The Conflict-Driven Answer Set Solver *clasp*: Progress Report

Martin Gebser, Benjamin Kaufmann, and Torsten Schaub[⋆]

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam

Abstract. We summarize the salient features of the current version of the answer set solver *clasp*, focusing on the progress made since version RC4 of *clasp*. Apart from enhanced preprocessing and search-supporting techniques, a particular emphasis lies on advanced reasoning modes, such as cautious and brave reasoning, optimization, solution projection, and incremental solving.

1 Introduction

The solver *clasp* for Answer Set Programming (ASP; [1]) is based upon advanced Boolean constraint solving technology. The theoretical foundations and basic algorithms underlying *clasp* can be found in [2, 3]. It is freely available as open source package at [4]. This paper reports on the progress made since the first system description of *clasp* [5] covering the features of version RC4: it mainly dealt with an empirical evaluation of *clasp*'s features related to conflict-driven nogood learning, comparing various strategies for restarts, nogood deletion, and decision heuristics. In the meantime, *clasp* won the solving categories *SCore* and *SLparse* at the first ASP system competition and is currently participating in the second one. Also, *clasp* qualified for this year's final round of the industrial Satisfiability checking (SAT) competition and competed in SAT-Race 2008 as well as in the 2007 Pseudo-Boolean (PB) evaluation[1].

2 Features

This section describes the major features of version 1.2.1 of *clasp* added since RC4.

Reasoning Modes. As almost all ASP solvers, *clasp* relies on a grounder providing a representation of a propositional logic program. Its major input format is *Lparse* output, provided by either *Lparse* [6] or *Gringo* [7]. Although *clasp*'s primary use case is the computation of a given number of answer sets, it also allows for computing the supported models of a logic program (via command line option `--supp-models`). As detailed below, in either case, options `--cautious` and `--brave` permit computing the intersection and union, respectively, of the respective types of models. Finally, the `--dimacs` option allows for using *clasp* as a SAT solver computing the classical models of a propositional formula supplied in DIMACS format.

[⋆] Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

[1] Thanks to Gayathri Namasivayam and Mirosław Truszczyński, University of Kentucky.

Preprocessing. At the beginning, a propositional logic program is subject to extensive preprocessing [8]. The idea is to simplify a logic program while identifying equivalences among its relevant constituents. These equivalences are then used for building a compact representation of the program (in terms of Boolean constraints). Notably, sometimes preprocessing is able to turn a non-tight program into a tight one (cf. [9]). Preprocessing is configured via option `--eq`, taking an integer value fixing the number of iterations. Once a program has been transformed into a set of Boolean constraints, it is subject to further preprocessing, mostly borrowed from the area of SAT [10]. SAT-based preprocessing is invoked with option `--sat-prepro` and further parameters. However, care must be taken when adapting such techniques from SAT because preprocessing must not eliminate variables that are relevant to the unfounded set checker or that occur in optimize statements or weight rules.

Dedicated Propagation. Not all parts of a logic program are turned into nogoods by *clasp* (in its default setting). Rather *clasp* employs specialized propagation algorithms and has a dedicated implementation for cardinality and weight constraints [11]. These are particular count and sum aggregates offered by *Lparse* and *Gringo*. As detailed in [11], their treatment involves a dedicated, source-pointer-based unfounded set algorithm that computes loop nogoods only on demand, while aiming at lazy unfounded set checking and backtrack-freeness. Although per default all cardinality and weight constraints are subject to dedicated propagation, their treatment can be configured through option `--trans-ext`. Propagation with loop nogoods is influenced by option `--loops` controlling their creation.

