Smodels with CLP and its Applications: A Simple and Effective Approach to Aggregates in ASP

Islam Elkabani, Enrico Pontelli, Tran Cao Son

Department of Computer Science
New Mexico State University
{ielkaban | epontell | tson}@cs.nmsu.edu

Abstract. In this work we propose a semantically well-founded extension of *Answer Set Programming (ASP)* with aggregates, which relies on the integration between state-of-the-art answer set solvers and constraint logic programming systems. The resulting system is efficient, flexible, extensible, and supports form of aggregation more general than those previously proposed in the literature. The system is developed as an instance of a general framework for the embedding of arbitrary constraint theories within ASP.

1 Introduction

In recent years we witnessed the rapid development of alternative logical systems, called *non-monotonic logics* [3], which allow new axioms to retract existing theorems; these logical systems are particularly adequate for common-sense reasoning and modeling of dynamic and incomplete knowledge. In particular, in the last few years a novel *programming paradigm* based on non-monotonic logics, has arisen, called *Answer Sets Programming (ASP)* [13], which builds on the mathematical foundations of *logic programming (LP)* and *non-monotonic reasoning*. ASP offers novel and declarative solutions in well-defined application areas, such as intelligent agents, planning, and diagnosis.

Many practical systems have been recently proposed to support execution of *Answer Set Programming (ASP)* [16, 7, 1]. The logic-based languages provided by these systems offer a variety of syntactic structures, aimed at supporting the requirements arising from different application domains. *Smodels* and *DLV* have pioneered the introduction of language-level extensions, such as choice-literals, weight and cardinality constraints, weak constraints [16, 7] to facilitate the declarative development of applications. Nevertheless, there are simple properties, commonly encountered in real-world applications, that cannot be conveniently handled within the current framework of ASP—such as properties dealing with arithmetic and aggregation. In particular, aggregations and other forms of set constructions have been shown [6, 4, 14, 10] to be essential to reduce the complexity of software development and to improve the declarative level of the programming framework. In the context of ASP, the lack of aggregation capabilities may lead to an exponential growth in the number of rules required for the development of a solution [2].

The objective of this work is to address some of these aspects within the framework of Answer Set Programming. In particular, the main concrete objective we propose to accomplish is to develop an extension of ASP which supports a semantically well-founded, flexible, and efficient implementation of aggregates. The model of aggregates we provide is more general than the form of aggregates currently present in the Smodels system [16] (limited to cardinality and weight constraints) and those proposed in the A-Prolog system [8]. The *DLV* system has recently reported an excellent development to allow aggregates in ASP [4]; the

DLV approach covers similar classes of aggregates as those described here, although our proposal follows a radically different methodology. The model we propose has the advantage of allowing developers to easily generalize to other classes of aggregates, to modify the strategies employed during evaluation, and even to accommodate for different semantics. A proposal for optimization of aggregate constraints have appeared in [15].

We follow a fairly general and flexible approach to address these issues. We start by offering a generic framework, called $\mathbb{ASP}$-$\mathbb{CLP}$, which provides a simple and elegant treatment of extensions of ASP w.r.t. *generic constraint domains*. We then instantiate this generic framework to the case of a constraint theory for aggregates. The resulting language, called $\mathbb{ASP}$-$\mathbb{CLP}$(Agg), is then implemented following the same strategy—i.e., by relying on the integration between a state-of-the-art ASP solver, specifically the Smodels system [16], and an external constraint solver. Instead of relying directly on an external constraint solver for aggregates, we make use of an external constraint solver for *Finite Domain* constraints, specifically the ECLiPSe system [18]. The implementation is simple and elegant, and it supports easy modifications of execution strategies and the introduction of new aggregates.

2 ASP with CLP: A General Perspective

Let us consider a signature $\Sigma_C = \langle \mathcal{F}_C, \mathcal{V}, \Pi_C \rangle$, where $\mathcal{F}_C$ is a set of constants and function symbols, $\mathcal{V}$ is a denumerable set of variables, and Π_C is a collection of predicate symbols. We will refer to Σ_C as the *constraint signature* and it will be used to build constraint formulae. A *primitive constraint* is an atom of the form $p(t_1, \ldots, t_n)$, where $p \in \Pi_C$ and $t_1, \ldots, t_n$ are terms built from symbols of $\mathcal{F}_C \cup \mathcal{V}$. A *$C$-constraint* is an arbitrary conjunction of primitive constraints and their negation.

Let us also assume a separate signature $\Sigma_P = \langle \mathcal{F}_P, \mathcal{V}', \Pi_P \rangle$, where $\mathcal{F}_P$ is a collection of constants, $\mathcal{V}'$ is a denumerable collection of variables and Π_P is a collection of predicate symbols. We will refer to Σ_P as the *ASP signature* and we will denote with $\mathcal{H}_P$ the Herbrand universe built from the symbol of Σ_P and with $\mathcal{B}_P$ the Herbrand base. We will refer to a formula $p(t_1, \ldots, t_n)$, where $t_i \in \mathcal{F}_P \cup \mathcal{V}'$, as an ASP-atom; an ASP-literal is either an ASP-atom or the negation (*not* A) of an ASP-atom. An ASP-clause is a formula $A :\!\!- B_1, \ldots, B_k$ where A is an ASP-atom and $B_1, \ldots, B_k$ are ASP-literals. An $\mathbb{ASP}$-$\mathbb{CLP}$ clause is a formula of the form $A :\!\!- C \,[\!]\, B_1, \ldots, B_k$ where A is an ASP-atom, C is a C-constraint, and $B_1, \ldots, B_k$ are ASP-literals. A program is a finite collection of $\mathbb{ASP}$-$\mathbb{CLP}$ clauses.

We assume the presence of a given interpretation structure $\mathcal{A}_C = \langle A, (\cdot)^C \rangle$ for the constraint signature, where A is the interpretation domain and $(\cdot)^C$ is a function mapping elements of $\mathcal{F}_C$ (Π_C) to functions (relations) over A. Given a primitive constraint c, we will use the notation $\mathcal{A}_C \models c$ iff $(c)^C$ is true; the notion can be easily generalized to constraints. Let P be a ground $\mathbb{ASP}$-$\mathbb{CLP}$ program and let $M \subseteq \mathcal{B}_P$. We define the $\mathbb{ASP}$-$\mathbb{CLP}$-reduct of P w.r.t. M as the set of ground clauses

$$P_M^C = \left\{ A :\!\!- B_1, \ldots, B_n \;\middle|\; \begin{array}{r} A :\!\!- C \,[\!]\, B_1, \ldots, B_n, not\ D_1, \ldots, not\ D_m \in P, \\ M \not\models D_i (1 \leq i \leq m), \\ \mathcal{A}_C \models C^M \end{array} \right\}$$

where C^M denotes the grounding of the constraint C w.r.t. the interpretation provided by M. M is an $\mathbb{ASP}$-$\mathbb{CLP}$-stable model of P iff M is the least Herbrand model of P_M^C.

