Evaluation Techniques and Systems for Answer Set Programming: A Survey

Martin Gebser,[1] **Nicola Leone,**[2] **Marco Maratea,**[3] **Simona Perri,**[2]
Francesco Ricca[2] and **Torsten Schaub**[1]
[1] University of Potsdam, Germany
[2] University of Calabria, Italy
[3] University of Genova, Italy

Abstract

Answer set programming (ASP) is a prominent knowledge representation and reasoning paradigm that found both industrial and scientific applications. The success of ASP is due to the combination of two factors: a rich modeling language and the availability of efficient ASP implementations. In this paper we trace the history of ASP systems, describing the key evaluation techniques and their implementation in actual tools.

1 Introduction

Answer Set Programming (ASP) [Brewka *et al.*, 2011] is a well-known approach to knowledge representation and reasoning, with roots in the areas of logic programming and non-monotonic reasoning [Gelfond and Lifschitz, 1991]. ASP is in close relationships to other formalisms such as Propositional Satisfiability (SAT), Satisfiability Modulo Theories (SMT), Quantified Boolean Formulas (QBF), Constraint Programming (CP), Planning and Scheduling, and many others.

The language of ASP allows for defining solutions to complex problems in a purely declarative way. The main construct is a rule, i.e., an expression of the form $Head \leftarrow Body$, where $Body$ is a logic conjunction possibly involving negation, and $Head$ is, in the basic formulation, either an atomic formula or a logic disjunction. ASP programs, i.e., collection of rules, are first order theories that can model uniformly solutions for a given problem over varying instances. The semantics of ASP programs is given in terms of answer sets (or stable models) [Gelfond and Lifschitz, 1991]. The basic language introduced in [Gelfond and Lifschitz, 1991] has been extended over the years by introducing many new constructs [Calimeri *et al.*, 2005; Gebser *et al.*, 2015a], that further simplify the modeling of complex problems. A standardized syntax for ASP has been then introduced in [Calimeri *et al.*, 2012].

The readability and expressiveness of the language combined with the availability of efficient implementations made ASP one of the major declarative paradigms for logic-based Artificial Intelligence (AI), suitably employed for realizing many AI applications [Erdem *et al.*, 2016].

The research of efficient evaluation techniques for the language of ASP started during the 90-ties [Lierler *et al.*, 2016], after a first period in which complexity arguments discouraged the implementation of systems. Indeed, the first serious attempts appeared about 10 years after the introduction of the stable model semantics. The subsequent introduction of new evaluation techniques, often fruit of cross-fertilization with the neighboring communities of SAT and CP, resulted in the design of more and more efficient ASP systems.

The advancement of the state of the art in ASP solving was consistent and continuous in the last few years, as witnessed by the results of the biannual ASP Competitions series (see [Calimeri *et al.*, 2016; Gebser *et al.*, 2017b] for the last events). This paper traces this history surveying the major contributions to the development of evaluation techniques and solvers for ASP that made it one of the most attractive paradigms of logic-based AI.

2 The History of ASP Evaluation

Almost all ASP implementations follow a two-step evaluation process [Kaufmann *et al.*, 2016]. The first step, called *grounding*, takes an ASP program as input and produces an equivalent variable-free (or *ground*) program that has the same answer sets as the input program. Such a ground program is the input of the second step of the process, called *solving*, having the role of searching for one or more answer sets, which coincide with the ones of the non-ground program.

In the following we first present the evolution of grounders, followed by the history of solvers; then, we mention the approaches blending existing techniques in portfolios, those capable of exploiting multi-processing, and, finally, we review some (relatively new) attempts deviating from the customary two-steps evaluation, called *grounding-less* systems, where grounding and solving are somehow interleaved.

2.1 Grounders

Grounders perform a complex task that may have a big impact on the performance of the whole system, as their output is the input for the solving phase: if input non-ground programs can be assumed to be fixed (data complexity), grounding is polynomial; however, as soon as variable programs are given in input the produced ground program is potentially of exponential size with respect to the input program. Thus, grounders employ smart procedures that are geared toward efficiently producing a ground program that preserves the semantics but is sensibly smaller than the one that could be obtained by a simple replacement of the variables with all the constants of the program.

One of the first released grounders was LPARSE [Syrjänen, 2001], a front-end grounder system, whose output encoded in a suitable numeric format, is intended to be given as input to a separated solver. LPARSE accepts logic programs respecting its ω-restrictedness condition, i.e. it enforces each variable in a rule to occur in a positive body literal, called domain literal, whose predicate (i) is not mutually recursive with the head, and (ii) is neither unstratified nor dependent (also transitively) on an unstratified predicate. A similar approach is used in the earlier versions of GRINGO [Gebser *et al.*, 2007], that bind non-global variables by domain predicates to enforce a λ-restrictedness condition, an extension of ω-restrictedness. Essentially, these restrictions are used in order to treat positive recursion among predicates, and guarantee to have a finite grounding.

