DEPARTMENT OF MATHEMATICS & COMPUTER SCIENCE

Master's degree in Artificial Intelligence & Computer Science

Master's Thesis

Implementation of a novel propagator for aggregates

Supervisors:

Prof. Carmine Dodaro

Prof. Dr. Thomas Eiter

Co-supervisor:

Dott. Tobias Geibinger

Candidate:

Salvatore Fiorentino

Mat. 242706

ACADEMIC YEAR 2023/2024

Inserire dedica e ringraziamenti...

Contents

Introduction

Answer Set Programming (ASP) is a highly utilized framework for knowledge representation and automated reasoning, as highlighted by *Marek and Truszczyński (1999)* [1] and *Niemelä (1999)* [2]. In ASP, combinatorial problems are formulated using logical rules that incorporate various linguistic constructs, which simplify the representation of complex knowledge. In its most basic form, ASP programs consist of normal logic rules, where each rule has a head atom and a body that is a conjunction of literals. Often, normal programs are extended by incorporating aggregates, as discussed by *Bartholomew et al. (2011)* [3], *Faber et al. (2011b)* [4], *Ferraris (2011)* [5], *Gelfond and Zhang (2014)* [6], *Liu et al. (2010)* [7], *and Simons et al. (2002)* [7]. Specifically, *SUM aggregates* are used in rule bodies, where literals are assigned weights, and the sum of the weights of the true literals must satisfy a specified (in)equality. When the head of the rule is false the aggregate becomes a constraint aggregate. Another kind of constraint is the *At Most One* constraint that essentially inhibits truth of pairs of literals in a given set. It is very common that these two constraints (SUM and AMO) are linked together. State of the art solvers treat this case ignoring the correlation between these two constraints. Our work aims to define a new costruct named *AMO-SUM* to efficiently handle this case. This efficiency improvement comes from the fact that we are able to treat the two constraint as a whole constraint.

Nowadays current ASP solver implements a (CDCL) algorithm with propagators, as explained by *Gebser et al. (2012)* [8]. CDCL (Conflict-Driven Clause Learning) is a contemporary form of non-chronological backtracking that follows the *choose-propagate-learn* pattern, as described by *Marques-Silva et al. (2021)* [9], [10].

This pattern consists of this three phases: **choose** phase, or decision phase, consists in picking a branking literal as true; **propagate** phase derives determinists consequences of the current state; **learn** phase is triggered when a conflict arises and aims at understanding from the conflict to not making the same mistake again. In the propagate phase specific procedures called *propagators* are used to derive such consequences. Propagators are required to explain why a consequence has been derived. This explanation is called reason, it is a set of literals that led to derive that consequence and it is used in the learning phase. When an aggregate is introduced inside the program then a specific propagator is required. Our work provides both a novel propagator for handling the new AMO-SUM construct and an algorithm to minimize the reasons of the aggregate-derived consequences. To provide a more comprehensive understanding of our work the

DA FINIRE

Chapter 1

Background

This chapter defines all the background needed to explain our work, trying to guide the reading through a clear and intuitive idea. Initially will be showed the syntax and semantics of normal program (section **??**), primarly focusing on notions relevant for this thesis, for instance on some extension such as the SUM constraints. Then some specific cases of SUM constraints, the so-called ALO and AMO constraints, will be explained in section (**??**). Afterwards, the current State-of-Art ASP solver algorithm to find a stable model will be discussed (section **??**), focusing on the concept of *propagator*.

1.1 Syntax and Sematincs

This section will proceed in a bottom-up fashion, introducing the most basic element to the concept of *program* and *stable model*.

