Integrating ASP and CLP solvers:
Computing Answer Sets from Partially Ground Programs

Proposal Defense

Veena S. Mellarkod

Planning Example

Ram is at his office and has a dentist appointment in one hour. For the appointment, he needs his insurance card which is at home and cash to pay the doctor. He can get cash from the nearby atm. The table shows the time needed to travel between locations: Dentist, Home, Office and Atm.

Minutes	Doctor	Home	Office	Atm
Doctor	0	20	30	40
Home	20	0	15	15
Office	30	15	0	20
Atm	40	15	20	0

(a). Find a plan which takes Ram to the doctor on time.

(b). Find a plan which takes Ram to the doctor at least 15 minutes early.

If the time variables ranged from [0..1440], then classical ASP solvers could not solve the problem.

Problem Definition

Classical ASP and CR-Prolog solvers compute answer sets of programs from their ground instantiations.

Such solvers are inapplicable to applications which contain variables that range over large domains.

We propose to solve this problem in two steps:

- designing a new language $\mathcal{AC}(\mathcal{C})$, which differs from A-Prolog by classifying predicates into different types.
- designing an algorithm for computing answer sets of $\mathcal{AC}(\mathcal{C})$ programs from their partial ground instantiations.

Planning Example - Representation

```
person(ram).
item(icard).      item(cash).
loc(dentist).     loc(office).
loc(home).        loc(atm).
step(1..4).

% Actions
action(go_to(P,L)) :- person(P), loc(L).

% Fluents
fluent(at_loc(P,L)) :- person(P), loc(L).
fluent(at_loc(I,L)) :- item(I), loc(L).
fluent(has(P,I)) :- person(P), item(I).
```

```
% If a person goes to loc L then he is at L in next step.
h(at_loc(P,L),S1) :-    person(P), loc(L),
                        next(S0,S1),
                        o(go_to(P,L),S0).

% If Ram is at loc L and item I is at loc L then he picks item I
h(has(P,I),S) :-    step(S),
                    h(at_loc(P,L),S),
                    h(at_loc(I,L),S).

% If a person has an item then it is at same loc as the person
h(at_loc(I,L),S) :- step(S),
                    h(has(P,I),S),
                    h(at_loc(P,L),S).
```

```
% Timing constraints

#csort time(1..1440).
#mixed at(step,time).

% assign times increasingly for steps.
:- next(S1, S2), at(S1,T1),  at(S2,T2), T1 - T2 > 0.

% minimum time for travel between dentist and home is 20 mins
:- h(at_loc(P, home), S1), o(go_to(P, dentist),S1),
    next(S1,S2), at(S1,T1), at(S2,T2), T1 - T2 > -20.

% minimum time for travel between home and atm is 15 minutes
:- h(at_loc(P, home), S1), o(go_to(P, atm),S1),
   next(S1,S2), at(S1,T1), at(S2,T2), T1 - T2 > -15.
```

```
%% Planning Module
1 { o(A,S) : action(A) } 1 :- step(S), not goal(S).

goal(S) :- h(at_loc(ram,dentist),S), h(has(ram,icard),S),
           h(has(ram,cash),S).

plan :- goal(S).
:- not plan.

% (a). He should be at the dentist in 60 minutes
:- goal(S), at(0,T1), at(S,T2), T2 - T1 > 60.

% Initial Situation
h(at_loc(ram,office),0).    h(at_loc(icard,home),0).
h(at_loc(cash,atm),0).
```

A-Prolog Equivalent

To get a program equivalent in A-Prolog, we can delete the rules:

```
#csort time(1..1440).
#mixed at(step,time).
```

and add the following rules:

```
time(1..1440).

1 { at(S,T) : time(T) } 1 :- step(S).
```

lparse was not able to ground the A-Prolog program.

Language $\mathcal{AC}(\mathcal{C})$

First goal is to design a language which classifies predicates into different types.

- syntax
- semantics

Requirements:

- syntax of $\mathcal{AC}(\mathcal{C})$ should be close to syntax of $A\text{-}Prolog$.
- semantics of $\mathcal{AC}(\mathcal{C})$ should be a natural extension of semantics of $A\text{-}Prolog$.

Language $\mathcal{AC}(\mathcal{C})$

Language $\mathcal{AC}(\mathcal{C})$ divides predicates into four types: $regular$, $constraint$, $defined$ and $mixed$.

Signature of a $\mathcal{AC}(\mathcal{C})$ program divides constants into $regular$ or $constraint$ sorts.

$regular$ predicates define relations with variables ranging over small domains.

$constraint$ predicates define primitive numerical relations with variables ranging over large domains.

$defined$ predicates define relations with variables ranging over large domains.

$mixed$ predicates define relationships between $regular$ and $defined$; and $regular$ and $constraint$ predicates.

$\mathcal{AC}solver$

Second goal is to design an algorithm $\mathcal{AC}solver$ for computing answer sets of $\mathcal{AC}(\mathcal{C})$ programs and prove its correctness.

Requirements: $\mathcal{AC}solver$ should compute answer sets from partially ground programs.

- ground $regular$ terms
- do not ground $constraint$ terms

Motivation

How huge a ground instantiation can get?

Let Π be a program as given below:

$time(0..1440).$
#domain $time(T_1;T_2).$
$q(1). \quad q(2).$
$p(X,Y) \leftarrow q(X),\ q(Y),\ X!=Y,\ not\ r(X,Y)$
$r(X,Y) \leftarrow q(X),\ q(Y),\ X!=Y,\ not\ p(X,Y)$
$\leftarrow r(X,Y),\ at(X,T_1),\ at(Y,T_2),\ T_1-T_2>3$
$\leftarrow p(X,Y),\ at(X,T_1),\ at(Y,T_2),\ T_1-T_2>10$
$1\ \{\ at(X,T)\ :\ time(T)\ \}\ 1 \leftarrow\ q(X)$

Number of rules in $lparse(\Pi) = 8226921.$

Motivation - 2

partial ground program

Let Π be a program as given below:

#csort $time(0..1440)$.
#mixed $at(q,\ time)$.
$q(1).\quad q(2).$
$p(X,Y)\ \leftarrow\ q(X),\ q(Y),\ X!=Y,\ not\ r(X,Y)$
$r(X,Y)\ \leftarrow\ q(X),\ q(Y),\ X!=Y,\ not\ p(X,Y)$
$\leftarrow\ r(X,Y),\ at(X,T_1),\ at(Y,T_2),\ T_1-T_2>3$
$\leftarrow\ p(X,Y),\ at(X,T_1),\ at(Y,T_2),\ T_1-T_2>10$

Number of rules in $\mathcal{P}(\Pi)=14$.

$\mathcal{P}(\Pi)$ Example

#csort $time(0..1440)$.

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

$p(1,2) \leftarrow not\ r(1,2)$

$p(2,1) \leftarrow not\ r(2,1)$

$r(1,2) \leftarrow not\ p(1,2)$

$r(2,1) \leftarrow not\ p(2,1)$

$\leftarrow\ r(1,2),\ at(1,T_1),\ at(2,T_2),\ T_1 - T_2 > 3$

$\leftarrow\ r(2,1),\ at(2,T_2),\ at(1,T_1),\ T_2 - T_1 > 3$

$\leftarrow\ r(1,1),\ at(1,T_1),\ at(1,T_1),\ T_1 - T_1 > 3$

$\leftarrow\ r(2,2),\ at(2,T_2),\ at(2,T_2),\ T_2 - T_2 > 3$

$\leftarrow\ p(1,2),\ at(1,T_1),\ at(2,T_2),\ T_1 - T_2 > 10$

$\leftarrow\ p(2,1),\ at(2,T_2),\ at(1,T_1),\ T_2 - T_1 > 10$

$\leftarrow\ p(1,1),\ at(1,T_1),\ at(1,T_1),\ T_1 - T_1 > 10$

$\leftarrow\ p(2,2),\ at(2,T_2),\ at(2,T_2),\ T_2 - T_2 > 10$

$\mathcal{AC}solver$

$\mathcal{AC}solver$ mainly consists of two parts.

