International Journal of Computer and Information Technology (ISSN: 2279 – 0764)
Volume 03 – Issue 06, November 2014

Answer Set Survey

Omar EL-Khatib
Computer Science Department
Taif University
Taif, SA
Email: omer.khatib {at} tu.edu.sa

***Abstract*— Answer set programming (ASP) is a form of declarative programming that is emerged from logic programming with negation and reasoning formalism that is based on the answer set semantics [1]. ASP is suited for solving hard combinatorial search problems, and has many applications, such as: bioinformatics [3, 4], configuration [5], database integration [6], diagnosis [7], hardware design [8], insurance industry [9], phylogenesis [10, 11], security protocols [12] and model checking [13], to name some.**
ASP has a rich yet simple modeling language with high performance solving capabilities, ability to reason with incomplete information, existence of well-developed mathematical theory and programming. A number of solvers have been proposed, such as: smodels [14, 15, 22], dlv [16], cmodels [15], assat [18, 24], and clasp [19]. ASP programs look like Prolog programs; however, they are treated in a different computation mechanism. ASP uses model generation instead of query evaluation as in Prolog. (Abstract)

Keywords-component; answer set programming, answer set semantics,

I. Introduction

Answer set programming (ASP) is a form of declarative programming that is emerged from logic programming with negation and reasoning formalism that is based on the answer set semantics [1]. ASP is considered in the late 1990s as a new programming paradigm [2]. This programming paradigm is suited for solving hard combinatorial search problems, and has many applications, such as: bioinformatics [3, 4], configuration [5], database integration [6], diagnosis [7], hardware design [8], insurance industry [9], phylogenesis [10, 11], security protocols [12] and model checking [13], to name some.

ASP has a rich yet simple modeling language with high performance solving capabilities, ability to reason with incomplete information, existence of well-developed mathematical theory and programming. A number of solvers have been proposed, such as: smodels [14, 15, 22], dlv [16], cmodels [17], assat [18, 24], and clasp [19]. In ASP, a search problem is represented by a logic program of which its answer sets correspond to solutions. An answer set solver is used for finding answer sets of the problem. ASP programs look like Prolog programs; however, they are treated in a different computation mechanism. ASP uses model generation instead of query evaluation as in Prolog.

II. Background

We briefly recall the basics about ASP. Let us consider a language composed of a set of propositional symbols (atoms) Σ_A. An ASP-program is a collection of rules of the form:

$$H \leftarrow L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_{m+n}. \qquad (1)$$

Where H is an atom, $m \geq 0$, $n \geq 0$, and for each $0 \leq i \leq m+n$, L_i is an atom. The symbol "*not*" stands for negation-as-failure. Given a rule *r* as in (1), we let *head(r)* denotes *{H}*, *body(r)* denotes the body of the rule r $\{L_1, \ldots L_m, \text{not } L_{m+1}, \ldots \text{not } L_{m+n}\}$, $body^+(r) = \{L_1, \ldots, L_m\}$, and $body^-(r) = \{L_{m+1}, \ldots, L_{m+n}\}$. If the *head(r)* = ϕ then the rule *r* is called a constraint. If the *body(r)* = ϕ then the rule *r* is called a fact. A program Π consists of rules of the form (1) is called a normal logic program. If all rules in a logic program have $body^-(r) = \phi$ then the logic program is called basic logic program or datalog program. All atoms in the logic program ∏ is called Atms(∏).

For a basic logic program ∏, a set of atoms X is closed under ∏ if for any rule $r \in \prod$, $head(r) \in X$ whenever $body^+(r) \subseteq X$. The answer set of ∏ is defined as the smallest set of atoms that is closed under ∏ this is denoted as Cn(∏). Answer set can be extended to normal logic programs as in [1]. First, let us introduce the reduct of a normal logic program ∏ as follows:

$$\prod^X = \{head(r) \leftarrow body^+(r) \mid r \in \prod, body^-(r) \cap X = \phi\}$$

$\prod^X$ is calculated by deleting:

- All rules having a *not* A in its body with $A \in X$, then
- All negative atoms of the form *not* A from the bodies of the remaining rules.

Clearly, $\prod^X$ is a basic logic program, and its answer set is defined as $Cn(\prod^X)$.

Give a logic program ∏ with answer set X. If $p \in X$, then we say that p is true under answer set X, otherwise p is false under answer set X. If $p \in X$ then there is at least one rule $r \in \prod$, such that: $p \in head(r)$ and $body^+(r) \in X$ and $body^-(r) \cap X = \phi$. We say that rule *r* supports atom p or p is provable by rule *r* in $\prod$ under answer set X. If $\{p\} \cap X=\phi$ then there is no rule supports p or p is not probable in $\prod$ under answer set X.

