EPiC Series in Computing

Volume 53, 2018, Pages 9–19

Vampire 2017. Proceedings of the 4th Vampire Workshop

Light-Weight Integration of SAT Solving into First-Order Reasoners – First Experiments

Stephan Schulz

DHBW Stuttgart
schulz@eprover.org

Abstract

We describe a light-weight integration of the propositional SAT solver PicoSAT and the saturation-based superposition prover E. The proof search is driven by the saturation prover. Periodically, the saturation is interrupted, and all first-order clauses are grounded. The resulting ground problem is converted to a propositional format and handed to the SAT solver. If the SAT solver reports unsatisfiability, the proof is extracted and reported on the first-order level. First experiments demonstrate the viability of the approach and suggest future extensions. They also yield interesting information about the structure of the search space.

1 Introduction

Nearly all modern theorem provers for first order logic use a refutational approach. They convert axioms and conjecture into a set of clauses that is unsatisfiable if and only if the conjecture is a logical conclusion of the axioms, and search for an explicit contradiction. We know from Herbrand's theorem (with a bit of help from the compactness theorem of propositional logic) that a set of first-order clauses is unsatisfiable if and only if it has a finite set of ground instances that is propositionally unsatisfiable. This immediately yields a complete proof procedure - enumerate the ground instances and periodically check for propositional unsatisfiability. An implementation of this approach was famously described by Davis and Putnam[3]. However, while theoretically sound and complete, in practice the set of ground instances grows too fast to be manageable, and calculi based on unification like Resolution [13], Paramodulation [12] and Superposition [1] became the dominant paradigm for first-order reasoning.

The propositional satisfiability algorithm underlying Davis' and Putnam's approach, on the other hand, developed into the DPLL procedure [4], and, with refinements of implementation and the addition of non-chronological back-tracking and conflict clause learning, lead to the modern generation of SAT solvers, which have evolved from pure back-tracking and unit propagation to *conflict driven clause learning* (CDCL) [21, 10]. These solvers have made incredible progress in the last two decades, routinely solving SAT problems with thousands or even millions of propositional atoms. This progress has created a desire to utilize modern SAT solvers for first-order-reasoning.

In this paper we describe first steps of integrating SAT solving into the superposition-based high-performance theorem prover E [15, 16]. Saturating provers, whether based on resolution or superposition, represent the proof state by a set of clauses and use an inference system to systematically derive new clauses that are subsequently added to the proof state. They thus combine the generation of (partially conflicting) instances (by unification) and the check for unsatisfiability (by eventual generating the empty clause) in a single procedure.

Compared to the naive enumeration of instances with a separate unsatisfiability test, this combination is one of the reasons for the success of saturation. However, concluding a proof requires both that the proper instances have been generated, and that the relevant clauses are selected for inferences to actually produce the empty clause.

All saturation-based provers we are aware of perform all possible inferences between a small, but growing subset of the proof state. Whether by level-saturation or by any of the versions of the *given-clause algorithm*, this results in a very large number of *passive* or *unprocessed* clauses that have already been generated, but did not yet have any chance to interact. Typically, the number of unprocessed clauses grows roughly quadratically with the number of processed clauses. The idea described in this paper is to periodically check if the (naively grounded) set of all clauses, both processed and unprocessed, is already propositionally contradictory. Since this check can often be performed very efficiently using a CDCL system, this may help uncover an already existing explicit contradiction much earlier than via the saturation procedure.

The rest of the paper is structured as follow: First, we give a short overview on related techniques. Then we describe the architecture and implementation of our system. Some initial experimental results validate the basic thesis of our work. We discuss various ways to continue and improve on this work, before we conclude with a short summary.

2 Related Work

There have been a number of recent approaches to utilize the power of SAT solvers in first-order reasoning. One of the older approaches is Plaisted's clause linking method [9]. In this method, complementary literals are unified and the corresponding *linking instances* of the clauses are recorded. The set of all instances is periodically checked for propositional unsatisfiability. In practice, this method suffered from lack of control - the number of possible links is enormous, and the value of any particular instance is hard to predict.