Partial Grounder $\mathcal{P}ground$

Inference Engine $\mathcal{AC}engine$

$\mathcal{P}ground$

Input: an $\mathcal{AC}(\mathcal{C})$ program Π.

Output: a partially ground (r-ground) program $\mathcal{P}(\Pi)$.

Proposition: Answer sets of Π are same as answer sets of $\mathcal{P}(\Pi)$.

Requirement: use $lparse$ to ground regular terms.

$\mathcal{AC}engine$

Input: an r-ground program Π

Output: returns $true$ and a (simplified) answer set of Π; otherwise returns $false$.

Proposition: If $\mathcal{AC}engine$ returns $true$ and a set M, then M is a simplified answer set of Π.

$\mathcal{AC}engine$

$\mathcal{AC}engine$ tightly couples a ASP solver and a constraint logic programming (CLP) solver.

The inferences of $regular$ part are computed by ASP solver.

The inferences of $defined$ and $constraint$ parts are computed by CLP solver.

The inferences of the mixed part are computed by communications between ASP and CLP solvers.

$\mathcal{AC}(\mathcal{C})$ instance: $\mathcal{AC}_0$

Consider an instance of $\mathcal{AC}(\mathcal{C})$ where $\mathcal{C} = \{X - Y > k\}$, $P_d = \emptyset$ and if $r \in \Pi_M$ then $head(r) = \emptyset$. Let us call this language $\mathcal{AC}_0$.

This language was studied in "S. Baselice, P. A. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. ICLP 2005".

We can expand the language using consistency restoring rules from $CR\text{-}Prolog$.

An algorithm to compute answer sets of programs in $\mathcal{AC}_0$, integrates

- Davis-Putnam type of algorithm for computing answer sets of ASP programs,
- the form of abduction from $CR\text{-}Prolog$, and
- incremental constraint satisfaction algorithm for difference constraints.

Language $\mathcal{AC}_0$

A program of $\mathcal{AC}_0$ consists of two modules

- regular rules of ASP and consistency restoring rules of $CR\text{-}Prolog$;
- denials with constraints of type $X - Y > k$.

The variables of a program Π that occur in constraints are called constraint variables and the rest are regular variables

The semantics of $\mathcal{AC}_0$ extends ASP, CR-Prolog and CASP semantics in a natural way.

Expressive Power of $\mathcal{AC}_0$

In addition to standard power of ASP, $\mathcal{AC}_0$ allows natural encoding of constraints and consistency restoring rules, including

- *simple temporal constraints* (John needs to be at the office in two hours),
- *disjunctive temporal constraints* (action a can occur either between 8-9 am or between 4-5 pm),
- *qualitative soft temporal constraints* (It is desirable for John to come to office 15 mins early),
- *disjunctive qualitative soft temporal constraints* (It is desirable for action a to occur either between 2-4 pm or between 6-8pm).

Computing answer sets of programs in $\mathcal{AC}_0$: $\mathcal{AD}solver$

$\mathcal{AD}solver$ consists of two parts:

(a). Partial grounder $\mathcal{P}ground$.

- $\mathcal{P}ground$ uses a modified version of $lparse$ to ground regular variables of input program Π of $\mathcal{AC}_0$.
- The output program $\mathcal{P}(\Pi)$ is much smaller when compared to $lparse(\Pi)$.
- **Proposition:** Answer sets of Π are same as answer sets of $\mathcal{P}(\Pi)$.

(b). Inference engine $\mathcal{AD}engine$.

- $\mathcal{AD}engine$ combines ASP, $CR\text{-}Prolog$ and difference constraint solving methods to compute answer sets of program $\mathcal{P}(\Pi)$ of $\mathcal{AC}_0$.

$\mathcal{AD}engine$

$\mathcal{AD}engine$ integrates a $CR\text{-}Prolog$ engine with a difference constraint solver, $\mathcal{D}solver$.

Given a set of difference constraints D of the form $X - Y \leq k$, $\mathcal{D}solver$ computes a solution for D.

$\mathcal{D}solver$ is an incremental constraint solver: given a set of constraints D, a solution S to D and a new constraint $X - Y \leq k$, $\mathcal{D}solver$ uses S to compute a solution for $D \cup \{X - Y \leq k\}$.

The underlying ASP engine and $\mathcal{D}solver$ are tightly coupled.

Planning with temporal constraints

Ram is at his office and has a dentist appointment in one hour. For the appointment, he needs his insurance card which is at home and cash to pay the doctor. He can get cash from the nearby atm.

The table shows the time needed to travel between locations: Dentist, Home, Office and Atm.

Minutes	Doctor	Home	Office	Atm
Doctor	0	20	30	40
Home	20	0	15	15
Office	30	15	0	20
Atm	40	15	20	0

(a). Find a plan which takes Ram to the doctor on time.

(b). Find a plan which takes Ram to the doctor at least 15 minutes early.

Example: continued

$\mathcal{P}(\Pi)$ has 610 rules and $\mathcal{AD}solver$ took 0.17 seconds to solve both problems.

An ASP program equivalent to Π with constraint variables ranged from $[0..1440]$, $lparse$ could not ground the program.

If constraint variables range from $[0..100]$, the ground instantiation is 366060 rules. Using ASP solvers, it takes 183.18 and 161.20 seconds to solve first and second problems respectively.

Domain size of constraint variables does not affect the efficiency of $\mathcal{AD}solver$.

Proposed Work

- Design an algorithm $\mathcal{AD}solver$, to compute answer sets of programs in $\mathcal{AC}_0$ and prove its correctness.
  - partial grounder uses intelligent grounding mechanisms.
  - the algorithm tightly couples ASP, $CR-Prolog$ and difference constraint solver.
  - difference constraint solver is incremental.
- Implement $\mathcal{AD}solver$.
- Show efficiency of $\mathcal{AD}solver$ over classical ASP and CR-$Prolog$ solvers for planning with temporal constraints.
- ? Compare between tightly coupled $\mathcal{AD}solver$ and a loosely coupled solver which computes answer sets of programs in $\mathcal{AC}_0$.

- Develop a collection of languages $\mathcal{AC}(\mathcal{C})$ parameterized over $\mathcal{C}$.
  - syntax of $\mathcal{AC}(\mathcal{C})$ is an extension to syntax of $A\text{-}Prolog$.
  - semantics of $\mathcal{AC}(\mathcal{C})$ is a natural extension of semantics of $A\text{-}Prolog$.
- Design an algorithm $\mathcal{AC}solver$, to compute answer sets of programs in $\mathcal{AC}(\mathcal{C})$ and prove its correctness.
  - partial grounder uses intelligent grounding mechanisms.
  - the algorithm tightly couples ASP, $CR-Prolog$ and a CLP solver (dependent on $\mathcal{C}$).
  - CLP solver is incremental.
- Implement $\mathcal{AC}solver$ for a particular $\mathcal{C}$.
  - $CLP(R)$

Structure of Talk

- Background
  - Answer Set Programming (ASP)
  - Constraint Logic Programming (CLP)
- Language $\mathcal{AC}(\mathcal{C})$
  - Syntax and Semantics
- $\mathcal{AC}solver$
  - $\mathcal{P}ground$
  - $\mathcal{AC}engine$
- An Instance $\mathcal{AC}_0$
- Proposed Work

Answer Set Programming

ASP is a declarative programming paradigm.

