# SMODELS$^A$ — A System for Computing Answer Sets of Logic Programs with Aggregates

Islam Elkabani, Enrico Pontelli, and Tran Cao Son

New Mexico State University, Las Cruces, NM 88003, USA
{tson, epontell, ielkaban}@cs.nmsu.edu

## 1 Introduction

In [2], we presented a system called $\mathbb{ASP}$-$\mathbb{CLP}$ for computing answer sets of logic programs with aggregates. The implementation of $\mathbb{ASP}$-$\mathbb{CLP}$ relies on the use of an external constraint solver (ECLiPSe) to deal with aggregate literals and requires some modifications to the answer set solver used in the experiment (SMODELS). In general, the system is capable of computing answer sets of arbitrary programs with aggregates, i.e., there is no syntactical restrictions imposed on the inputs to the system. This makes $\mathbb{ASP}$-$\mathbb{CLP}$ different from DLV$^A$ (built BEN/5/23/04) [1], which deals with stratified programs only. $\mathbb{ASP}$-$\mathbb{CLP}$, however, is based on a semantics that does not guarantee minimality of answer sets. Furthermore, our experiments with $\mathbb{ASP}$-$\mathbb{CLP}$ indicate that the cost of communication between the constraint solver and the answer set solver proves to be significant in large instances.

In this work, we explore an alternative to $\mathbb{ASP}$-$\mathbb{CLP}$ and develop a new system for computing answer sets of logic programs with aggregates. We begin with the definition of a new semantics for programs with aggregates that has the following characteristics:

- It applies to *arbitrary* programs with aggregates, e.g., no syntactic restrictions on the use of aggregates, and it is as intuitive as the traditional answer set semantics.
- It *does not* explicitly require the satisfaction of desirable properties of answer sets (such as being *closed, supported,* or *minimal*), but the answer sets resulting from the new definition *naturally* satisfy such properties.
- It can handle aggregates as head of rules (not supported yet in our implementation).
- It can be implemented by integrating the definition directly in state-of-the-art answer set solvers. In particular, it requires only the addition of a module to determine the "solutions" of an aggregate, without any modifications to the mechanisms to computer answer sets.

The syntax of the language is similar to $\mathbb{ASP}$-$\mathbb{CLP}$—where a new type of literals (aggregate literals) is used; an aggregate literal has the form $F(\{X \mid p(X_1,\ldots,X_n)\})\, op\, Val$ where $F$ is an aggregate function (e.g., SUM), and *op* is a relational operator (e.g., $=$, $\leq$). A similar literal with multisets is also available. The semantics and comparison with other approaches can be found in [3]. Its main features are:

- it defines the concept of *solution* of an aggregate $\ell$ as a pair $\langle X, Y\rangle$ such that every model $M$ of the program satisfying $X \subseteq M$ and $Y \cap M = \emptyset$ also satisfies $\ell$;
- it defines the *unfolding* of a program based on the notion of solution.

The unfolding of a program with aggregates is a normal logic program whose answer sets can be computed using off-the-shelf systems. A set of atoms $M$ is an answer set of a program with aggregates $P$ iff it is an answer set of *unfolding*$(P)$. We illustrate the semantics through the following examples.

*Example 1.* Let $P_1$ be the program

$$p(1). \qquad p(2). \qquad p(3) \leftarrow q. \qquad q \leftarrow sum(\{X \mid p(X)\}) > 5.$$

The only aggregate solution of $sum(\{X \mid p(X)\}) > 5$ is $\langle\{p(1), p(2), p(3)\}, \emptyset\rangle$ and *unfolding*$(P_1)$ contains:

$$p(1). \qquad p(2). \qquad p(3) \leftarrow q. \qquad q \leftarrow p(1), p(2), p(3).$$

which has $M_1 = \{p(1), p(2)\}$ as its only answer set. $M_1$ is the only answer set of $P_1$.