Model Enumeration. Different ways of enumerating models are supported by *clasp*. In fact, solution enumeration is non-trivial in the context of backjumping and nogood learning. A popular approach consists in recording solutions as nogoods and exempting them from nogood deletion. Although *clasp* supports this via option `--solution-recording`, it is prone to blow up in space in view of an exponential number of solutions in the worst case. Unlike this, the default enumeration algorithm of *clasp* runs in polynomial space [3]. Both approaches also allow for projecting solutions on a subset of atoms [12]; invoked with `--project` and configured via the well-known directives `#hide` and `#show` of *Lparse* and *Gringo*. For example, the program consisting of the choice rule `{a,b,c}.` has eight (obvious) answer sets. When augmented with directive `#hide c.`, still eight solutions are obtained, yet including four duplicates. Unlike this, invoking *clasp* with `--project` yields only four duplicate-free solutions. This option is of great practical value whenever one faces overwhelmingly many answer sets, involving solution-irrelevant variables having a proper combinatorics. As regards implementation, it is interesting to note that *clasp* offers a dedicated interface for enumeration. This allows for abstracting from how to proceed once a model was found and thus makes the search algorithm independent of the concrete enumeration strategy. One further strategy implemented via the enumeration interface consists of computing the intersection or union of all answer sets of a program (via `--cautious` and `--brave`, respectively). Rather than computing a set of (possibly) exponentially many answer sets, the idea is to compute a first answer set, record a constraint eliminating it from further solutions, then compute a second answer set, strengthen the constraint to represent the intersection (or union) of the first two answer sets, and to continue in this way until no more answer sets are obtained. This process involves computing at most as many answer sets as there are atoms in the input program. Either the cautious or the brave consequences are then given by the atoms captured by the final constraint.

Optimization. Another application-oriented feature is optimization. As common in *Lparse*-like languages, an objective function is specified by a sequence of `#minimize` and `#maximize` statements. For finding optimal solutions, *clasp* offers several options. First, *clasp* allows for computing one or all (`--opt-all`) optimal solutions. Second, the objective function can be initialized via `--opt-value`. The latter turns out to be useful when one is interested in computing consequences belonging to all optimal solutions (in combination with `--cautious`). One starts with a search for an optimum and then re-launches *clasp* by bounding its search with the value of the optimum. Doing the latter with `--cautious` yields all consequences true in all optimum answer sets. On applications, it turned out to be very useful to optimize using the option `--restart-on-model` (making *clasp* restart after each (putative) minimum solution) in order to ameliorate the convergence to an optimum solution. Again, optimization is implemented via the aforementioned enumeration interface. When a solution is found, the optimization constraint is updated by the corresponding value. Then, the decision level invalidating the updated constraint is identified and backtracked; if the constraint is violated on the top-level, search terminates. Furthermore, it is worth mentioning that *clasp* also propagates over optimization statements. For this, optimization statements are themselves stored as Boolean constraints [5] in the solver. As such, they can derive (and provide reasons for) implications during unit propagation.

Restarts. The robustness of *clasp* is boosted by advanced restart strategies. Apart from the policies already discussed in [5], namely, geometric, fixed-interval, and Luby-style policies, a nested policy, first used in *picosat* [13], is meanwhile also offered by *clasp*. This policy takes three parameters x, y, z and makes restarts follow a two dimensional pattern that increases geometrically in both dimensions. The geometric restart sequence $x * y^i$ is repeated when it reaches an outer limit $z * y^j$, where i counts the number of restarts and j how often the outer limit was hit so far. Usually, restart strategies as listed above are based on a global number of conflicts. Moreover, *clasp* features local restarts [14]. Here, one counts the number of conflicts at each decision level in order to localize the measure of difficulty. For this, we maintain a counter $c(d)$ for each decision level d. When a new decision level d is created, $c(d)$ is set to the global number of conflicts. When backtracking to level d, a restart is only initiated if the difference between the global number of conflicts and $c(d)$ is now larger than the strategy-dependent threshold. It is worth noting that despite the fact that recent SAT solvers use rather aggressive restart strategies (cf. Section 3), *clasp* still defaults to a more conservative geometric policy because this performs better on our ASP-specific benchmarks.