3 ASP with CLP: Aggregates in ASP

Our objective is to introduce 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, 6, 4]. We expect to gain similar advantages from the introduction of different forms of aggregations in ASP.

Example 1 ([14]). 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).
control(X, Z, Y, N) :- control(X,Z), owns(Z,Y,N).
fraction(X,Y,N)     :- sum({{M: (control(X,Z,Y,M):company(Z))}}) = 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, 17, 14, 5]). 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 (e.g., example 1 above). 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 aggregates have been developed, including work on the use of aggregates in ASET [8], work on sets and grouping in logic programming [6], and a recently proposed implementation of aggregates in the *DLV* system [4].

The specific approach proposed in this work accomplishes the same objectives as [4, 8]. The novelty of our approach lies in the technique adopted to support aggregates. Following the spirit of our previous efforts [6], we rely on the integration of different *constraint solving* technologies to support the management of different flavors of sets and aggregates. In this paper, we describe a back-end inference engine—obtained by integrating *Smodels* with a finite-domain constraint solver—capable of executing *Smodels* programs with aggregates. The back-end is meant to be used in conjunction with front-ends capable of generating constraints and performing high-level constraint handling of sets and aggregates (as in [6]). We will refer to the resulting system as $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) hereafter.

4 The Language

Now we will give a formal definition of the syntax and semantics of the language accepted by the $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) system. This language is an extension of the language accepted by the *Smodels* system, with the addition of aggregate functions. Observe that by making $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) an extension of *Smodels*, we accept all the constructs provided by the *Smodels* front-end (e.g., cardinality constraints).

Syntax. The input language accepted by our system is analogous to the language of *Smodels* with the exception of a new class of literals—*the aggregate literals.*

Definition 1. *An* extensional set (multiset) *is a set (multiset) of the form* $\{a_1, \ldots, a_k\}$ *(*$\{\!\{a_1, \ldots, a_k\}\!\}$*) where* a_i *are terms. An* intentional set *is a set of the form* $\{X : Goal[X, \bar{Y}]\}$*; an* intentional multiset *is a multiset of the form* $\{\!\{X : Goal[X, \bar{Y}]\}\!\}$*. In both definitions,* X *is*

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

the grouping variable *while* $\bar{Y}$ *are* existentially quantified variables. *Following the syntactic structure of* Smodels, $Goal[X, \bar{Y}]$ *is an expression of the form:* $p(X)$ *(*$\bar{Y}$ *is empty), where* $p \in \Pi_P$, *or* $p(X, \bar{Y})$ *:* $q(\bar{Y})$, *where* $p, q \in \Pi_P$ *and* $q(\bar{Y})$ *is the domain predicate for* $\bar{Y}$ *[16]. An intensional set (multiset) is ground if* $vars(Goal[X, \bar{Y}]) = \{X, \bar{Y}\}$. *An* aggregate term *is of the form* $aggr(S)$, *where* S *is either an extensional or intensional set or multiset, and aggr is a function. We will mostly focus on the handling of the "traditional" aggregate functions, i.e., count, sum, min, max, avg, times.*

With respect to the generic syntax of Section 2, in $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) we assume that $\mathcal{F}_C$ contains a collection of function symbols of the type $F^{\{\}}_{Goal[X,\bar{Y}]}$ and $F^{\{\!\{\}\!\}}_{Goal[X,\bar{Y}]}$. The arity of each such function symbol corresponds to the number of free variables different from the grouping variable and the existentially quantified variables present in $Goal[X, \bar{Y}]$—i.e.,

$$arity\left(F^{\{\}}_{Goal[X,\bar{Y}]}\right) = |vars(Goal[X, \bar{Y}]) \setminus \{X, \bar{Y}\}| = arity\left(F^{\{\!\{\}\!\}}_{Goal[X,\bar{Y}]}\right)$$

Thus, $F(\{X : Goal[X, \bar{Y}]\})$ and $F(\{\!\{X : Goal[X, \bar{Y}]\}\!\})$ are syntactic sugars for the terms: $F^{\{\}}_{Goal[X,\bar{Y}]}(Z_1, \ldots, Z_n)$ and $F^{\{\!\{\}\!\}}_{Goal[X,\bar{Y}]}(Z_1, \ldots, Z_n)$, where $\{Z_1, \ldots, Z_n\} = vars(Goal[X, \bar{Y}]) \setminus \{X, \bar{Y}\}$. Similar definitions apply to the case of aggregate terms built using extensional sets.

Definition 2. *An* aggregate literal *is of the form* $aggr(S)$ `op` *Result, where* `op` *is a relational operator drawn from the set* $\{=, !=, <, >, <=, >=\}$ *and Result is a variable or a number.*

The assumption is that the language of Section 2 is instantiated with $\Pi_C = \{=, \neq, \leq, \geq, >, <\}$. Observe that 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 other variables appearing in an aggregate literal (e.g., *Result*) are treated in the same way as variables appearing in a negative literal, requiring their presence within a domain atom in the body of the rule.

Definition 3. *An* $\mathbb{ASP}$-$\mathbb{CLP}$*(Agg) rule is in the form:* $A \leftarrow L_1, \ldots, L_n$ *where* A *is a positive literal and* $L_1, \ldots, L_n$ *are either standard literals—i.e., atoms or negated atoms—or aggregate literals.*[2] *An* $\mathbb{ASP}$-$\mathbb{CLP}$*(Agg) program is a set of* $\mathbb{ASP}$-$\mathbb{CLP}$*(Agg) rules.*

We are assuming, for simplicity, that the body of each $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) rule contains at most one aggregate literal. In $\mathbb{ASP}$-$\mathbb{CLP}$(Agg), we have opted for relaxing the stratification requirements proposed in [8], which prevent the introduction of recursion through aggregates. The price to pay is the possibility of generating non-minimal models [6, 12]; on the other hand, the literature has highlighted situations where stratification of aggregates prevents natural solutions to problems [14, 5].

Semantics. Now we will provide the stable model semantics [9] of the language, based on the interpretation of the aggregate atoms. The construction is simply an appropriate instantiation of the general definition provided in Section 2. Let us start with some terminology. Given a set A, we denote with $\mathcal{M}(A)$ the set of all finite multisets composed of elements of A, and let us denote with $\mathcal{P}(A)$ the set of all finite subsets of A.