*Example 2.* Consider the program $P_2$:

$$p(2). \qquad p(1) \leftarrow min(\{X \mid p(X)\}) \geq 2.$$

The aggregate literal $min(\{X \mid p(X)\}) \geq 2$ has a unique solution $\langle\{p(2)\}, \{p(1)\}\rangle$.

$$\textit{unfolding}(P_2) = \{p(2). \qquad p(1) \leftarrow p(2), \textit{not}\ p(1).\}$$

*unfolding*$(P_2)$ does not have answer sets, i.e., $P_2$ does not have answer sets.

We will now describe SMODELS$^A$ that implements the new semantics. Source code of the system can be found at `www.cs.nmsu.edu/~ielkaban/asp-aggr.html`.

## 2 The SMODELS$^A$ System

Our main goal in developing SMODELS$^A$ is to test the feasibility of a new approach to computing the answer sets of programs with aggregates by *(i)* computing the solutions of aggregate literals; *(ii)* computing the unfolding; and *(iii)* using standard answer set solvers to compute the answer sets. For this reason, we add to LPARSE and SMODELS two new modules. One for the preprocessing and another for the computation of the unfolding program. The overall structure of our system is shown in Fig. 1. The current implementation is built using SMODELS v.2.28 and LPARSE v.1.0.13.

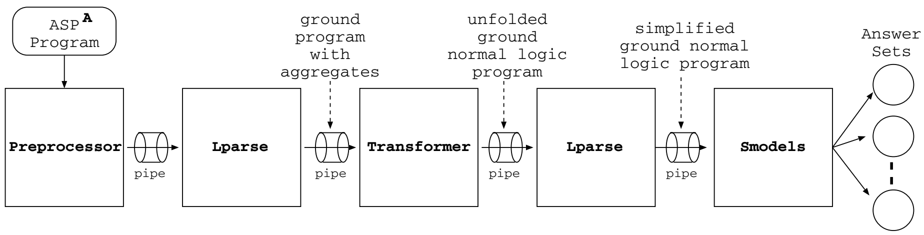
ASP^A Program
ground program with aggregates
unfolded ground normal logic program
simplified ground normal logic program
Answer Sets
Preprocessor
Lparse
Transformer
Lparse
Smodels
pipe
pipe
pipe
pipe

**Fig. 1.** Overall System Structure

Similar to the SMODELS system, the computation of answer sets of a program with aggregates is piped through several stages. In the 2nd and 4th stage, LPARSE is used. In the last stage, SMODELS is used. Let us detail the modules used in the other stages.

### 2.1 The Preprocessor

The *Preprocessor* is used to perform a number of simple syntactic transformations of the input program. These transformations are mostly aimed at rewriting the aggregate literals in a format acceptable by LPARSE. An aggregate literal of the form $f(\{X \mid p(X,Y)\})$ op $R$ is transformed into an atom, *t-aggregate atom*, of the form

```
''$agg''(f, ''$x'', p(''$x'', Y), R, op)
```

and a choice rule

$\{$`''$agg''(f, ''$x'', p(''$x'', Y), R, op)`$\} \leftarrow$ `type(Y)`

where `type(Y)` is the domain predicate specifying the possible values of $Y$. For example, the rule

$$q \leftarrow sum(\{X \mid p(X)\}) > 3.$$

is transformed to:

```
q ← ''$agg''(sum,''$x'',p(''$x''),3,greater).
{''$agg''(sum,''$x'',p(''$x''),3,greater)}.
```

The resulting program is processed by LPARSE and by the *Transformer Module*.

## 2.2 The Transformer Module

The *Transformer Module* is the major component of SMODELS$^A$. It is responsible for the computing of the unfolding of the input programs and has four components: *Reader*, *Dependencies Analyzer*, *Aggregate Solver*, and *Rules Expander*. The overall organization of the *Transformer Module* is shown in Fig. 2. The *Transformer* is completely written in Prolog.

