Smodels with CLP—A Treatment of Aggregates in ASP

Enrico Pontelli Tran Cao Son

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
{epontell|tson}@cs.nmsu.edu

1 Introduction

In recent years we have witnessed a renovated interest towards the development of practical, effective, and efficient implementations of different flavors of logic programming. A significant deal of research has been invested in the development of systems that support logic programming under *answer set semantics*, favoring the creation of a novel programming paradigm, commonly referred to as *Answer Set Programming (ASP)* (e.g. [13]).

Nice practical and efficient systems have been proposed to support execution of ASP programs, e.g., *smodels* [16], *dlv* [9], *cmodels* [1], and *ASSAT* [14]. The logic-based languages provided by these systems offer a variety of syntactic structures, aimed at supporting the requirements arising from different application domains.

The objective of this work is to investigate the introduction of different types of *aggregates* in ASP. Database query languages (e.g., SQL) use aggregate functions—such as *sum*, *count*, *max*, and *min*—to obtain summary information from a database. Aggregates have been shown to significantly improve the compactness and clarity of programs in various flavors of logic programming [12, 7]. We expect to gain similar advantages from the introduction of different forms of aggregations in ASP. In the next two examples, we demonstrate the use of aggregates in ASP.

Example 1. Consider the simplest problem in job scheduling where we have a total number of available machines (`resource(Max)`) and a number of jobs to be scheduled. Each job uses one machine and lasts for a certain duration (`duration(Job,Time)`). Once a job starts, it has to be completed. Intuitively, we can define a predicate $active(Job, Time)$ and use the aggregate $count$ to record that Job is active and to calculate the number of active jobs at the time instance $Time$, respectively. The number of active jobs cannot be greater than the number of machines at any time. This can be represented by the next two rules:

active(Job,Time) :- time(Time), duration(Job, Len), start(Job,Init), Time >= Init, Time < Init+Len.
:- time(T), resource(Max), count(T, active(Job,T)) > Max.

We note that a previous encoding of the job scheduling problem in ASP has been developed in [3]. The encoding is, however, quite complicated due to the lack of aggregates.

Example 2. (From [15], demonstrating the need of aggregates in recursive definition)
Let $owns(X, Y, N)$ denote the fact that company X owns a fraction N of the shares of the company Y. We say that a company X *controls* a company Y if the sum of the shares it owns in Y together with the sum of the shares owned in Y by companies controlled by X is greater than half of the total shares of Y.[1]

control(X, X, Y, N) :- owns(X,Y,N). *%% X directly owns N of Y*
control(X, Z, Y, N) :- control(X,Z), owns(Z,Y,N). *%% indirect*
fraction(X,Y,N) :- sum(M,control(X,Z,Y,M)) = N.
control(X,Y) :- fraction(X,Y,N), N >0.5.

A significant body of research has been developed in the database and in the constraint programming communities exploring the theoretical foundations and, in a more limited fashion, the algorithmic properties of aggregation constructs in logic programming (e.g. [12, 18, 15, 6]). More limited attention has been devoted to the more practical aspects related to computing in logic programming in presence of aggregates. In [2], it has been shown that aggregate functions can be encoded in ASP. The main disadvantage of this proposal is that the obtained encoding contains several intermediate variables, thus making the grounding phase quite expensive in term of space and time. Recently, a number of proposals to extend logic programming with aggregate functions have been developed. We list them below:

- A preliminary investigation of aggregation in ASP has been proposed by Gelfond [11] in the ASET system—a version of A-Prolog enriched with aggregation on sets.
- Manipulation of grouping facilities in logic programming have been studied by various authors (e.g., [17]); a semantic characterization of grouping under answer set semantics, along with algorithms for efficient reduction of grouping and aggregates, have been recently proposed by the authors [8].

[1] For the sake of simplicity we omitted the domain predicates required by *smodels*.

- Very recently Dell'Armi et al. [5] have proposed an implementation of aggregates in the *dlv* ASP engine. The proposal provides syntactic capabilities analogous to the ones proposed in this paper. We employ a different execution model—which has the potential of covering a slightly larger set of program than those dealt with in [5].

The specific approach proposed in this work is aimed at accomplishing the same objectives as similar proposals recently appeared in the literature [5, 11]. The novelty of our approach lies in the technique adopted to support aggregates. Following the spirit of our previous efforts [8, 4, 7], we rely on the integration of different constraint solving technologies to support the management of different flavors of sets and aggregations. In particular, in this work we describe a backend inference engine, obtained by the integration of *smodels* with a finite-domain constraint solver, capable of executing *smodels* program with aggregates. The backend is meant to be used in conjunction with front-ends capable of performing high-level constraint handling of sets and aggregates (as described in [8]).