The grounder of the DLV system [Faber *et al.*, 2012], and the more recent versions of GRINGO (starting from 3.0) instead imposes the less restrictive condition of *safety*, requiring that each variable in a rule appears in some positive body literal. These grounders are based on semi-naive database evaluation techniques [Ullman, 1988] for avoiding duplicate work during grounding. Grounding is seen as an iterative bottom-up process guided by the successive expansion of a program's term base, that is, the set of variable-free terms constructible from the signature of the program at hand.

Recently a brand new version of the DLV grounder has been released, as the stand-alone grounder I-DLV [Calimeri *et al.*, 2017]. The new grounder relies on the theoretical foundations of its predecessor but has been completely redesigned, integrating new optimizations, and usability features such as, annotations, python interface, interoperability mechanisms.

2.2 Solvers

The search for (optimal) answer sets (i.e., ASP solving) basically amounts to Boolean constraint solving. This task was pioneered in the area of SAT by the classical Davis-Putnam-Logemann-Loveland (DPLL) procedure [Davis *et al.*, 1962]; more recently, it has been solved by resorting to the Conflict-Driven Clause Learning (CDCL) [Marques-Silva and Sakallah, 1999] algorithm. Similarly, the so-called *native* answer set solvers are based on the same backtracking procedures used in SAT solving, yet refined to the semantics and additional constructs of ASP programs. Alternatively, *translation-based* solvers can be obtained by applying transformations to ground programs to obtain instances of other formalisms featuring efficient solvers, and exploiting these solvers for computing answer sets.

Native Approaches. The first developed ASP solvers, namely DLV [Leone *et al.*, 2006] and SMODELS [Simons *et al.*, 2002], were pioneered in the late '90s and pursued "native" approaches based on the classical DPLL procedure. While DPLL augments basic backtracking search with unit propagation on clauses, DLV and SMODELS adjust such techniques to ASP programs. In particular, this includes dedicated inference mechanisms to detect and falsify so-called unfounded sets [Van Gelder *et al.*, 1991; Leone *et al.*, 1997], which particularly address positive recursion in case of non-tight programs. Later on, DLV was extended with backjumping techniques [Ricca *et al.*, 2006], in place of basic backtracking, and look-back heuristics [Maratea *et al.*, 2008] that take advantage of the backjumping process [Prosser, 1993]. Similarly, the SMODELSCC solver [Ward and Schlipf, 2004] extended the algorithm of SMODELS with backjumping, while further adding mechanisms for conflict-driven clause learning, as pioneered by CDCL in the area of SAT.

The second generation of native ASP solvers, including CLASP [Gebser *et al.*, 2012a] and WASP [Alviano *et al.*, 2015a], integrates CDCL-style search with propagation principles dedicated to ASP programs. Implementation features shared with modern SAT solvers include, e.g., watched literals, activity-based heuristics, and rapid restarts [SAT, 2009]. Such basic features are accompanied by techniques for dealing with unfounded sets, aggregates, and optimization in order to cover the range of modeling concepts and computational tasks available in ASP [Kaufmann *et al.*, 2016]. While the IDP system [Bruynooghe *et al.*, 2015] has been conceived as a model generator for first-order theories extended with inductive definitions, it has much in common with the aforementioned ASP grounders and solvers. That is, it includes a grounder, GIDL [Wittocx *et al.*, 2008], a solver, MINISATID [Mariën *et al.*, 2008], and handles (positive) recursion among atoms in inductive definitions.

For solving a program P which is not-Head-Cycle-Free (non-HCF), i.e. where checking whether a model of P is $\subseteq$-minimal w.r.t. P^I is in general coNP-complete, native ASP solvers employ a two-level architecture in which DPLL- or CDCL-style search is used for (i) generating candidate models of P and (ii) checking for the existence of smaller counter-models of P^I. To this end, the propagation principles of the first respective solver, DLV, are capable of handling disjunctive rules [Calimeri *et al.*, 2006], while the check for counter-models is delegated to a SAT solver. The GNT system [Janhunen *et al.*, 2006] pursues a corresponding approach by casting the tasks of generating and checking candidate models to ASP programs processed with separate instances of SMODELS. Similarly, CLASP (or its nowadays deprecated sibling CLASPD) and WASP couple complementary instances of their CDCL-style search engines to perform model generation or checking, respectively. Both CLASP and WASP encode the checking task for arbitrary candidate models [Gebser *et al.*, 2013; Alviano *et al.*, 2015b], and perform checking via assumption-based reasoning [Eén and Sörensson, 2003].

