Grounding and Solving in Answer Set Programming

Benjamin Kaufmann
Universität Potsdam, Germany
kaufmann@cs.uni-potsdam.de

Nicola Leone
University of Calabria, Italy
leone@mat.unical.it

Simona Perri
University of Calabria, Italy
perri@mat.unical.it

Torsten Schaub*
Universität Potsdam, Germany, and INRIA Rennes, France
torsten@cs.uni-potsdam.de

Abstract

Answer Set Programming is a declarative problem solving paradigm that rests upon a workflow involving modeling, grounding, and solving. While the former is described in (Gebser and Schaub 2016), we focus here on key issues in grounding, or how to systematically replace object variables by ground terms in a effective way, and solving, or how to compute the answer sets of a propositional logic program obtained by grounding.

Introduction

Answer Set Programming (ASP) combines a high-level modeling language with effective grounding and solving technology. Moreover, ASP is highly versatile by offering various elaborate language constructs and a whole spectrum of reasoning modes. The work flow of ASP is illustrated in Figure 1. At first, a problem is expressed as a logic program. A grounder systematically replaces all variables in the program by (variable-free) terms, and the solver takes the resulting propositional program and computes its answer sets (or aggregations of them).

ASP's success is largely due to the availability of a rich modeling language (cf (Gebser and Schaub 2016)) along with effective systems. Early ASP solvers *smodels* (Simons, Niemelä, and Soininen 2002) and *dlv* (Leone et al. 2006) were followed by SAT[1]-based ones, such as *assat* (Lin and Zhao 2004) and *cmodels* (Giunchiglia, Lierler, and Maratea 2006), before genuine conflict-driven ASP solvers such as *clasp* (Gebser, Kaufmann, and Schaub 2012a) and *wasp* (Alviano et al. 2015) emerged. In addition, there is a continued interest in mapping ASP onto solving technology in neighboring fields, like SAT or even MIP[2] (Janhunen, Niemelä, and Sevalnev 2009; Liu, Janhunen, and Niemelä 2012), and in the automatic selection of the appropriate solver by heuristics (Maratea, Pulina, and Ricca 2014).

On the other hand, modern grounders like (the one in) *dlv* (Faber, Leone, and Perri 2012) or *gringo* (Gebser et al. 2011) 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. Other grounding approaches are pursued in *gidl* (Wittocx, Mariën, and Denecker 2010), *lparse* (Syrjänen 2001), and earlier versions of *gringo* (Gebser, Schaub, and Thiele 2007). The latter two bind non-global variables by domain predicates to enforce ω- or λ-restricted (Syrjänen 2001; Gebser, Schaub, and Thiele 2007) programs that guarantee a finite grounding, respectively.

In what follows, we describe the basic ideas and major issues of modern ASP grounders and solvers, also in view of supporting its language constructs and reasoning modes.

Grounding

Modern ASP systems perform their computation by first generating a ground program that does not contain any variable but has the same answer sets as the original program. This phase, usually referred to as *grounding* or *instantiation*, solves a complex problem. In the case in which input non-ground programs can be assumed to be fixed (data complexity), this task is polynomial. However, as soon as variable programs are given in input, grounding becomes EXPTIME-hard, and the produced ground program is potentially of exponential size with respect to the input program. To give an idea of that, consider the following program containing only one rule, and two facts:

```
obj(0).  obj(1).
tuple(X1,...,Xn) :- obj(X1), ..., obj(Xn).
```

The ground instantiation of the rule contains 2^n ground rules, corresponding to the number of n-tuples, over a set of two elements. For more details about complexity of ASP the reader may refer to (Dantsin et al. 2001).

Grounding, hence, may be computationally very expensive having a big impact on the performance of the whole system, as its output is the input for an ASP solver, that, in the worst case, takes exponential time in the size of the input. Thus, a naïve grounding which replaces the variables with all the constants appearing in the program (thus producing the full instantiation) is undesirable from a computational point of view. Indeed, most of the ground atoms appearing

*Affiliated with Simon Fraser University, Canada, and the Griffith University, Australia.