2 Integrating a Constraint Solver to An Answer Set Solver

We now describe the most relevant aspects of our system that will be referred as *smodels–ag* hereafter. The general idea of our solution is to employ finite domain constraints to encode the aggregates present in a program. Each atom appearing in an aggregate is represented as a variable with domain $0..1$; the whole aggregate is expressed as a constraint involving such variables. E.g., given a program containing the atoms $p(1), p(2), p(3)$, the aggregate $sum(A, p(A))<3$ will lead to the overall constraint $X[1]::0..1, X[2]::0..1, X[3]::0..1, X[1]*1+X[2]*2+X[3]*3\#<3$ where $X[1], X[2], X[3]$ are constraint variables corresponding to $p(1), p(2), p(3)$ respectively.

2.1 Syntax

The input language accepted by the *smodels-ag* system is analogous to the language used by *smodels*, with the exception of a new class of comparison literals—the *aggregate* literals. Aggregate literals are of the form $F(X, \bar{Y}, Goal[X, \bar{Y}])\ \mathbf{Op}\ Result$, where

- F is the aggregate function—currently the system accepts the aggregate functions sum, $count$, min, max;
- X is the grouped variable;
- $\bar{Y}$ are variables that are meant to be existentially quantified in the aggregate operation;
- $Goal[X, \bar{Y}]$ is either a simple atom (and in such case $\bar{Y}$ should be empty) or an expression of the type $atom_1[X, \bar{Y}] : atom_2[\bar{Y}]$;
- $\mathbf{Op}$ is one of the relational operators drawn from the set $\{=, !=, <, >, <=, >=\}$;
- $Result$ is either a variable or a numeric constant.

The variables $X, \bar{Y}$ are locally quantified within the aggregate. At this time, the aggregate literal cannot play the role of a domain predicate—thus any other variables appearing in an aggregate literal are treated in the same way as variables appearing in a negative literal in the body of a rule.

It is worth noticing that in *smodels-ag* we have opted for relaxing the stratification requirement present in [5, 11], which avoids the presence of recursion through aggregates. The price to pay is the possibility of generating non-minimal models [8]; on the other hand, the literature has highlighted situations where stratification of aggregates prevents from expressing natural solutions to problems (e.g., [15]).

2.2 System Architecture

The overall structure of *smodels-ag* is shown in Figure 1. The current implementation is built using the *smodels* system (vers. 2.27) and the ECLiPSe constraint solver (vers. 5.4). At this stage it is a prototype aimed at investigating the feasibility of the proposed ideas.

Preprocessing. The Pre-processing module is composed of three sequential steps. In the *first* step, a program – called *Pre-Analyzer* – is used to perform a number of simple syntactic transformations of the input program. The transformations are mostly aimed at rewriting the aggregate literals in a format acceptable by *lparse*. The *second* step executes the *lparse* program on the output of the pre-analyzer, producing a grounded version of the program encoded in the format required by *smodels* (i.e., with a separate representation of rules and atoms). The *third* step is performed by the *Post-Analyzer* program whose major activities are:

- Identification of the dependencies between aggregate literals and atoms contributing to such aggregates; these dependencies are explicitly included in the output file. (The *lparse* output format is extended with a fourth section, which accommodates a description of these dependencies.)
- Generation of the constraint formulae encoding the behavior of each aggregate; for example, an entry like *57 sum(x,use(8,x),3,greater)* in the atom table (describing the aggregate literal *sum(X,use(8,X)) > 3*) is converted to

*57 sum(3,[16,32,48],"X16 * 2 + X32 * 1 + X48 * 4 + 0 #> 3")* ($16, 32, 48$ are the indices of the atoms contributing to the aggregate).

- Simplification of the constraints making use of the truth values discovered by *lparse*.