Ganzinger and Korovin [6] developed what they called *instance-based theorem proving*, culminating in the Inst-Gen calculus [8] and its implementation in iProver [7]. In this method, the selection of the linking instances is driven by a propositional model. The clause set is grounded and a proposition solver is used to check the ground set for satisfiability. If the grounded set is unsatisfiable, so is the original clause set. If not, the method tries to lift the ground model up to the original (usually non-ground) clause set. If the lifted interpretation is a model, the clause set is satisfiable. If not, the method uses unification to find literals that conflict on the non-ground model, and adds linking instances between the involved clauses. This process of propositional unsatisfiable checking and refinement of the clause set is repeated until the proof search is terminated with a contradiction, a model, or a resource limit.

Both clause linking and Inst-Gen are new and complete calculi for first-order reasoning. Inst-Gen, in its iProver incarnation, is competitive with saturation-based provers in some classes, e.g. the effectively propositional EPR class. It does not reach their performance in the general case, and, in particular, suffers from the fact that there has not yet found a satisfactory solution for handling the equality relation.

AVATAR, the *Advanced Vampire Architecture for Automated Reasoning* [20, 11], combines

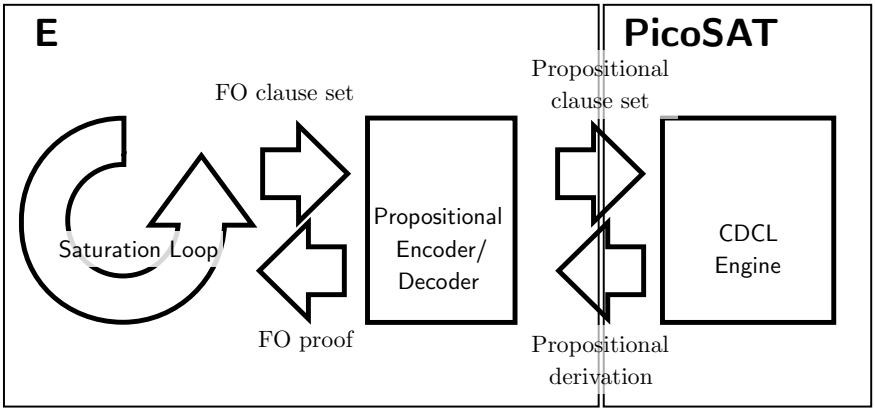

Figure 1: Architectural Overview

propositional reasoning and superposition-based saturating reasoning in a very elegant framework. The propositional structure of the first-order clause set is extracted by associating each independent sub-clause (i.e. set of literals not sharing any variables with other literals in the clause) with a propositional variable. The resulting propositional clause set is given to a SAT solver. If the propositional clause set is unsatisfiable, so is the first-order problem. If not, the model returned is used to determine which sub-clauses are assumed true. The set of all these assumed true subclauses is saturated. If it is found unsatisfiable, the propositional clause set is updated with a learned clause eliminating the underlying model, and the process starts over.

The AVATAR approach provides a seamless transition from superposition to propositional reasoning. For a unit-equational problem, it degenerates into unfailing completion. For a problem without splitable clauses, it corresponds to plain superposition. And for a ground problem, it performs essentially like a pure CDCL prover. AVATAR has been implemented for Vampire, and is believed to be one of the reasons for the increased performance of Vampire in the last few years. The main downside is that AVATAR, at least if well-implemented, requires a significant development effort. Our work in this paper is an attempt to reap some of the same benefits with a simpler, more light-weight approach.

3 Background and architecture

We assume the usual conventions and terminology of classical first-order clausal logic (with equality). A *signature* is a tuple (F, P, V), where F is a finite set of *function symbols* with associated arities, P is a set of predicate symbols with arities, and V is an enumerable set of *variables*. *Constants* are function symbols with arity 0. The main objects of concern are *terms*. A term is recursively defined as either a variable $x \in V$ or is composed by combining a function symbol of arity n (written $f/n \in F$) and n existing terms $t_1, \ldots t_n$ to form $f(t_1, \ldots t_n)$. Note that this definition makes a constant with zero argument terms also a term - we typically write this as c, not $c()$.