[1] Satisfiablity Testing

[2] Mixed Integer Programming

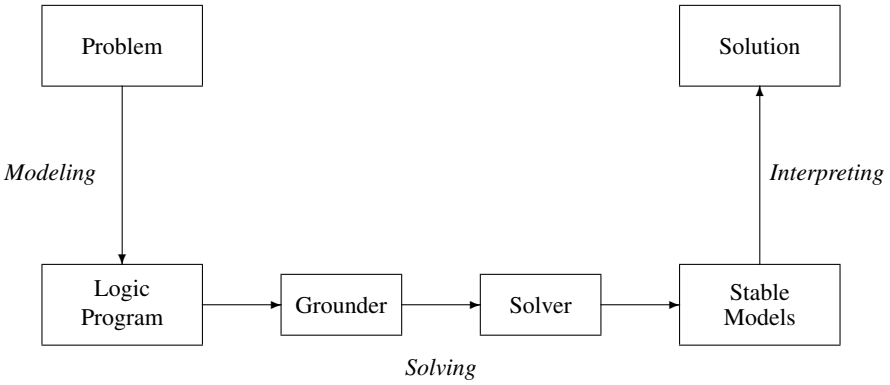

Figure 1: The work flow of Answer Set Programming

in the full instantiation are not derivable from the program rules, and all generated ground rules containing these atoms in the positive bodies are useless for answer set computation. For instance, consider the following program:

```
c(1,2).
a(X) | b(Y) :- c(X,Y).
```

The full instantiation of the only rule appearing in the program contains four ground instances:

```
a(1) | b(1) :- c(1,1).
a(2) | b(1) :- c(2,1).
a(2) | b(2) :- c(2,2).
a(1) | b(2) :- c(1,2).
```

However, the first three ground rules are useless. They will never be applicable because their bodies contain atoms `c(1,1)` and `c(2,1)`, and `c(2,2)` that are not derivable from the program (they do not appear in the head of any rule).

ASP grounders, like *gringo* or the *dlv* instantiator, employ smart procedures that are geared toward efficiently producing a ground program that is considerably smaller than the full instantiation but preserves the semantics. In the following, we first give an informal description of the grounding computation. Then we introduce the problem of dealing with function symbols, which may lead to infinite groundings. Finally we overview some optimization strategies.

The Instantiation Procedure

In this subsection, we provide a description of the basic instantiation procedure, which is adopted by the most popular grounders, *gringo* and the *dlv* instantiator. For clarity, the description is informal, and presents a simplified version of the actual instantiation strategy. For instance, we do not take into account extensions of the basic language like choice rules or aggregates (Lifschitz 2016; Alviano, Leone 2015; 2016). Full details can be found in (Faber, Leone, and Perri 2012; Gebser et al. 2011).

The core of the grounding phase is the process of rule instantiation. Given a rule r and a set of ground atoms S, which represents the extensions of the predicates, it generates the ground instances of r. Such task can be performed by iterating on the body literals looking for possible substitutions for their variables. Grounders impose a safety condition which requires that each rule variable appears also in a positive body literal. Thus, for the instantiator, it is enough to have a substitution for the variables occurring in the positive literals.

To clarify this process, consider the following (non-ground) rule:

```
a(X) | b(Y) :- p(X,Z), q(Z,Y).
```

Now, assume that the set of extensions $S = \{$`p(1,2), q(2,1), q(2,3)`$\}$ is given. Then, the instantiation starts by looking for a ground atom in S matching with `p(X,Z)`. Therefore `p(X,Z)` is matched with `p(1,2)` and the substitution for `X` and `Z` is propagated to the other body literals, thus leading to the partially-ground rule body `p(1,2), q(2,Y)`. Then, `q(2,Y)` is instantiated with the matching ground atom `q(2,1)` and a ground rule `a(1)| b(1):- p(1,2), q(2,1)` is generated. Now, in order to find other possible rule instances, a backtracking step is performed, the binding for variable `Y` is restored and a new match for `q(2,Y)` is searched, finding `q(2,3)`. The new match is applied, leading to another rule instance `a(1) | b(3) :- p(1,2), q(2,3)`. Then, the process goes on, by backtracking again to `q(2,Y)`, and then to `p(X,Z)`, because there are no more matches for `q(2,Y)`. Given that also no further matches are possible for `p(X,Z)`, the instantiation of the rule terminates, producing only two ground rules:

```
a(1) | b(1) :- p(1,2), q(2,1).
a(1) | b(3) :- p(1,2), q(2,3).
```