For illustration, consider the following program:

$$\prod_1 = \{p \leftarrow \text{not } q.\\ q \leftarrow \text{not } p.\\ \}$$

There are four answer set candidates for $\prod_1$: $\{\{p\}, \{q\}, \{p, q\}, \phi\}$. To verify which of the four candidates are answer sets, we use the following table.

X	$\prod_1$	$Cn(\prod_1^X)$	$X=Cn(\prod_1^X)$
ϕ	{p. q.}	{p, q}	No
{p}	{p.}	{p}	Yes
{q}	{q.}	{q}	Yes
{p, q}	ϕ	ϕ	No

Therefore, we have two answer sets: {p} and {q}.
Another example is the following program:

$$\prod_2 = \{p \leftarrow \text{not } p.\}$$

We have two answer set candidates: $\{ \{p\}, \phi\}$. Verifying which of the two candidates is an answer set:

X	Π_1	$Cn(\Pi_1^X)$	$X=Cn(\Pi_1^X)$
ϕ	{p.}	{p}	No
{p}	ϕ	ϕ	No

Therefore, we have no answer set for the $\prod_2$. The rule in $\prod_2$ is called integrity constraint and can be written as $\leftarrow p$, which means eliminate any candidate answer set that contains the atom p.

Consider the following program:

$$\prod_3 = \{p \leftarrow \text{not } q.\\ q \leftarrow \text{not } p.\\ \leftarrow p. \}$$

There are four answer set candidates $\{\phi, \{p\}, \{q\}, \{p, q\}\}$. The rule $\leftarrow p$ eliminates all answer set candidates that contain the atom p. So, we are left with two answer set candidates $\{\phi, \{q\}\}$. Clearly, program $\prod_3$ has one answer set {q}.

Let $\prod$ be a normal logic program and let $G(\prod) = (Atms(\prod), E)$ be the positive atom dependency graph of $\prod$. If $(a, b) \in E$, then there is a rule $r \in \prod$, such that $a \in head(r)$ and $b \in body^+(r)$. A loop in $\prod$, is the set of atoms $L \subseteq Atms(\prod)$ such that it induces a strongly connected subgraph of $G(\prod)$. Recall that a strongly connected graph is a graph where there is a path of non-zero length between every two atoms in the graph. We denote the set of all loops in $\prod$ by $Loop(\prod)$.
The process of eliminating all variables from a logic program and converting it into propositional logic program is called grounding. For example, Let $\prod_4 = \{d(1).\ d(2).\ \ p(X) \leftarrow d(X)\}$, then grounding of $\prod_4$ is: $\{d(1).\ d(2).\ \ p(1) \leftarrow d(1).\ \ p(2) \leftarrow d(2).\ \}$. The predicate d(1) and d(2) are called domain predicate, which are predicates used to find all variable binding. While Atom p(X) is not a domain predicate.

III. Modeling

To write a program Π in ASP, we divide Π into two sets of rules: (1) generate rules which generate all possible answer sets, and (2) integrity constraint rules which eliminate unwanted answer sets. As an example, consider the n-queen search problem. The goal is to place an n-queen on an nxn chessboard, so that no two queens appear on the same row, column or diagonal.
To define the board length and width, the following fact is defined:

board(1..N, 1..N).

Here, NxN facts were defined. Alternatively, to reduce the number of facts, we can define the board as:

d(1..N).

Which define both the length of the board and the width of the board.
The generate set of rules is as follows:

$$queenOn(X, Y) \leftarrow d(X), d(Y), \text{not } empty(X, Y).$$
$$empty(X,Y) \leftarrow d(X), d(Y), \text{not } queenOn(X,T).$$

The two rules above select either to position the queen on location (X, Y) on the chessboard (represented by the predicate queenOn(X,Y)) or leave the location (X,Y) empty (i.e. with no queen on the location (X,Y), represented by the predicate empty(X,Y)).
The two rules are similar to rules in Π_1 where atom p is replaced by atom queenOn(X,Y) and atom q is replaced by empty(X,Y).

For a 2x2 chessboard, there will be 16 answer set candidates, among them are the following:

empty(1,1), empty(1,2), empty(2,1), empty(2,2)

empty(1,1), queenOn(1,2), empty(2,1), empty(2,2)

	Q

empty(1,1), queenOn(1,2), queenOn(2,1), empty(2,2)

	Q
Q	

queenOn(1,1), queenOn(1,2), queenOn(2,1), queenOn(2,2)

Q	Q
Q	Q

The following integrity constraint eliminates all answer sets that contain two queens positioned in the same row with different column locations.

$$\leftarrow \text{queenOn}(X, Y), d(X), d(Y), \text{queenOn}(X, Y1), d(Y1), Y \neq Y1.$$

The following integrity constraint eliminates all answer sets that contain two queens positioned in the same column with different row locations.

$$\leftarrow \text{queenOn}(X,Y), d(X), d(Y), \text{queenOn}(X1, Y), d(X1), X \neq X1.$$