An atom is composed similar to a term by combining a predicate symbol p/n with the suitable number of subterms. A literal is either an atom $p(t_1, \ldots t_n)$ or a negated atom $\neg p(t_1, \ldots t_n)$. Finally, a clause is a multiset of literals, usually written as a disjunction, as in the example $C = p(x) \vee q(x, y) \vee \neg p(y)$.

A term, atom, literal, clause is called *ground*, if it contains no variables. A ground atom can also be interpreted as a propositional atom, thus a set of ground clauses can be seen as a proposition clause set whose satisfiability can be decided with the appropriate methods.

A substitution *sigma* is a finite mapping from variables to terms. It is continued to a function on terms, atoms, literals, clauses, and clause sets in the obvious way, i.e. by replacing the affected variables by the corresponding terms in these structures. A substitution is called *ground* if it replace variables only by ground terms, and *grounding* for a structure if it maps all variables in that structure to ground term. As an example, $\sigma = \{x \mapsto a, y \mapsto f(a)\}$ is a grounding substitution for the clause C, and $\sigma(X) = p(a) \vee q(a, f(a)) \vee \neg p(a)$.

Our approach is based on the assumption that in saturating provers, the instances necessary for an explicit contradiction may often be generated before they are combined to form the empty clause. Thus, we periodically interrupt the saturation process, apply a grounding substitution to all clauses in the proof state, encode the resulting ground clauses in a format suitable for a propositional prover, and use an efficient propositional method to check the result for unsatisfiability. By Herbrand's theorem, if this set is unsatisfiable, so is the original clause set, and the proof is complete.

If the propositional prover provides an unsatisfiable core, or even a proof object, we can lift this back to the first-order level with a little bit of book-keeping. Figure 1 shows the basic architecture of the system based on this idea.

Note that the likelihood of a proposition conflict being detected depends in part on the nature of the grounding substitutions. Variables in clauses are implicitly universally quantified, so we are free to chose any mapping. Since propositional unsatisfiability is based on conflicting constraints on the same atom, it makes sense to minimize the number of atoms by substituting all variables with the same constant. Going back to the original first-order problem, the normal case is that the axiomatization is satisfiable, i.e. any contradiction must involve clauses from the (negated) conjecture. Thus, it is reasonable to pick the constant used for grounding from those that occur in conjecture clauses.

4 Implementation

To implement the approach described above, we have integrated our theorem prover E and the propositional prover PicoSAT [2].

PicoSAT is a modern SAT solver based on conflict-driven clause learning. It has demonstrated quite good performance, is available under a permissive MIT-style license, and can produce easily parsable proof traces. It is also implemented in C and provided a C API, which makes it a good match for E.

E is a saturating theorem prover based on the superposition calculus [1]. Among its outstanding features are the use of a purely equational representation (non-equational literals are encoded as equations of terms of a special sort), aggressively shared terms (every subterm is represented only once), and a very powerful mechanism to build and describe search heuristics.

The prover implements the DISCOUNT variant of the given-clause algorithm. As described in the the introduction, it saturation core represents the proof state via two sets of clauses, the set P of processed clauses, and the set U of unprocessed clauses. Clause are moved from U to P. For each *given clause* g so moved, all generating inferences where g is at least one premise and all additional premises come from P are performed, and the resulting new clauses are added to U. This maintains the invariant that all generating inferences between clauses in P have been performed. In addition to this main invariant, E also maintains the invariant that clauses in P are maximally interreduced (all simplifying inferences between clauses in P have