The first element is the set of *atoms*, let $\mathcal{A}$ be such set. $\neg$ is a symbol representing the common negation in logic. A *literal* is an atom with possibly the negation symbol in front of it. For instance $a \in \mathcal{A}$ is a literal (and an atom) and $\neg b$ with $b \in \mathcal{A}$ is a literal (but not an atom); a is said to be a *positive literal* instead b is a negative literal. In a more formal way: given a literal $\ell \in L$, ℓ is positive if $\ell \in \mathcal{A}$ otherwise it is negative. Let L be a set of literals. Given $\ell \in L$ then $\overline{\ell}$ denotes its complement, if ℓ is a positive literal, i.e. $\ell = a \in \mathcal{A}$, then $\overline{\ell} = \neg a$ else when $\ell = \neg a$ (negative literal) with $a \in \mathcal{A}$ then $\overline{\ell} = a$. A set of literal L can be negated, written $\overline{L}$; $\overline{L}$ is equivalent to the set of literals of L where each literal is negated, that is, $\overline{L} = \{\overline{\ell} \mid \ell \in L\}$.

Each atom can be mapped to a truth value (boolean value) by an *intepretation*. An *intepretation* (or *assignment*) I is a set of literals where $I \cap \overline{I} = \emptyset$. If $\mathcal{A} \subseteq (I \cup \overline{I})$ then I is called *total-intepretation*, otherwise it is a *partial-intepretation*. On one hand if $\ell \in I$ then ℓ is *true* under I, on the other hand if $\ell \in \overline{I}$ then it is *false* under I. Abusing of notation, If $\ell \in I$ let $I(\ell) := 1, 0$ otherwise; if $\ell \notin I$ let $I^{\uparrow}(\ell) := 1, 0$ otherwise. If either $\ell \notin I$ and $\ell \notin \overline{I}$ then ℓ is said to be *undefined*.

Now the first main brick can be presented: the rule. A rule is a classic implication in propositional logic.

$$r: \quad p \leftarrow \ell_1, \ldots, \ell_n \tag{1.1}$$

where p is an atom and $\ell_1, \ldots, \ell_n$ with $n \geq 0$ are literals. The rule r **??** is equivalent to $\ell_1 \wedge \ldots \wedge \ell_n \rightarrow p$. As in propositional logic p and $\ell_1, \ldots, \ell_n$ are named respectively *head* and *body* of the rule. The head of r is defined by the symbol $H(r) := p$ and the body by the set $B(r) := \{\ell_1, \ldots, \ell_n\}$. Each rule has a *positive* and *negative* part, and it is stricly linked with the concept of positive and negative literal. On one hand, the positive body of the rule r, named $B^+(r)$, is the set of positive literals of r, that is, $B^+(r) = \{\ell \mid \{\ell\} \cap \mathcal{A} \neq \emptyset\}$. On the other hand, the negative body of the rule r, named $B^-(r)$, is the set of negative literals of r, namely, $B^-(r) = \{\ell \mid \{\ell\} \cap \mathcal{A} = \emptyset\}$

Now the relation $\models$ (satisfies, or is model of) will be inductly defined: let I be an intepretation, if $\ell \in I$ then $I \models \ell$; if $\ell \in \overline{I}$ then $I \models \overline{\ell}$; if $I \models \ell_+$ for every $\ell_+ \in B^+(r)$ and $I \models \ell_-$ for every $\ell_- \in B^-(r)$, then $I \models B(r)$; if whenever $I \models B(r)$ also $I \models H(r)$ then $I \models r$. A (normal) program Π is a set of rules.

Let's now shift to another concept that will allow us to transition from a normal program to an extension of it: the concept of *SUM constraint*. A SUM constraint (or simply constraint) has the following form

$$\text{SUM}\{w_1 : \ell_1;\ \cdots\ ;\ w_n : \ell_n\} \geq b \tag{1.2}$$

where $n \geq 0$, $\{\ell_1, \ldots, \ell_n\}$ is a set of literals such that $\ell_i \neq \ell_j$ for all $i, j \in \{1, \ldots, n\}$ such that $i \neq j$ and $\{b, w_1, \ldots, w_n\}$ is a set of naturals numbers.