The last integrity constraint eliminates all answer sets that two queens positioned on the diagonal.

$$\leftarrow \text{queenOn}(X,Y), d(X), d(Y), \text{queenOn}(X1, Y1), d(X1), d(Y1), |X\text{-}X1| \neq |Y\text{-}Y1|.$$

The following integrity constraint is needed to make sure that every row has a queen placed in some column number.

$$\text{rowHasQueen}(X) \leftarrow \text{queenOn}(X, Y), d(X), d(Y). \quad (2)$$
$$\leftarrow \text{rowHasQueen}(X), d(X). \quad (3)$$

Rule (2) finds all row numbers that have queen in some column for that row. Integrity constraint (3) eliminates all answer sets that contains rows that do not have queen in that row of the chessboard.

IV. ANSWER SET EXTENSIONS

Various extensions to the basic paradigm exists, in the following subsections many answer set extensions is presented.

Classical Negation:

In reference [20], another kind of negation is defined which is classical negation. It is represented as $\neg$ or $-$. A literal is either an atom *p* or a classical negated atom $\neg p$. A logic program with literals is called an extended logic program. An extended logic program is contradictory if both *p* and $\neg p$ are derivable. Classical negation can be eliminated by replacing $\neg$p with a new atom p' and adding the integrity constraint $\leftarrow$ p, p'. Then we have a normal logic program and the answer set is defined in the same way as defined in normal logic programs. The difference between negation as failure *not* p and classical $\neg$p is that, in case of negation as failure *not* p in a logic program $\prod$, if p is not supported by any rule in $\prod$ under answer set X, then by closed world assumption *not* p is derivable from $\prod$ under answer set X. For example, consider the logic program $\prod_5$ = {cross $\leftarrow$ not car}, the answer set X of $\prod_4$ is {cross}. This is because car is not supported in $\prod_5$. So, cross is true under answer set X of $\prod_5$. However, in case of classical negation $\neg$p, then $\neg$p belongs to answer set X, if $\neg$p is supported in $\prod$ under answer set X. For example, consider the logic program $\prod_6$ = {cross $\leftarrow$ $\neg$car}, then the answer set X is ϕ since $\neg$car is not supported in $\prod_6$ under answer set X. therefore, atom cross is not supported in $\prod_6$ under answer set X. Using classical negation, an atom fails if its negation succeeds. Using negation as failure, an atom fails if it does not succeed.

Disjunctive Logic Programs

Disjunctive logic program is a normal logic program where the head of the rule is a disjunctive of atoms [20]. The rules are of the form:

$$h_1 \mid h_2 \mid \ldots \mid h_k \leftarrow L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, L_{m+n}.$$

To define answer set models, first we consider a disjunctive program $\prod$ without negation as failure "*not*", rules are of the form:

$$h_1 \mid h_2 \mid \ldots \mid h_k \leftarrow L_1, \ldots, L_m.$$

The minimum set of atoms X that is closed under $\prod$ is an answer set. Therefore, if for every rule $r \in \prod$, if body(r) $\subseteq$ X, then for some $0 \leq i \leq k$, $L_i \subseteq X$. Now, consider a program with disjunctive rules and contains negation as failure "*not*" in the body. Let X be a set of atoms, define $\prod^X$ to be disjunctive program by:

- deleting all rules r $\in \prod$ where *not* L $\in$ body(r) and L $\in$ X, then
- Delete remaining *not* L from the rest of the rules' bodies.

Clearly, $\prod^X$ is disjunctive logic program with no negation as failure "*not*". X is an answer set if X is a minimal set that is closed under $\prod^X$. for example, let $\prod_7$ = {p | q $\leftarrow$ }, then $\prod_6$ has two answer sets {p} and {q}. The set {p, q} is not an answer set of $\prod_7$ since it is not a minimal set closed under $\prod_7$.

Cardinality Atoms:

Another extension is a cardinality atom [34, 35] which is as follows:

$$L \ \{L_1, \ldots, L_m\} \ U$$

Where L is a lower bound integer number and U is an upper bound integer. The semantics of a cardinality atom is that, if the number of atoms that belong to an answer set X of a logic program $\prod$ are between the lower bound L and upper bound U (inclusive), then the cardinality atom is considered true under the answer set X. Otherwise, the cardinality atom is considered false under the answer set X. Rules that have cardinality atom in the body of a rule are called cardinality rule. Rules that have cardinality atom in the head of the rule is called a choice rule (see below).

Weighted Atoms:

Weighted atoms [34, 35] are of the form:

$$L \ [\ L_1{=}a_1, \ldots, L_m{=}a_m \ ]U$$

Where L is a lower weight bound integer number, and U is an upper weight bound integer. Each literal L_i where $1 \leq i \leq m$, in the weighted atom, have integer weights a_i. The semantics of a weighted atom is that, if the sum of weights of the literals that are in answer set X is between the lower bound L and the upper bound U (inclusive), then the weighted atom is considered true under the answer set X. Otherwise, the weighted atom is considered false under the answer set X. Rules that have weighted atom in the body of a rule are called weighted rule.