Search state: (U, P)
U contains *unprocessed* clauses, P contains *processed* clauses.
Initially, P is empty and all clauses are in U.
The *given clause* is denoted by g.

```
while U ≠ {}
    if prop_trigger(U,P)
        if prop_unsat_check(U,P)
            SUCCESS, Proof found
    g = extract_best(U)
    g = simplify(g, P)
    if g == □
        SUCCESS, Proof found
    if g is not subsumed by any clause in P (or otherwise redundant w.r.t. P)
        P = P\{c ∈ P | c subsumed by (or otherwise redundant w.r.t.) g}
        T = {c ∈ P | c can be simplified with g}
        P = (P\T) ∪ {g}
        T = T ∪ generate(g, P)
        foreach c ∈ T
            c = cheap_simplify(c, P)
            if c is not trivial
                U = U ∪ {c}
SUCCESS, original U is satisfiable
```

Remarks: The basic DISCOUNT loop is printed in black. The SAT solver integration is printed in gray.

Figure 2: The *given-clause* algorithm as implemented in E

been performed), and it simplifies new clauses with respect to P at the time they are created. Fig. 2 shows a sketch of both the original algorithm and the modifications we made in the work described here.

The modification of the existing proof procedure for integrating PicoSAT is quite minimal. At each iteration of the main saturation loop, we test if a *trigger condition* is met. If yes, the full proof state is grounded and handed to a propositional encoder. This encoder converts the problem to a propositional problem and writes it as a file in DIMACS cnf format [5]. It also maintains a mapping from first-order clause to propositional clause and from grounded first-order literal to propositional literal.

The prover then starts PicoSAT (with a fixed time limit) on the generated file and monitors the output via a UNIX pipe. If PicoSAT terminates, the output is analyzed. If PicoSAT has found a model or has timed out, the attempt was unsuccessful (but see the the future work section). If PicoSAT found a proof, it is analyzed and the unsatisfiable core of the propositional problem is extracted. These propositional clauses are mapped back to their respective original first-order clauses, which are then used to construct a (currently quote simple) proof for the empty clause. This is returned to the main proof procedure, which, as always, terminates successfully when encountering the empty clause.

At the heart of the implementation is the grounding module and the encoder/decoder. Both make heavy use of E's aggressively shared term representation.

The grounding module first picks a grounding constant for variables[1]. There are a number of simple strategies. The simplest one picks an arbitrary variable, and interprets it as a constant. The strategy used for the experiments picks the constant that appears with the lowest frequency in the conjecture (with fallbacks if this does not exist or is not unique). Once the grounding term has been determined, the grounding substitution is constructed and at the same time applied to all clauses by mapping all variables to the grounding term. Since variables are shared and variable bindings are recorded in the actual variable structure, this results in a global grounding without the need to explicitly apply the substitution to clauses, literals, or terms.

The encoder simply re-inserts the substituted atoms (encoded as terms) into the shared term bank. This yields a unique pointer and identifier for each ground atom. These markers are re-mapped to the range 1-n and then interpreted as propositional atoms as required in the DIMACS format, which uses positive integers as positive literals, and negative integers as the corresponding negative literals. This mapping from first-order to (integer) propositional literals applied to all clauses, yielding a list of propositional clauses. Each propositional clause also maintains a pointer to the corresponding first-order clause and an implicit sequential number. To reduce the size of the input files, we also implement a simple purity check [3]. We already maintain a list of all ground atoms for translation purposes. We traverse the whole grounded clause set, and annotate each atom with markers indicating its occurrence as a positive or negative literal. Before exporting a propositional clauses, we check if all literals occur with both polarities. Otherwise the clause is pure, and can be ignored.

When PicoSAT returns a proof object, this contains a sequence of numbered propositional clauses with links to the clauses used to derive each clause (none for the axioms). This list is extracted by E, which then uses the (numerical) identifiers of the unsatisfiable core to lift the propositional proof back to the first-order level.

We have so far implemented three different trigger conditions, all based on the amount of work done by the prover. The *ProcInterval* trigger monitors the number of given clauses selected and processed. It triggers a propositional satisfiability check whenever this limit crosses a multiple of the configurable trigger limit. The *GenInterval* trigger similarly monitors the number of newly generated clauses, and triggers a check whenever this limit crosses a multiple of the configurable limit. The last trigger, *TTInsert*, monitors the number of attempts to insert a new term into E's shared term bank data structure. In our experience, this is a reasonably good measure for overall work done, i.e. it correlates well (if not perfectly) with CPU time usage. For this limit, we let the trigger value grow exponentially from the configurable initial limit - more concretely, the limit is doubled after each propositional check.