Translation-Based Approaches. From Clark (1978), we know that answer sets of a tight program [Erdem and Lifschitz, 2003] P coincide with propositional models of P's completion the answer sets of tight programs can be computed by running SAT solvers. By relying on this result, the first version of the SAT-based solver CMODELS [Giunchiglia *et al.*, 2006] was based on this correspondence. As a generalization to the non-tight case, [Lin and Zhao, 2004] proposed loop formulas whose addition to a program's completion establishes correspondence between propositional models and answer sets. Since the number of required loop formulas can be exponential [Lifschitz and Razborov, 2006], the

SAT-based solvers ASSAT [Lin and Zhao, 2004] and CMODELS, from its second version on, add loop formulas incrementally to eliminate models that are no answer sets. In fact, loop formulas deny unfounded sets [Lee, 2005], which are also handled by native systems, so that there is a close proximity between native and SAT-based solvers utilizing loop formulas, and both kinds of systems are based on similar search procedures. This also carries forward to non-HCF programs [Lee and Lifschitz, 2003], where the third version [Lierler, 2005] of CMODELS utilizes SAT solvers also for stability checking.

The translation by LP2SAT [Janhunen, 2006; Janhunen and Niemelä, 2011], instead, is based on so-called level rankings [Niemelä, 2008] to check $\subseteq$-minimality w.r.t. the reduct of an HCF program in the non-tight case. Such level rankings are encoded a priori, rather than incrementally, and expressing them in SAT requires sub-quadratic instead of exponential space. Technically, the tool LP2ACYC [Gebser *et al.*, 2014a] instruments an ASP program such that propositional models of its completion subject to an acyclicity condition match the answer sets of the program. The required acyclicity can then be established via level rankings, where linear representations are feasible in several target formalisms, including ASP, Pseudo-Boolean Constraints/Optimization, or SAT Modulo Acyclicity [Bomanson *et al.*, 2016; Gebser *et al.*, 2014a], SMT with Difference or Bit-Vector Logic [Janhunen *et al.*, 2009; Nguyen *et al.*, 2011], and Mixed Integer Programming [Liu *et al.*, 2012]. In fact, the translation-based systems participating in the Sixth ASP Competition [Gebser *et al.*, 2017c] are based on this infrastructure, while SAT-based solvers utilizing loop formulas have come out of fashion.

2.3 Portfolio Approaches

Automated algorithm selection techniques [Rice, 1976] aim at robustness across a range of heterogeneous inputs. Inspired by SATZILLA in the area of SAT, the CLASPFOLIO system [Gebser *et al.*, 2011a; Hoos *et al.*, 2014] uses support vector regression to learn scoring functions approximating the performance of several CLASP variants in a training phase. Given an instance, CLASPFOLIO then extracts features and evaluates such functions in order to pick the most promising CLASP variant for solving the instance. This algorithm selection approach was particularly successful in the Third ASP Competition [Calimeri *et al.*, 2014], held in 2011, where CLASPFOLIO won the first place in the NP category and the second place overall (without participating in the Beyond-NP category). The ME-ASP system [Maratea *et al.*, 2014; 2015b] goes beyond the solver-specific setting of CLASPFOLIO and chooses among different grounders as well as solvers. Grounder selection traces back to [Maratea *et al.*, 2013], and similar to the QBF solver AQME [Pulina and Tacchella, 2009], ME-ASP uses a classification method for performance prediction. Notably, “bad” classifications can be treated by adding respective instances to the training set of ME-ASP [Maratea *et al.*, 2015a], which enables an adjustment to new problems or instances thereof. In the Seventh ASP Competition [Gebser *et al.*, 2017a], the winning system was I-DLV+S that utilizes I-DLV for grounding and automatically selects back-ends for solving through classification between CLASP and WASP.

Going beyond the selection of a single solving strategy from a portfolio, the ASPEED system [Hoos *et al.*, 2015] indeed runs different solvers, sequentially or in parallel, as successfully performed by PPFOLIO in the 2011 SAT Competition. Given a benchmark set, a fixed time limit per instance, and performance results for candidate solvers, the idea of ASPEED is to assign time budgets to the solvers such that a maximum number of instances can be completed within the allotted time. In other words, the goal is to divide the total runtime per computing core among solvers such that the number of instances on which at least one solver successfully completes its run is maximized. The portfolio then consists of all solvers assigned a non-zero time budget along with a schedule which solvers to run on the same computing core. Calculating such an optimal portfolio for a benchmark set is an Optimization problem addressed with ASP in ASPEED.

2.4 Multi-processing in ASP Systems

We below describe approaches taking advantage of multi-core/processor support during both grounding and/or solving.

Concerning grounding, parallel grounding techniques were developed as extensions of LPARSE [Balduccini *et al.*, 2005] and the DLV grounder [Perri *et al.*, 2013]. The former approach was designed for distributing LPARSE incarnations, working in local memory, on Beowulf clusters, while the latter aims at shared-memory parallelism on multi-core/processor machines. In particular, the parallel version of the DLV grounder allows for a concurrent instantiation at several levels: it allows for instantiating in parallel subprograms of the program in input, rules within a given subprogram, and also for parallelizing the evaluation of a single rule. This is accompanied by techniques for granularity control and dynamic load balancing to achieve an efficient parallelization.