Given a ground intensional set (multiset) s term $\{X : Goal[X, \bar{Y}\}$ ($\{\!\{X : Goal[X, \bar{Y}]\}\!\}$), and given an interpretation $M \subseteq \mathcal{B}_P$, the grounding of s w.r.t. M (denoted by s^M) is the ground extensional set (multiset) term: $\{a_1, \ldots, a_k\}$ ($\{\!\{a_1, \ldots, a_k\}\!\}$) where

$$\{(a_1, \bar{b}_1), \ldots, (a_k, \bar{b}_k)\} = \{(x, \bar{y}) \mid (x, \bar{y}) \in \ldots \wedge M \models Goal[x, \bar{y}]\}$$

[2] For simplicity we do not distinguish between the constraint and non-constraint part of each rule.

As we have seen in Section 2, we assume the existence of a predefined interpretation structure $\mathcal{A}_C = \langle A, (\cdot)^C \rangle$ for the constraint part of the language. In our case, the interpretation is meant to describe the meaning of the aggregate function and relations of $\mathbb{ASP}$-$\mathbb{CLP}$(Agg).

Let us start by defining the interpretation for the function symbols F^a, where F is an aggregate operation (e.g., *sum*, *count*) and a is either $\{\}$ or $\{\!\{\}\!\}$. Intuitively, each aggregate function is interpreted as a function over sets or multisets of integers. In particular, we assume that standard aggregate functions are interpreted according to their usual meaning, i.e., (a is an element of $\{\ \{\},\ \{\!\{\}\!\}\ \}$)

- $(sum^a)^C$ is the function that maps a set (multiset) of integers to their sum;
- $(count^a)^C$ is a function that maps a set (multiset) of integers to its cardinality;
- $(min^a)^C$ ($(max^a)^C$) is a function that maps a set (multiset) of integers to the minimum (maximum) value present in it;
- $(avg^a)^C$ is a function that maps a set (multiset) of integers to the average of its values.[3]

If s is the extensional set (multiset) term $\{a_1, \ldots, a_k\}$ ($\{\!\{a_1, \ldots, a_k\}\!\}$), then $(F_s^{\{\}})^C$ ($(F_s^{\{\!\{\}\!\}})^C$) is simply defined as the element of $\mathbb{Z}$:

$$(F_s^{\{\}})^C = (F^{\{\}})^C(\{a_1^C, \ldots, a_k^C\}) \qquad (F_s^{\{\!\{\}\!\}})^C = (F^{\{\!\{\}\!\}})^C(\{a_1^C, \ldots, a_k^C\})$$

Given an interpretation $M \subseteq \mathcal{B}_P$ and given $Goal[X, \bar{Y}]$ such that $vars(Goal[X, \bar{Y}]) = \{X, \bar{Y}\}$, for each $F^{\{\}}_{Goal[X,\bar{Y}]} \in \mathcal{F}_C$ and for each $F^{\{\!\{\}\!\}}_{Goal[X,\bar{Y}]} \in \mathcal{F}_C$ we assume that

$$(\ F^{\{\}}_{Goal[X,\bar{Y}]}\ )^C = (F^{\{\}}_{Goal[X,\bar{Y}]^M})^C \qquad (\ F^{\{\!\{\}\!\}}_{Goal[X,\bar{Y}]}\ )^C = (F^{\{\!\{\}\!\}}_{Goal[X,\bar{Y}]^M})^C$$

Similarly, we assume that the constraint structure interprets the various relational operators employed in the construction of aggregate atoms—i.e., $=, \neq, \leq, \geq, >, <$—according to their intuitive meaning as comparisons between integer numbers. We can summarize the notion of satisfaction as follows:

Definition 4 (Aggregate Satisfaction). *Given $M \subseteq \mathcal{B}_P$ and a ground aggregate atom $F(\{X : Goal[X, \bar{Y}]\})$ $\mathtt{op}$ R, then $\mathcal{A}_C \models (F(\{X : Goal[X, \bar{Y}]\})\ \mathtt{op}\ R)^M$ iff*

$$(F^{\{\}})^C\left(\{a \in \mathbb{Z} \mid \exists \bar{b} \in \mathcal{H}_P^*.\ M \models Goal[a, \bar{b}]\}\right)\ \mathtt{op}\ R$$

is true w.r.t. $\mathcal{A}_C$. Similarly, given a ground aggregate atom $F(\{\!\{X : Goal[X, \bar{Y}]\}\!\})$ $\mathtt{op}$ R, then $\mathcal{A}_C \models (F(\{\!\{X : Goal[X, \bar{Y}]\}\!\})\ \mathtt{op}\ R)^M$ iff $(F^{\{\!\{\}\!\}})^C(\{\!\{a \in \mathbb{Z} \mid \exists \bar{b} \in \mathcal{H}_P^*.\ M \models Goal[a, \bar{b}]\}\!\})\ \mathtt{op}\ R$ is true w.r.t. $\mathcal{A}_C$.

The remaining aspects of the notion of stable models are immediately derived from the definitions in Section 2, using the notion of entailment of constraints defined in Definition 4.

Observe that in the case of aggregates, this semantics definition essentially coincides with the semantics proposed by a number of authors, e.g., [12, 8, 7]. Observe also that this semantics characterization has some drawbacks—in particular, there are situations where recursion through aggregates leads to non-minimal models [12, 5], as shown in the following example.

Example 2. Consider the following program:

$$\begin{array}{llll} p(1). & p(2). & p(3). & p(5) \text{ :- } q. \\ q \text{ :- } sum(\{X : p(X)\}) > 10. & & & \end{array}$$

This program contains recursion through aggregates. It has two answer sets: $A_1 = \{p(1), p(2), p(3)\}$ and $A_2 = \{p(1), p(2), p(3), p(5), q\}$. The model A_2 is not a minimal model.

[3] Currently, *avg* returns an integer; it is straightforward to generalize it to return a real.

5 Practical Integration of Aggregates in ASP

Let us now describe how the language proposed in the previous section can be effectively and efficiently implemented. The implementation scheme we propose follows naturally from the discussion presented in the previous sections. The current prototype is available at `http://www.cs.nmsu.edu/~ielkaban/smodels-ag.html`.