Let σ be a constraint of the form **??** then bnd_σ represents the *bound* of the constraint *sigma*; $w_1, \ldots, w_n$ are the weights of each literal in σ; $lits_\sigma$ is the set of literals of σ. Let's specify the relation $\in$ as $(w_i, \ell_i) \in \sigma$ to be read as $(w_i : \ell_i)$ is an element in (the aggregation set of) σ. The function $wh_\sigma(\ell_i) = w_i$, namely, this is a function that maps every literal of σ to its weight. Intuitively, the constraint is satisfied w.r.t. an interpretation I if summing the weight of the true literals under I the value is greater or each to the bound. More formally, extending the relation $\models$, σ is satisfied if $\sum_{i=1}^{n} wh_\sigma(l_i) \times I(l_i) \geq bnd_\sigma$, written $I \models \sigma$. Note that σ may be omitted from the above notation if its meaning is clear from context.

Just as a side note: in ASP-Core-2 standard [11] the constraint **??** is written as the headless rule $| : - \quad \#sum\{w_1, \ell_1 : \ell_1; ...; w_n, \ell_n : \ell_n\} < b.|$ Now a more formal definition of program can be given: a program Π is a set of rules and constraint, referred as $rules(\Pi)$ and $constraints(\Pi)$ respectively. The set of rules and constraints define the set of atoms of the program and they are named $atoms(\Pi)$. Finally, I satisfies Π, written $I \models \Pi$, if $I \models r$ for all $r \in rules(\Pi)$ and $I \models \sigma$ for all $\sigma \in constraints(\Pi)$.

fare più carino

To make the discussion made so far more understandable, we will introduce the following example:

Example 1 (Running example)**.** Let Π_{run} be the following:

$$
\begin{array}{lll}
r_\alpha : & \alpha \leftarrow \neg\alpha' & \alpha \in \{x, y, z\} \\
r_{\alpha'} : & \alpha' \leftarrow \neg\alpha & \alpha \in \{x, y, z\} \\
\sigma_1 : & \text{SUM}\{1 : \overline{x};\ 1 : \overline{y} \qquad \} \geq 1 & \\
\sigma_2 : & \text{SUM}\{1 : x;\ 2 : y;\ 2 : z\} \geq 3 &
\end{array}
$$

Note that there are six atoms (x, y, z, x', y', z'), six rules and two constraints. $wh_{\sigma_1}(\overline{x}) = 1, wh_{\sigma_1}(\overline{y}) = 1, wh_{\sigma_2}(x) = 1, wh_{\sigma_2}(y) = 2, wh_{\sigma_2}(z) = 2, bnd_{\sigma_1} = 1, bnd_{\sigma_2} = 3$.

A possible (total) interpretation I satisfying Π_{run} is: $I_1 = \{x, z, \overline{y}, x', y', z'\}$. Since, $\sum_{i=1}^{2} wh_{\sigma_1}(l_i) \times I_1(l_i) = \sum 1 \times I_1(\overline{x}) + 1 \times I_1(\overline{y}) = 1 \geq bnd_{\sigma_1}$ and $\sum 1 \times I_1(x) + 2 \times I_1(y) + 2 \times I_1(z) = 3 \geq bnd_{\sigma_2}$. Given a program Π and an intepretation I, its *reduct* is defined as follows: $\Pi^I = \{H(r) \leftarrow B^+(r) \mid r \in rules, I \models B(r)\}$, please note that $constraints(\Pi^I) = \emptyset$. One interpretation

may differ from another due to its stability. An interpretation I is a stable model of a program Π if $I \models \Pi$ and there is no $J \subset I$ such that $J \models \Pi^I$. Let $SM(\Pi)$ denote the set of stable models of Π. Taking in consideration example **??** we can notice that I_1 is not a stable model, that is, $I_1 \notin SM(\Pi_{run})$. This is because taking the partial assignment $J_1 = \{y'\}$ then $J_1 \models \Pi_{run}^{I_1}$ since $\Pi_{run}^{I_1} = \{y' \leftarrow\}$ and $J_1 \subset I_1$. Instead $I_2 = \{x, z, \overline{y}, \overline{x'}, y', \overline{z'}\} \in SM(\Pi_{run})$.