Language $A\text{-}Prolog$, is a knowledge representation language with roots in the semantics of logic programming languages and non-monotonic reasoning.

The language is expressive, and has a well understood methodology for representing defaults, causal properties of actions and fluents, various types of incompleteness, etc.

There are several derivatives and expansions of $A\text{-}Prolog$, we describe the $A\text{-}Prolog$ language that is most generally used.

A-Prolog syntax

The syntax of A-Prolog is determined by a typed $signature\ \Sigma$ consisting of types, typed object constants, typed variables, typed function symbols and typed predicate symbols.

We assume that the signature contains symbols for numbers and for the standard functions and relations of arithmetic. $Terms$ are built as in first-order languages.

$Atoms$ are expressions of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol with arity n and t_i's are terms of suitable types.

Atoms formed by arithmetic relations are called $arithmetic$ atoms.
Atoms formed by non-arithmetic relations are called $plain$ atoms.

A-Prolog syntax continued

Literals are atoms and negated atoms, i.e. expressions of the form $\neg p(t_1, \ldots, t_n)$.

A rule r of *A-Prolog* is a statement of the form:

$$h_1 \ or \ \ h_2 \ or \ \ldots or \ \ h_k \ \leftarrow l_1, \ \ldots l_m, \ not \, l_{m+1}, \ \ldots, not \, l_n. \qquad (1)$$

where $l_1, \ldots, l_m$ are literals, and h_i's and $l_{m+1}, \ldots, l_n$ are plain literals.

A *A-Prolog* program is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of rules.

A-Prolog semantics

A plain literal l is satisfied by a consistent set of plain literals S, $S \models l$, if $l \in S$.
An extended plain literal $not\ l$ is satisfied by S if $S \not\models l$.
A set of literals L is satisfied by S if each element of L is satisfied by S.

A consistent set of plain literals S *is closed under a program* Π *not containing default negation* if, for every rule, $r \in \Pi$, if $body(r)$ is satisfied by S, we have $head(r) \cap S \neq \emptyset$.

Answer Set of a program without default negation: A consistent set of plain literals, S, is an answer set of a program Π not containing default negation if S is minimally (set theoretic) closed under all the rules of Π.

A-Prolog semantics continued

Reduct of a *A-Prolog* program: Let Π be an arbitrary *A-Prolog* program. For any set S of plain literals, let Π^S be the program obtained from Π by deleting:

- each rule, r, such that $neg(r) \cap S \neq \emptyset$;
- all formulas of the form $not\ l$ in the bodies of the remaining rules.

Answer Set of a *A-Prolog* program: A set of plain literals, S, is an answer set of a *A-Prolog* program Π if it is an answer set of Π^S.

Example

Constraint Logic Programming

Constraint Logic Programming (CLP) began as a natural merger of two declarative paradigms: constraint solving and logic programming. This combination helps make CLP programs both expressive and flexible.

Constraint logic programming can be said to involve the incorporation of constraints and constraint "solving" methods in logic-based language. This characterization suggests the possibility of many interesting languages, based on different constraints like $CLP(R)$, $CLP(FD)$.

Prolog can be said to be a CLP language where the constraints are equations over the algebra of terms (the Herbrand domain).

CLP syntax - 1

The language $CLP(\mathcal{D})$ is defined over a $signature$ $\Sigma = \langle P, \mathcal{D} \rangle$, where P and $\mathcal{D}$ are defined as sets of typed constants, variables, function and predicate symbols (with specified arity for each symbol and type for each parameter).

The simplest form of constraint we can define from $\mathcal{D}$ is a $primitive\ constraint$. A primitive constraint consists of a constraint relation symbol from $\mathcal{D}$ together with appropriate number of arguments.

A $constraint$ is of the form $c_1 \wedge \ldots \wedge c_n$ where $n \geq 0$ and $c_1, \ldots, c_n$ are primitive constraints.

CLP syntax - 2

Terms of the language are built as in first order languages.

An atom is of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol from P and $t_1, \ldots, t_n$ are terms from P and $\mathcal{D}$.

A rule is of the form

$$A_0 \leftarrow \alpha_1, \alpha_2, \ldots, \alpha_k.$$

where each α_i, $1 \leq i \leq k$, is either a primitive constraint or an atom.

A $CLP(\mathcal{D})$ $program$ is defined as a finite collection of rules.

$CLP(R)$ example

```
fib(0, 1).
fib(1, 1).
fib(N, X1 + X2) :- N > 1, fib(N - 1, X1), fib(N - 2, X2).
```

A goal (query) which asks for a number A such that fibonacci of A lies between 80 and 90 is:

? $80 \leq B, B \leq 90, fib(A, B)$.

answer constraints: $A = 10, B = 89$.

$CLP(\mathcal{D})$ operation model

The semantics of $CLP(\mathcal{D})$ is defined by its operation model.

Let C be a set of primitive constraints referred to as a *constraint store*.

Given a program Π, a query $Q = l_1, \ldots, l_n$, and a constraint store C, there is a *derivation step* from a pair $\langle Q, C \rangle$ to another pair $\langle Q_1, C_1 \rangle$ if:

- $Q_1 = l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n$, where l_i is a primitive constraint and C_1 is a (possibly simplified) set of constraints equivalent to $C \cup \{l_i\}$. Furthermore, C_1 is solvable;
- or $Q_1 = l_1, \ldots, l_{i-1}, b_1, \ldots, b_m, l_{i+1}, \ldots, l_n$, where there is a rule $r \in \Pi$ such that head(r) can be unified with l_i and $body(r)$ is unified with the set of literals $\{b_1, \ldots, b_m\}$; and $C_1 = C$.

$clp(\mathcal{D})$ operational model

A *derivation sequence* is a possibly infinite sequence of pairs $\langle Q_0, C_0\rangle, \langle Q_1, C_1\rangle \ldots$, starting with an initial query Q_0 and initial constraint store C_0, such that there is a derivation step to each pair $\langle Q_i, C_i\rangle, i > 0$, from the preceding pair $\langle Q_{i-1}, C_{i-1}\rangle$.

Let Π be a program as follows:

$$d(X,Y) \leftarrow X^2 \geq Y,\ e(X).$$
$$e(X) \leftarrow X * 4 \leq 20.$$
$$e(X) \leftarrow X = 2.$$

Let $Q = d(X,Y)$ and let $C = \emptyset$. The following is a derivation sequence:

$$\langle Q,\ C\rangle = \langle\ \{d(X,Y)\},\ \emptyset\ \rangle$$
$$\langle Q_1, C_1\rangle = \langle\ \{X^2 \geq Y, e(X)\},\ \emptyset\ \rangle$$
$$\langle Q_2, C_2\rangle = \langle\ \{e(X)\},\ \{X^2 \geq Y\}\ \rangle$$
$$\langle Q_3, C_3\rangle = \langle\ \{X * 4 \leq 20\},\ \{X^2 \geq Y\}\ \rangle$$
$$\langle Q_4, C_4\rangle = \langle\ \emptyset,\ \{X^2 \geq Y,\ X \leq 5\}\ \rangle$$

$clp(\mathcal{D})$ operational model

A sequence is $successful$ if it is finite and its last element is $\langle \emptyset, C_n \rangle$, where C_n is a set of solved constraints.