Concerning ASP solvers, these have been also extended to exploit multi-core/multi-processor machines by introducing parallel evaluation methods. The first approaches in this direction [Finkel *et al.*, 2001; Pontelli *et al.*, 2003; Balduccini *et al.*, 2005; Gressmann *et al.*, 2006] were based on SMODELS and divided its DPLL-style search among cluster machines or multiple threads, primarily using guiding paths [Zhang *et al.*, 1996], pioneered in the area of SAT, to process separate subproblems in parallel. Cluster and multi-threaded versions of the CDCL-based solver CLASP [Gebser *et al.*, 2011b; 2012b], however, turned out to be particularly successful when applying a parallel portfolio of CLASP variants to a common problem. Recent versions of CLASP [Gebser *et al.*, 2015b] further extend the multi-threaded search infrastructure to non-HCF programs [Gebser *et al.*, 2013]. The aforementioned ASPEED system allows for scheduling solver runs on multiple computing cores. Also, translation-based systems may readily exploit parallelism provided by respective back-end solvers, as done in the 2013 and 2014 editions of the ASP Competition. Finally, [Dovier *et al.*, 2016] provide an approach to parallel CDCL-style ASP solving utilizing GPUs.

2.5 Grounding-less ASP Systems

In contraposition with the traditional ground&solve approach, systems such as *Gasp* [Dal Palù *et al.*, 2009], *Asperix* [Lefèvre *et al.*, 2017], and *Omiga* [Dao-Tran *et al.*, 2012] perform a *lazy grounding* technique in which grounding and solving steps are interleaved, and rules are grounded on-demand during solving. These systems try to overcome the so called *grounding bottleneck*, that occurs on problems for which the instantiation is inherently so huge that the traditional approach is not suitable, and this occurs also when problems can be solved in polynomial space [Eiter *et al.*, 2007]. However, in general, they suffer from poor search performance since they do not perform conflict-driven learning and backjumping for reducing the search space. Recently, a new ASP system, namely *Alpha* [Weinzierl, 2017], has been developed aiming at blending lazy-grounding and learning of non-ground clauses. Finally, [de Cat *et al.*, 2015] proposed *lazy model expansion* within the FO(ID) formalism supporting partially lazy instantiation of constraints.

3 System Interfaces

Given that the implementations of ASP systems are nowadays highly optimized and sophisticated, modifying a system to support a richer language or dedicated reasoning techniques is non-obvious. In order to facilitate making such extensions, modern ASP systems like CLINGO [Gebser *et al.*, 2014b; 2016] and WASP [Alviano *et al.*, 2013; 2015a] provide APIs.

The interface of CLINGO offers customized control about grounding and solving, where specific routines can be developed in *C* as well as the scripting languages *lua* and *python*. One major target application is *incremental solving* [Gebser *et al.*, 2008], where parts of an ASP program are successively (re)grounded and search is repeatedly performed w.r.t. the growing ground instantiation. Corresponding techniques have been successfully utilized in areas like automated planning [Dimopoulos *et al.*, 2017] and finite model finding [Gebser *et al.*, 2011c], where the horizon needed for a problem solution is a priori unknown or its theoretical bound prohibitively large, respectively. The second main application of CLINGO's interface consists of its extension by *external propagators* [Drescher and Walsh, 2012], implementing customized reasonings on top of the basic CDCL-style search.

Along the same lines, the interface of WASP allows for equipping its search procedure with external propagators, which can be implemented in *C++*, *perl* and *python*, for extending the basic reasoning capabilities. As demonstrated in [Cuteri *et al.*, 2017], the incorporation of propagators can overcome the *grounding bottleneck* on several combinatorial problems. Moreover, WASP's interface supports the integration of custom *search heuristics* [Dodaro *et al.*, 2016], and respective methods were successfully applied to the industrial Partner Units [Aschinger *et al.*, 2011] and Combined Configuration [Gebser *et al.*, 2015c] problems. A first attempt of combining I-DLV and WASP in a monolithic design has been described in [Alviano *et al.*, 2017]. The resulting project is called DLV2, and combines the python interface of WASP with the one offered by I-DLV with the aim of easier the development of propagators or heuristics.