We have implemented the above in a pre-release version of E 2.1. It will is available for developers via GitHub (`https://github.com/eprover`) now and will be officially released in the very near future via `https://www.eprover.org`.

5 Experimental Results

We have performed a preliminary evaluation of our implementation on the 16048 CNF and FOF problems of the TPTP problem library [19], release 7.0.0. Experiments were run on the StarExec cluster [18], i.e. on machines with an Intel Xeon E5/2.40 GHz processor and at least 128 GB of main memory (enough to ensure that lack of memory is never a concern, given the other parameters of the experiments).

[1] As of E 2.0, the prover supports a many-sorted logic, so this presentation is slightly simplified. In reality, the described process is repeated for each sort.

Strategy	Solutions	Proofs	SAT proofs	No SAT	SAT P/A
WF51/X	7782	6972	0	6972	0
WF51/T	7824	7014	88	6192	822
WF51/P	7831	7022	138	6100	922
Evo/X	9468	8647	0	8647	0
Evo/T	9502	8681	81	7525	1156
Evo/P	9506	8686	109	7225	1461

Table 1: Base performance - solutions and proofs within 300 seconds

We used a total CPU time limit of 300 seconds for the combined system, and a limit of 3 seconds for each individual run of PicoSAT. We are considering data for a total of 6 search strategies - 2 base strategies combined with 3 different ways to determine when to use the SAT solver.

The two base strategies differ in the way the given clause is picked. The first one, *WF51*, uses the conventional interleaving of picking 5 small (by symbol count) clauses, then one old clause (i.e. one from a FIFO queue), and keep repeating this scheme. The second strategy, *Evo*, is the strongest single heuristic we have found so far. It originates from self-optimization of E via genetic algorithms [14] and interleaves a total of 5 selection strategies, 3 different goal-directed strategies, a set-of-support-simulating size-based strategy, and a FIFO queue.

The three different ways to use the SAT solver are not at all, establishing a baseline (denoted by X), every 5000 processed clauses (P), or using the exponentially growing term insertion threshold described above, with a base value of 5000000 insertions for the first SAT solver attempts (T). The 6 strategies are thus *WF51/X*, *WF51/P*, *WF51/T*, *Evo/X*, *Evo/P*, and *Evo/T*.

Table 1 shows the basic performance of the strategies. The *Solutions* column shows the total number of proofs and models found by the respective strategy. Note that the SAT module currently can only contribute to finding proofs. Thus, the *Proofs* column lists the number of proofs found. The next column, *SAT proofs*, shows how many of these proofs were found by the SAT solver, while the next one, *No SAT*, shows the number of proofs found before the first invocation of the SAT solver. *SAT P/A* finally shows the number of proofs found in runs with at least one SAT attempt.

The first observation is that all strategies which use the SAT solver improve the performance of the base strategy. For both base strategies, this effect is marginally bigger for the */P* variant (based on processed clause count) than for the /T variant. The improvement is slightly weaker for the strong *Evo* strategy than for the *WF51* strategy. The relative number of proofs found with the help of PicoSAT is not immediately impressive. However, the main reason for that is that most of the proofs are found before PicoSAT is invoked for the first time. If we only consider proof attempts where PicoSAT was invoked at least once, the SAT solver is responsible for approximately 10-15% of proofs for *WF51* and for 7% for the *Evo* strategies.

Overall, 416 proofs for 211 distinct problems have been found with the help of PicoSAT. 55 of these problems are not solved by either of the base strategies. 3 problems are only solved by non-SAT strategies, probably due to the overhead of the SAT attempts.