A sequence is $conditionally\ successful$ if it is finite and its last element is $\langle \emptyset, C_n \rangle$, where C_n consists of delayed and possibly some solved constraints.

A sequence is $finitely\ failed$ if it is finite, neither successful nor conditionally successful, and such that no derivation step is possible from its last element.

Given a program Π and a query Q, $CLP(\mathcal{D})solver$ returns: $true$, if it finds a successful sequence; $maybe$ if it finds a conditionally successful derivation and $false$ if all sequences are finitely failed.

$\mathcal{AC}(\mathcal{C})$ syntax - 1

$\mathcal{AC}(\mathcal{C})$ is a collection of languages parametrised over a collection $\mathcal{C}$ of constraints (constructed in a natural way).

The syntax of $\mathcal{AC}(\mathcal{C})$ is determined by a sorted signature Σ, consisting of sorts $S_1, \ldots, S_n$, properly typed predicate symbols, variables and function symbols.

Variables of $\mathcal{AC}(\mathcal{C})$ have a sort assigned to them. Each variable ranges over objects constants from their sort. A term of Σ is either a constant, a variable, or an expression $f(t_1, \ldots, t_n)$, where f is a function symbol of arity n, $t_1, \ldots, t_n$ are terms of proper sorts.

Terms also have sorts and are assigned in the natural way. An $atom$ is of the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms of proper sorts.

$\mathcal{AC}(\mathcal{C})$ syntax - 2

Each sort is further distinguished as either a $regular\ sort$ or a $constraint\ sort$.

The object constants from regular sorts and constraint sorts are called $r\text{-}constants$ and $c\text{-}constants$ respectively.

A variable is an $r\text{-}variable$ ($c\text{-}variable$) if it ranges over constants from a regular (constraint) sort.

Predicate symbols of Σ are divided into four disjoint sets called regular, constrained, mixed and defined, denoted by P_r, P_c, P_m and P_d respectively.

The c-predicate symbols P_c and c-function symbols F_c in Σ are subsets of predicate and function symbols from $\mathcal{C}$ respectively.

$\mathcal{AC}(\mathcal{C})$ syntax - 3

An atom $p(t_1, \ldots, t_n)$ is

- an $r\text{-}atom$ if $p \in P_r$ and $t_1, \ldots, t_n$ are r-terms;
- a $c\text{-}atom$ if $p \in P_c$ and $t_1, \ldots, t_n$ are c-terms;
- a $m\text{-}atom$ if $p \in P_m$ and $t_1, \ldots, t_n$ are r-terms or c-terms with at least one from each;
- a $d\text{-}atom$ if $p \in P_d$ and where each parameter t_i is either an r-term or a c-term.

A $literal$ is either an atom a or its negation $\neg a$.

An $extended\ literal$ is either a literal l, or its negation $not\ l$.

$\mathcal{AC}(\mathcal{C})$ syntax - 4

A $rule\ r$ of $\mathcal{AC}(\mathcal{C})$ over Σ is a statement of the form:

$$h_1\ or\ \ldots or\ \ h_k \leftarrow l_1,\ \ldots,\ l_m,\ not\ \ l_{m+1},\ \ldots,\ not\ \ l_n \qquad (2)$$

where $h_1, \ldots, h_k$ are r-literals or d-literals and $l_1, \ldots, l_n$ are arbitrary literals.

A rule is an $r\text{-}rule$ if the literals in the head and body of the rule are all r-literals. A rule is a $d\text{-}rule$ if all literals in the head are d-literals. The rest of the rules are called $m\text{-}rule$s.

A $program$ of $\mathcal{AC}(\mathcal{C})$ is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a collection of rules over Σ.

Example

Let Σ be a signature with constants $C_r = \{a, b\}$ and $C_c = \{0, \ldots, 100\}$; $P_r = \{p(C_r, C_r),\ q(C_r)\}$, $P_m = \{\ at(C_r, C_c)\ \}$, $P_c = \{>,\ =,\ \leq,\ \geq\}$ and $P_d = \{\ equal(C_c, C_c)\ \}$; variables $V_r = \{X,\ Y\}$ range over C_r and $V_c = \{T_1,\ T_2\}$ range over C_c.

Π_1 :

$$q(a).$$
$$q(b).$$
$$p(X, Y) \leftarrow q(X),\ q(Y),\ at(X, T_1),\ at(Y, T_2),\ T_1 > T_2.$$

Π_2 :

$$q(a).$$
$$q(b).$$
$$p(X, Y) \leftarrow q(X),\ q(Y),\ at(X, T_1),\ at(Y, T_2),\ equal(T_1, T_2).$$
$$equal(T_1, T_2) \leftarrow T_1 = T_2.$$

$\mathcal{AC}(\mathcal{C})$ semantics - 1

To give semantics of programs in $\mathcal{AC}(\mathcal{C})$, first we transform an arbitrary program Π into its ground instantiation $ground(\Pi)$. The semantics of ground(Π), will be viewed as the semantics of program Π.

A *ground instance* of a rule r is a rule obtained from r by:

1. replacing variables of r by ground terms from respective sorts;
2. replacing all numerical terms by their values.

$\mathcal{AC}(\mathcal{C})$ semantics - 2

A set of ground literals, S satisfies the head of a ground rule r if at least one of the literals in the head of r is satisfied by S. S satisfies the body of r if literals $l_1, \ldots, l_m$ belong to S and literals $l_{m+1}, \ldots, l_n$ do not belong to S. S satisfies rule r, if the head of r is satisfied by S whenever the body of r is satisfied by S.

$$q(a).$$
$$q(b).$$
$$p(a,b) \leftarrow q(a),\ q(b),\ at(a,2),\ at(b,3),\ 2<3.$$

$S_1 = \{q(a), q(b)\}$ satisfies first two rules

$S_2 = \{q(a), q(b), p(a,b), at(a,2), at(b,3), <(2,3)\}$ satisfies all rules.

$\mathcal{AC}(\mathcal{C})$ semantics - 3

Candidate Mixed Set: Let X be a set of ground m-atoms such that for every mixed predicate $p \in P_m$ and every list of ground r-terms $\bar{t}_r$, there is exactly one list of ground c-terms $\bar{t}_c$ such that $p(\bar{t}_r, \bar{t}_c) \in X$. We call a set of mixed atoms that satisfies this condition a *candidate-mixed* set.

Let mixed predicates of Σ, $P_m = \{at(C_r, C_c)\}$, where $C_r = \{a, b, c\}$ and $C_c = [0 \ldots 1000]$.

$X_1 = \{at(a, 1), at(b, 2), at(a, 3), at(c, 45)\}$ is not a *candidate-mixed* set.

$X_2 = \{at(a, 48), at(b, 32)\}$ is not a *candidate-mixed* set.

$X_3 = \{at(a, 10), at(b, 23), at(c, 45)\}$ is a *candidate-mixed* set.

$\mathcal{AC}(\mathcal{C})$ semantics - 4

Intended Interpretation: Let M_c be the set of all ground c-atoms that are true in the intended interpretation of corresponding predicate symbols in P_c.

Let constraint predicates in Σ, $P_c = \{<(C_c, C_c), \ \ >(C_c, C_c)\}$, with $C_c = [0..100]$.