*Reader.* The *Reader* gets the output of the first LPARSE processing and constructs three tables: the *Atoms Table*, the *Rules Table*, and the *Aggregates Table*. These tables store the ground atoms, the ground rules, and the ground t-aggregate atoms (called aggregate atoms hereafter). For each aggregate atom, the *Reader* also stores other information, such as its aggregate function (e.g., SUM, COUNT, etc.), its relational operator (e.g., $>$, $<$, etc.), the compared value, the grouped variable, and the dependent atoms skeleton (e.g., $p(X)$ where $X$ is the grouped variable). For example, the values for these attributes are SUM, $>$, 3, “$x”, p(“$x”), respectively, for the aggregate atom “$agg”(sum, “$x”,p(“$x”),3,greater).

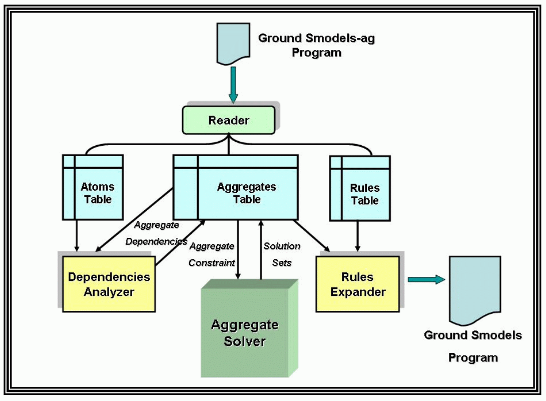

**Fig. 2.** Transformer Module

*Dependencies Analyzer (DA).* The *DA* is responsible for the identification of the dependencies between aggregate atoms and atoms contributing to such aggregates. For each aggregate literal, the *DA* searches the *Atoms Table* for its atom dependencies, constructs a set (implemented as a list) of pointers to these atoms, and stores it as a part of the aggregate information in the *Aggregates Table*. These dependencies represent the domain from which the solutions of an aggregate constraint are built. For example, the set of dependencies of the atom “\$agg”(sum, “\$x”,p(“\$x”),3,greater) consists of all the atoms of the form $p(X)$ and is $\{p(1), p(2), p(3)\}$ in the previous example.

Aggregate Solver (AS). The main task of the *AS* is to compute a minimal solution set for each aggregate atom in the program. It contains several constraint solving procedures, one for each aggregate function. Presently, it supports SUM, AVG, MIN, MAX, and COUNT and the basics relational operators $>, <, \geq, \leq, =, \neq$. For every aggregate atom in the *Aggregates Table*, *AS* identifies its aggregate function and sends it, together with its set of dependencies, to the appropriate constraint solving procedure which produces either *(i)* a minimal set of solutions needed for the unfolding of the atom, if the aggregate literal has some solutions; or *(ii) false* otherwise. This information is then stored in the *Aggregates Table*. If we consider the previous example, *AS* will return the set $\{\langle\{p(1), p(2), p(3)\}, \emptyset\rangle\}$ for the aggregate atom “\$agg”(sum, “\$x”,p(“\$x”),3,greater) with the set of dependencies $\{p(1), p(2), p(3)\}$. If the constant 3 in the aggregate literal is changed to 7, the *AS* will returns *false*.

Rules Expander (RE). The *RE* module completes the job of the *Transformer Module*, by computing the unfolding of the program. For each rule $r$ in the *Rules Table*, it generates *unfolding*$(r)$, the set of rules obtained from $r$ by simultaneously replacing each aggregate literal in $r$ by the unfolding of one of its solutions (stored in the *Aggregate Table*). The *RE* also simplifies the code to remove the temporary choice rules introduced by the *Reader*. *RE* also performs some optimizations, such as removing rules whose body contains an unsatisfiable aggregate literal. For the program $P_1$, the result of this step is the following program:

```
p(1).    p(2).    p(3) ← q.    q ← p(1),p(2),p(3).
```