5.1 Overall Design

The overall design of the proposed $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) system is illustrated in Figure 1. The structure resembles the typical structure of most ASP solvers—i.e., a preprocessing phase, which is employed to simplify the input program and produce an appropriate ground version, is followed by the execution of an actual solver to determine the stable models of the program. The preprocessor is enriched with a module used to determine dependencies between constraints present in the input program and regular ASP atoms; in our case, the preprocessor detects the dependences between aggregates used in the program and the atoms that directly contribute to such aggregates. The result of the dependence analysis is passed on to the solver (along with the ground program) to allow the creation of internal data structures to manage the constraints.

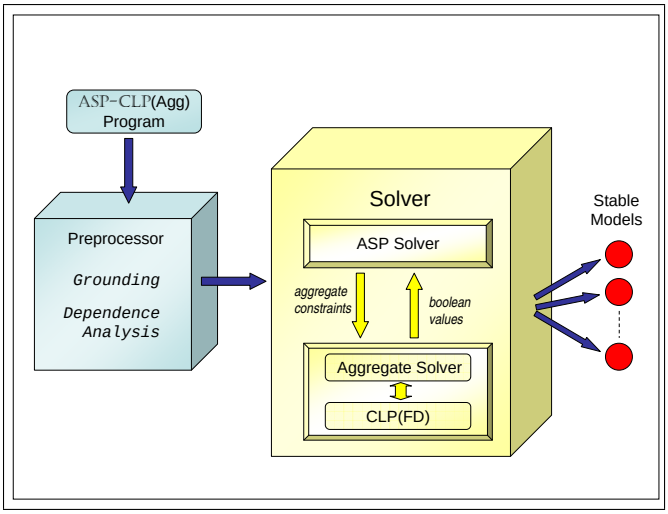

Fig. 1: Overall Design

The solver is a combination of a traditional ASP solver—in charge of handling the program rules and controlling the flow of execution—and a constraint solver; the ASP solver sends the constraints to be evaluated to the external solver, which in turn returns boolean values, representing instantiation of the boolean variables representing the components of the constraint (see Section 5.2). Intuitively, the constraint solver is employed to determine under what conditions (in terms of truth values of standard ASP atoms) a certain constraint will be satisfied. The result of the constraint processing will be used by the ASP to modify the structure of the stable model currently under construction and to proceed with the execution. Thus, the constraint solver is a "helper" in the computation of the stable models; at any point in time, relations between standard atoms exist within the data structures of *Smodels* while numerical relations expressed by aggregates exist within the constraint store. As shown in the Figure, in the specific case of $\mathbb{ASP}$-$\mathbb{CLP}$(Agg), the solver used to handle aggregate constraints is itself implemented using another constraint solver, a constraint solver over finite domains.

5.2 Representing Aggregates as Finite Domain Constraints

As described in Figure 1, each aggregate constraint in a $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) program is managed through a finite domain constraint solver. This section discusses how the encoding of aggregate constraints to finite domain constraints has been performed.

First, each atom appearing in an aggregate is represented as a domain variable with domain 0..1; the whole aggregate is then expressed as a constraint involving such variables.

The intuition behind this transformation is to take advantage of the powerful propagation capabilities of finite domain constraint solver to automatically test the satisfiability of an aggregate and prune alternatives from its solution search space. In this work we rely on the finite domain constraint solver provided by the ECLiPSe system [18]. Let us provide a brief summary of the encoding of the most relevant forms of aggregates used in $\mathbb{ASP}$-$\mathbb{CLP}$(Agg):

- **Count Aggregate:** An aggregate atom in the form $count(\{\!\{X : Goal[X, \bar{Y}]\}\!\})$ op $Result$ is represented as a finite domain constraint in the form:
$$X[i_1] \ + \ X[i_2] \ + \ \ldots \ + \ X[i_n] \ con\_op \ Result$$
where the $X[i]$'s are finite domain constraint variables representing all the ground atoms of $Goal[X, \bar{Y}]$, the i's are the indices of the ground atoms in the atom table and con_op is the ECLiPSe operator corresponding to the relational operator op. E.g., given the atoms $p(1), p(2), p(3)$, the aggregate $count(\{\!\{A : p(A)\}\!\}) \ < \ 3$ will lead to the constraint
$$X[1]::0..1, \ X[2]::0..1, \ X[3]::0..1, \ X[1]+X[2]+X[3] \ \#< 3$$
where $X[1], X[2], X[3]$ are constraint variables corresponding to $p(1), p(2), p(3)$ respectively. The handling of the corresponding aggregate in presence of sets instead of multisets is very similar and it relies on the preprocessor identifying in advance atoms that provide the same contribution, and combine them with a logical or statement. For example, assume that the extension of p contains the atoms $p(1, a), p(1, b), p(2, c)$ and we have an aggregate of the type $count(\{X : p(X, Y) : domain(Y)\})\# > 1$ it will be encoded as
$$X[i_1] :: 0..1, X[i_2] :: 0..1, X[i_3] :: 0..1, (X[i_1]\#= 1 \#\backslash/ X[i_2]\#= 1)\# <=> B1, (B1 + X[i_3])\# > 1.$$
- **Sum Aggregate:** An aggregate atom in the form $sum(\{\!\{X : Goal[X, \bar{Y}]\}\!\})$ op $Result$ is represented as a finite domain constraint in the form:
$$X[i_1] \ * \ v_{i_1} \ + \ X[i_2] \ * \ v_{i_2} \ + \ \ldots \ + \ X[i_n] \ * \ v_{i_n} \ con\_op \ Result$$
where the $X[i]$'s are finite domain constraint variables representing all the ground atoms of $Goal[X, \bar{Y}]$, the i's are the indices of the ground atoms in the atom table, v_i's are the values of X satisfying the atom $Goal[X, \bar{Y}]$ and con_op is the ECLiPSe operator corresponding to the relational operator op. E.g., given the atoms $p(1), p(2), p(3)$, the aggregate $sum(\{\!\{A : p(A)\}\!\}) < 3$ will lead to the 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. The handling of the aggregates based on intensional sets instead of multisets follows the same strategy highlighted in the case of the *count* aggregate.
- **Max Aggregate:** An aggregate atom in the form $max(\{\!\{X : Goal[X, \bar{Y}]\}\!\})$ op $Result$ is represented as a finite domain constraint in the form:
$$maxlist([\ X[i_1] * v_{i_1}, \ X[i_2] * v_{i_2}, \ldots, X[i_n] * v_{i_n} \ ]) \ con\_op \ \ Result$$
where the $X[i]$'s are finite domain constraint variables representing all the ground atoms of $Goal[X, \bar{Y}]$, the i's are the indices of the ground atoms in the atom table, v_i's are the constants instantiating the atom $Goal[X, \bar{Y}]$ and con_op is the ECLiPSe operator corresponding to the relational operator op. E.g., given the atoms $p(1), p(2), p(3)$, the aggregate $max(\{\!\{A : p(A)\}\!\})<5$ will lead to the constraint
$$X[1]::0..1, \ X[2]::0..1, \ X[3]::0..1, \ maxlist([X[1] * 1, X[2] * 2, X[3] * 3]) \ \#< 5$$
where $X[1], X[2], X[3]$ are constraint variables corresponding to $p(1), p(2), p(3)$ respectively. Observe that in this case there is no difference in the encoding if the aggregate is defined on intensional sets instead of multisets. Observe also that the contributions of the various atoms will have to be shifted to ensure that no negative contributions are present.