Roughly, the body literals are instantiated from left to right, starting from the first one. The instantiation of the generic body literal L consists in searching in S for a ground atom A matching with L; if such a matching is found, then the variables in L are bound with the constants in A, the substitution is propagated to the other body literals, and the next literal in the body is considered. If such a matching atom is not found, a backtracking step to a previous literal L' is performed, some variable bindings are restored, and the process goes on by looking for another matching for L'. When all body literals have been instantiated, an instance for the rule

r is found and the process continues by backtracking again to some previous literal, in order to find other substitutions. A crucial aspect of this process is how the set of ground atoms S containing the extensions of the predicates is computed. When a program is given as input to a grounder, it usually contains also a set of ground atoms, called *Facts*. It constitutes the starting point of the computation. In other words, initially S = *Facts*. During instantiation, the set S is expanded with the ground atoms occurring in the head of the newly generated ground rules. For instance, in the previous example, the ground atoms `a(1)` and `b(1)` are added to S and they will possibly be used for the instantiation of other rules. Thus, the extensions of the predicates are built dynamically. In order to guarantee the generation of all useful ground instances a particular evaluation order should be followed. If a rule r_1 defines (i.e., has in the head) a predicate p, and another rule r_2 contains p in the positive body, then r_1 has to be evaluated before r_2 since r_1 produces ground atoms needed for instantiating r_2. Complying with such evaluation orders ensures that the produced ground program has the same answer sets of the full instantiation, but is possibly smaller (Faber, Leone, and Perri 2012).

To produce proper evaluation orders, grounders make use of structural information provided by a directed graph, called *Dependency Graph*, that describes how predicates depend on each other. This graph induces a partition of the input program into subprograms, associated with the strongly connected components, and a topological ordering over them. The subprograms are instantiated one at a time starting from the ones associated with the lowest components in the topological ordering.

Recursive rules within a subprogram, i.e. rules where some body predicate depends, directly or transitively, on a predicate in the head, are instantiated according to a seminaïve database technique (Ullman 1988). Their evaluation produces ground atoms needed for their own evaluation, thus, several iterations are performed, until a fixpoint is reached. At each iteration, for the predicates involved in the recursion, only the ground atoms newly derived during the previous iteration are taken into account.

To illustrate this, consider the following problem, called *Reachability*: Given a finite directed graph, compute all pairs of nodes (a, b) such that b is reachable from a through a nonempty sequence of arcs.

This problem can be encoded by the following ASP program:

```
reach(X,Y) :- arc(X,Y).
reach(X,Y) :- arc(X,U), reach(U,Y).
```

The set of arcs is represented by the binary relation `arc`. A fact `arc(a,b)` means that the graph contains an arc from a to b; the set of nodes is not explicitly represented.

The program computes a binary relation $reach$ containing all facts `reach(a,b)` such that b is reachable from a through the arcs of the input graph G. In particular, the first (non-recursive) rule states that b is directly reachable from a, if there is an arc from a to b; whilst the second (recursive) rule states that b is transitively reachable from a, if there is a path in the graph from a to b.

The instantiation of this program is performed by first evaluating the non-recursive rule on the set S containing the arcs. Assuming that $S = \{$ `arc(1,2), arc(2,3), arc(3,4)` $\}$ three ground instances are produced:

```
reach(1,2) :- arc(1,2).
reach(2,3) :- arc(2,3).
reach(3,4) :- arc(3,4).
```

The ground atoms `reach(1,2)`, `reach(2,3)`, and `reach(3,4)` are added to set S and the evaluation of the recursive rule starts. The first iteration is performed, producing rules

```
reach(1,3) :- arc(1,2), reach(2,3).
reach(2,4) :- arc(2,3), reach(3,4).
```

Then, `reach(1,3)`, `reach(2,4)` are added to S and another iteration starts. To avoid duplicate rules, for the recursive predicate $reach$, only the two newly generated ground atoms are used, producing:

```
reach(1,4) :- arc(1,2), reach(2,4).
```

Now, `reach(1,4)` is added to S. Another iteration is performed. Nothing new can be produced. The fixpoint is reached and the evaluation terminates.

