Integrating Answer Set Reasoning with Constraint Solving Techniques

Veena S. Mellarkod and Michael Gelfond
{veena.s.mellarkod, michael.gelfond}@ttu.edu

Texas Tech University

Abstract. The paper introduces a collection of knowledge representation languages, $\mathcal{V}(\mathcal{C})$, parametrised over a class $\mathcal{C}$ of constraints. $\mathcal{V}(\mathcal{C})$ is an extension of both CR-Prolog and CASP allowing the separation of a program into two parts: a regular program of CR-Prolog and a collection of denials[1] whose bodies contain constraints from $\mathcal{C}$ with variables ranging over large domains. We study an instance $\mathcal{AC}_0$ from this family where $\mathcal{C}$ is a collection of constraints of the form $X - Y > K$. We give brief implementation details of an algorithm computing the answer sets of programs of $\mathcal{AC}_0$ which does not ground constraint variables and tightly couples the "classical" ASP algorithm with an algorithm checking consistency of difference constraints. We present several examples to show the methodology of representing knowledge in $\mathcal{AC}_0$. The work makes it possible to solve problems which could not be solved by pure ASP or constraint solvers.

1 Introduction

Language CR-Prolog has been shown to be a useful tool for knowledge representation and reasoning [6]. The language is expressive, and has a well understood methodology inherited from Answer Set Prolog (ASP)[9], for representing defaults, causal properties of actions and fluents, various types of incompleteness, etc. In addition it allows reasoning with complex exceptions to defaults and hence avoids the occasional inconsistencies of ASP. CR-Prolog allows natural encoding of "rare events". These events are normally ignored by a reasoner associated with the program and only used to restore consistency of the reasoner's beliefs. For instance a program
$\neg p(X) \leftarrow not\ p(X).$
$q(a) \leftarrow \neg p(a).$
$[r(X)]\ :\ p(X) \stackrel{+}{\leftarrow}.$
consists of two regular rules of Answer Set Prolog and the consistency restoring rule, $[r(X)]$, which says that in some rare cases, $p(X)$ may be true. The rule is ignored in the construction of the answer set $\{\neg p(a), q(a)\}$ of this program. If however the program is expanded by $\neg q(a)$ the rule $r(a)$ will be used to avoid inconsistency. The answer set of the new program will be $\{p(a), \neg q(a)\}$.

[1] By a denial we mean a logic programming rule with an empty head.

CR-Prolog solvers built on top of the ASP solvers: Smodels [11] and Surya [10], proved to be sufficiently efficient for building industrial size applications related to intelligent planning and diagnostics [6]. Neither ASP nor CR-Prolog however, can deal with applications which require a combination of, say, planning and scheduling. This happens because scheduling normally requires programs which include variables with rather large numerical domains. ASP and CR-Prolog solvers compute answer sets of a ground instance of the input program. If a program contains variables with large domains such an instance can be too large, which renders the program unmanageable for the solver, despite the use of multiple optimization procedures.

A step toward resolving this problem was made in [7], where the authors introduced a language $CASP$. The algorithm for computing answer sets of $CASP$ programs only performs a partial grounding of variables and computes answer sets of the resulting, partially ground program by combining the classical ASP algorithm and a constrained solver for the constraints of $\mathcal{C}$. The CASP solver built loosely couples off-the-shelf ASP solver Smodels [11] and constraint solver GNU-Prolog [2].

In this paper we expand the idea to CR-Prolog. In particular, we introduce a collection, $\mathcal{V}(\mathcal{C})$, of languages parametrised over a class $\mathcal{C}$ of constraints. $\mathcal{V}(\mathcal{C})$ is an extension of both, CR-Prolog and CASP. We study an instance $\mathcal{AC}_0$ of the resulting language where $\mathcal{C}$ is a collection of constraints of the form $X - Y > K$. We design and implement an algorithm computing the answer sets of programs of $\mathcal{AC}_0$ which does not ground constraint variables and tightly couples the classical ASP algorithm with constraint solving mechanisms. To our knowledge the solver built is the first tightly coupled solver integrating ASP reasoning mechanisms and constraint solving techniques to compute answer sets from partially ground programs. This makes it possible to declaratively solve problems which could not be solved by pure ASP or by pure constraint solvers. The use of the language and the efficiency of its implementation is demonstrated by a number of examples. The paper is organized as follows: In section 2 we define the syntax and semantics of $\mathcal{V}(\mathcal{C})$ and $\mathcal{AC}_0$. Section 3 contains a brief description of the algorithm for computing answer sets of programs in $\mathcal{AC}_0$. Section 4 gives examples of knowledge representation and reasoning in $\mathcal{AC}_0$ and gives experimental results of on the use of $\mathcal{AC}_0$ for solving a sizable planning and scheduling problem related to the decision support system for the space shuttle controllers.