4 Systems for ASP Extensions

Advanced frameworks that extend the common syntax and semantics of ASP programs include Constraint ASP (CASP) [Baselice *et al.*, 2005; Banbara *et al.*, 2017; Balduccini and Lierler, 2017], ASP Modulo Theories (ASPMT) [Susman and Lierler, 2016], Bound-Founded ASP (BFASP) [Aziz *et al.*, 2013], and Higher-order EXternal (HEX) programs [Eiter *et al.*, 2016]. The idea of CASP systems like ACSOLVER, CLINGCON, EZCSP, IDP, INCA, and MINGO, whose key features are analyzed and compared in [Lierler, 2014], is to augment the Boolean problem representations of ASP with constraints over multi-valued variables in order to compactly formulate quantitative conditions. Corresponding implementation techniques are inspired by SAT Modulo Theories (SMT) solvers, and ASPMT systems like ASPMT2SMT and EZSMT furnish translations to exploit SMT solvers as search backends. The BFASP system CHUFFED extends a Constraint Programming (CP) solver with a propagator for bound-founded integer variables that default to either the smallest or largest value available in their domains, thus generalizing the minimality condition on answer sets from Boolean to multi-valued variables. HEX programs allow for equipping ASP programs with external sources of knowledge and/or computation, and the DLVHEX system features techniques to integrate the evaluation of external sources into the search of ASP solvers. Application domains making use of ASP extensions as furnished by the CASP, ASPMT, BFASP, and HEX frameworks include, e.g., hybrid task and motion planning in robotics, time scheduling, as well as ontological reasoning.

5 Conclusion

The research of techniques for implementing the stable model semantics started about twenty years ago. This paper retraces this history surveying the major contributions that made ASP one of the most attractive paradigms of logic-based AI.

References

[Alviano *et al.*, 2013] M. Alviano, C. Dodaro, W. Faber, N. Leone, and F. Ricca. WASP: A native ASP solver based on constraint learning. In *LPNMR13*, volume 8148 of *LNCS*, pages 54–66. Springer, 2013.

[Alviano *et al.*, 2015a] M. Alviano, C. Dodaro, N. Leone, and F. Ricca. Advances in WASP. In *LPNMR15*, volume 9345 of *LNCS*, pages 40–54. Springer, 2015.

[Alviano *et al.*, 2015b] M. Alviano, C. Dodaro, and F. Ricca. Reduct-based stability check using literal assumptions. In *ASPOCP'15*, 2015.

[Alviano *et al.*, 2017] M. Alviano, F. Calimeri, C. Dodaro, D. Fuscà, N. Leone, S. Perri, F. Ricca, P. Veltri, and J. Zangari. The ASP system DLV2. In *LPNMR 2017*, volume 10377 of *LNCS*, pages 215–221. Springer, 2017.

[Aschinger *et al.*, 2011] M. Aschinger, C. Drescher, G. Friedrich, G. Gottlob, P. Jeavons, A. Ryabokon, and E. Thorstensen. Optimization methods for the partner units problem. In *CPAIOR'11*, volume 6697 of *LNCS*, pages 4–19. Springer, 2011.

[Aziz *et al.*, 2013] R. A. Aziz, G. Chu, and P. J. Stuckey. Stable model semantics for founded bounds. *TPLP*, 13(4-5):517–532, 2013.

[Balduccini and Lierler, 2017] M. Balduccini and Y. Lierler. Constraint answer set solver EZCSP and why integration schemas matter. *TPLP*, 17(4):462–515, 2017.

[Balduccini *et al.*, 2005] M. Balduccini, E. Pontelli, O. El-Khatib, and H. Le. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing*, 31(6):608–647, 2005.

[Banbara *et al.*, 2017] M. Banbara, B. Kaufmann, M. Ostrowski, and T. Schaub. Clingcon: The next generation. *TPLP*, 17(4):408–461, 2017.

[Baselice *et al.*, 2005] S. Baselice, P. A. Bonatti, and M. Gelfond. A preliminary report on integrating of answer set and constraint solving. In *Answer Set Programming*, volume 142 of *CEUR Workshop Proceedings*, 2005.

[Bomanson *et al.*, 2016] J. Bomanson, M. Gebser, T. Janhunen, B. Kaufmann, and T. Schaub. Answer set programming modulo acyclicity. *Fund. Infor.*, 147(1):63–91, 2016.

[Brewka *et al.*, 2011] G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[Bruynooghe *et al.*, 2015] M. Bruynooghe, H. Blockeel, B. Bogaerts, B. De Cat, S. De Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker, and S. Verwer. Predicate logic as a modeling language: Modeling and solving some machine learning and data mining problems with IDP3. *TPLP*, 15(6):783–817, 2015.

[Calimeri *et al.*, 2005] F. Calimeri, W. Faber, N. Leone, and S. Perri. Declarative and Computational Properties of Logic Programs with Aggregates. In *IJCAI-05*, pages 406–411, 2005.

[Calimeri *et al.*, 2006] F. Calimeri, W. Faber, N. Leone, and G. Pfeifer. Pruning operators for disjunctive logic programming systems. *Fund. Infor.*, 71(2-3):183–214, 2006.

[Calimeri *et al.*, 2012] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. ASP-Core-2: Input language format. `https://www.mat.unical.it/aspcomp2013/ASPStandardization/`, 2012.