Choice Rules:

Choice rules [34, 35] have cardinality atom or weighted atom in the head of a rule. Choice rule is of two forms:

$$L \{ h_1, \ldots, h_m \} U \leftarrow body^+, body^-. \quad (4)$$

Or

$$L [h_1=a_1, \ldots, h_m=a_m] U \leftarrow body^+, body^- \quad (5)$$

Where L is a lower bound integer and U is the upper bound integer, and for each $0 \leq i \leq m$, each a_i is an integer represents the weight of the atom h_i. The semantic of choice rules is as follows: for an answer set X, if $body^+ \in X$ and $body^- \cap X = \phi$, then:

- In case of rules of the form (4), an arbitrary number of atoms between lower bound L and upper bound U (inclusive) from the atoms $\{h_1, \ldots, h_m\}$ is in the answer set X.
- In case of rules of the form (5), an arbitrary atoms with sum of their weights is between the lower bound L and the upper bound U (inclusive) from the atoms $\{ h_1=a_1, \ldots, h_m=a_m\}$ is in the answer set.

Conditional Literals:

A conditional literal [34, 35] is of the form:

$$p(X) : q(X)$$

Where p(X) is a literal and q (X) is a domain predicate. The conditional literal is expanded to a conjunction of literals. For example, consider the following program:

$$\prod_8 = \{ d(1). \quad d(2).$$

$$h \leftarrow p(X) : d(X).$$

$$\}$$

Program $\prod_8$ is expanded as follows:

$$d(1). \quad d(2).$$

$$h \leftarrow p(1), p(2).$$

Weak Constraint:

Weak constraints [33] are rules that should be satisfied but their violation in an answer set X does not reject the answer set X. The answer sets of a logic program ∏ with a set W of weak constraints are those answer sets of X which minimize the number of violated weak constraints. Weak constraints can be weighted according to their importance (the higher the weight, the more important the constraint). In the presence of weights, the answer sets minimize the sum of the weights of the violated weak constraints. Weak constraints can also be prioritized. Under prioritization, the semantics minimizes the violation of the constraints of the highest priority level first; then the lower priority levels are considered one after the other in descending order.

Syntactically, weak constraints are specified as follows.

$$:\sim L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_{m+n} \quad [Weight:Level]$$

Where Weight and Level are integers, L_i are literals, for $0 \leq i \leq m+n$.

Ordered Disjunction Rules:

Logic programs with ordered disjunction rules are an extension to normal logic programs. The new connective × that represents ordered disjunction allowed to appear in the head of rules only [31, 32]. The ordered disjunction rule is of the form:

$$C_1 \times C_2 \times \ldots \times C_n \leftarrow body^+, body^-.$$

For an answer set X of a logic program ∏. If the rule is in ∏, $body^+ \in X$ and $body^- \cap X= \phi$, then try to include C_1 in the answer set X if possible, otherwise try to include C_2 in answer set X if possible, ... otherwise try to include C_n in answer set X. For example, Let

$$\prod_9 = \{p \times q \leftarrow \text{not } r,$$
$$s \leftarrow \text{not } s, p \}$$

Then the answer set of $\prod_9$ is ${p}$. On the other hand, the program:

$$\prod_{10} = \{p \times q \leftarrow \text{not } r,$$
$$s \leftarrow \text{not } s, p\}$$

has an answer set of {q}, since we cannot find an answer set {p} because of the second rule.

Aggregate Functions:

Aggregate functions like *sum*, *count*, *max*, and *min*, has been added to answer set [21]. These arithmetic operators increase the expressiveness of answer set. In [21], A symbolic set is a pair *{Vars: Conj}*, where *Vars* is a list of variables and *Conj* is a conjunction of literals (p or ¬p). An aggregate function is of the form *f(S)*, where *S* is a symbolic set and *f* is arithmetic operator that belongs to {sum, count, max, min}. An aggregate atom is of the form:

$$L \ \Delta_1 \ f(S) \ \Delta_2 \ R$$

Where f(S) is an aggregate function, $\Delta_1, \Delta_2 \in \{=, <, >, \leq, \geq\}$, and L and R are terms (either a constant of a variable) called guard of an aggregate function. One of "L Δ_1" or "Δ_2 R" can be omitted.

A (DLP^A) rule is of the form:

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_k, \text{not } b_{k+1}, \ldots, \text{not } b_m$$

Where $a_1, \ldots, a_n$ are atoms and $b_1, \ldots, b_m$ are atoms or aggregate atom. A (DLP^A) program is a set (DLP^A) rules. As an example, consider the following program with some facts about a(X, Y) and b(X).