Optimizations

Substantial effort has been spent on sophisticated algorithms and optimization techniques aimed at improving the performance of the instantiation process. In the following we briefly recall the most relevant ones.

The *dynamic magic sets* technique (Alviano et al. 2012a) is a rewriting-based optimization strategy used by the *dlv* system. It extends the Magic Sets technique originally defined for standard Datalog for optimizing query answering over logic programs. Given a query, the Magic Sets technique rewrites the input program to identify a subset of the program instantiation which is sufficient for answering the query. The restriction of the instantiation is obtained by means of additional "magic" predicates, whose extensions represent relevant atoms w.r.t. the query. Dynamic Magic Sets, specifically conceived for disjunctive programs, inherit the benefits provided by standard magic sets and additionally allow to exploit the information provided by the magic predicates also during the answer set search. Magic sets turned out to be very useful in many application domains, even on some co-NP complete problems like consistent query answering (Manna, Ricca, and Terracina 2015).

Other techniques have been developed for optimizing the rule instantiation task (Faber, Leone, and Perri 2012). In particular, since rule instantiation is essentially performed by evaluating the relational join of the positive body literals, an optimal ordering of literals in the body is a key issue for the efficiency of the procedure, just like for join computation. Thus, an efficient body reordering criterion specifically conceived for the rule instantiation task has been proposed. Moreover, a *backjumping algorithm* has been developed (Perri et al. 2007), which reduces the size of the ground programs, avoiding the generation of useless rules, but fully preserving the semantics.

In the last few years, in order to make use of modern multi-core/multi-processor computers, a *parallel instantiator* has been developed. It is based on a number of strategies allowing for the concurrent evaluation of parts of the program, and is endowed with advanced mechanisms for dealing with load balancing and granularity control (Perri, Ricca, and Sirianni 2013).

Dealing with function symbols

Function symbols are widely recognized as an important feature for ASP. They increase the expressive power and in some cases improve the modeling capabilities of ASP, allowing the support of complex terms like lists, and set terms. Functions can also be employed to represent, via scholemization, existential quantifiers, which are receiving an increasing attention in the logic programming and database communities (Gottlob, Manna, and Pieris 2015). However, the presence of function symbols within ASP programs has a strong impact on the grounding process, which might even not terminate. Consider, for instance, the program:

```
p(0).
p(f(X)) :- p(X).
```

The instantiation is infinite; indeed the grounding of the recursive rule, at the first iteration adds to the set of extensions S the ground atom `p(f(0))`, that is used in the next iteration, producing `p(f(f(0))` and so on. Despite this, grounders like the one in *dlv* and *gringo* allow to deal with recursive function symbols, and guarantee termination whenever the program belongs to the class of the so called *finitely-ground* programs (Calimeri et al. 2008). Intuitively, for each program $\mathcal{P}$ in this class, there exists a finite ground program $\mathcal{P}'$ having exactly the same answer sets as $\mathcal{P}$. Program $\mathcal{P}'$ is computable for finitely-ground programs, thus answer sets of $\mathcal{P}$ are computable as well. Notably, each computable function can be expressed by a finitely-ground program; membership in this class is not decidable, but it has been proven to be semi-decidable (Calimeri et al. 2008).