[Calimeri *et al.*, 2014] F. Calimeri, G. Ianni, and F. Ricca. The third open answer set programming competition. *TPLP*, 14(1):117–135, 2014.

[Calimeri *et al.*, 2016] F. Calimeri, M. Gebser, M. Maratea, and F. Ricca. Design and results of the fifth answer set programming competition. *Artif. Intell.*, 231:151–181, 2016.

[Calimeri *et al.*, 2017] F. Calimeri, D. Fuscà, S. Perri, and J. Zangari. I-DLV: the new intelligent grounder of DLV. *Intell. Artif.*, 11(1):5–20, 2017.

[Clark, 1978] K. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[Cuteri *et al.*, 2017] B. Cuteri, C. Dodaro, F. Ricca, and P. Schüller. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *TPLP*, 17(5-6):780–799, 2017.

[Dal Palù *et al.*, 2009] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. GASP: Answer set programming with lazy grounding. *Fund. Infor.*, 96(3):297–322, 2009.

[Dao-Tran *et al.*, 2012] M. Dao-Tran, T. Eiter, M. Fink, G. Weidinger, and A. Weinzierl. Omiga : An open minded grounding on-the-fly answer set solver. In *JELIA 2012*, pages 480–483, 2012.

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[de Cat *et al.*, 2015] B. de Cat, M. Denecker, M. Bruynooghe, and P. J. Stuckey. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res.*, 52:235–286, 2015.

[Dimopoulos *et al.*, 2017] Y. Dimopoulos, M. Gebser, P. Lühne, J. Romero, and T. Schaub. plasp 3: Towards effective ASP planning. In *LPNMR'17*, volume 10377 of *LNAI*, pages 286–300. Springer, 2017.

[Dodaro *et al.*, 2016] C. Dodaro, P. Gasteiger, N. Leone, B. Musitsch, F. Ricca, and K. Schekotihin. Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). *TPLP*, 16(5-6):653–669, 2016.

[Dovier *et al.*, 2016] A. Dovier, A. Formisano, E. Pontelli, and F. Vella. A GPU implementation of the ASP computation. In *PADL'16*, volume 9585 of *LNCS*, pages 30–47. Springer, 2016.

[Drescher and Walsh, 2012] C. Drescher and T. Walsh. Answer set solving with lazy nogood generation. In *Technical Communications of ICLP'12*, volume 17 of *LIPIcs*, pages 188–200. SD, 2012.

[Eén and Sörensson, 2003] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoret. Comput. Science*, 89(4):543–560, 2003.

[Eiter *et al.*, 2007] T. Eiter, W. Faber, M. Fink, and S. Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *AMAI*, 51(2-4):123–165, 2007.

[Eiter *et al.*, 2016] T. Eiter, M. Fink, G. Ianni, T. Krennwallner, C. Redl, and P. Schüller. A model building framework for answer set programming with external computations. *Theory and Practice of Logic Progeamming*, 16(4):418–464, 2016.

[Erdem and Lifschitz, 2003] E. Erdem and V. Lifschitz. Tight logic programs. *TPLP*, 3(4-5):499–518, 2003.

[Erdem *et al.*, 2016] E. Erdem, M. Gelfond, and N. Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016.

[Faber *et al.*, 2012] W. Faber, N. Leone, and S. Perri. The intelligent grounder of DLV. In *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *LNCS*, pages 247–264. Springer, 2012.

[Finkel *et al.*, 2001] R. Finkel, V. Marek, N. Moore, and M. Truszczyński. Computing stable models in parallel. In *ASP'01*, 2001.

[Gebser *et al.*, 2007] M. Gebser, T. Schaub, and S. Thiele. Gringo : A new grounder for answer set programming. In *LPNMR'07*, volume 4483 of *LNCS*, pages 266–271. Springer, 2007.

[Gebser *et al.*, 2008] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In *ICLP'08*, volume 5366 of *LNCS*, pages 190–205. Springer-Verlag, 2008.

[Gebser *et al.*, 2011a] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller. A portfolio solver for answer set programming: Preliminary report. In *LPNMR'11*, volume 6645 of *LNCS*, pages 352–357. Springer, 2011.

[Gebser *et al.*, 2011b] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, and B. Schnor. Cluster-based ASP solving with claspar. In *LPNMR'11*, volume 6645 of *LNCS*, pages 364–369. Springer, 2011.

[Gebser *et al.*, 2011c] M. Gebser, O. Sabuncu, and T. Schaub. An incremental answer set programming based system for finite model computation. *AI Communications*, 24(2):195–212, 2011.

[Gebser *et al.*, 2012a] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187-188:52–89, 2012.

[Gebser *et al.*, 2012b] M. Gebser, B. Kaufmann, and T. Schaub. Multi-threaded ASP solving with clasp. *TPLP*, 12(4-5):525–545, 2012.