$\prod_{11} = \{\ q(1) \vee p(2, 2).$

$q(2) \vee p(2, 1).$

$t(X) \leftarrow q(X), \#sum\{Y: p(X, Y)\} > 1.$

$\}$

Ground($\prod_{11}$) is:

$\{\ q(1) \vee p(2, 2).$

$q(2) \vee p(2, 1).$

$t(1) \leftarrow q(1), \#sum\{Y: p(1, Y)\} > 1.$

$t(2) \leftarrow q(2), \#sum\{Y: p(2, Y)\} > 1.$

$\}$

The first two rules of program $\prod_{11}$ will generate 4 answer sets candidates: {{q(1), q(2)}, {q(1), p(2, 1)}, (p(2, 2), q(2)}, {p(2, 2), p(2, 1)}}. From the third rule atom t(1) will not be in any answer set candidates since sum{Y: p(1, Y)} is always zero not greater than 1. From the last rule t(2) will be in an answer set when q(2) is an answer set and sum{Y: p(2, Y)} is greater than one. Therefore, program $\prod_{11}$ has four answer set {{q(1), q(2)}, {q(1), p(2, 1)}, {t(2), p(2, 2), q(2)}, {p(2, 2), p(2, 1)}}.

A rule r is safe if the following conditions hold: (i) each variable appears in the head of the rule, must also appear in a positive atom in the body of rule r. (ii) each variable appears in the symbolic set {*Vars: Conjs*} must appear in a positive literal *Conjs*. (iii) each guard of an aggregate function must be either a constant or a variable that appear in the head of the rule r.

V. ANSWER SET SYSTEMS

Several answer set computation systems have been developed, such as: smodels, dlv, assat, cmodels, and clasp. The computation of answer sets is done in two phases: (i) grounding of the logic program ($\prod$): which is eliminating variables to obtain a propositional program ground($\prod$). (ii) Computation of answer sets on the propositional program ground($\prod$).

$$\begin{tikzcd}
\text{Logic program} \arrow[r, "\text{Grounder}"] & \text{Propositional logic program} \arrow[r, "\text{Answer set solver}"] & \text{Answer sets(s)}
\end{tikzcd}$$

Figure 1, computation of answer set of a logic programs.

The first phase (grounding) is done using lparse program or gringo or the grounder in dlv. The program lparse and gringo both can handle classical negation, disjunction logic programs, cardinality atoms, weight atoms, choice rules, conditional literals, and aggregates. The grounder of dlv handles classical negation, disjunction logic programs, weak constraint, and aggregates. We will consider the second phase which is the generation of answer set of a propositional logic program. The normal logic programs are considered only for simplicity. Therefore, all atoms in the answer set must appear in the head of some rule in the logic program. For a normal logic program $\prod$, a partial model $\wp$ is a 3-valued model with true atoms, false atoms and undefined atoms. So,

$$\wp = \langle T, F, U \rangle$$

Where T is the true atoms, F is the false atoms, and U is the undefined atoms. A total model is a partial model where all atoms Atms($\prod$) is either true or false and the undefined atoms is empty, i.e. $\wp = \langle T, F, \phi \rangle$ where $T \cup F$ = Atms($\prod$). The algorithm is as follows: we start with all atoms appears in the logic program as undefined atoms U. Therefore, $\wp = \langle \phi, \phi, Atms \rangle$. Then we use propagation techniques to extend the partial order to increase the number of true and false atoms. If the extended partial model is a total model (i.e. $U=\phi$) then it is an answer set. If the extended partial model contains an atom that is true and false at the same time (i.e. $T \cap F = \phi$), then a contradictory partial model occur and no answer set situation occurred. Otherwise, an atom is selected non-deterministically from the undefined atoms U and branch on its true or false truth value. This is done recursively by adding the selected atom to the partial model as true atom once and again as false atom. There are many ASP heuristics to select the atom from the set U to branch on. The ASP heuristics tries to minimize the depth of the search tree in finding the answer sets. Please, see the [23] for more information about ASP heuristics.

The smodels systems: The smodels [14] algorithms is as follows:

```
Smodels(T, F, U)
    1. Expand(T, F, U)
    2. If $T \cap F \neq \phi$ then return fail.
    3. If $U=\phi$ then T is an answer set, print T
    4. A = select(U)
    5. Smodels($T \cup \{A\}, F, U \setminus \{A\}$);
    6. Smodels($T, F \cup \{A\}, U \setminus \{A\}$);
```

The smodels function takes the partial model which is the three sets: the true atoms T, false atoms F and the undefined atoms U. Then it calls the Expand function in step (1), which uses the propagation techniques to increase the number of true atoms and false atoms. In step 2, the function checks that if the partial model is contradictory, it return fail. In step 3, the function checks if the partial model is a total model, then it prints that model as answer set. In step 4, the function selects an atom A from the undefined atom. In step 5, it calls itself recursively to try to find answer set $T \cup \{A\}$. In step 6, the function smodels calls itself recursively to try to find answer set with $F \cup \{A\}$. Initially, smodels is called with $T=\phi$, $F=\phi$ and U=Atms.