For applications in which termination needs to be guaranteed a priori, the ASP grounders can make use of a preprocessor implementing a decidable check, which allows the user to statically recognize whether the input program belongs to a smaller subclass of the finitely-ground programs (Syrjänen 2001; Gebser, Schaub, and Thiele 2007; Lierler and Lifschitz 2009; Calimeri et al. 2008). For instance, the grounder of *dlv* is endowed with a checker (which can also be disabled) for recognizing argument-restricted programs (Lierler and Lifschitz 2009). Earlier version of *gringo* in order to guarantee finiteness, accepted input programs with a domain restriction, namely λ-restricted programs (Gebser, Schaub, and Thiele 2007). From series 3, *gringo* removed domain restrictions and the responsibility to check whether the input program has a finite grounding is left to the user.

Solving

Modern ASP solvers rely upon advanced conflict-driven search procedures, pioneered in the area of Satisfiablity testing (SAT; (Biere et al. 2009)).[3] Conflicts are analyzed and recorded, decisions are taken in view of conflict scores, and back-jumps are directed to the origin of a conflict.

While the general outline of search in ASP is arguably the same as in SAT, the extent of ASP requires a much more elaborate approach. First, the stable models semantics enforces that atoms are not merely true but provably true (Lifschitz 2016). Second, the rich modeling language of ASP comes with complex language constructs. In particular, disjunction in rule heads and non-monotone aggregates lead to an elevated level of computational complexity, which imposes additional search efforts. Finally, ASP deals with various reasoning modes. Apart from satisfiability testing, this includes enumeration, projection, intersection, union, and (multi-objective) optimization of answer sets, and moreover combinations of them, for instance, the intersection of all optimal models.[4] The first two issues bring about additional inferences, the latter require flexible solver architectures.

The restriction of modern SAT solvers to propositional formulas in Conjunction Normal Form allows for reducing inferences to unit propagation along with the usual choice operations. In contrast, traditional ASP solving deals with an abundance of different inferences for propagation, which makes a direct adaption of conflict-driven search procedures virtually impossible. The key idea is thus to map inferences in ASP onto unit propagation on nogoods[5] (Gebser, Kaufmann, and Schaub 2012a), which traces back to a characterization of answer sets in propositional logic (Lin and Zhao 2004). Let us illustrate this by program P (thereby restricting ourselves to normal rules): [6]

$$P = \left\{ \begin{array}{lll} \texttt{a :- not b,} & \texttt{b :- not a,} \\ \texttt{x :- a,not c,} & \texttt{x :- y,} \\ \texttt{y :- x,b} & \end{array} \right\}$$

Interpreting this program in propositional logic results in the set $RF(P)$ of implications:

$$RF(P) = \left\{ \begin{array}{ll} a \leftarrow \neg b, & b \leftarrow \neg a, \\ x \leftarrow a \wedge \neg c \vee y, & \\ y \leftarrow x \wedge b & \end{array} \right\}$$

Note that we replaced default negation `not` by classical negation $\neg$ and combined both rules with head x, while leaving the direction of the implications untouched (for readability). Now, the set $RF(P)$ has twelve classical models, many of which contain atoms not supported by any rule. (This is important because the stable models semantics insists on provably true atoms.) For instance, c is not supported by any rule, as is b whenever a is true as well.

Models containing unsupported atoms are eliminated by turning the implications in $RF(P)$ into equivalences (Clark

[3] This technology is usually referred to as *Conflict-driven Clause Learning*.

[4] Actually, this is a frequent reasoning mode used in underspecified application domains such as bio-informatics (Erdem, Gelfond, and Leone 2016).

[5] Nogoods express inadmissible assignments (Dechter 2003).

[6] Below, $RF(P)$, $CF(P)$, and $LF(P)$ stand for the rule, completion, and loop formulas of P.

1978). Doing so for each atom yields the set $CF(P)$ of equivalences:

$$CF(P) = \left\{ \begin{array}{ll} a \leftrightarrow \neg b, & b \leftrightarrow \neg a, \\ x \leftrightarrow a \wedge \neg c \vee y, & \\ y \leftrightarrow x \wedge b, & c \leftrightarrow \bot \end{array} \right\}$$