We get:
$M_c = \{0 < 1, \ldots, 0 < 100, 1 < 2, \ldots, 99 < 100, 1 > 0, \ldots, 100 > 99\}$

$\mathcal{AC}(\mathcal{C})$ semantics - 5

[**Answer Set**] Let (Σ, Π) be an $\mathcal{AC}(\mathcal{C})$ program and X be a $candidate\text{-}mixed$ set; a set S of ground literals over Σ is an $\mathcal{AC}(\mathcal{C})$ answer set of Π if S is an ASP answer set of $ground(\Pi) \cup X \cup M_c$.

Let Π_1 and Π_2 be programs as shown before.

An answer set for program Π_1 is $S = A \cup X \cup M_c$, where
$A = \{q(a),\ q(b)\}$, $X = \{at(a,3),\ at(b,3)\}$ and
$M_c = \{1 > 0, \ldots, 100 > 99, 0 = 0, 1 = 1, \ldots, 100 = 100\}$.

An answer set for program Π_2 is $S = A \cup X \cup D \cup M_c$, where
$A = \{q(a), q(b), p(a,b), p(b,a)\}$, $X = \{at(a,2), at(b,2)\}$,
$M_c = \{1 > 0, \ldots, 100 > 99, 0 = 0, 1 = 1, \ldots, 100 = 100\}$, and
$D = \{equal(0,0), equal(1,1), equal(2,2), \ldots\}$.

$\mathcal{AC}(\mathcal{C})$ semantics - 6

simplified answer set: Let $M = A \cup X \cup D \cup M_c$ be an answer set of a program Π, where A, X, D are sets of regular, mixed and defined atoms respectively and M_c is the set of c-atoms representing the intended interpretation of c-predicates in P_c. The set $A \cup X$ is called a *simplified answer set* of Π.

$\mathcal{AC}solver$

Input: an $\mathcal{AC}(\mathcal{C})$ program Π

Output: returns $true$ and an $\mathcal{AC}(\mathcal{C})$ simplified answer set of Π; or returns $false$

$\mathcal{AC}solver$

The solver for computing answer sets of programs in $\mathcal{AC}(\mathcal{C})$ is called $\mathcal{AC}solver$. It consists of three parts.

1. $\mathcal{P}ground$
2. $\mathcal{T}ranslator$
3. $\mathcal{AC}engine$

Given a $\mathcal{AC}(\mathcal{C})$ program Π, $\mathcal{AC}solver$ first calls $\mathcal{P}ground$ to ground r-terms of Π, the resulting r-ground program, $\mathcal{P}(\Pi)$, is transformed by $\mathcal{T}ranslator$ into an r-ground program $\mathcal{T}(\Pi)$. The $\mathcal{AC}engine$ combines constraint solving techniques with ASP techniques to compute answer sets of $\mathcal{T}(\Pi)$.

Terminology

Π can be divided into three disjoint sets of rules.

The set consisting of r-rules of Π is called the $regular$ part of Π denoted by Π_R. Similarly, the set consisting of m-rules of Π is called the $middle$ part of Π denoted by Π_M and the set consisting of d-rules of Π is called the $defined$ part of Π denoted by Π_D.

The collection of rules of a program Π whose heads are formed by a predicate p, is called the $definition$ of p in Π. A predicate p is called a $domain\ predicate$ with respect to Π, if the definition of p in Π has no negative recursion.

Syntax Restrictions

1. *There is only one literal in the head (non-disjunctive programs).* This restriction allows for a simpler description of the algorithm.
2. *Given a rule $r \in \Pi_M$, every c-variable of r should occur in m-literals of body(r).* This restriction ensures the correctness of the algorithm $\mathcal{AC}engine$.
3. *Each r-variable occurring in $r \in \Pi_D$, occurs in head(r).* The consequences of Π_D will be computed using constraint programming techniques. This restriction will ensure that a r-variable of a rule $r \in \Pi_D$ will be ground at the time of solving the head of r; thus allowing to perform lazy grounding.
4. *The only extended m-literals, d-literals and c-literals allowed in Π are atoms.* This restriction simplifies the description of the algorithm.

5. *Mixed literals do not occur in rules of* Π_D. This restriction simplifies the description of the algorithm.

6. $\Pi_R \cup \Pi_M$ *is r-domain restricted in the sense that every r-variable in a rule* $r \in \Pi_R \cup \Pi_M$, *must appear in an r-atom formed by a domain predicate in the body of* r. This restriction is similar to domain restriction of the grounder $lparse$ [] and differs by restricting only r-variables. This restriction allows $\mathcal{P}ground$ to use $lparse$ to ground r-terms of $\Pi_R \cup \Pi_M$.

7. *There are no cyclic definitions between d-literals and r-literals.* This restriction simplifies the algorithm $\mathcal{AC}solver$ and ensures the correctness of the algorithm $\mathcal{AC}engine$.

Example - 1

$\Sigma = \{C_r = \{a, b\},\ C_c = [1, \cdots, 100],\ P_r = \{p(C_r, C_r),\ q(C_r),\ r(C_r)\},$
$P_m = \{at(C_r, C_c)\},\ P_c = \{>\},\ P_d = \{d(C_c, C_c), d_2(C_r, C_c)\}\}$.

Program Π_3 does not satisfy restrictions (1), (2), (3) and (6):

$q(a)\ or\ \ q(b).$ $\qquad\qquad$ $r(a).$

$p(X, Y)\ \leftarrow\ q(X),\ at(X, T_1),\ d(T_1, T_2).$

$d(T_1, T_2)\ \leftarrow\ not\ r(X),\ T_1 > T_2.$

Program Π_4 does not satisfy restrictions (4), (5), (6) and (7):

$q(a) \leftarrow\ not\ q(b).$

$q(b) \leftarrow\ not\ q(a).$

$r(a).$ $\qquad\qquad$ $r(b).$

$p(X, Y)\ \leftarrow\ r(X),\ not\ q(Y),\ not\ at(X, T_1),\ at(Y, T_2),\ d(T_1, T_2).$

$d(T_1, T_2)\ \leftarrow\ r(X),\ not\ d_2(X, T_1),\ T_1 > T_2.$

$d_2(X, T)\ \leftarrow\ p(X, X),\ \neg at(X, T),\ T > 50.$

Example - 2

$\Sigma = \{C_r = \{a, b\},\ C_c = [1, \cdots, 100],\ P_r = \{p(C_r, C_r),\ q(C_r),\ r(C_r)\},$
$P_m = \{at(C_r, C_c)\},\ P_c = \{>\},\ P_d = \{d(C_r, C_c, C_c)\}\}$. Programs Π_1 and Π_2 satisfy the syntax restrictions:

Program Π_1:

$q(a).$ $\quad$ $q(b).$ $\quad$ $r(a).$

$p(X, Y) \leftarrow q(X),\ q(Y),\ at(X, T_1),\ at(Y, T_2),\ d(X, T_1, T_2).$

$d(X, T_1, T_2) \leftarrow not\ r(X),\ T_1 > T_2.$

Program Π_2:

$q(a) \leftarrow not\ q(b).$

$q(b) \leftarrow not\ q(a).$

$r(a).$ $\quad$ $r(b).$

$p(X, Y) \leftarrow r(X),\ r(Y), not\ q(Y),\ at(X, T_1),\ at(Y, T_2),\ T_1 > T_2.$

$\mathcal{P}ground$

Given a $\mathcal{AC}(\mathcal{C})$ program Π, $\mathcal{P}ground$ returns a r-ground program $\mathcal{P}(\Pi)$. *Answer sets of* $\mathcal{P}(\Pi)$ *are same as answer sets of* Π.