- **Min Aggregate:** It might seem that the representation of the $min(\{\!\{X : Goal[X,\bar{Y}]\}\!\})$ op$Result$ aggregate atom as a finite domain constraint is analogous to that of the max aggregate with the only difference of using $minlist/1$ instead of $maxlist/1$. This is not absolutely true. We have noticed a problem that might evolve when we represent the min aggregate in the same way as we did with the max aggregate. The problem is that we might have one or more values of the X_i's are set to 0, which are the X_i's that represent ground atoms having false truth values, this might lead to a wrong answer when we compute the minimum value in a list, since the result will be 0 all the time, although the real minimum value could be another value rather than 0 (the minimum value of the v_i's that correspond to the $X[i]$'s representing ground atoms having true truth values). E.g. , given the atoms $p(3), p(4), p(5)$, if we already knew that $p(3)$ and $p(4)$ are true, while $p(5)$ is false, in this case if we use the same representation as the max aggregate in representing the aggregate $min(\{\!\{A : p(A)\}\!\})<2$ that will lead to the constraint

$$X[1]::0..1,\ X[2]::0..1,\ X[3]::0..1,\ minlist([X[1]*3, X[2]*4, X[3]*5])\ \#< 2$$

This representation is wrong, since in this case the result for this constraint will be true, since the result from applying minlist will be 0 and 0 is less than two, but the correct answer should be false, since the minimum of the values that correspond to ground atoms having true truth value is 3 which is not less than 2. In order to overcome this problem, we have suggested the following representation of the aggregate atom $min(\{X : Goal(X)\})$ *op Result* as a finite domain constraint:

$$\begin{aligned}
&Y[i_1]\ \#=\ (X[i_1]{*}1){+}1,\\
&\qquad\vdots\\
&Y[i_n]\ \#=\ (X[i_n]{*}n){+}1,\\
&\quad element(Y[i_1], [\ M,\ v_{i_1}, \ldots,\ v_{i_n}\ ],\ Z[i_1]),\\
&\quad element(Y[i_2], [\ M,\ v_{i_1}, \ldots,\ v_{i_n}\ ],\ Z[i_2]),\\
&\qquad\vdots\\
&\quad element(Y[i_n], [\ M,\ v_{i_1}, \ldots,\ v_{i_n}\ ],\ Z[i_n]),\\
&\quad minlist([\ Z[i_1],\ Z[i_2],\ \ldots,\ Z[i_n]\ ])\ \ con_op\ \ Result
\end{aligned}$$

where M is a constant such that $M > v_i$, for all possible values of i, the Y_i's are selector indices that are used to select a value from the list $[\ M,\ v_{i_1}, \ldots,\ v_{i_n}\ ]$ to be assigned to the $Z[i]$'s by using the fd-library constraint $element/3$ and the $Z[i]$'s are the new list of $X[i] * v_i$ with the exception that each $X[i] * v_i$ that corresponds to an atom with a false truth value is changed to M. E.g. , by applying this treatment for the previous example, we will find that $Z[1]$ is assigned 3, $Z[2]$ is assigned 4 and $Z[3]$ is assigned a large number. In this case the result of the constraint $minlist([Z[1], Z[2], Z[3]])\ \#< 2$ is false, which is a correct answer (since $3 < 2$ is false).

The other aggregates follow similar ideas; we omit the description for lack of space.

5.3 Implementation

The implementation of $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) has been realized by introducing localized modifications in the *Smodels* (V. 2.27) system [16] and by using the ECLiPSe system (V. 5.4) as a solver for finite domain constraints. In particular, the implementation makes use of both *Smodels*—i.e., the actual answer set solver—and *lparse*—the front-end used by *Smodels* to

intelligently ground the input program. In the rest of this section we provide some details regarding the structure of the implementation.

Preprocessing. The Preprocessing 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 ground version of the program encoded in the format required by *Smodels*—i.e., with a numerical encoding of rules and with the creation of an explicit atom table. 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, describing these dependencies.)
- Generation of the constraint formulae encoding the aggregate; e.g., an entry like *"57 sum(x,use(8,x),3,multiset,greater)"* in the atom table (describing the aggregate $sum(\{\!\{X : use(8,X)\}\!\}) > 3$) is converted to *"57 sum(3,[16,32,48],"X16 * 2 + X32 * 1 + X48 * 4 + 0 #> 3")"* ($16, 32, 48$ are indices of *use(8,_)*).
- Simplification of the constraints making use of the truth values discovered by *lparse*.

Data Structures. Now we will describe in more details the modifications done to the *Smodels* system data structures, in order to extend it with aggregate functions and make it capable of communicating the ECLiPSe constraint solver. As in *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. Each standard atom includes also a list of pointers to all the aggregate literals depending on such atom.

Atom: Most of the new data structures that have been added in the new $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) system are extensions of the class Atom—used by *Smodels* to represent one atom. This is because we are introducing a new type of atoms (aggregate literals) which has its own properties. To represent these properties we have augmented the class Atom with the following fields:

- *Atom ***** **dependents**: If this atom is an aggregate constraint, *dependents* is the list of atoms this aggregate depends on.
- *Atom ***** **constraints** stores the list of aggregate literals that depends on this atom.
- *int* **met_dependents**: If this atom is an aggregate constraint, *met_dependents* is the number of its dependent atoms that still have unknown truth value.
- *EC_word* **PosCon** (**NegCon**) is an ECLiPSe data structure that holds the positive (negative) constraint to be posted into the constraint store. (e.g., X[12] #= 1).
- *EC_word* **conterm**: It is an ECLiPSe data structure that holds the aggregate constraint that will be posted into the ECLiPSe constraint store.
- *enum* {*notaggr,sum,count,min,max*} **aggr_type**: a flag describing the type of aggregate.
- *EC_ref* **hook**: It is one domain variable, representing a reified version of the constraint associated to the current aggregate atom.

