the case of calling for satisfiability check on an plugged-in solver.
(iii) Adding a new external SAT solver inside the framework of Vampire is not complicated. One has to take care about how the first-order reasoning part and the SAT solver communicate and do some book keeping. The issue arises when one has to fine-tune the SAT solver so that it performs well in it's new environment. Usually state of the art SAT solvers are highly optimized in order to behave well on big problems coming from industry but sometimes seem to get stuck in small problems. Also one important component of a state of the art solver is preprocessing [4], which for our purpose has to be turned off in order to ensure that we do not eliminate variables that might be used for splitting in the first-order reasoner. Due to the fact that we do not use all the power of a SAT solver we have to answer the question whether it is the case that state of the art, or commercial, solvers behave better in this context. In the case of the AVATAR architecture for an automated theorem prover producing different models for the SAT problems means that a clause gets splitted in different ways. That also translates into the fact that in some cases the use of an "handcrafted" SAT solver might produce the right model, but it also means that in other cases the state of the art solver produces the right model at the right time. This results in either solving the problem really fast or not at all. Some interesting fact that we have noticed during the experiments is the fact that the AVATAR architecture is really sensitive towards the models produced by the SAT solver.

5 Conclusion and future work

Starting from the initial results of the newly introduced AVATAR framework for an automatic theorem prover, we investigate how does a state of the art SAT solver behave

in this framework. We describe the process of integrating a new SAT solver in the framework of Vampire using the AVATAR architecture. We also present a couple of decisions that have been made in order to better integrate the first-order proving part of Vampire with SAT solving. From our experiments we noticed that using a state of the art SAT solver like Lingeling inside the framework of an automated theorem prover based on AVATAR is useful and behaves well on TPTP problems. However there are also cases where using Lingeling as background solver Vampire does not perform as good as using a less efficient SAT solver.

We believe that further refinements on the SAT solver part and better fine tuning of the solver will produce even better results. We are investigating different ways of combining the Vampire built-in SAT solver with the external SAT solver such that we do not restart upon every newly added clause. Besides splitting, Vampire uses SAT solving also for instance generation [7] and indexing. We are therefore also interested in finding out whether the use of a state of the art SAT solver improves the performance of Vampire.

References

1. http://hpc.uvt.ro/infrastructure/infragrid/. HPC Center - West University of Timisoara
2. Biere, A.: Picosat essentials. JSAT 4(2-4), 75–97 (2008)
3. Biere, A.: Lingeling, plingeling and treengeling entering sat competition 2013. In: SAT Competition 2013. pp. 51–52 (2013)
4. Eén, N., Biere, A.: Effective preprocessing in sat through variable and clause elimination. In: SAT. pp. 61–75 (2005)
5. En, N., Srensson, N.: An extensible sat-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer Berlin Heidelberg (2004)
6. Hoder, K., Voronkov, A.: The 481 ways to split a clause and deal with propositional variables. In: CADE. pp. 450–464 (2013)
7. Korovin, K.: Inst-gen - a modular approach to instantiation-based automated reasoning. In: Programming Logics. pp. 239–270 (2013)
8. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: Proc. of CAV. pp. 1–35 (2013)
9. Nieuwenhuis, R., Hillenbrand, T., Riazanov, A., Voronkov, A.: On the evaluation of indexing techniques for theorem proving. In: IJCAR. pp. 257–271 (2001)
10. Riazanov, A., Voronkov, A.: Splitting without backtracking. In: IJCAI. pp. 611–617 (2001)
11. Riazanov, A., Voronkov, A.: Limited resource strategy in resolution theorem proving. J. Symb. Comput. 36(1-2), 101–115 (2003)
12. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning sat solvers. In: Handbook of Satisfiability, pp. 131–153 (2009)
13. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. J. Autom. Reasoning 43(4), 337–362 (2009)
14. Sutcliffe, G.: Tptp, tstp, casc, etc. In: CSR. pp. 6–22 (2007)
15. Voronkov, A.: Avatar: The architecture for first-order theorem provers. In: CAV. pp. 696–710 (2014)
16. Weidenbach, C.: Combining superposition, sorts and splitting. In: Handbook of Automated Reasoning, pp. 1965–2013 (2001)