[Gebser *et al.*, 2013] M. Gebser, B. Kaufmann, and T. Schaub. Advanced conflict-driven disjunctive answer set solving. In *IJCAI'13*, pages 912–918. IJCAI/AAAI Press, 2013.

[Gebser *et al.*, 2014a] M. Gebser, T. Janhunen, and J. Rintanen. Answer set programming as SAT modulo acyclicity. In *ECAI'14*, volume 263 of *FAIA*, pages 351–356. IOS Press, 2014.

[Gebser *et al.*, 2014b] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. In *Technical Communications of ICLP'14*, volume 14(4-5) of *TPLP Online Supplement*, 2014.

[Gebser *et al.*, 2015a] M. Gebser, A. Harrison, R. Kaminski, V. Lifschitz, and T. Schaub. Abstract gringo. *TPLP*, 15(4-5):449–463, 2015.

[Gebser *et al.*, 2015b] M. Gebser, R. Kaminski, B. Kaufmann, J. Romero, and T. Schaub. Progress in clasp series 3. In *LPNMR'15*, volume 9345 of *LNCS*, pages 368–383. Springer, 2015.

[Gebser *et al.*, 2015c] M. Gebser, A. Ryabokon, and G. Schenner. Combining heuristics for configuration problems using answer set programming. In *LPNMR'15*, volume 9345 of *LNCS*, pages 384–397. Springer, 2015.

[Gebser *et al.*, 2016] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko. Theory solving made easy with clingo 5. In *Technical Communications of ICLP'16*, volume 52, pages 2:1–2:15. OASICS, 2016.

[Gebser *et al.*, 2017a] M. Gebser, M. Maratea, and F. Ricca. The design of the seventh answer set programming competition. In *LPNMR 2017*, volume 10377 of *LNCS*, pages 3–9. Springer, 2017.

[Gebser *et al.*, 2017b] M. Gebser, M. Maratea, and F. Ricca. The sixth answer set programming competition. *J. Artif. Intell. Res.*, 60:41–95, 2017.

[Gebser *et al.*, 2017c] M. Gebser, M. Maratea, and F. Ricca. The sixth answer set programming competition. *J. Artif. Intell. Res.*, 60:41–95, 2017.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Giunchiglia *et al.*, 2006] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

[Gressmann *et al.*, 2006] J. Gressmann, T. Janhunen, R. E. Mercer, T. Schaub, S. Thiele, and R. Tichy. On probing and multi-threading in platypus. In *ECAI'06*, volume 141 of *FAIA*, pages 392–396. IOS Press, 2006.

[Hoos *et al.*, 2014] H. Hoos, M. Lindauer, and T. Schaub. claspfolio 2: Advances in algorithm selection for answer set programming. *TPLP*, 14(4-5):569–585, 2014.

[Hoos *et al.*, 2015] Holger Hoos, Roland Kaminski, Marius Lindauer, and Torsten Schaub. aspeed: Solver scheduling via answer set programming. *TPLP*, 15(1):117–142, 2015.

[Janhunen and Niemelä, 2011] T. Janhunen and I. Niemelä. Compact translations of non-disjunctive answer set programs to propositional clauses. In *LP, KR, and NMR*, volume 6565 of *LNCS*, pages 111–130. Springer, 2011.

[Janhunen *et al.*, 2006] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J. You. Unfolding partiality and disjunctions in stable model semantics. *ACM TOCL*, 7(1):1–37, 2006.

[Janhunen *et al.*, 2009] T. Janhunen, I. Niemelä, and M. Sevalnev. Computing stable models via reductions to difference logic. In *LPNMR'09*, volume 5753 of *LNCS*, pages 142–154. Springer, 2009.

[Janhunen, 2006] T. Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *J. of Appl. Non-Class. Log.*, 16(1-2):35–86, 2006.

[Kaufmann *et al.*, 2016] B. Kaufmann, N. Leone, S. Perri, and T. Schaub. Grounding and solving in answer set programming. *AI Magazine*, 37(3):25–32, 2016.

[Lee and Lifschitz, 2003] J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs. In *ICLP'03*, volume 2916 of *LNCS*, pages 451–465. Springer, 2003.

[Lee, 2005] J. Lee. A model-theoretic counterpart of loop formulas. In *IJCAI'05*, pages 503–508. Professional Book Center, 2005.

[Lefèvre *et al.*, 2017] C. Lefèvre, C. Béatrix, I. Stéphan, and L. Garcia. ASPeRiX, a first-order forward chaining approach for answer set computing. *TPLP*, 17(3):266–310, 2017.

[Leone *et al.*, 1997] N. Leone, P. Rullo, and F. Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 135(2):69–112, 1997.

[Leone *et al.*, 2006] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM TOCL*, 7(3):499–562, 2006.