Table 2 summarizes the distribution of SAT proofs over TPTP domains. The first data column lists the number of proofs found with the help of the SAT solver in the respective domain. Since we collate data from 4 strategies, there can be up to 4 proofs per problem. Thus, the second data col-um shows the distribution of problems solved with the help of the SAT solver. The last data column restricts this distribution to the 55 problems solved *only* with

TPTP domain	SAT proofs	SAT solved problems	Only SAT solved
ALG	18	10	1
BIO	4	1	1
BOO	3	2	0
COL	3	2	0
CSR	115	72	10
FLD	12	5	0
GEO	5	3	0
GRP	6	5	0
HEN	2	1	0
HWV	42	14	9
LAT	34	12	7
LCL	19	6	4
MSC	3	1	1
NUM	40	17	6
PLA	4	2	1
REL	2	2	0
RNG	8	5	0
SCT	13	6	2
SET	13	8	2
SEU	22	11	3
SWB	9	4	0
SWC	4	2	0
SWV	18	11	4
SWW	12	6	3
SYN	5	3	1

Table 2: SAT solver proofs by TPTP domain

Strategy	SAT attempts	SAT models	Sat proofs	SAT timeouts
WF51/T	2891	2717	88	86
WF51/P	5312	5112	138	62
Evo/T	3653	3476	81	96
Evo/P	8190	7991	109	90
Overall	20046	19296	425	334

Table 3: SAT solver proof attempts

the help of the SAT solver. As we can see, the SAT solver can find proofs in all domains, although the contribution in CSR (large common sense knowledge bases), HWV (hardware verification) and NUM (number theory) stand out. We suspect that for the CSR problems, only minimal instantiation is needed, and the main task is detecting the conflict in the knowledge base. For HWV, often a significant part of the specification is propositional.

Table 3 summarizes the number of attempts by the SAT solver (on successful searches only) and their outcome. The most interesting point here might be that the generated propositional problems seem to be quite simple for PicoSAT - within the 3 second time limit, it managed to decide 98.3% of all problems - the vast majority by finding a model.

Finally, Table 4 shows the statics of the 416 proofs found by PicoSAT. As a first observation,

Statistic	Min	1st q.	Median	3rd q.	Max	Average	Std. dev.
Clauses	3825	65972	160999	296951	2107682	267881.401	+/-357106.458
Non-pure	2	1297	10478	36739	861260	54376.373	+/-120934.703
Unsat core	2	3	4	10	1705	33.683	+/-160.312

Table 4: Statistics of propositional proofs found

we can see that the original problems are quite large, with an average of 267000 input clauses. However, the purity reduction make the problems much easier to handle, reducing them, on average, by about 80%. Moreover, this statistic, like all in this table, is skewed by a few extreme outliers. If we look at more normal cases, purity reduction removes more than 93% of all clauses.

Another interesting measure is the size of the unsatisfiable core, i.e. the number of clauses actually conflicting. Again, the average of 34 clauses is misleading. More than half of the problems have 4 or less clauses in the conflict. This is another indicator that a perfect clause selection heuristic would be able to find these proofs quite easily. All 7 proofs with a core size of 500 or more are generated for only two TPTP problems: `MSC007-1.008.p`, a propositional pigeon-hole problem, and `PLA044-1.p`, an effectively propositional problem translated from QBF.

6 Future Work

Our existing implementation already shows quite promising results. However, there are a number of obvious further avenues of development.

First, our very preliminary evaluation has only covered a small part of the parameter space. Further experiments may give us a better understanding of good instantiation strategies, and of the best way to determine when and how often to perform the propositional satisfiability test. Also, our evaluation has only collected detailed data for successful proof attempts. To get additional insights it would be useful to also collect information for failed proof attempts.

An obvious weakness of our current implementation is that the propositional satisfiability test is not taking into account any equational theory. This could, at least conceptually, be overcome by replacing the SAT solver with a suitable SMT solver, in particular one supporting equality over uninterpreted functions (EUF). Indeed, this might be a way to introduce other forms of theory reasoning into E.

A promising way to further strengthen the approach would be to follow iProver's idea and lift information from propositional models to the first order level, not by explicitly linking clauses with conflicting literals, but by heuristically preferring clauses that invalidate the lifted model.