2 Syntax and Semantics of $\mathcal{V}(\mathcal{C})$

2.1 Syntax

The language $\mathcal{V}(\mathcal{C})$ contains a sorted signature Σ, with sorts partitioned into two classes: *regular*, s_r, and *constraint*, s_c. Intuitively, the former are comparatively small but the latter are too large for the ASP grounders. Functions defined on regular (constraint) classes are called r-functions (c-functions). Terms are built as in first-order languages. Predicate symbols are divided into three disjoint sets called *regular*, *constrained* and *mixed* and denoted by P_r, P_c and P_m respectively. Constraint predicate symbols

are determined by $\mathcal{C}$. Parameters of regular and constraint predicates are of sorts s_r and s_c respectively. Mixed predicates have parameters from both classes. Atoms are defined as usual. A literal is an atom a or its negation $\neg a$. An extended literal is a literal l or *not* l, where *not* stands for *default negation*. Atoms formed from regular, constraint, and mixed predicates are called r-atoms, c-atoms and m-atoms respectively. Similarly for literals. We assume that predicates of P_c have a predefined interpretation, represented by the set M_c of all true ground c-atoms. For instance, if $' >' \in P_c$, and ranges over integers, M_c consists of $\{...0 > -1, 1 > 0, 2 > 0, ..., 2 > 1, 3 > 1, ...\}$. The c-literals allowed in $\mathcal{V}(\mathcal{C})$ depend on the class $\mathcal{C}$. The $\mathcal{V}(\mathcal{C})$ rules over Σ are defined as follows.

Definition 1. [rules]

1. *A regular rule (r-rule) ρ is a statement of the form:*

$$h_1 \ or \ \cdots or \ \ h_k \leftarrow l_1, \ \cdots, l_m, \ not \ \ l_{m+1}, \cdots, not \ l_n$$

   where $k >= 0$; h_i's and l_i's are r-literals.
2. *A constraint rule (c-rule) is a statement of the form:*

$$\leftarrow l_1, \ \cdots, l_m, \ not \ \ l_{m+1}, \cdots, not \ l_n$$

   where at least one l_i is non-regular.
3. *A consistency restoring rule (cr-rule) is a statement of the form:*

$$r: \ h_1 \ or \ \cdots or \ \ h_k \stackrel{+}{\leftarrow} l_1, \ \cdots, l_m, \ not \ \ l_{m+1}, \cdots, not \ l_n$$

   where $k > 0$, r is a term which uniquely denotes the name of the rule and h_i's and l_i's are r-literals.

$head(r) = h_0 \ or \ \cdots or \ \ h_k$; $body(r) = \{l_1, \ \cdots, l_m, \ not \ \ l_{m+1}, \cdots, not \ l_n\}$; and $pos(r)$, $neg(r)$ denote, respectively, $\{l_1, \cdots, l_m\}$ and $\{l_{m+1}, \cdots, l_n\}$. A regular rule and constraint rule have the same intuitive reading as standard rules of ASP. The intuitive reading of a cr-rule is: *if one believes in $l_1, \ldots l_m$ and have no reason to believe $l_{m+1}, \ldots, l_n$, then one may possibly believe one of $h_1, \ldots, h_k$*. The implicit assumption is that this possibility is used as little as possible, and only to restore consistency of the agent's beliefs.

Definition 2. [program] *A $\mathcal{V}(\mathcal{C})$ program is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a sorted signature and Π is a set of $\mathcal{V}(\mathcal{C})$ rules over Σ.*