This strengthening results in three models of $CF(P)$, one entailing atom b only, another making b, x, y true, and finally one in which a, x hold, respectively. The first two models differ in making both x and y true or not. A closer look at the original program P reveals that x and y support each other in a circular way. Whether or not such a circular derivation is harmful depends upon the existence of a valid external support (Lin and Zhao 2004), provided by an applicable rule whose head is in the loop but none of its positive antecedents belongs to it. In our case, this can be accomplished by the formula in $LF(P)$:

$$LF(P) = \{ \ (x \vee y) \rightarrow a \wedge \neg c \ \}$$

The formula expresses that an atom in the loop (consisting of x and y) can only be true if an external support of x or y is true. Here the only external support is provided by rule $x \leftarrow a, \texttt{not}\ c$ in P, as reflected by the consequent in $LF(P)$. That is, x or y can only be true if the latter rule applies. Since no other loops occur in P, the set $CF(P) \cup LF(P)$ provides a characterization of P's answer sets (Lin and Zhao 2004), one making atom b true and another a, x.

Note that in general the size of $CF(P)$ is linear in that of a program P, whereas the size of $LF(P)$ may be exponential in P (Lifschitz and Razborov 2006). Fortunately, satisfaction of $LF(P)$ can be tested in linear time for logic programs facing no elevated complexity (see above), otherwise this test is co-NP-complete (Leone, Rullo, and Scarcello 1997).

The translation of programs into nogoods employed by modern ASP solvers follows the above characterization but takes the space issue into account. Given a program P, the nogoods expressing $CF(P)$ are explicitly represented in an ASP solver, while the ones in $LF(P)$ are only made explicit upon violation. This violation is detected by so-called unfounded set algorithms (Leone, Rullo, and Scarcello 1997; Gebser et al. 2012). Although we do not detail this here, we mention that aggregates are treated in a similar way by dedicated mechanisms unless they can be translated into nogoods in a feasible way (Gebser et al. 2009).

Finally, let us make this more concrete by looking at the system architecture of *clasp*, depicted in Figure 2. The preprocessing component takes a (disjunctive) logic program and translates it into an internal representation. This is done in several steps. First, the given program, P, is simplified by semantic preserving translations as well as equivalence detection (Gebser et al. 2008). The simplified program P' is then translated into nogoods expressing $CF(P')$, which are subject to clausal simplifications adapted from corresponding SAT techniques. The resulting static nogoods are kept in the shared context component, as are parts of the dependency graph of P in order to reconstruct members of $LF(P)$ on demand. Often more than three quarters of the nogoods obtained from $CF(P')$ are binary or ternary. Hence, such short nogoods are stored in dedicated data structures (and shared during parallel solving). Each solver instance implements a conflict-driven search procedure, as sketched at the outset of this section. Of particular interest is propagation, distinguishing between unit and post propagation. The former computes a fixed point of unit propagation. More elaborate propagation mechanisms can be added via post propagators. For instance, for programs with loops, this list contains a post propagator implementing the unfounded set checking procedure. Similarly, *clasp*'s extension with constraint processing, *clingcon* (Ostrowski and Schaub 2012), as well as *dlvhex* (Eiter et al. 2006) use its post propagation mechanism to realize additional theory-specific propagations. The parallel execution of *clasp* allows for search space splitting as well as running competitive strategies. In both cases, learned conflict nogoods (as well as bounds in case of optimization) are exchanged between solver instances, each of which can be configured individually (see (Gebser, Kaufmann, and Schaub 2012b) for details on multi-threading). Finally, the enumerator is in charge of handling the various reasoning modes; once a solver finds a model, the enumerator tells it how to continue. This can be done by finding a next model in case of enumeration, or a better model in case of optimization.

Conclusion

Answer Set Programming combines a high-level modeling language with effective grounding and solving technology. This materializes in off-the-shelf ASP systems, whose grounding and solving engines can be used as black-box systems with standardized interfaces. Also, ASP is highly versatile by offering various complex language constructs and reasoning modes. As a side-effect, many ASP solvers can also be used for MAX-SAT[7], SAT, and PB[8] solving. As a consequence, ASP faces a growing range of applications, as detailed in (Erdem, Gelfond, and Leone 2016).