The dlv system: The dlv system is a deductive database system, based on disjunctive logic programming, which offers front-ends to several advanced KR formalisms. The system can handle weak constraints, aggregates functions, functions, lists and sets. You can call dlv from java programs and you can call c++ functions from dlv. An outline of the general architecture of the dlv system is shown in Figure 2. The input logic program ($\prod$) is fed into the instantiator, which eliminates the variables and generates the propositional logic program ground($\prod$). This process is called instantiator or grounding. Then the model generator generates an answer set candidate which is verified by the model checker whether it is an answer set. The algorithm for Model generator is as follows:

```
ModelGenerator (T, F, U):
    1. <T, F, U>= Propagate(T, F, U);
    2. If $(T \cap F = \phi)$ then fail.
    3. If $(U=\phi)$ then print answer set T.
    4. L = select(U)
    5. ModelGenerator($T \cup \{L\}$, F, $U \setminus \{L\}$);
    6. ModelGenerator(T, $F \cup \{L\}$, $U \setminus \{L\}$);
```

The algorithm of Model generator is similar to the smodels function.

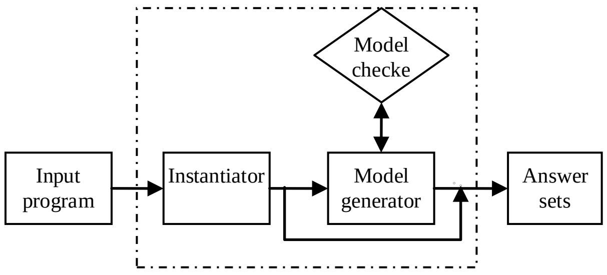

Figure 2, dlv system general architecture

The assat and cmodels systems: Both answer set solver assat and cmodels converting the answer set logic program into propositional satisfiability (SAT) program. This reduces the problem of computing answer sets to SAT problem (using Clark's completion [25]), then invoke any of the SAT solver to generate the answer sets. To handle the normal logic programs, a loop formula is added [18, 24]. However, whenever a logic program is mapped to equivalent propositional formulas, the size might grow exponentially [26]. The formal definition of loop formula is as follows. Let $\prod$ be a normal logic program, assume a set of atoms $L \subseteq Atms(\prod)$, we define an external support of L for $\prod$ as: $ES_{\prod}(L) = \{r \in \prod \mid head(r) \in L, body(r) \cap L=\phi\}$. Then the loop formula of a loop L for $\prod$ is:

$$LF_{\prod}(L)=\{\bigwedge_{A\in L}\neg A \leftarrow \bigwedge_{r\in ES(L)}\neg(\bigwedge_{A\in body+(r)} A \wedge \bigwedge_{A\in body-(r)} \neg A)$$

For example: Let

$$\begin{aligned}\prod = \{&a \leftarrow b, \text{not } c.\\ &b \leftarrow a, \text{not } d.\\ &a \leftarrow e.\\ \}&\end{aligned}$$

Then the loop L of $\prod$ is {a, b}, then the external support for L is: $ES_{\prod}(L) = \{a \leftarrow e\}$. The loop formula $LF_{\prod}(L)$ is:

$$\{\neg a, \neg b \leftarrow \neg e\}.$$

The Postdam system: The postdam answer set solving collection is a combination of grounder called gringo and an answer set solver called clasp [27, 28]. The primary clasp algorithm has been developed for answer set solving based on conflict-driven nogood learning [30]. Assignments and nogoods are set of assigned atoms, i.e. entities of the form T_p or F_p denoting that atom p has been assigned true or false, respectively. Given an assignment A, we denote $A^T = \{p \mid T_p \in A\}$ and $A^F = \{p \mid F_p \in A\}$. The assignments A is a total assignment if it assign a truth value to all atoms in Atms($\prod$), otherwise it is a partial assignment. Given an assignment A and a nogood δ, we say that δ is violated if $\delta \subseteq A$. In turn, a partial assignment A is a solution for a set of nogoods Δ if no $\delta \in \Delta$ is violated by A.

The concept of nogood can be also used during deterministic propagation phases (a.k.a. unit propagation) to determine additional assignments. Given a nogood δ and a partial assignment A such that $\delta\backslash A = \{F_p\}$ ($\delta\backslash A = \{T_p\}$), then we can infer the need to add T_p (F_p) to A in order to avoid violation of δ. In the context of ASP computation, we distinguish two types of nogoods: Clark's completion nogoods [25, 30], which are derived from Clark's completion of a logic program (denoted with Δ_{cc} the set of Clark's completion nogoods for the program $\prod$), and loop formula nogoods [25], which are derived from the loop formula of $\prod$ (denoted by $\Lambda_{\prod}$). The two fundamental results associated Clark's completion and loop formula is as follows: (see [28]).