Continuing talking about the above example, a last consideration that will let us to move towards the next chapter has to be addressed. The constraint σ_1 is a special one, it says: at least 1 of the 2 literals has to be satisfied. Notice that all the literals are flipped, so referring to the literals without being flipped it is actually saying 'at least 1 of the 2 have to be falsified' , and since $2 = 1 - 1$ it can be reformulated as 'at most 1 of them can be satisfied'. Thus, with an *At least One (ALO)* sum constraint is possible to define an *At Most One (AMO)* constraint.

1.2 ALO (clauses) and AMO as a Special Cases

Given a set $\{\ell_1, \ldots, \ell_n\}$, with $n \geq 1$, an *At Least One (ALO)* constraint over this set will enforce to have at least one literal true. As in the example **??** it can be expressed as:

$$\text{SUM}\{1 : \ell_1;\ \cdots\ ;\ 1 : \ell_n\} \geq 1 \tag{1.3}$$

When it is not ambiguous this constraint can be written as $\{\ell_1, \ldots, \ell_n\}$, usually this constraint is named also *clause* (following the *CNF* notation of the propositional logic). Modern ASP solvers enrich the input program with clauses enforcing I to be a model of rules of the form (**??**) (i.e., $\{p, \overline{\ell_1}, \ldots, \overline{\ell_n}\}$). If over this set instead is enforced an *At Most Constraint*, i.e., at most one literal is true, the following constraint is introduced:

$$\text{SUM}\{1 : \overline{\ell_1};\ \cdots\ ;\ 1 : \overline{\ell_n}\} \geq n - 1 \tag{1.4}$$

As I attempted to introduce in the precedent chapter, to enforce that at most one literal is true of a given set it is enough to enforce that at least $n - 1$

literals are falsified. Since, intuitively, if two different literals are satisfied then $n-2$ literals are not falsified. More formally given an interpretation I, I satisfies at most 1 literal if $\sum_{i=1}^{n} I(\overline{\ell_i}) \geq n-1$, or equivalently $\sum_{i=1}^{n} I(\ell_i) \leq 1$. The AMO constraint (**??**) is compactly written $\text{AMO}\{\ell_1, \ldots, \ell_n\}$.

Example 2 (Continuing Example **??**)**.** Note that σ_1 is the clause $\{\overline{x}, \overline{y}\}$, or also the AMO constraint $\text{AMO}\{x, y\}$.

1.3 Stable Model Search, Propagators and Learning

Currently ASP solvers employ a *conflict-driven clause learning* (CDCL) algorithm [8] to search a stable model. This is an algorithm also used in a lot of SAT solvers and it has been revealed to be a very effective one. The CDCL follows a pattern called *choose-propagate-learn*, where: the *choose* phase consists in deciding a branching literal to become true; *propagate* phase involves to deterministically derive new consequences from the current state (interpretation); *learn* phase, when a conflict arises, understands some new *clause* (constraint) that was not explicity defined in the program. To dive into each of these phases, understanding the CDCL, some previous concepts have to be mentioned.

A *conflict* occurr when two literals $\ell, \overline{\ell}$ are together in the interpretation. Given two clauses C, D (as described in **??**) and a literal ℓ such that $\ell \in C$ and $\overline{\ell} \in D$ then the *resolution* step of C and D upon ℓ, written $C \otimes_\ell D$, is equal to $(C \setminus \{\ell\}) \cup (D \setminus \{\overline{\ell}\})$. Intuitively, if a program Π has such clauses (constraints) C and D then since an intepretation cannot be simultaneously satisfies ℓ and $\overline{\ell}$ then $C \setminus \{\ell\}$ or $D \setminus \{\overline{\ell}\}$ have to be satisfied, thus the following clause $(C \setminus \{\ell\}) \cup (D \setminus \overline{\ell})$ has to be satisfied. Please note that if an intepretation $I \models \Pi$ then $I \models C \otimes_\ell D$. So $C \otimes_\ell D$ is implied by the program, ad it is said to be *redundant*.