Progress Saving. Another search-related feature of *clasp* is progress saving, as described in [15]. The idea is as follows. On backjumping (or restarting), the values of variables whose assignment is about to be erased are saved for all but those variables assigned on the last decision level. The saved values are then used during decision making. That is, when a variable for which a value was saved is selected by the decision heuristic, it is assigned to that value. The intuition behind this strategy is that the assignments made prior to the last decision level did not lead to a conflict and may have satisfied some subproblem. Hence, repeating those assignments may help to avoid solving subproblems multiple times. Progress saving is invoked with option `--save-progress`; its computational impact depends heavily on the structure of the application at hand.

Application Programming Interface. A major yet internal feature of *clasp* is that it can be used in a stateful way. That is, *clasp* may keep its state, involving program representation, learned constraints, heuristic values, etc, and be invoked under additional (temporary) assumptions and/or by adding new atoms and rules. The corresponding interfaces are fundamental for supporting incremental ASP solving as realized in *iClingo* [16], a combination of *Gringo* and *clasp* for incremental grounding and solving. Furthermore, they allow for solving under assumptions [17]; an important feature that is, for example, used in our parallel ASP solver *claspar* [18].

3 Fine-Tuning

Advanced Boolean constraint solving technology adds a multitude of degrees of freedom to ASP solving. For instance, currently, *clasp* has roughly 40 options, half of which control the search strategy. Although considerable efforts were taken to find default parameters for optimizing robustness and speed, the default setting still leaves room for drastic improvements on specific benchmark classes by fine-tuning the parameters. The question arises how to deal with this vast "configuration space" and how to conciliate it with the idea of declarative problem solving. Currently, there seems to be no alternative to manual fine-tuning when addressing highly demanding application problems.

As rules of thumb, we usually start by investigating the following options:

- `--heuristic`: Try *VSIDS* instead of *clasp*'s default *BerkMin*-style heuristic.
- `--sat-prepro`: SAT-based preprocessing works best on tight programs with few cardinality and weight constraints. It should (almost) always be used if extended rules are transformed into nogoods (via `--trans-ext`).
- `--restarts`: Try aggressive restart policies, like *Luby-256* or the *nested policy*, or try disabling restarts, whenever a problem is deemed to be unsatisfiable.
- `--save-progress`: Progress saving typically works nicely if the average backjump length (or the #choices/#conflicts ratio) is high ($\geq$10). It usually performs best if combined with aggressive restarts.
- `--trans-ext`: Applicable if the program contains extended rules, that is, rules including cardinality and weight constraints. Try at least the *dynamic* transformation.

The impact of simple fine-tuning can be seen on the following examples. As shown in [19], *clasp* times out on satisfiable *4-Coloring* problems. However, with `--save-progress`, *clasp* solves all instances in less than 2 sec (the average backjump length is $>$60). For another example, consider the benchmark *WeightBounded-DominatingSet* from the second ASP competition. The default configuration of *clasp* results in six timeouts, all of which vanish once aggressive restarts are used. Similar effects are observed on application problems featuring yet different characteristics.

Although such fine-tuning may greatly improve the efficiency of *clasp*, it is hard to accomplish for an unpracticed user, and after all it takes us away from the ideals of declarative problem solving. To this end, we advocate an extension of *clasp*, called *claspfolio*, that maps benchmark features to solver configurations (via machine learning techniques). It is interesting future work to see whether this allows for an automatic selection of effective parameter settings.

4 Discussion

Since its inception in 2007, *clasp* has become an efficient, full-fledged ASP solver. Beyond its computational power, it meanwhile features various reasoning modes that make it an attractive tool for knowledge representation and reasoning. This is witnessed by an increasing number of applications relying on *clasp* or derivatives as reasoning engine, e.g., [20–25]. *clasp* constitutes a central component of *Potassco*, the Potsdam Answer Set Solving Collection bundling tools for Answer Set Programming developed at the University of Potsdam. An extension of *clasp*, called *claspD* [26], allows for dealing with disjunctive ASP programs. Meanwhile, the family has grown and two new systems, *Clingo* and *iClingo* [16], have emerged. *Clingo* is a monolithic combination of *clasp* and *Gringo*. *iClingo* is an ASP system that allows for dealing incrementally with parametrized problems, as encountered for instance in bioinformatics, planning, and model checking. The latest addition to the family is *Clingcon* [27], augmenting *Clingo* with (non-Boolean) constraint processing capacities. Also, there is a distributed version of *clasp*, called *claspar* [18], designed for running on clusters with MPI. Sources (and binaries) of our systems are publicly available at [4].