Example 1. To represent conditions: "John goes to work either by car which takes 30 to 40 minutes, or by bus which takes at least 60 minutes", we start by defining the signature $\Sigma = \{C_r = \{start, end\}, \ P_r = \{by\_car, by\_bus\}, \ C_c = \{D_c = [0..1439], R_c = [-1439..1439]\}, \ V_c = \{T_s, T_e\}, \ F_c = \{-\}, P_c = \{>\}, \ P_m = \{at\}\}$. The sets C_r and P_r contain regular constants and predicates; elements of C_c, V_c, F_c, and P_c are constrained constants, variables, functions and predicate symbols. P_m is the set of mixed predicates. Values in D_c represent time in minutes. Consider one whole day from 12:00am to 11:59pm mapped to 0 to 1439 minutes. Regular atom "*by_car*" says that "John travels by car"; mixed

atom $at(start, T)$ says that "John starts from home at time T". Similarly for "*by_bus*" and "$at(end, T)$". Function "$-$" has the domain D_c and range R_c; T_s, T_e are variables for D_c. The rules below represent the information from the story.
% 'John travels either by car or bus' is represented by an r-rule
$r_a:\ by\_car\ or\ \ by\_bus.$
% Travelling by car takes between 30 to 40 minutes. This information is encoded by two c-rules
$r_b:\ \leftarrow by\_car,\ at(start, T_s),\ at(end, T_e), T_e - T_s > 40.$
$r_c:\ \leftarrow by\_car,\ at(start, T_s),\ at(end, T_e),\ T_s - T_e > -30.$
% Travelling by bus takes at least 60 minutes
$r_d:\ \leftarrow by\_bus,\ at(start, T_s),\ at(end, T_e),\ T_s - T_e > -60.$

Example 2. Let us expand the story from example 1 by new information: *'John prefers to come to work before 9am'*. We add new constant $'time0'$ to C_r of Σ which denotes the start time of the day, regular atom 'late' which is true when John is late and constrained variable T_t for D_c. Time 9am in our representation is mapped to 540^{th} minute. We expand example 1 by the following rules:
% Unless John is late, he comes to work before 9am
$r_e:\ \leftarrow at(time0, T_t),\ at(end, T_e),\ \neg late,\ T_e - T_t > 540$
% Normally, John is not late
$r_f:\ \ \neg late\ \leftarrow\ not\ late$
% On some rare occasions he might be late, which is encoded by a cr-rule
$r_g:\ \ late \stackrel{+}{\leftarrow}$

In this paper, we study an instance $\mathcal{AC}_0$ of $\mathcal{V}(\mathcal{C})$, where $\mathcal{C}$ consists of constraints of type $X - Y > K$, where X, Y are variables and K is a number. Examples 1 and 2 are examples of $\mathcal{AC}_0$ programs.

2.2 Semantics

We denote the sets of r-rules, cr-rules and c-rules in Π by Π^r, Π^{cr} and Π^c respectively. A rule r of $\langle \Pi, \Sigma \rangle$ will be called *r-ground* if regular terms in r are ground. A program is called *r-ground* if all its rules are r-ground. A rule r^g is called a *ground instance* of a rule r if it is obtained from r by: (1). replacing variables by ground terms of respective sorts; (2). replacing the remaining terms by their values. For example, $3+4$ will be replaced by 7. The program $ground(\Pi)$ with all ground instances of all rules in Π is called the *ground instance* of Π. Obviously ground(Π) is an r-ground program.
We first define semantics for programs without cr-rules. For the definition, we use the term *asp answer set* to refer to the definition of answer sets in answer set prolog [9].

Definition 3. [answer set 1] *Given a program (Σ, Π), where Π contains no cr-rules, let X be a set of ground m-atoms such that for every predicate $p \in P_m$ and every ground r-term t_r, there is exactly one c-term t_c such that $p(\bar{t}_r, \bar{t}_c) \in X$. A set S of ground atoms over Σ is an answer set of Π if S is an asp answer set of $ground(\Pi) \cup X \cup M_c$.*

Example 3. Consider Example 1 and let $X = \{at(start, 430), at(end, 465)\}$. The set $S = \{by\_car, at(start, 430), at(end, 465)\} \cup M_c$ is an asp answer set of $ground(\Pi) \cup X \cup M_c$ and therefore is an answer set of Π. According to S, John starts to travel by car at 7:10am and reaches work at 7:45am. Of course there are other answer sets where John travels by car and his start and end times differ but satisfy given constraints. There are also answer sets where John travels by bus.

Now we give the semantics for programs with cr-rules. By $\alpha(r)$, we denote a regular rule obtained from a cr-rule r by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$; α is expanded in a standard way to a set R of cr-rules. Recall that according to [6], a minimal (with respect to set theoretic inclusion) collection R of cr-rules of Π such that $\Pi^r \cup \Pi^c \cup \alpha(R)$ is consistent (i.e. has an answer set) is called an *abductive support* of Π.