**Table 1.** Benchmarks with Aggregates (times in sec.)

| Program | Sample Size | SMODELS$^A$ Time | Transformer Time | DLV$^A$ Time |
|---|---|---|---|---|
| Company Control | 20 | 0.010 | 0.080 | N/A |
| Company Control | 40 | 0.020 | 0.340 | N/A |
| Company Control | 80 | 0.030 | 2.850 | N/A |
| Company Control | 120 | 0.040 | 12.100 | N/A |
| Shortest Path | 20 | 0.220 | 0.740 | N/A |
| Shortest Path | 30 | 0.790 | 2.640 | N/A |
| Shortest Path | 50 | 3.510 | 13.400 | N/A |
| Shortest Path (All Pairs) | 20 | 6.020 | 35.400 | N/A |
| Party Invitations | 40 | 0.010 | 0.010 | N/A |
| Party Invitations | 80 | 0.020 | 0.030 | N/A |
| Party Invitations | 160 | 0.050 | 0.050 | N/A |
| Seating | 16/4/4 | 11.40 | 0.330 | 4.337 |
| Employee Raise | 15/5 | 0.57 | 0.140 | 2.750 |
| Employee Raise | 21/15 | 2.88 | 1.770 | 6.235 |
| Employee Raise | 24/20 | 3.13 | 2.420 | 26.50 |
| NM1 | 125 | 0.11 | 0.10 | N/A |
| NM1 | 150 | 0.16 | 0.13 | N/A |
| NM2 | 125 | 1.44 | 0.80 | N/A |
| NM2 | 150 | 2.08 | 1.28 | N/A |

## 3 Experiments and Benchmarks

We have experimented SMODELS$^A$ with various benchmarks (some from the literature and some newly created) and compared it with DLV$^{\mathcal{A}}$ whenever possible (Table 1). The experiments have been performed on a Linux P4 (3.06GHz, 512MB). The column SMODELS$^A$ reports the time for computing answer sets of the unfolded program, while **Transformer Time** reports the unfolding time. The performance results are acceptable in most cases; on stratified programs, our system is occasionally faster than DLV, and occasionally slower, depending on the type of aggregate (some have many solutions, that we precompute, and that are not required during answer set computation).

## 4 Discussion

We presented a new system for computing answer sets of logic programs with aggregates. The new system differs from our previous system in two ways: *(i)* it implements a different, intuitive, semantics, which leads only to minimal models; and *(ii)* it does not modify LPARSE and SMODELS. The result of our initial experimentation shows that this direction is promising. The system has not been optimized for performance and this will be our focus in the near future. In particular, we plan to

1. Improve the preprocessor, e.g., by using more sophisticated data structures (e.g., to speedup search of atoms during the *DA* phase) and to eliminate redundant aggregate atoms in the *Aggregate Table*.
2. Improve the aggregate solver to allow more than one grouping variable and additional aggregate functions (presently, it handles only one grouping variable and allows only basic aggregate functions);
3. Improve the rule expander to reduce the size of the unfolding program.

A more important work, that is in progress, is to extend our system to support a second characterization of our aggregate semantics, equivalent to the one mentioned here, which relies on unfolding w.r.t. a specific answer set; this is expected to reduce the size of the unfolding for many aggregates and simplifies the handling of aggregates in the head of the rules. Furthermore, from our experiments, it is obvious that there are aggregates that are better handled with the approach described in this paper (as they lead to a small unfolding), and others that would benefit from additional knowledge about the answer set we are building (i.e., delay the unfolding until the actual answer set computation). We plan to develop classification methods that will select the appropriate unfolding approach.

## References

1. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics,Complexity,and Implementation in DLV. *IJCAI*, 2003.
2. I. Elkabani, E. Pontelli, and T.C. Son. Smodels with clp and its applications: A simple and effective approach to aggregates in asp. *Int. Conference Logic Programming*, pp. 73–89, 2004.
3. T.C. Son, E. Pontelli, and I. Elkabani. A Translational Semantics for Aggregates in Logic Programming. Technical Report NMSU-CS-2005-005, New Mexico State University, 2005.