Finally, there are quality-of-implementation issues to solve. At the moment, E extracts an unsatisfiable core from the prover and lifts it back to the first order level to integrate it into a proof object. It does not, however, construct an actual first-order proof object. Also, the coupling between E and PicoSAT currently takes place via external files and a pipe. It would be both more efficient and more robust to directly link PicoSAT and use it as a library via its documented API.

7 Conclusion

We have described a simple way to integrate propositional reasoning and first-order saturating theorem provers. Our lightweight approach requires only very limited and quite localized changes to the core prover, which makes it quite accessible to other developers.

The experimental results are promising, especially given the limited number of experiments so far. We believe that results can be significantly improved if we explore the parameter space. On the other hand, the fact that most of the found proofs have very small cores, so that a perfect strategy would need only very few steps to complete the proof via saturation, strongly suggests that there still is significant potential for better search heuristics (a point also illustrated from a very different perspective in [17].

While AVATAR probably has a more seamless transition from (unsplittable) first-order to pure propositional problems, our periodic SAT check can also be beneficial in cases where the AVATAR abstraction cannot recover propositional structure.

Finally, we think that there are two exciting lines of research to continue this work. The first is the use of propositional model information to guide the saturation, creating a a new calculus that effectively hybridizes Superposition and Inst-Gen. Secondly the replacement of the purely propositional solver in the architecture by an EUF-SMT solver, or even a more general SMT systems may lead to a lightweight path to integrate theory reasoning and saturation.

Acknowledgements

Many thanks to Laura and Andrei, who not only invited me to write this paper, thus forcing me to order my thoughts on the topic, but who also were unusually patient in waiting for the result.

References

[1] Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[2] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[3] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(1):215–215, 1960.

[4] Martin Davis, Georg Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[5] DIMACS. Satisfiability Suggested Format, 1993.

[6] Harald Ganzinger and Konstantin Korovin. New directions in instantiation-based theorem proving. In *Proc. 18th LICS, Ottawa*, pages 55–64. IEEE Computer Society Press, June 2003.

[7] Konstantin Korovin. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proc. of the 4th IJCAR, Sydney*, volume 5195 of *LNAI*, pages 292–298. Springer, 2008.

[8] Konstantin Korovin. Inst-Gen - A Modular Approach to Instantiation-Based Automated Reasoning. In *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *LNCS*, pages 239–270. Springer, 2013.

[9] S.-J. Lee and D.A. Plaisted. Eliminating Dupliction with the Hyper-Linking Strategy. *Journal of Automated Reasoning*, 9(1):25–42, 1992.

[10] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satis-*

fiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[11] Giles Reger, Martin Suda, and Andrei Voronkov. Playing with AVATAR. In Amy P. Felty and Aart Middeldorp, editors, *Proc. of the 25th CADE, Berlin, Germany*, volume 9195 of *LNAI*, pages 399–415. Springer, 2015.

[12] G. Robinson and L. Wos. Paramodulation and Theorem Proving in First-Order Theories with Equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.

[13] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[14] Simon Schäfer and Stephan Schulz. Breeding theorem proving heuristics with genetic algorithms. In Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov, editors, *Proc. of the Global Conference on Artificial Intelligence, Tibilisi, Georgia*, volume 36 of *EPiC*, pages 263–274. EasyChair, 2015.

[15] Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[16] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[17] Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In Nicola Olivetti and Ashish Tiwari, editors, *Proc. of the 8th IJCAR, Coimbra*, volume 9706 of *LNAI*, pages 330–345. Springer, 2016.

[18] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A Cross-Community Infrastructure for Logic Solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Proc. of the 7th IJCAR, Vienna*, volume 8562 of *LNCS*, pages 367–373. Springer, 2014.

[19] Geoff Sutcliffe. The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[20] Andrei Voronkov. AVATAR: The architecture for first-order theorem provers. In *Proc. of the 26th CAV, Vienna, Austria*, volume 8559 of *LNCS*, pages 696–710. Springer, 2014.

[21] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proc. of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pages 279–285. IEEE Press, 2001.