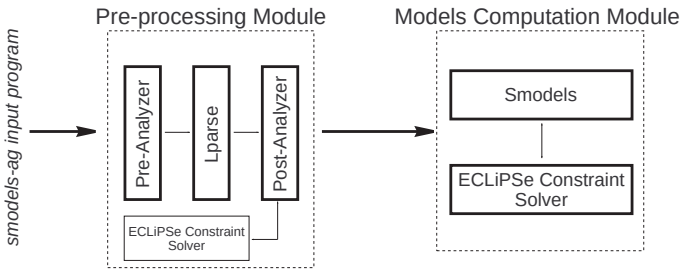

Fig. 1. Overall System Structure

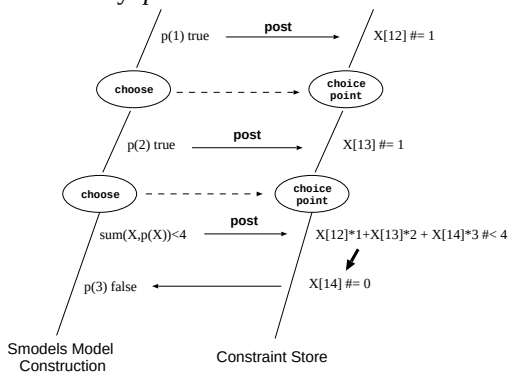

Fig. 2. Communication *smodels* to *ECLiPSe*

Models Computation. The Model Computation module (Fig. 1) is in charge of generating the models from the input program. The module consists of a modified version of *smodels* interacting with an external finite domain constraint solver (in this specific instance, the ECLiPSe solver).

Following the spirit of *smodels*, each atom in the program has a separate internal representation—including aggregate literals. In particular, each aggregate literal representation maintains information regarding what program rules it appears in. The representation of each aggregate literal is similar to that of a standard atom, with the exception of some additional fields; these are used to store an ECLiPSe structure representing the constraint associated to the aggregate. In addition, each standard atom includes a list of pointers to all the aggregate literals that depend on such atom.

The main flow of execution is directed by the *smodels* engine. In parallel with the construction of the model, our system builds a *constraint store* in the ECLiPSe engine. The constraint store maintains *one conjunction* of constraints, representing the level of aggregate instantiation achieved so far. Each time a standard (i.e., non-aggregate) atom is made true or false, a new constraint is posted in the constraint store. If i is the index of such atom within *smodels*, and the atom is made true (false), then the constraint $X[i]\#{=}1$ ($X[i]\#{=}0$) is posted. (Fig. 2, first two `post` operations).

The structure of the computation developed by *smodels* is reflected in the structure of the constraints store (see Fig. 2). In particular, each time *smodels* generates a choice point (e.g., as effect of guessing the truth value of an atom), a corresponding choice point has to be generated in the store. Similarly, whenever *smodels* detects a conflict and initiates backtracking, a failure has to be triggered in the store as well. Observe that choice points and failures can be easily generated in the store using, for example, the Prolog *repeat* and *fail* predicates.

Since aggregate literals are treated by *smodels* as standard program atoms, they can be made true, false, or guessed. The only difference is that, whenever their truth value is decided, a different type of constraint will be posted to the store—i.e., the constraint that encodes the aggregate (Fig. 2, third posting). If the aggregate literal is made false, then a negated constraint will be posted (negated constraints are obtained by applying the $\#\backslash+$ ECLiPSe operator).

The *expand* procedure of *smodels* requires some minor modifications as well. Aggregate literals may become true or false not only as the result of the declarative closure computation, but also because enough evidence has been accumulated to prove its status. E.g., if the truth value of all the atoms involved in the aggregate has already been established, then the aggregate can be immediately evaluated.

It is important to observe that the constraints posted to the store have an active role during the execution:

- constraints are used in a forward manner to prune *smodels* executions (failure in the store leads to failure in *smodels*)
- constraints can also provide feedback to *smodels* by forcing truth or falsity of previously uncovered atoms (truth value is unknown at that time). E.g., if the constraint $X[12]{*}1 + X[13]{*}2 + X[14]{*}3\#{<}4$ is posted to the store (corresponding to the aggregate $sum(X, p(X)){<}10$) and $X[12]\#{=}1$ and $X[13]\#{=}1$ have been previously posted (e.g., $p(1)$ and $p(2)$ are true), then it will force $X[3]\#{=}0$, i.e., $p(3)$ to be false (Fig. 2, last step).
- constraints may lead to failures in the constraint store; in this case, the failure has to be propagated back to the *smodels* computation.