Acknowledgments The first and last author were partially funded by DFG grants SCHA 550/8 and SCHA 550/9. The second and third author were partially supported by MIUR under PON project "SI-LAB BA2KNOW – Business Analitycs to Know", and by Regione Calabria, programme POR Calabria FESR 2007-2013, projects "ITravel PLUS" and "KnowRex: Un sistema per il riconoscimento e l'estrazione di conoscenza".

References

Alviano, M.; Faber, W.; Greco, G.; and Leone, N. 2012a. Magic Sets for Disjunctive Datalog Programs. *Artificial Intelligence* 187:156–192.

Alviano, M.; Dodaro, C.; Leone, N.; and Ricca, F. 2015. Advances in WASP. In Calimeri, F.; Ianni, G.; and Truszczyński, M., eds., *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, 40–54. Springer.

[7] Maximum Satisfiability Problem

[8] Pseudo-Boolean

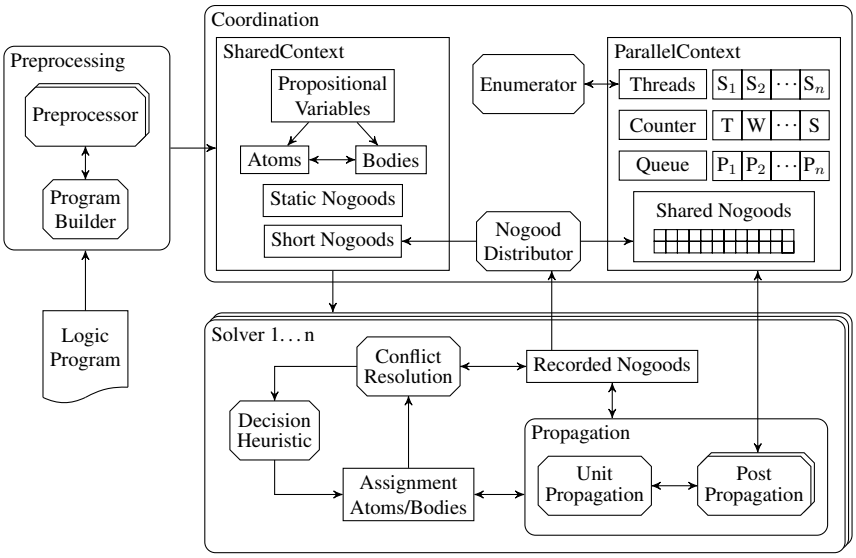

Figure 2: The multi-threaded architecture of the ASP solver *clasp*

Alviano, M.; Leone, N. 2015. Complexity and Compilation of GZ-aggregates in Answer Set Programming. *Theory and Practice of Logic Programming* 15(4-5):574–587.

Alviano, M.; Leone, N. 2016. On the Properties of GZ-Aggregates in Answer Set Programming. *Proceedings of the 25-th International Joint Conference on Artificial Intelligence (IJCAI-16)*. AAAI Press.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*. IOS Press.

Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. 2008. Computable Functions in ASP: Theory and Implementation. In Garcia de la Banda, M., and Pontelli, E., eds., *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, 407–424. Springer.

Clark, K. 1978. Negation as Failure. In Gallaire, H., and Minker, J., eds., *Logic and Data Bases*. Plenum Press. 293–322.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys* 33(3):374–425.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.

Eiter, T.; Ianni, G.; Schindlauer, R.; and Tompits, H. 2006. DLVHEX: A Prover for Semantic-web Reasoning under the Answer-set Semantics. In *Proceedings of the International Conference on Web Intelligence (WI'06)*, 1073–1074. IEEE Computer Society.

Erdem, E.; Gelfond, M.; and Leone, N. 2016. Applications of ASP. *AI Magazine*. This issue.

Faber, W.; Leone, N.; and Perri, S. 2012. The Intelligent Grounder of DLV. In Erdem, E.; Lee, J.; Lierler, Y.; and Pearce, D., eds., *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*. Springer. 247–264.