In the *choose* phase a branching (undefined) literal is selected. Every branching (or decision) literal is paired with a *decision level*. To understand the decision level is enougth to now that at each choose phase the

corresponding literal is added into a 'list' and the decision level is the index into the list (index starting from 1). The term *backjumping* to a certain level l refers to the process of 'forgetting' every decision made after decision level l, including the respective propagated literals, and then resuming the decision-making process starting from level l.

In the *propagate* phase at the instant when a literal ℓ is derivated (or propagated) then a *reason*, written $reason(\ell)$, is specified. This reason defines *why* ℓ has been propagated. More in detail, the reason is a clause or this form $\{\ell\} \cup \{\overline{\ell_1}, \ldots, \overline{\ell_n}\}$. It represents the implication rule $\ell_1 \wedge \ldots \wedge \ell_n \rightarrow \ell$ and it specifies that when all the literals $\ell_1 \wedge \ldots \wedge \ell_n$ are true then also ℓ must be true. Can happen that inferring a literal ℓ creates a conflic (i.e., $\overline{\ell} \in I$), in that case ℓ is a *conflict literal*. Moreover ℓ is also paired with a decision literal that it is inherited from the current decision level of the search. *Unit Propagation* is a specific kind of propagation, it is applied when given a clause C all its literals are falses except for a literal ℓ, that is, $C \setminus \overline{I} = \{\ell\}$. In this case the only way to satisfy C is setting ℓ to true, thus ℓ is propagate to true; the reason is exactly because the other literals are falses, so $reason(\ell) = C$. Please not the $reason(\ell)$ respresents $\overline{\ell_1} \wedge \ldots \wedge \overline{\ell_n} \rightarrow \ell$, where $\{\ell_1 \wedge \ldots \wedge \ell_n\} = C \cap \overline{I}$.

The whole algorithm, initially starts with an empty assignment (all literals undefined) and a the decision level (dl) to 0 , then the *propagate* phases takes place, inferring all the consequences. If a conflict is detected it means that the program is not satisfiable, since without any choice we got a conflict. Else if no conflict is deteceted then the *choose* phase decides the new branching literal. An important note is that just the branching literals are without a reason, since they do not follows from any logic reasoning. Then, again the propagate phase is executed. If a conflict is detected then a process named *conflict analysis* starts. Starting from the reason of the conflict literal a (*backward*) resolution upon a literal of the last decision level is perfomed. This step is iteratively done until the new obtained (redundant) clause contains just one literal of the current decision level, this clause in called *Unique Implication Point (UIP)*. Given the UIP then a *backjump* is perfomed to the second highest decision level (*assertion level*).

The assertion level is special in the sense that it is the deepest level at which adding the conflict-driven clause (UIP) would allow unit resolution to derive a new implication using that clause. Since the literal with highest dl after backjumping is the only undefined literal inside UIP then the unit propagation will inferr that literal, that is, will flip the previous value. When the highest literal in the UIP has 0 as dl then the assertion level is -1 by default. Please note that the starting reason cannot cannot but have at least one literal of the last decision level, otherwise the unit propagation cannot be perfomed. After this a new choice is made until either all atoms are defined or the assertion level is -1. The above described algorithm is defined below.

Algorithm 1 Typical CDCL algorithm

Input: An ASP program Π

```
1  begin
2      I ← [ ]
       dl ← 0
       if (Propagate(Π, I) == CONFLICT) then
3          return Unsat
4      while not AllVariablesAssigned(Π, I) do
5          ℓ = PickBranchingLiteral(Π, I)
           dl ← dl + 1
           store(I, ℓ, dl)
           if (UnitPropagation(Π, I) == CONFLICT) then
6              β = ConflictAnalysis(Π, I)
               if (β < 0) then
7                  return Unsat
8              else
9                  Backjump(Π, I, β)
                   dl ← β
10     return I
```

Let's assume for now that the *Propagate* function is exactly equal to *Unit propagation*, this is usefull for the next example.