Observe that, for certain aggregates, we envision the possibility of having completely different constraints for the case the constraint is required to be true and the case the aggregate is required to be false. The field **conterm** is employed to support backtracking.

Finite Domain Variables: The communication between the *Smodels* system and the ECLiPSe is a two-way communication. The *Smodels* system is capable of posting constraints into the ECLiPSe constraint solver. On the other hand, ECLiPSe is communicating with *Smodels* by either sending the truth value of a posted completed aggregate constraint or by sending back values of labeled variables appearing in a constraint corresponding to a non-completed aggregate. These types of communication require *Smodels* to be able to directly access values of finite domain variables present in the constraint store managed by ECLiPSe. This can be done by using the ECLiPSe data types *EC_refs* and *EC_ref*. We have added the following data structures as global variables in order to handle this situation:

- *EC_refs* * **X**: It is an ECLiPSe data structure that holds n references to n ECLiPSe variables, where n is the number of atoms present in the ground program. Thus, each atom is represented in the constraint store by a separate domain variable—these variables are declared as domain variables with domain $0 \dots 1$ at the beginning of the execution. The discovery of the truth value of an atom within *Smodels* can be communicated to the constraint store by binding the corresponding domain variable; on the other hand, constraints in the store can force variables to a selected value, which will be retrieved by *Smodels* and transformed in truth value assignment.

Execution Control. In this section we will describe the flow of execution of $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) in greater details. The main flow of execution is directed by *Smodels*. In parallel with the construction of the model, our system builds a *constraint store* within ECLiPSe. The constraint store maintains *one conjunction* of constraints, representing the level of aggregate instantiation achieved so far. The implementation of the $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) system required localized changes to various modules of *Smodels*. During our description for the control of execution, we are going to highlight some of the main changes that have been applied to the *Smodels* modules.

Main: during the initialization of the data structures, an additional step is performed by $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) related to the management of the aggregates. A collection of declarations and constraints are immediately posted to the ECLiPSe constraint store; these include:

- If i is the internal index of one atom in *Smodels*, then a domain variable $X[i]$ is created and the declaration $X[i] :: 0..1$ posted;
- If an aggregate is present in the program and the preprocessor has created the constraint c for such aggregate, then the constraint $B_i :: 0..1, c\#<=> B_i$ is posted in the store. The variable B_i is stored in the Atom structure for the aggregate.

These two steps are illustrated in the `post` operations (1) and (2) in Figure 2.

Expand: The goal of the *Smodels* **expand** module is to deterministically extend the set of atoms whose truth values are known (true/false) as much as possible. In our $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) system we extend the expand module in such a way that, each time an aggregate dependent 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 in the ECLiPSe constraint store. (Fig. 2, `post` operations (3) and (4)). If the ECLiPSe returns `EC_fail` this means that a conflict is detected (inconsistency), so the control returns to *Smodels* where the conflict is handled. Otherwise, ECLiPSe returns `EC_succeed` and the control returns to the **expand** module.

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 representation of the aggregate. For each aggregate, its constraint representation is reified and posted during

the initialization. If the aggregate is determined to be true (false), then we simply need to post a constraint of the type $B_i\#= 1$ ($B_i\#= 0$), where B_i is the variable reifying the constraint for the aggregate (Fig. 2, `post` operation (5)).

Observe that the constraints posted to the store have an active role during the execution:

- Constraints can provide feedback to *Smodels* by forcing a truth value for previously uncovered atoms. This means that ECLiPSe can return an answer, in terms of instantiation of previously unbound variables, to *Smodels*. This instantiation is converted into a truth value for atoms in the *Smodels* and then the control returns to the expand module again. E.g., if the constraint $(X[12]{*}2 + X[13]{*}4\#<5)\# <=> B$ is posted to the store during initialization (corresponding to the aggregate $sum(\{\!\{X : p(X)\}\!\})<4$) and $X[12]\#= 1$ has been previously posted (i.e., $p(2)$ is true), then requiring the aggregate to be true (by posting $B\#= 1$) will force $X[3]\#= 0$, i.e., $p(3)$ to be false (Fig. 2, `post` operation (5)). If there are more answers for the aggregate constraint, the control must return back to ECLiPSe for backtracking and generating another answer; this happens after *Smodels* computes the stable model containing the previous answer or fails to get a stable model containing the previous answer and backtracks.
- Inconsistencies in the constraint store have to be propagated to *Smodels*. E.g., if we have $(X[12] + X[13] + X[14]\# > 2)\# <=> B1$ (corresponding to $count(\{\!\{X : p(X)\}\!\}) > 2$) and $X[13]\#= 0$ (corresponding to $p(2)$ being false), and we finally request the aggregate to be true (posting $B1\#= 1$), then ECLiPSe will return a failure, that will activate backtracking in *Smodels*.

Check aggregate completion: An aggregate literal may become true/false not only as the result of the deductive closure computation of the *Smodels* **expand** procedure, but also because enough evidence has been accumulated to prove its status. In this case, every time an aggregate dependent atom is made true or false, the aggregate literal it appears in should be checked for truth/falsity. The test can be simply performed by verifying the value of the variable B_i attached to the reification of the aggregate constraint. If the value of B_i is 1 (0), then the aggregate can be immediately evaluated to true (false), regardless of the still unknown truth values of the rest of its dependent atoms E.g., if the constraint $(X[16]{*}1 + X[17]{*}2 + X[14] * 3\#>2)\# <=> B2$ is posted to the store (corresponding to the aggregate $sum(\{\!\{X : q(X)\}\!\})>2$) and $X[14]\#= 1$ (i.e., $q(3)$ is true), then in this case ECLiPSe instantiates $B2$ to 1, which should be translated to a true value for the atom representing the aggregate in *Smodels* (while $q(1)$ and $q(2)$ are still unknown) (Fig. 2, `check` operation (7)).

Pick: 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 constraint store (see Fig. 2, `post` operation (6)). 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 constraint store using the *repeat* and *fail* predicates of ECLiPSe. In our $\mathbb{ASP}$-$\mathbb{CLP}$(Agg) system, we have extended the *Smodels* pick module to allow aggregate atoms to be picked and its truth value is guessed in the same manner as in the case of non-aggregate atoms. Obviously, aggregate atoms that are picked are non-completed aggregate atoms since, as we mentioned previously, aggregate atoms are checked for their completion every time a dependent atom is made true or false. In this case, the picked aggregate atom is set to true (by simply posting the constraint $B\#= 1$, where B is the variable associated to the reification of the aggregate).