Gebser, M., and Schaub, T. 2016. Modeling and Language Extensions. *AI Magazine*. This issue.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2008. Advanced Preprocessing for Answer Set Solving. In Ghallab, M.; Spyropoulos, C.; Fakotakis, N.; and Avouris, N., eds., *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, 15–19. IOS Press.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2009. On the Implementation of Weight Constraint Rules in Conflict-driven ASP Solvers. In Hill, P., and Warren, D., eds., *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, 250–264. Springer.

Gebser, M.; Kaminski, R.; König, A.; and Schaub, T. 2011. Advances in Gringo Series 3. In Delgrande, J., and Faber, W., eds., *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, 345–351. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2012a. Conflict-driven Answer Set Solving: From Theory to Practice. *Artificial Intelligence* 187-188:52–89.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2012b. Multi-threaded ASP Solving with Clasp. *Theory and Practice of Logic Programming* 12(4-5):525–545.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. Gringo: A new Grounder for Answer Set Programming. In Baral, C.; Brewka, G.; and Schlipf, J., eds., *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, 266–271. Springer.

Giunchiglia, E.; Lierler, Y.; and Maratea, M. 2006. An-

swer Set Programming Based on Propositional Satisfiability. *Journal of Automated Reasoning* 36(4):345–377.

Gottlob, G.; Manna, M.; and Pieris, A. 2015. Polynomial Rewritings for Linear Existential Rules. In *Proceedings of the 24-th International Joint Conference on Artificial Intelligence (IJCAI-15)*, 2992-2998. AAAI Press.

Janhunen, T.; Niemelä, I.; and Sevalnev, M. 2009. Computing Stable Models via Reductions to Difference Logic. In Erdem, E.; Lin, F.; and Schaub, T., eds., *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, 142–154. Springer.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic* 7(3):499–562.

Leone, N.; Rullo, P.; and Scarcello, F. 1997. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics, and Computation. *Information and Computation* 135(2):69–112.

Lierler, Y., and Lifschitz, V. 2009. One More Decidable Class Of Finitely Ground Programs. In Hill, P., and Warren, D., eds., *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, 489–493. Springer.

Lifschitz, V., and Razborov, A. 2006. Why Are There So Many Loop Formulas? *ACM Transactions on Computational Logic* 7(2):261–268.

Lifschitz, V. 2016. Answer Sets and the Language of Answer Set Programming. *AI Magazine*. This issue.

Lin, F., and Zhao, Y. 2004. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. *Artificial Intelligence* 157(1-2):115–137.

Liu, G.; Janhunen, T.; and Niemelä, I. 2012. Answer Set Programming via Mixed Integer Programming. In Brewka, G.; Eiter, T.; and McIlraith, S., eds., *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, 32–42. AAAI Press.

Manna, M.; Ricca, F.; and Terracina, G. 2015. Taming Primary Key Violations to Query Large Inconsistent Data via ASP. *Theory and Practice of Logic Programming* 15(4-5):696–710.

Maratea, M.; Pulina, L.; and Ricca, F. 2014. A Multi-engine Approach to Answer-set Programming. *Theory and Practice of Logic Programming* 14(6):841–868.

Ostrowski, M., and Schaub, T. 2012. ASP Modulo CSP: The Clingcon System. *Theory and Practice of Logic Programming* 12(4-5):485–503.

Perri, S.; Scarcello, F.; Catalano, G.; and Leone, N. 2007. Enhancing DLV Instantiator by Backjumping Techniques. *Annals of Mathematics and Artificial Intelligence* 51(2-4):195–228.

Perri, S.; Ricca, F.; and Sirianni, M. 2013. Parallel Instantiation of ASP Programs: Techniques and Experiments. *Theory and Practice of Logic Programming* 13(2):253–278.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence* 138(1-2):181–234.

Syrjänen, T. 2001. Omega-restricted Logic Programs. In Eiter, T.; Faber, W.; and Truszczyński, M., eds., *Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, 267–279. Springer.

Ullman, J. 1988. *Principles of Database and Knowledge-Base Systems*. Computer Science Press.

Wittocx, J.; Mariën, M.; and Denecker, M. 2010. Grounding FO and FO(ID) with Bounds. *Journal of Artificial Intelligence Research* 38:223–269.