Let Σ be formed by $C_r = \{1, 2\}$, $C_c = [0..100]$, $P_r = \{p, q, r, s, <, \neq\}$, $P_m = \{at\}$, $P_c = \{\leq, =\}$, $P_d = \{d\}$ and variables $V_r = \{X,\ Y\}$ and $V_c = \{T_1,\ T_2\}$. Π is as follows:

$$
\begin{array}{lll}
r_{a_{[1,2]}}: & q(1). & q(2). \\
r_b: & p(X,Y) \leftarrow & q(X),\ q(Y),\ r(X,Y),\ X < Y \\
r_c: & r(X,Y) \leftarrow & q(X),\ q(Y),\ not\ s(X,Y),\ X \neq Y \\
r_d: & s(X,Y) \leftarrow & q(X),\ q(Y),\ not\ r(X,Y),\ X \neq Y \\
r_e: & p(X,Y) \leftarrow & q(X),\ q(Y),\ at(X,T_1),\ at(Y,T_2), \\
 & & T_1 \leq T_2,\ d(X,Y,T_1,T_2) \\
r_f: & d(X,Y,T_1,T_2) \leftarrow & s(X,Y),\ X < Y,\ T_1 = T_2 + 10
\end{array}
$$

$\mathcal{P}ground$

Given a program Π of $\mathcal{AC}(\mathcal{C})$, we construct an r-ground program $\mathcal{P}(\Pi)$ as follows:

1. replace c-variables in Π_M by constants called $tc\_constants$,
   $\Pi_1 = tc(\Sigma_\Pi, \Pi_M) \cup \Pi_R \cup \Pi_D$

$$
\begin{array}{ll}
r_{a_{[1,2]}}: & q(1). \quad q(2). \\
r_b: & p(X,Y) \leftarrow q(X),\ q(Y),\ r(X,Y),\ X < Y \\
r_c: & r(X,Y) \leftarrow q(X),\ q(Y),\ not\ s(X,Y),\ X \neq Y \\
r_d: & s(X,Y) \leftarrow q(X),\ q(Y),\ not\ r(X,Y),\ X \neq Y \\
r_{e_1}: & p(X,Y) \leftarrow q(X),\ q(Y),\ at(X,t_1),\ at(Y,t_2), \\
 & \qquad\qquad t_1 \leq t_2, d(X,Y,t_1,t_2) \\
r_f: & d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X < Y,\ T_1 = T_2 + 10
\end{array}
$$

2. remove c-literals occurring in Π_{1_M} and store them in a set C,
   $\Pi_2 = \Pi_R \cup \Pi_D \cup (\Pi_{1_M} \setminus c\text{-}lits(\Pi_{1_M}))$

$$
\begin{array}{lll}
r_{a_{[1,2]}}: & q(1). \quad q(2). & \\
r_b: & p(X,Y) & \leftarrow\ q(X),\ q(Y),\ r(X,Y),\ X<Y \\
r_c: & r(X,Y) & \leftarrow\ q(X),\ q(Y),\ not\ s(X,Y),\ X \neq Y \\
r_d: & s(X,Y) & \leftarrow\ q(X),\ q(Y),\ not\ r(X,Y),\ X \neq Y \\
r_{e_1}: & p(X,Y) & \leftarrow\ q(X),\ q(Y),\ at(X,t_1),\ at(Y,t_2), \\
 & & \quad d(X,Y,t_1,t_2) \\
r_f: & d(X,Y,T_1,T_2) & \leftarrow\ s(X,Y),\ X<Y,\ T_1 = T_2 + 10
\end{array}
$$

and $C = \{t_1 \leq t_2\}$.

3. add disjunctive rules for mixed and defined predicates and get $\Pi_3 = \Pi_2 \cup addr(\Pi_2)$

$$
\begin{aligned}
r_{m_1}: &\quad at(X,Z) \leftarrow not\ n\_at(X,Z).\\
r_{m_2}: &\quad n\_at(X,Z) \leftarrow not\ at(X,Z).\\
r_{m_3}: &\quad d(X,Y,Z_1,Z_2) \leftarrow not\ n\_d(X,Y,Z_1,Z_2).\\
r_{m_4}: &\quad n\_d(X,Y,Z_1,Z_2) \leftarrow not\ d(X,Y,Z_1,Z_2).
\end{aligned}
$$

where, variables Z, Z_1, Z_2 range over tc_constants $\{t_1, t_2\}$ and variables X, Y range over $\{1, 2\}$.

The predicates n_at and n_d are new predicates not in Σ_2.

4. ground using $lparse$, $\Pi_4 = lparse(\Pi_{3_R} \cup \Pi_{3_M} \cup addr(\Pi_2)) \cup \Pi_{3_D}$

$rg_{a_{[1,2]}}$: $q(1).\quad q(2).$

rg_{b_1} : $p(1,2) \leftarrow r(1,2)$

$rg_{c_{[1,2]}}$: $r(1,2) \leftarrow not\ s(1,2).\quad r(2,1) \leftarrow not\ s(2,1).$

$rg_{d_{[1,2]}}$: $s(1,2) \leftarrow not\ r(1,2).\quad s(2,1) \leftarrow not\ r(2,1).$

rg_{e_1} : $p(1,1) \leftarrow at(1,t_1),\ at(1,t_2),\ d(1,1,t_1,t_2)$

rg_{e_2} : $p(1,2) \leftarrow at(1,t_1),\ at(2,t_2),\ d(1,2,t_1,t_2)$

rg_{e_3} : $p(2,1) \leftarrow at(2,t_1),\ at(1,t_2),\ d(2,1,t_1,t_2)$

rg_{e_4} : $p(2,2) \leftarrow at(2,t_1),\ at(2,t_2),\ d(2,2,t_1,t_2)$

rg_{m_1} : $at(1,t_1) \leftarrow not\ n\_at(1,t_1).\ \cdots$

rg_{m_k} : $n\_at(2,t_2) \leftarrow not\ at(2,t_2).$

rg_{d_1} : $d(a,a,t_1,t_1) \leftarrow not\ n\_d(a,a,t_1,t_1).\ \cdots$

rg_{d_j} : $n\_d(b,b,t_2,t_2) \leftarrow not\ n\_d(b,b,t_2,t_2).$

r_f : $d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X < Y,\ T_1 = T_2 + 10$

5. remove ground instantiations of $addr(\Pi_2)$ to get
   $\Pi_5 = \Pi_4 \setminus ground(addr(\Pi_2))$

$$
\begin{array}{lll}
rg_{a_{[1,2]}}: & q(1). \quad q(2). & \\
rg_{b_1}: & p(1,2) & \leftarrow \; r(1,2) \\
rg_{c_1}: & r(1,2) & \leftarrow \; not\ s(1,2) \\
rg_{c_2}: & r(2,1) & \leftarrow \; not\ s(2,1) \\
rg_{d_1}: & s(1,2) & \leftarrow \; not\ r(1,2) \\
rg_{d_2}: & s(2,1) & \leftarrow \; not\ r(2,1) \\
rg_{e_1}: & p(1,1) & \leftarrow \; at(1,t_1),\ at(1,t_2),\ d(1,1,t_1,t_2) \\
rg_{e_2}: & p(1,2) & \leftarrow \; at(1,t_1),\ at(2,t_2),\ d(1,2,t_1,t_2) \\
rg_{e_3}: & p(2,1) & \leftarrow \; at(2,t_1),\ at(1,t_2),\ d(2,1,t_1,t_2) \\
rg_{e_4}: & p(2,2) & \leftarrow \; at(2,t_1),\ at(2,t_2),\ d(2,2,t_1,t_2) \\
r_f: & d(X,Y,T_1,T_2) & \leftarrow \; s(X,Y),\ X<Y,\ T_1 = T_2 + 10
\end{array}
$$