Example 3. Let Π have, among others, the rules

$$x \leftarrow \neg z \quad y \leftarrow \neg z \quad w \leftarrow x, y$$

Hence, a modern ASP solver materializes the clauses

$$\{x, z\} \quad \{y, z\} \quad \{w, \overline{x}, \overline{y}\}$$

For readability at each literal is associated the relative decision level as superscript. Let's assume that the current interpretation is $I = [\overline{w}^1]$ and the decision level is $dl = 1$. Since no propagation is performed the algorithm goes directly to the propagate phase, let's assume that $\overline{z}$ is selected. Then, by unit propagation, x and y are inferred, thanks to the first two clauses. The correspective reasons are: $reason(x) = \{x, z\}$, $reason(y) = \{y, z\}$. Now, the current intepretation I is equal to $[\overline{w}^1, \overline{z}^2, y^2, x^2]$. The last clause $\{w, \overline{x}, \overline{y}\}$ is not satisfied by I so a conflict k is detected and the $reason(k) = \{w, \overline{x}, \overline{y}\}$. Since there are more than 1 literal of the decision level 2 (the last one) a backward resolution step is perfomed. Let's take arbitrarily $\overline{x}$: $\{w, \overline{x}, \overline{y}\} \otimes_{\overline{x}} \{y, z\} = \{w, \overline{y}, z\}$. Since both $\overline{y}, z$ are with decision level 2 then let's take arbitrarily $\overline{y}$: $\{w, \overline{y}, z\} \otimes_{\overline{y}} \{y, z\} = \{w, \overline{y}, z, z\} = \{w, \overline{y}, z\}$.
$\{w, \overline{y}, z\}$ is *UIP* so let's backtrack to the assertion level 1. This will unit propagate z with $reason(z) = \{w, \overline{y}, z\}$. So the intepretation will look like: $I = [\overline{w}^1, z^1]$

The main difference between a SAT solver and a ASP solver can be summarized focusing on the function $Propagate(\Pi, I)$. SAT solvers employ mainly unit propagation inside the *Propagate* function, instead ASP solvers use, in addition on unit propagation, other two fundamental propagation functions: *Unfounded-free propagation* and *AggregatesPropagation*. The first one (*Unfounded-free propagation*) asses that the intepretation is a stable model, but its details are out of the scope of this work. The second one (*AggregatesPropagation*) is used to derive new consequences from the aggregates present in the program. For a SUM constraint σ of the form (**??**), solvers typically employ a specific *aggregate propagator* [12], [13], which essentially adds to I the literal ℓ_i ($i \in [1..n]$) if ℓ_i is required to (possibly)

reach the bound b, i.e., if

$$\sum_{j\in[1..n],j\neq i} w_j \cdot I^{\uparrow}(\ell_j) < b \tag{1.5}$$

In this case $\mathit{reason}(\ell_i) = \{\ell_i\} \cup \mathit{lits}_\sigma \cap \overline{I}$. Intuitively, the falses literals are the only justification why the bound bnd_σ could be not reached if ℓ_i was not added to I. In the special case of (**??**), that is, if σ is $\text{AMO}\{\ell_1, \ldots, \ell_n\}$, the literal $\overline{\ell_i}$ ($i \in [1..n]$) is added to I if there is ℓ_j, with $j \neq i$, such that $\ell_j \in I$. In this case, $\mathit{reason}(\overline{\ell_i})$ is $\{\overline{\ell_i}, \overline{\ell_j}\}$.

Example 4 (Continuing Example **??**)**.** If I is empty, no literal can be inferred from σ_1 and σ_2. If I is $[\overline{z}]$, then the application of (**??**) to the literals of σ_2 gives

$$2 \cdot [\overline{z}]^{\uparrow}(y) + 2 \cdot [\overline{z}]^{\uparrow}(z) = 2 \cdot 1 + 2 \cdot 0 = 2 < 3$$
$$1 \cdot [\overline{z}]^{\uparrow}(x) + 2 \cdot [\overline{z}]^{\uparrow}(z) = 1 \cdot 1 + 2 \cdot 0 = 1 < 3$$
$$1 \cdot [\overline{z}]^{\uparrow}(x) + 2 \cdot [\overline{z}]^{\uparrow}(y) = 1 \cdot 1 + 2 \cdot 1 = 3 \not< 3$$