As mentioned, a choice point is generated (using the `repeat` predicate) into the ECLiPSe constraint store before posting the picked aggregate. If a conflict is detected, it is propagated

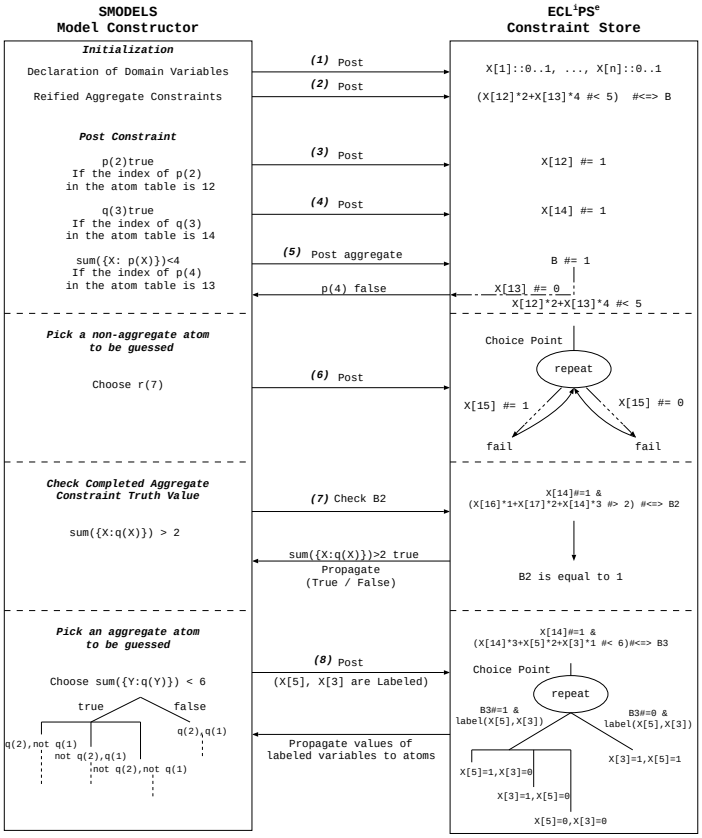

Fig. 2. Communication between *Smodels* and ECLiPSe

to the ECLiPSe constraint store (by posting the `fail` constraint to the constraint store) where a failure is generated to force backtracking to the choice point. Backtracking to a choice point will require posting the complementary constraint to the constraint store—e.g., if originally the constraint generated was $X[i]\#= 1$ ($B\#= 1$) then upon backtracking the constraint $X[i]\#= 0$ ($B\#= 0$) will be posted (see the `post` operation (6) in Fig. 2). If no conflicts were detected, then the *Smodels* will continue the computation of the model and a backtracking will take place for constructing a new model. At this point the control will return back to the ECLiPSe where a new answer is generated.

$\mathbb{ASP}$-$\mathbb{CLP}$(Agg) supports two modalities for picking aggregate atoms. Under the *lazy* modality, the truth value of an aggregate atom is guessed by simply instantiating the variable associated to the corresponding reification. E.g., if we want to guess the truth value of the aggregate $count(\{\!\{X : p(X)\}\!\}) < 2$, which was initially reified as $(X[3]+X[5]+X[6]\#< 2)\#<=> B$, then the pick operation will simply generate a choice point and post the constraint $B\#= 1$ (and $B\#= 0$ during backtracking). Note that this may not immediately provide enough information to the constraint solver to produce values for the variables $X[3], X[5], X[6]$. Under the *eager* modality, we expect the constraint solver to immediately start enumerating the possible variable instantiations satisfying the constraint; thus when the aggregate is picked, we also request the constraint store to *label* the variables in the constraint (Fig. 2, `post` operation (8)). In the previous example, when the aggregate is picked we not only request its constraint to be true (by posting $B\#= 1$) but we also post a $labeling([X[3], X[5], X[6]])$.

Evaluation. The implementation of the resulting system has been completed, and it is available at `www.cs.nmsu.edu/~ielkaban/smodels-ag.html`. The current prototype, built using *Smodels* and ECLiPSe is stable and it was used to successfully implement a number of benchmark programs. The execution speed is good, thanks to the good implementation of the ECLiPSe interface (which limits the communication overhead between the two systems). Furthermore, the system has demonstrated excellent ability to reduce the search space for programs that contain a number of aggregates related to the same predicates—their representations as constraints and the propagation mechanisms of ECLiPSe allows to automatically prune a number of irrelevant alternatives.

Work is in progress to optimize the implementation and to perform formal efficiency comparisons with other relevant systems (e.g., with *Smodels* for programs that can be encoded using limited forms of aggregation—e.g., only cardinality constraints—and with *DLV*).

6 Example

In this section we discuss a problem presented in [14, 11]. The main idea of the problem is to send out party invitations considering that some people will not accept the invitation unless they know that at least k other people from their friends accept it too. Suppose a situation as the one represented in the program below. In this case, Mary will not accept the invitation unless Sue does and Sue will not accept it unless Mary does. According to the semantic of our language (discussed in Section 4) we are expecting two situations. In the first situation we assume that there is a bad communication between Mary and Sue and in this case a deadlock situation will occur and neither of them will accept the invitation. The other situation is that both of them simultaneously accept. We assume here that each person has a unique name. The relation requires(X, K) is true when an invited person X requires at least K other people of X's friends to accept the invitation. The relation friends(X, Y) is true when a person X is a friend of person Y. The relation person(X) is used as a domain predicate.

```
requires(ann,0).  requires(rose,0). requires(mary,1).  requires(sue,1).
person(mary).    person(sue).    person(ann).   person(rose).
friend(mary,sue).     friend(sue,mary).
coming(X) :- requires(X,0).
coming(X) :- requires(X,K), count({{ Y: kc(X,Y)}}) >= K.
kc(X,Y)   :- friend(X,Y), coming(Y).
```

The result of running the previous program on the *Pre-Analyzer* is:

```
requires(ann,0). requires(rose,0). requires(mary,1). requires(sue,1).
person(mary).    person(sue).    person(ann).   person(rose).
friend(mary,sue).   friend(sue,mary).
coming(X) :-  requires(X,0).
coming(X) :-  requires(X,K), count(y,kc(X,y),K,multiset,greateq).
{count(y,kc(X,y),K,multiset,greateq)} :- requires(X,K).
kc(X,Y)   :-  friend(X,Y),coming(Y).
```

The result from the *Pre-Analyzer* is passed to *lparse*, and the result obtained from *lparse* on the *Pre-Analyzer* output is passed to the *Post-Analyzer*. The Model Computation module on the *Post-Analyzer* output produces the following models (we highlight the relevant parts):

```
Stable Model:                          Stable Model:
    coming(mary), coming(sue)               coming(ann)
    coming(ann), coming(rose)               coming(rose)
```

7 Discussion

Various proposals have been put forward to provide alternative semantics for logic programming with aggregates [5, 14], trying to address some of the limitations of the simple semantics adopted in the previous section. A natural alternative semantics, which removes the presence of non-minimal models, can be defined as follows.