6. (a) put back c-literals removed in step (2), to get $\Pi_{6a} = \beta(\Pi_5, C, \Sigma_2)$

   (b) replace tc_constants by c-variables,

$$\Pi_6 = \Pi_{5_R} \cup \Pi_{5_D} \cup reverse\_tc(\Pi_{5_M}, T_c)$$

$$rg_{a_{[1,2]}}: \quad q(1). \quad q(2).$$
$$rg_{b_1}: \quad p(1,2) \leftarrow r(1,2)$$
$$rg_{c_1}: \quad r(1,2) \leftarrow not\ s(1,2)$$
$$rg_{c_2}: \quad r(2,1) \leftarrow not\ s(2,1)$$
$$rg_{d_1}: \quad s(1,2) \leftarrow not\ r(1,2)$$
$$rg_{d_2}: \quad s(2,1) \leftarrow not\ r(2,1)$$
$$rg_{e_{1'}}: \quad p(1,1) \leftarrow at(1,T_1),\ at(1,T_2),\ T_1 \leq T_2,\ d(1,1,T_1,T_2)$$
$$rg_{e_{2'}}: \quad p(1,2) \leftarrow at(1,T_1),\ at(2,T_2),\ T_1 \leq T_2,\ d(1,2,T_1,T_2)$$
$$rg_{e_{3'}}: \quad p(2,1) \leftarrow at(2,T_1),\ at(1,T_2),\ T_1 \leq T_2,\ d(2,1,T_1,T_2)$$
$$rg_{e_{4'}}: \quad p(2,2) \leftarrow at(2,T_1),\ at(2,T_2),\ T_1 \leq T_2,\ d(2,2,T_1,T_2)$$
$$r_f: \quad d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X < Y,\ T_1 = T_2 + 10$$

7. compute rename(Π_{6_M},Y), to get $\Pi_7 = rename(\Pi_{6_M}, Y) \cup \Pi_{6_R} \cup \Pi_{6_D}$ where Y is a r_ground_mixed set of Π. We get $\mathcal{P}(\Pi)$ as:

$rg_{a_{[1,2]}}: \quad q(1). \quad q(2).$

$rg_{b_1}: \quad p(1,2) \leftarrow r(1,2)$

$rg_{c_1}: \quad r(1,2) \leftarrow not\ s(1,2)$

$rg_{c_2}: \quad r(2,1) \leftarrow not\ s(2,1)$

$rg_{d_1}: \quad s(1,2) \leftarrow not\ r(1,2)$

$rg_{d_2}: \quad s(2,1) \leftarrow not\ r(2,1)$

$rg_{e_a}: \quad p(1,1) \leftarrow at(1,V_1),\ at(1,V_1),\ V_1 \leq V_1,\ d(1,1,V_1,V_1)$

$rg_{e_b}: \quad p(1,2) \leftarrow at(1,V_1),\ at(2,V_2),\ V_1 \leq V_2,\ d(1,2,V_1,V_2)$

$rg_{e_c}: \quad p(2,1) \leftarrow at(2,V_2),\ at(1,V_1),\ V_2 \leq V_1,\ d(2,1,V_2,V_1)$

$rg_{e_d}: \quad p(2,2) \leftarrow at(2,V_2),\ at(2,V_2),\ V_2 \leq V_2,\ d(2,2,V_2,V_2)$

$r_f: \quad d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X < Y,\ T_1 = T_2 + 10$

Negative Literals

Recall that $\mathcal{P}ground$ uses $lparse$ for intelligent grounding of r-terms of a program Π. The system $lparse$ while grounding also transforms the program with $negative$ literals ($\neg$) to an equivalent program (with respect to answer sets) without negative literals. Given a literal l and its negation $\neg l$, $lparse$ does the following transformation:

- replaces each occurrence of $\neg l$ in Π by a new literal l'
- adds the following rule to the program.

  $\leftarrow l,\ l'.$

Therefore, to simplify the description of the solver, we assume that programs with negated literals undergo the above transformation and $\mathcal{P}(\Pi)$ *does not contain any negative literals.*

$\mathcal{T}ranslator$

The $\mathcal{T}ranslator$ transforms $\mathcal{P}(\Pi)$ into a new r-ground program $\mathcal{T}(\Pi)$.

Answer sets of $\mathcal{P}(\Pi)$ have one to one correspondence with answer sets of $\mathcal{T}(\Pi)$.

The solver $\mathcal{AC}engine$ then computes answer sets of $\mathcal{T}(\Pi)$ using constraint programming techniques integrated with ASP techniques.

The program $\mathcal{P}(\Pi)$ is translated to $\mathcal{T}(\Pi)$ in order to eliminate disjunctive queries to the CLP system.

Terminology

Let Π be a r-ground program with signature Σ and B be a set of ground extended r-literals of Σ.

- We will identify an expression $not\ (not\ a))$ with a.
- $pos(B) = \{a \in Atoms(\Sigma) \mid a \in B\}$,
  $neg(B) = \{a \in Atoms(\Sigma) \mid not\ a \in B\}$,
  $Atoms(B) = pos(B) \cup neg(B)$.
- A set, M, of atoms $agrees$ with B if $pos(B) \subseteq M$ and $neg(B) \cap M = \emptyset$.
- B $covers$ a set of atoms M, $covers(B, M)$, if $M \subseteq Atoms(B)$
- B is $inconsistent$ if $pos(B) \cap neg(B) \neq \emptyset$.

$\mathcal{AC}engine$

Input: a r-ground $\mathcal{AC}(\mathcal{C})$ program Π and a set of ground extended r-literals S.

Output: returns $true$ and a simplified $\mathcal{AC}(\mathcal{C})$ answer set of Π agreeing with S; otherwise returns $false$

Proposition: If $\mathcal{AC}engine$ returns $true$ and a set M, then M is a simplified $\mathcal{AC}(\mathcal{C})$ answer set of Π.

Functions used by $\mathcal{AC}engine$

- function expand
  - takes as input a r-ground program $\Pi_R \cup \Pi_M$ and a set of ground extended r-literals S; and
  - returns a consistent set of ground extended r-literals which is the set of consequences of $\Pi_R \cup \Pi_M$ and S.
- function pick
  - takes as input the r-atoms of Π and a set of ground extended r-literals S; and
  - returns a literal l such that $l \notin S$ and $not\ l \notin S$.

- function c_solve
  - takes as input a non-ground program Π_D, a set of ground extended r-literals S and a query Q which is a set of r-ground d-literals and c-literals; and
  - returns $true$, $maybe$ or $false$. If c_solve returns $true$ then it also returns a set of constraints A.

Let Π_D be the defined part of a program Π and S be a set of ground extended r-literals. Let $Q = query(\Pi, S)$, D be the set of d-literals in Q and $\bar{V}_Q$ be the set of variables in Q.

- If c_solve(Π_D, S, Q, A) returns $true$ then every answer set of Π agreeing with S contains $D|_\theta^{\bar{V}_Q}$, where $\theta = a\_solution(A)$.
- If c_solve(Π_D, S, Q, A) returns $false$ then there is no answer set of Π agreeing with S.