Hence, x and y are inferred with $\mathit{reason}(x) = \{x, z\}$ and $\mathit{reason}(y) = \{y, z\}$. Note that, once $I = [\overline{z}, x, y]$, the application of (**??**) to σ_1 gives

$$1 \cdot [\overline{z}, x, y]^{\uparrow}(\overline{y}) = 1 \cdot 0 = 0 < 1$$
$$1 \cdot [\overline{z}, x, y]^{\uparrow}(\overline{x}) = 1 \cdot 0 = 0 < 1$$

Therefore, a conflict k is raised, say because $\overline{y}$ (or similarly $\overline{x}$) is added to I with $\mathit{reason}(k) = \{\overline{x}, \overline{y}\}$.

Chapter 2

AMOSUM constraint

Bibliography

[1] V. Marek and M. Truszczyński, "Stable models and an alternative logic programming paradigm," in *The Logic Programming Paradigm: a 25-year Perspective*, 1999, pp. 375–398. DOI: 10.1007/978-3-642-60085-2_17.

[2] I. Niemelä, "Logic programming with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3,4, pp. 241–273, 1999. DOI: 10.1023/A:1018930122475.

[3] M. Bartholomew, J. Lee, and Y. Meng, "First-order semantics of aggregates in answer set programming via modified circumscription," in *Logical Formalizations of Commonsense Reasoning, AAAI Spring Symposium*, AAAI, 2011. [Online]. Available: http://www.aaai.org/ocs/index.php/SSS/SSS11/paper/view/2472.

[4] W. Faber, G. Pfeifer, and N. Leone, "Semantics and complexity of recursive aggregates in answer set programming," *Artif. Intell.*, vol. 175, no. 1, pp. 278–298, 2011. DOI: 10.1016/j.artint.2010.04.002. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2010.04.002.

[5] P. Ferraris, "Logic programs with propositional connectives and aggregates," *ACM Trans. Comput. Log.*, vol. 12, no. 4, p. 25, 2011. DOI: 10.1145/1970398.1970401. [Online]. Available: http://doi.acm.org/10.1145/1970398.1970401.

[6] M. Gelfond and Y. Zhang, "Vicious circle principle and logic programs with aggregates," *Theory and Practice of Logic Programming*, vol. 14,

no. 4-5, pp. 587–601, 2014. DOI: 10.1017/S1471068414000222. [Online]. Available: http://dx.doi.org/10.1017/S1471068414000222.

[7] L. Liu, E. Pontelli, T. C. Son, and M. Truszczynski, "Logic programs with abstract constraint atoms: The role of computations," *Artif. Intell.*, vol. 174, no. 3-4, pp. 295–315, 2010. DOI: 10.1016/j.artint.2009.11.016. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2009.11.016.

[8] M. Gebser, B. Kaufmann, and T. Schaub, "Conflict-driven answer set solving: From theory to practice," *Artif. Intell.*, vol. 187, pp. 52–89, 2012. DOI: 10.1016/j.artint.2012.04.001. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2012.04.001.

[9] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. FAIA, vol. 336, IOS Press, 2021, pp. 133–182.

[10] D. Carmine, "Design and implementation of modern cdcl asp solvers," 2014.

[11] F. Calimeri, W. Faber, M. Gebser, *et al.*, "Asp-core-2 input language format," *Theory Pract. Log. Program.*, vol. 20, no. 2, pp. 294–309, 2020. DOI: 10.1017/S1471068419000450. [Online]. Available: https://doi.org/10.1017/S1471068419000450.

[12] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "On the implementation of weight constraint rules in conflict-driven ASP solvers," in *ICLP*, ser. LNCS, vol. 5649, Springer, 2009, pp. 250–264.

[13] W. Faber, N. Leone, M. Maratea, and F. Ricca, "Look-back techniques for ASP programs with aggregates," *Fundam. Informaticae*, vol. 107, no. 4, pp. 379–413, 2011.