2.3 Discussion

The first prototype implementing these ideas has been completed and successfully used on a pool of benchmarks. Performance is acceptable, but we expect to obtain significant improvements by refining the interface with ECLiPSe. Combining a constraint solver to *smodels* brings a number of advantages. We list some of them below:

- since we are relying on an external constraints solver to effectively handle the aggregates, the only step required to add new aggregates (e.g., $times$, avg) is the generation of the appropriate constraint formula during preprocessing;
- the constraint solvers are very flexible; for example, by making use of Constraint Handling Rules (CHR) [10] we can easily implement different strategies to handle constraints as well as new constraint operators;
- the constraint solvers allow certain optimizations to be done automatically (see [5] for desirable optimizations in presence of aggregates);
- it is a straightforward extension to allow the user to declare certain aggregate instances as *eager*; in this case, instead of posting only the corresponding constraint to the store, we will also post a *labeling*, forcing the immediate resolution of the constraint store (i.e., guess the possible combination of truth values of the atoms involved in the aggregate). In this way, the aggregate will act as a generator of solutions instead of just a pruning mechanism.

3 Conclusions

In this work we have explored an alternative approach to the problem of handling aggregates in ASP. We developed a system which interfaces *smodels* with an external constraint solver (*ECLiPSe* in this particular case). The power of *smodels* is preserved by treating aggregates as standard literals; on the other hand, each aggregate is concurrently manipulated as a constraint within the constraint solver. *smodels* and the constraint solver interact (in a bi-directional way) by exchanging information about truth values of program atoms and information about success and failure of the current computation. The generality of the constraint solver allows us to use aggregates not only as pruning mechanisms but also as generators of solutions.

We believe this approach has advantages over previous proposals. The use of a general constraint solver allows us to easily understand and customize the way aggregates are handled (e.g., allow the user to select eager vs. non-eager treatment); it also allows us to easily extend the system to include new form of aggregates, by simply adding new type of constraints. Furthermore, the current approach relaxes some of the syntactic restriction imposed in other proposals (e.g., stratification of aggregations). The implementation requires minimal modification to the *smodels* system and introduces insignificant overheads for regular programs.

The prototype provided us with a confirmation of the feasibility of this approach. Future work includes:

- further relaxation of some of the syntactic restrictions. For example, the use of labeling, during constraint solving, allows the aggregates to "force" solutions, thus allowing the aggregate to act as a generator of values and removing the need to include domain predicates to cover the result of the aggregate (e.g., the *safety* condition used in *dlv*).
- development of an independent grounding front-end; the use of a pre-analyzer in the current implementation is dictated by the need to overcome the limitations of *lparse* in dealing with syntactic extensions.

References

1. Y. Babovich and V. Lifschitz. Computing Answer Sets Using Program Completion.
2. C. Baral. *Knowledge Representation, reasoning, and declarative problem solving with Answer sets*, Cambridge Uni. Press.
3. C. Baral et al. Reasoning about actions in presence of resources: applications to planning and scheduling, Proc. of ICIT, 2001.
4. A. Dal Palu', A. Dovier, E. Pontelli, G. Rossi. Integrating Finite Domain Constraints and CLP with Sets. PPDP, ACM, 2003.
5. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, G. Pfeifer. . Aggregate Functions in Disjunctive Logic Programming. IJCAI, 2003.
6. M. Denecker et al. Ultimate well-founded and stable semantics for logic programs with aggregates. In *ICLP*, 212–226. 2001.
7. A. Dovier et al. Constructive Negation and Constraint Logic Programming with Sets. *New Generation Computing*, 19(3).
8. A. Dovier, E. Pontelli, and G. Rossi. Intensional Sets in CLP. Int. Conference on Logic Programming, Springer, 2003.
9. T. Eiter et al. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *KRR*, pages 406–417, 1998.
10. T. Frühwirth. Theory and Practice of Constraint Handling Rules. Journal of Logic Programming, 37(1–3), 1998.
11. M. Gelfond. Representing Knowledge in A-Prolog. Computational Logic: Logic Programming and Beyond, Springer, 2002.
12. D. B. Kemp and P. J. Stuckey. Semantics of Logic Programs with Aggregates. In *ILPS*, pages 387–401. MIT Press, 1991.
13. V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1–2):39–54, 2002.
14. F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In *AAAI*, 2002.
15. K. A. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. *JCSS*, 54:79–97, 1997.
16. P. Simons, N. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *AIJ*, 138(1–2):181–234.
17. O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of Set Terms in the Logic Data Language (LDL). *JLP*, 12(1/2):89–119.
18. A. Van Gelder. The Well-Founded Semantics of Aggregation. In *11th PODS*, pages 127–138. ACM Press, 1992.