Clark's completion of $\prod$:

$$\begin{aligned}\prod\nolimits_{cc} = \{\ &\beta_r \leftrightarrow \wedge_{a\in body+(r)}\, a \wedge \wedge_{b\in body-(r)} \neg b \mid r \in \prod\} \cup\\ &\{ p \leftrightarrow \vee_{r\in body\prod(p)}\beta_r \mid p\in atoms(\prod)\}\end{aligned}$$

Where β_r is a new variable introduced for each rule $r \in \prod$.

Example:

Let $\prod = \{ a \leftarrow b, \text{not } c.\ \ a \leftarrow d.\ \ b \leftarrow \text{not } e.\ \ e \leftarrow \text{not } b. \}$, then

$$\begin{aligned}\prod\nolimits_{cc} = \{\ &\beta_1 \leftrightarrow b, \neg c.\ \ \beta_2 \leftrightarrow d.\ \ \beta_3 \leftrightarrow \neg e.\ \beta_4 \leftrightarrow \neg b.\} \cup\\ &\{ a \leftrightarrow \beta_1 \vee \beta_2.\ \ b \leftrightarrow \beta_3.\ \ e \leftrightarrow \beta_4. \}\end{aligned}$$

The Clark's completion nogood reflect the structure of the implications present in the definition of $\prod_{cc}$. In particular:

- For the original rule $p \leftarrow body(r)$, the set of nogood is $\{F_{\beta r}\} \cup \{T_a \mid a \in body+(r)\} \cup \{F_b \mid b \in body^-(r)\}$.
- In addition, for each rule, the body should be false if any of its element is falsified, leading to the set of nogoods of the form: $\{T_{\beta r}, F_a\}$ for each $a \in body^+(r)$ and $\{T_{\beta r}, T_b\}$ for each $b \in body^-(r)$.
- The closure of an atom definition leads to a nogood expressing that the atom is true if any of its rule is true: $\{F_p, T_{\beta r}\}$ for each $r \in body_{\prod}(p)$.
- Similarly, the atom cannot be true if all its rules have a false body. This yields the nogood $\{T_p\} \cup \{F_{\beta r} \mid r \in body_{\prod}(p)\}$.

Δ_{cc} is the set of all the nogoods defined as above.

The loop formula nogoods derive instead from the need to capture loop formulae, thus avoiding positive cycles. Let a loop L be a set of atoms and $EB_{\prod}(L)$ be the external support of L for $\prod$. Then, for each atom $p \in L$, the loop nogoods are: $\{T_p\} \cup \{F_{\beta r} \mid \beta r \in EB_{\prod}(L)\}$ We demote with $\Lambda_{\prod}$ the set of all loop formula nogoods and $\Delta_{\prod}$ the whole set of nogoods: $\Delta_{\prod} = \Delta_{cc} \cup \Lambda_{\prod}$.

References

[1] M. Gelfond and V. Lifschitz, "the Stable Model Semantics for Logic Programming," ICLP/SLP, pp. 1070-1080, 1988.

[2] V. Marek and M. Truszczyński, "Stable models and an alternative logic programming paradigm," In Apt, Krzysztof R. The Logic programming paradigm: a 25-year perspective, pp. 169-181, Springer. 1991.

[3] N. Tran and C. Baral, C, "Reasoning about triggered actions in AnsProlog and its application to molecular interactions in cells," In Dubois, D., Welty, C., Williams, M., eds.: Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR'04), pp. 554–564, AAAI Press 2004

[4] S. Dworschak, S. Grell, V. Nikiforova, T. Schaub and J. Selbig, "Modeling biological networks by action languages via answer set programming," Constraints 13 (1-2), pp. 21–65, 2008

[5] T. Soininen and I. Niemela, "Developing a declarative rule language for applications in product configuration, " In Gupta, G., ed.: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99), pp. 305–319, Springer 1999.

[6] N. Leone, G. Greco, G. Ianni, V. Lio, G. Terracina, T. Eiter, W. Faber, M. Fink, G. Gottlob, R. Rosati, D. Lembo and M. Lenzerini, M. Ruzzi, E. Kalka, B. Nowicki. and W. Staniszkis, "The INFOMIX system for advanced integration of incomplete and inconsistent data," In Ozcan, F., ed.: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'05), ACM Press (2005), pp. 915–917.

[7] T. Eiter, W. Faber, N. Leone, and G. Pfeifer, "The diagnosis frontend of the dlv system," AI Communications 12 (1-2), 99-111, 1999.

[8] E. Erdem, E. and M. Wong, "Rectilinear Steiner tree construction using answer set programming," In: Proc. of ICLP, 386–399, 2004.