Acknowledgments. This work was partially funded by DFG under Grant SCHA 550/8-1 and by the GoFORSYS project under Grant 0313924.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M., ed.: Proc. of IJCAI'07, AAAI Press (2007) 386–392
3. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. [28] 136–148
4. `http://potassco.sourceforge.net/`
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A conflict-driven answer set solver. [28] 260–265
6. Syrjänen, T.: Lparse 1.0 user's manual.
7. Gebser, M., Schaub, T., Thiele, S.: GrinGo: A new grounder for answer set programming. [28] 266–271
8. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Advanced preprocessing for answer set solving. In Ghallab, M., Spyropoulos, C., Fakotakis, N., Avouris, N., eds.: Proc. of ECAI'08, IOS Press (2008) 15–19
9. Babovich, Y., Lifschitz, V.: Computing answer sets using program completion. (2003)
10. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: Proc. of SAT'05, Springer (2005) 61–75

11. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: On the implementation of weight constraint rules in conflict-driven ASP solvers. [29] 250–264
12. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected Boolean search problems. In van Hoeve, W., Hooker, J., eds.: Proc. of CPAIOR'09, Springer (2009) 71–86
13. Biere, A.: PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation **4** (2008) 75–97
14. Ryvchin, V., Strichman, O.: Local restarts. In Kleine Büning, H., Zhao, X., eds.: Proc. of SAT'08, Springer (2008) 271–276
15. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In Marques-Silva, J., Sakallah, K., eds.: Proc. of SAT'07, Springer (2007) 294–299
16. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. [30] 190–205
17. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4) (2003)
18. Ellguth, E., Gebser, M., Gusowski, M., Kaminski, R., Kaufmann, B., Liske, S., Schaub, T., Schneidenbach, L., Schnor, B.: A simple distributed conflict-driven answer set solver. In Erdem, E., Lin, F., Schaub, T., eds.: Proc. of LPNMR'09, Springer (2009) To appear
19. Janhunen, T., Niemelä, I., Sevalnev, M.: Computing stable models via reductions to Boolean circuits and difference logic. In Denecker, M., ed.: Proc. of LaSh'08, (2008) 16–30
20. Mileo, A., Merico, D., Bisiani, R.: A logic programming approach to home monitoring for risk prevention in assisted living. [30] 145–159
21. Boenn, G., Brain, M., de Vos, M., Fitch, J.: Automatic composition of melodic and harmonic music by answer set programming. [30] 160–174
22. Gebser, M., Schaub, T., Thiele, S., Usadel, B., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. [30] 130–144
23. Ishebabi, H., Mahr, P., Bobda, C., Gebser, M., Schaub, T.: Answer set vs integer linear programming for automatic synthesis of multiprocessor systems from real-time parallel programs. Journal of Reconfigurable Computing (2009) To appear
24. Kim, T., Lee, J., Palla, R.: Circumscriptive event calculus as answer set programming. In Boutilier, C., ed.: Proc. of IJCAI'09, AAAI Press (2009) To appear
25. Thielscher, M.: Answer set programming for single-player games in general game playing. [29] To appear
26. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-driven disjunctive answer set solving. In Brewka, G., Lang, J., eds.: Proc. of KR'08, AAAI Press (2008) 422–432
27. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. [29] 235–249
28. Baral, C., Brewka, G., Schlipf, J., eds.: Proc. of LPNMR'07. Springer (2007)
29. Hill, P., Warren, D., eds.: Proc. of ICLP'09. Springer (2009)
30. Garcia de la Banda, M., Pontelli, E., eds.: Proc. of ICLP'08. Springer (2008)