[Lierler *et al.*, 2016] Y. Lierler, M. Maratea, and F. Ricca. Systems, engineering environments, and competitions. *AI Magazine*, 37(3):45–52, 2016.

[Lierler, 2005] Y. Lierler. Disjunctive answer set programming via satisfiability. In *LPNMR'05*, volume 3662 of *LNCS*, pages 447–451. Springer, 2005.

[Lierler, 2014] Y. Lierler. Relating constraint answer set programming languages and algorithms. *Artif. Intell.*, 207:1–22, 2014.

[Lifschitz and Razborov, 2006] V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM TOCL*, 7(2):261–268, 2006.

[Lin and Zhao, 2004] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.

[Liu *et al.*, 2012] G. Liu, T. Janhunen, and I. Niemelä. Answer set programming via mixed integer programming. In *KR'12*, pages 32–42. AAAI Press, 2012.

[Maratea *et al.*, 2008] M. Maratea, F. Ricca, W. Faber, and Nicola Leone. Look-back techniques and heuristics in DLV: Implementation, evaluation and comparison to QBF solvers. *J. of Algor.*, 63(1-3):70–89, 2008.

[Maratea *et al.*, 2013] M. Maratea, L. Pulina, and F. Ricca. Automated selection of grounding algorithm in answer set programming. In *AI*IA'13*, volume 8249 of *LNCS*, pages 73–84. Springer, 2013.

[Maratea *et al.*, 2014] M. Maratea, L. Pulina, and F. Ricca. A multi-engine approach to answer-set programming. *TPLP*, 14(6):841–868, 2014.

[Maratea *et al.*, 2015a] M. Maratea, L. Pulina, and F. Ricca. Multi-engine ASP solving with policy adaptation. *Journal of Logic and Computation*, 25(6):1285–1306, 2015.

[Maratea *et al.*, 2015b] M. Maratea, L. Pulina, and F. Ricca. Multi-level algorithm selection for ASP. In *LPNMR'15*, volume 9345 of *LNCS*, pages 439–445. Springer, 2015.

[Mariën *et al.*, 2008] M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In *SAT'08*, volume 4996 of *LNCS*, pages 211–224. Springer, 2008.

[Marques-Silva and Sakallah, 1999] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans on Comp.*, 48(5):506–521, 1999.

[Nguyen *et al.*, 2011] M. Nguyen, T. Janhunen, and I. Niemelä. Translating answer-set programs into bit-vector logic. In *INAP'11 and WLP'11*, volume 7773 of *LNCS*, pages 95–113. Springer, 2011.

[Niemelä, 2008] I. Niemelä. Stable models and difference logic. *AMAI*, 53(1-4):313–329, 2008.

[Perri *et al.*, 2013] S. Perri, F. Ricca, and M. Sirianni. Parallel instantiation of ASP programs: Techniques and experiments. *TPLP*, 13(2):253–278, 2013.

[Pontelli *et al.*, 2003] E. Pontelli, M. Balduccini, and F. Bermudez. Non-monotonic reasoning on Beowulf platforms. In *PADL'03*, volume 2562 of *LNCS*, pages 37–57. Springer, 2003.

[Prosser, 1993] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[Pulina and Tacchella, 2009] L. Pulina and A. Tacchella. A self-adaptive multi-engine solver for quantified Boolean formulas. *Constraints*, 14(1):80–116, 2009.

[Ricca *et al.*, 2006] F Ricca, W. Faber, and N. Leone. A backjumping technique for disjunctive logic programming. *AI Communications*, 19(2):155–172, 2006.

[Rice, 1976] J. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[SAT, 2009] *Handbook of Satisfiability*, volume 185 of *FAIA*. IOS Press, 2009.

[Simons *et al.*, 2002] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

[Susman and Lierler, 2016] B. Susman and Y. Lierler. Smt-based constraint answer set solver EZSMT (system description). In *ICLP (Technical Communications)*, volume 52 of *OASICS*, pages 1:1–1:15. SD, 2016.

[Syrjänen, 2001] T. Syrjänen. Omega-restricted logic programs. In *LPNMR'01*, volume 2173 of *LNCS*, pages 267–279. Springer, 2001.

[Ullman, 1988] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[Van Gelder *et al.*, 1991] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.

[Ward and Schlipf, 2004] J. Ward and J. Schlipf. Answer set programming with clause learning. In *LPNMR'04*, volume 2923 of *LNCS*, pages 302–313. Springer, 2004.

[Weinzierl, 2017] A. Weinzierl. Blending lazy-grounding and CDNL search for answer-set solving. In *LPNMR 2017*, pages 191–204, 2017.

[Wittocx *et al.*, 2008] J. Wittocx, M. Mariën, and M. Denecker. GidL: A grounder for FO+. In *NMR'08*, pages 189–198, 2008.

[Zhang *et al.*, 1996] H. Zhang, M. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.