$\mathcal{AC}engine$(Π: r-ground program, B: set of r-literals)

$[a]$ $S := expand(\Pi_R \cup \Pi_M, B)$

$[b]$ **if** $inconsistent(S)$ **return** $false$

$[c]$ **if** $covers(S, rAtoms(\Pi))$ **then**

$[d]$ $V := c\_solve(\Pi_D,\ S,\ \text{query}(\Pi,S),\ A)$

$[e]$ **if** $V =$ true **return** true $\{$a-set: $pos(S) \cup m\_atoms|_A\}$

$[f]$ **else return** false

$[g]$ **else** $V := c\_solve(\Pi_D,\ S,\ \text{pos}(\text{query}(\Pi,S)),\ A)$

$[h]$ **if** $V =$ false **return** false

$[i]$ $pick(l, \bar{S})$

$[j]$ **if** $\mathcal{AC}engine(\Pi, S \cup \{l\})$ **then return** true

$[k]$ **else return** $\mathcal{AC}engine(\Pi, S \cup \{not\ l\})$

$\mathcal{AC}(\mathcal{C})$ instance: $\mathcal{AC}_0$

Consider an instance of $\mathcal{AC}(\mathcal{C})$ where $\mathcal{C} = \{X - Y > k\}$, $P_d = \emptyset$ and if $r \in \Pi_M$ then $head(r) = \emptyset$. Let us call this language $\mathcal{AC}_0$.

This language was studied in "S. Baselice, P. A. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. Proceedings of ICLP 2005".

We can expand the language using consistency restoring rules from $CR\text{-}Prolog$.

An algorithm to compute answer sets of programs in $\mathcal{AC}_0$, integrates

- Davis-Putnam type of algorithm for computing answer sets of ASP programs,
- the form of abduction from $CR\text{-}Prolog$, and
- incremental constraint satisfaction algorithm for difference constraints.

Language $\mathcal{AC}_0$

A program of $\mathcal{AC}_0$ consists of two modules

- regular rules of ASP and consistency restoring rules of $CR\text{-}Prolog$;
- denials with constraints of type $X - Y > k$.

The variables of a program Π that occur in constraints are called constraint variables and the rest are regular variables

The semantics of $\mathcal{AC}_0$ extends ASP, CR-Prolog and CASP semantics in a natural way.

Expressive Power of $\mathcal{AC}_0$

In addition to standard power of ASP, $\mathcal{AC}_0$ allows natural encoding of constraints and consistency restoring rules, including

- *simple temporal constraints* (John needs to be at the office in two hours),
- *disjunctive temporal constraints* (action a can occur either between 8-9 am or between 4-5 pm),
- *qualitative soft temporal constraints* (It is desirable for John to come to office 15 mins early),
- *disjunctive qualitative soft temporal constraints* (It is desirable for action a to occur either between 2-4 pm or between 6-8pm).

Computing answer sets of programs in $\mathcal{AC}_0$: $\mathcal{AD}solver$

$\mathcal{AD}solver$ consists of two parts:

(a). Partial grounder $\mathcal{P}ground$.

- $\mathcal{P}ground$ uses a modified version of $lparse$ to ground regular variables of input program Π of $\mathcal{AC}_0$.
- The output program $\mathcal{P}(\Pi)$ is much smaller when compared to $lparse(\Pi)$.
- **Proposition:** Answer sets of Π are same as answer sets of $\mathcal{P}(\Pi)$.

(b). Inference engine $\mathcal{AD}engine$.

- $\mathcal{AD}engine$ combines ASP, $CR\text{-}Prolog$ and difference constraint solving methods to compute answer sets of program $\mathcal{P}(\Pi)$ of $\mathcal{AC}_0$.

$\mathcal{AD}engine$

$\mathcal{AD}engine$ integrates a $CR\text{-}Prolog$ engine with a difference constraint solver, $\mathcal{D}solver$.

Given a set of difference constraints D of the form $X - Y \leq k$, $\mathcal{D}solver$ computes a solution for D.

$\mathcal{D}solver$ is an incremental constraint solver: given a set of constraints D, a solution S to D and a new constraint $X - Y \leq k$, $\mathcal{D}solver$ uses S to compute a solution for $D \cup \{X - Y \leq k\}$.

The underlying ASP engine and $\mathcal{D}solver$ are tightly coupled.

Planning with temporal constraints

Ram is at his office and has a dentist appointment in one hour. For the appointment, he needs his insurance card which is at home and cash to pay the doctor. He can get cash from the nearby atm.

The table shows the time needed to travel between locations: Dentist, Home, Office and Atm.

Minutes	Doctor	Home	Office	Atm
Doctor	0	20	30	40
Home	20	0	15	15
Office	30	15	0	20
Atm	40	15	20	0

(a). Find a plan which takes Ram to the doctor on time.

(b). Find a plan which takes Ram to the doctor at least 15 minutes early.

Example: continued

The domain description of the problem contains direct effects, in-direct effects, executability conditions of actions, and temporal constraints like:

```
% Time taken to travel between office and atm is at least 20 mins
:- h(at_loc(P, office), S), o(go_to(P, atm),S),
   at_time(S,T1), at_time(S+1,T2), T1 - T2 > -20.
```

$\mathcal{P}(\Pi)$ has 610 rules and $\mathcal{AD}solver$ took 0.17 seconds to solve both problems.

An ASP program equivalent to Π with constraint variables ranging from $[0..100]$ has a ground instantiation of 366060 rules. Using ASP solvers, it takes 183.18 and 161.20 seconds to solve first and second problems respectively.

If constraint variables ranged from $[0..1440]$ (the number of minutes in a day), $lparse$ could not ground the program.

Domain size of constraint variables does not affect the efficiency of $\mathcal{AD}solver$.

Reasoning in $\mathcal{AC}_0$

$\mathcal{AD}solver$ was tested for complex planning and scheduling problems using a decision support system for space shuttle controllers: *USA-Advisor*.

USA-Advisor was expanded by:

(a). adding temporal information on the time taken for fuel and oxidant to flow from one valve to another;

(b). temporal constraints to describe valve being stabilized before opening; and

(c). soft temporal constraint to describe, *if possible, it is desirable to fire the jets by 30 seconds*.

step, time domain	$\mathcal{P}ground$ seconds	$\mathcal{AD}solver$ seconds	$lparse$ seconds	$Surya$	$Smodels$
3, 0..400	8.033	18.158	error	-	-
3, 0..400	7.980	20.395	error	-	-
3, 0..31	8.033	18.158	10min	>2hr	>2hr
3, 0..31	7.980	20.395	10min	>2hr	>2hr
4, 0..31	10.682	31.807	10min	>2hr	>2hr
3, 0..31	8.023	15.770	10min	>2hr	>2hr
3, 0..31	7.989	12.011	10min	>2hr	>2hr
3, 0..31	7.986	19.297	10min	>2hr	>2hr

The table shows timing results of looking for plans with different initial situations and final goals.

Using longer steps results in $lparse$ unable to ground the program.

Proposed Work

- Develop a tight algorithm $\mathcal{AD}solver$
- Implement $\mathcal{AD}solver$ using incremental constraint solver $\mathcal{D}solver$
- Show efficiency of $\mathcal{AD}solver$ over classical ASP and $CR\text{-}Prolog$ solvers for planning with temporal constraints.
- Develop a tight algorithm $\mathcal{AC}solver$
- Prove correctness of $\mathcal{AC}solver$
- Implement $\mathcal{AC}solver$ using a CLP system