[9] C. Beierle, O. Dusso and G. Kern-Isberner, "Using answer set programming for a decision support system," In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05), Springer (2005), pp.374–378.

[10] J. Kavanagh, d. Mitchell, E. Ternovska, J. Manuch, X. Zhao and A. Gupta, "Constructing Camin-Sokal phylogenies via answer set programming," In Hermann, M., Voronkov, A., eds.: Proceedings of the Thirteenth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'06), Springer (2006), pp. 452–466.

[11] D. Brooks, E. Erdem, and S. Erdogan, J. Minett and D. Ringe, "Constructing Camin-Sokal phylogenies via answer set programming,"
Inferring phylogenetic trees using answer set programming. Journal of Automated Reasoning 39 (4), pp. 471–511, 2007.

[12] L. Aiello and F. Massacci, "Verifying security protocols as planning in logic programming," ACM Transactions on Computational Logic 2 (4), pp. 542–580, 2001.

[13] K. Heljanko and I. Niemela, "Bounded LTL model checking with stable models,“ Theory and Practice of Logic Programming 3 (4-5), pp. 519–550, 2003.

[14] P. Simons. "Efficient implementation of the stable model semantics for normal logic programs," Research Report 35, Helsinki University of Technology, September 1995.

[15] I. Niemelä and P. Simons. "Efficient implementation of the well-founded and stable model semantics," *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 289-303, Bonn, Germany, September 1996.

[16] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. "The DLV system for knowledge representation and reasoning," ACM Transactions on Computational Logic, 7(3):499–562, July 2006.

[17] Yu. Lierler and M. Maratea, "Cmodels-2: SAT-based answer set solver enhanced to non-tight programs," In Proc. of LPNMR-7, 2004.

[18] F. Lin and Yu. Zhao, ASSAT: "Computing answer sets of a logic program by SAT solvers," In Proc. of AAAI-02 .

[19] M. Gebser, B. Kaufmann, A. Neumann and T. Schaub, "clasp: A Conflict-Driven Answer Set Solver," LPNMR'07, 2007

[20] M. Gelfond and V. Lifschitz. "Classical negation in logic programs and disjunctive databases," New Generation Computing , 9(3/4):365–386, 1991.

[21] T. Dell Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. "Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV," In Proceedings of the 18th International Joint Conference on Arti- ficial Intelligence (IJCAI) 2003, pp. 847 852, Acapulco, Mexico, August 2003. Morgan Kaufmann Publishers.

[22] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 420-429, Dagstuhl, Germany, July 1997.

[23] W. Faber, N. Leone, and G. Pfeifer. "Experimenting with heuristics for answer set programming," In B. Nebel, editor, Proceedings of the International Joint Conference on Artificial Intelligence, pp. 635–640. Morgan Kaufmann, 2001.

[24] F. Lin and Y. Zhao. "Assat: Computing answer sets of a logic program by sat solvers," Artificial Intelligence, 157:115–137, 2004.

[25] K. Clark. "Negation as failure," In H. Gallaire and J. Minker, editors, Logic and Data Bases, pp. 293–322. Plenum Press, 1978.

[26] V. Lifschitz and A. Razborov. Why are there so many loop formulas? ACM Transactions on Computational Logic, pp 261-268, 2006.

[27] M. Gebser, B. Kaufmann, and T. Schaub. "Conflict-driven Answer Set Solving: From Theory to Practice," Artificial Intelligence 187: 52–89, 2012.

[28] M. Gebser et al. "Answer Set Solving in Practice," Morgan & Claypool, 2012.

[29] F. Fages. "Consistency of Clark's Completion and Existence of Stable Models," Journal of Methods of Logic in Computer Science, 1(1):51–60, 1994.

[30] R. Dechter. Constraint Processing. Morgan Kaufmann, 2003.

[31] G. Brewka , I. Niemelä , T.Syrjänen. "Logic programs with ordered disjunction," Computational Intelligence, 20(2): 335-357, May 2004.

[32] G. Brewka. "Logic Programming with Ordered Disjunction," In Proceedings of AAAI, pp. 100—105, 2002.

[33] DLV user manual: http://www.dlvsystem.com/html/DLV_User_Manual.html

[34] I. Niemela and P. Simons. "Extending the Smodels system with cardinality and weight constraints," In Minker, J., ed.,Logic-Based Artificial Intelligence. Kluwer Academic Publishers. 2000

[35] C. Baral. "Knowledge Representation, Reasoning and Declarative Problem Solving," Cambridge University Press, 2003.

[36] F. Vella, A. Dal Palu, A. Dovier, A. Formisano, E. Pontelli. "CUD@ASP: Experimenting with GPUs in ASP solving," CILC, volume 1068 of CEUR Workshop Proceedings, page 163-177. CEUR-WS.org, (2013).