Definition 5 (Aggregate Solution Set). *Let us consider a ground aggregate literal α of the form $F\{\!\{X : Goal[X, \bar{Y}]\}\!\}$ `op` Result. Let us denote with $\mathcal{S}(\alpha)$ the following set:*

$$\mathcal{S}(\alpha) = \left\{ \{(a_1, \bar{b}_1), \ldots, (a_n, \bar{b}_n)\} \;\middle|\; \begin{array}{r} a_i, \bar{b}_i \text{ are ground terms and} \\ Goal[a_i, \bar{b}_i] \text{ is a legal grounding of } Goal[X, \bar{Y}] \text{ and} \\ \mathcal{A}_C \models (F\{\{a_1, \ldots, a_n\}\} \texttt{ op } Result) \end{array} \right\}$$

We will refer to $\mathcal{S}(\alpha)$ as the Aggregate Solution Set.

Definition 6 (Aggregate Unfolding). *Let α be the ground aggregate $F\{\!\{X : Goal[X, \bar{Y}]\}\!\}$ `op` Result. We define the* unfolding *of α (*unfold(α)*) as the set of formulae*

$$unfold(\alpha) = \left\{ \bigwedge_{(a,\bar{b}) \in S} Goal[a, \bar{b}] \wedge \bigwedge_{(a,\bar{b}) \notin S} not\, Goal[a, \bar{b}] \;\middle|\; S \in \mathcal{S}(\alpha) \right\}$$

We also define the unfold of a non-aggregate literal A as the set containing only A (i.e., $unfold(A) = \{A\}$). The unfolding of a clause $H :\!- B_1, \ldots, B_n$ is defined as the set of clauses:

$$unfold(H :\!- B_1, \ldots, B_n) = \{(H :\!- \beta_1, \ldots, \beta_n) \mid \beta_i \in unfold(B_i), 1 \leq i \leq n\}$$

*The unfolding of a program P (*unfold*(P)) is obtained by unfolding each clause in P.*

Definition 7 (Alternative Stable Model Semantics for Programs with Aggregates). *Let M be an Herbrand interpretation and let P be a program with aggregates. M is a stable model of P iff M is a stable model of $unfold(P)$.*

Example 3. Consider the program

```
p(1).                p(2).                p(3).
p(5) :- q.           q :- sum({{X:p(X)}}) > 10.
```

The unfold of this program yields a program which is identical except for the last rule:

```
q :- p(1), p(2), p(3), p(5).
```

since $\{\!\{1, 2, 3, 5\}\!\}$ is the only multiset that satisfies the aggregate. The resulting program has a single answer set: $\{p(1), p(2), p(3)\}$, thus the non-minimal model accepted in the former semantic characterization (see Example 2) is no longer a stable model of the program.

This alternative semantics characterization can be supported with minimal changes in the proposed system. In particular, the construction and handling of the constraints encoding aggregate computations is unchanged. The only changes required are in the management of the declarative closure computation in presence of aggregates within *Smodels*. The presence of non-minimal models derive from true aggregates being treated as facts, loosing the dependencies between the aggregate and the atoms it depends on. These dependencies can be restored by dynamically introducing a rule upon satisfaction of an aggregate—where the body of the rules includes the true atoms satisfying the aggregates (readily available from the data structures provided by the preprocessor).

8 Conclusions and Future Work

A prototype implementing these ideas has been completed and used on a pool of benchmarks. Performance is acceptable, but we expect significant improvements by refining the interface with ECLiPSe. Combining a constraint solver with *Smodels* brings many advantages:

- 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; e.g., by making use of Constraint Handling Rules we can implement different strategies to handle constraints and new constraint operators;
- it is a straightforward extension to allow the user to declare 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 combinations of truth values of selected atoms involved in the aggregate). In this way, the aggregate will act as a generator of solutions instead of just a pruning mechanism.

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 modifications to *Smodels* and introduces insignificant overheads for regular programs. The prototype confirmed the feasibility of this approach.

In our future work, we propose to further relax some of the syntactic restrictions. E.g., the use of labeling allows the aggregates to "force" solutions, so that the aggregate can act as a generator of values; this may remove the need to include domain predicates to cover the result of the aggregate (e.g., the *safety* condition used in *DLV*).

References

1. Y. Babovich and V. Lifschitz. Computing Answer Sets Using Program Completion.
2. C. Baral. *Knowledge Representation, reasoning, and problem solving*, Cambridge, 2003.
3. C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *JLP*, 19/20, 1994.
4. T. Dell'Armi et al. . Aggregate Functions in DLV. *2nd Intl. ASP Workshop*, 2003.
5. M. Denecker et al. Ultimate well-founded and stable semantics for logic programs with aggregates. In *ICLP*, Springer. 2001.
6. A. Dovier, E. Pontelli, and G. Rossi. Intensional Sets in CLP. *ICLP*, Springer Verlag, 2003.
7. T. Eiter et al. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. *KRR*, 1998.
8. M. Gelfond. Representing Knowledge in A-Prolog. *Logic Programming&Beyond*, Springer, 2002.
9. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. *ILPS*, MIT, 1988.
10. S. Greco. Dynamic Programming in Datalog with Aggregates. *TKDE*, 11(2), 1999.
11. D. Kemp et al. Efficient Recursive Aggregation and Negation in Deductive Databases. *TKDE*, 10(5), 1998.
12. D. Kemp and P. Stuckey. Semantics of Logic Programs with Aggregates. *ILPS*, MIT Press, 1991.
13. V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, pages 375–398, 1999.
14. K. A. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. *JCSS*, 54:79–97, 1997.
15. K. Ross et al. Foundations of Aggregation Constraints. *TCS*, 193(1-2), 1998.
16. P. Simons et al. Extending and Implementing the Stable Model Semantics. *AIJ*, 138(1–2), 2002.
17. A. Van Gelder. The Well-Founded Semantics of Aggregation. In *PODS*, ACM Press, 1992.
18. M. Wallace, S. Novello, J. Schimpf. ECLiPSe: a Platform for Constraint Logic Programming. IC-Parc, Imperial College, 1997.