Definition 4. [answer set 2] *A set S is called an answer set of Π if it is an asp answer set of program $\Pi^r \cup \Pi^c \cup \alpha(R)$ for some abductive support R of Π.*

Example 4. Consider Example 2 and let $X = \{at(start, 430), at(end, 465)\}$. The set $S = \{by\_car, \neg late,\ at(time0, 0),\ at(start, 430),\ at(end, 465)\} \cup M_c$ is an answer set of $ground(\Pi) \cup X \cup M_c$ and therefore is an answer set of Π. According to S, John starts by car at 7:10am and reaches work at 7:45am and is not late. The cr-rule was not applied and $\alpha(\emptyset) = \emptyset$.

3 $\mathcal{AD}solver$

In this section we describe the algorithm which takes a program $\langle \Sigma, \Pi \rangle$ of $\mathcal{AC}_0$ as input and returns a simplified answer set $A \cup X$ (regular and mixed atoms) such that $M = A \cup X \cup M_c$ is an answer set of Π where M_c is the intended interpretation of c-predicates. The algorithm works for a class of $\mathcal{AC}_0$ programs satisfying the following syntax restrictions:

- *There are no disjunctions in the head of rules.*
- *Every c-rule of the program contains exactly one c-literal in the body.*

$\mathcal{AD}solver$ consists of a partial grounder $\mathcal{P}ground_d$ and an inference engine $\mathcal{AD}engine$. Given a $\mathcal{AC}_0$ program Π, $\mathcal{AD}solver$ first calls $\mathcal{P}ground_d$ to ground r-terms of Π, to get an r-ground program, $\mathcal{P}_d(\Pi)$. The $\mathcal{AD}engine$ combines constraint solving techniques with answer set reasoning and abduction techniques to compute simplified answer sets of $\mathcal{P}_d(\Pi)$.

3.1 $\mathcal{P}ground_d$

Given a $\mathcal{AC}_0$ program Π, $\mathcal{P}ground_d$ grounds the r-variables in Π and outputs a r-ground program $\mathcal{P}_d(\Pi)$. The implementation of $\mathcal{P}ground_d$ uses intelligent grounder $lparse$ [14]. To allow for partial grounding by $lparse$, we need intermediate transformations before and after grounding by $lparse$. The transformations ensure that c-variables are not ground and rules containing m-atoms are not removed by $lparse$. The transformations remove and store c-variables, m-atoms and c-atoms from Π before grounding and then restore them back after grounding.

Example 5. Let $a1$ and $a2$ be two actions. For representing the condition "a1 should occur 30 minutes before a2", we begin by defining a signature. $\Sigma = \{C_r = \{\{a1, a2\}, \{1, 2\}\}, V_r = \{S\}, P_r = \{o\}, P_m = \{at\}, C_c = \{D_c = \{0..1440\}, R_c = \{-1440..1440\}\}, V_c = \{T_1, T_2\}, F_c = \{-\}, P_c = \{>\}\}$ and Π be the following rules:

```
step(1..2).
% only one action can occur at each step
o(a1, S) :- step(S), not o(a2, S).
o(a2, S) :- step(S), not o(a1, S).
% an action can occur at most once
:- step(S1), step(S2), o(a1, S1), o(a1, S2), S1 != S2.
:- step(S1), step(S2), o(a2, S1), o(a2, S2), S1 != S2.
% define 'time' as a csort, and 'at' as a mixed predicate
#csort time(0..1440).
#mixed at(step,time).
% time should be increasingly assigned to steps
:- step(S1), step(S2), at(S1,T1), at(S2,T2), S1<S2, T1-T2 > 0.
% a1 should occur 30 minutes before a2
:- step(S1), step(S2), o(a1, S1), o(a2, S2),
   at(S1, T1), at(S2, T2), T1 - T2 > -30.
```

We get $\mathcal{P}_d(\Pi)$ as follows:

```
step(1).                          step(2).
o(a1, 1) :-  not o(a2, 1).        o(a1, 2) :-  not o(a2, 2).
o(a2, 1) :-  not o(a1, 1).        o(a2, 2) :-  not o(a1, 2).
:-  o(a1, 1), o(a1, 2).           :-  o(a2, 1), o(a2, 2).
#csort time(0..1440).
:- at(1, V1), at(2, V2), V1 - V2 > 0.
:- o(a1, 1), o(a2, 2), at(1, V1), at(2, V2), V1 - V2 > -30.
:- o(a1, 2), o(a2, 1), at(2, V2), at(1, V1), V2 - V1 > -30.
```

Note that $V1$ and $V2$ are constraint variables with domain [0..1440].

3.2 *$\mathcal{AD}$engine*

The *$\mathcal{AD}$engine* integrates a standard CR-Prolog solver and a difference constraint solver. CR-Prolog solver consists of a meta layer and computes answer sets by using an underlying ASP inference engine. For *$\mathcal{AD}$engine*, we use *Surya* [10] as the underlying inference engine.
Suppose there are no c-rules in a program Π, then Π is a CR-Prolog program. Typical CR-Prolog solvers available now, compute answer sets of Π as follows:

1. a meta-layer selects a minimal set R of cr-rules of Π called a candidate abductive support of Π;
2. an ASP inference engine is used to check program $\Pi^r \cup \alpha(R)$ for consistency and compute an answer set.
3. if an answer set is found at step (2) then R is an abductive support with respect to Π and the answer set computed is an answer set of Π and is returned[2]; otherwise the solver loops back to step(1) to find another minimal set R not tried so far.

[2] This algorithm is a simplification of the actual algorithm [5], which requires additional checking due to dynamic and special preference rules allowed in the language.

To compute answer sets of $\mathcal{AC}_0$ programs, we modify the solver to accept c-rules; and then change step(2) of the algorithm. Given a $\mathcal{AC}_0$ program Π, we modify the underlying inference engine Surya to compute answer sets of $\Pi^r \cup \Pi^c \cup \alpha(R)$. Note that the program $\Pi^r \cup \Pi^c \cup \alpha(R)$ does not contain cr-rules but only r-rules and c-rules.

$\mathcal{AD}$*engine* integrates a form of abductive reasoning using the meta-layer with answer set reasoning and constraint solving of the underlying inference engine. Surya has been modified to tightly couple with a difference constraint solver (for constraint solving). The algorithm presented in [7] uses constraint solving techniques for checking consistency of constraints with respect to a partial model computed. Our algorithm uses constraint solving techniques for checking consistency and for computing consequences with respect to a given program and a partial model computed. The solver implemented for constraint solving is an incremental difference constraint solver that computes solutions of a set of constraints (constraint store) using a previous solution and changes to the constraint store. This method is more efficient than computing solutions from scratch.

To our knowledge this is the first tightly coupled solver for integrating answer set reasoning and constraint solving to compute answer sets from partially ground programs. In the next section, we show that the solver can efficiently compute answer sets for a large complex system and it makes it possible to solve problems which could not be solved by pure ASP, CR-Prolog or constraint solvers. $\mathcal{AD}$*solver* is available at [1].

4 Representing Knowledge in $\mathcal{AC}_0$

$\mathcal{AC}_0$ is good for representing planning and scheduling problems. Given a task of executing n actions and time restrictions on their executions, a scheduling problem consists of finding times $T_1, \ldots, T_n$ such that action 'a_i occurs at time T_i' and satisfies all the time restrictions. The timing restrictions can be temporal distance constraints between any two actions. Such constraints can be represented in $\mathcal{AC}_0$ as follows. Let $a_1, \ldots, a_n$ be n actions and $S_1, \ldots, S_n$ be variables in domain $[1..n]$. The r-atom $occurs(a_i, S_i)$ is read as, "action a_i occurs at step S_i". The step S_i is a number and denotes a time point T_i and is represented by an m-atom $at(S_i, T_i)$. Atom $at(S, T)$ is read as 'step S occurs at time T'. The domain of a step S (r-variable) is comparatively smaller to domain of a time variable T (c-variable).

When actions have durations, the scheduling problem finds the start and end time points of actions such that all timing restrictions are satisfied. One method of representing constraints on action durations in $\mathcal{AC}_0$ is as follows. Let $a_1, \ldots, a_n$ be actions and $S_1, \ldots, S_n$ be variables from the domain [1..n]. The r-atom $occurs(a_i, S_i)$ is read as, "action a_i occurs at step S_i". The variable S_i is a number which denotes a time interval $[T_{si}, T_{ei}]$. The time interval of step S_i is represented by two m-atoms $at(S_i, start, T_{si})$ and $at(S_i, end, T_{ei})$. Atom $at(S, start, T)$ is read as 'step S starts at T'. We can write temporal constraints using the c-variables T_{si} and T_{ei}.

Example 6. [12] **[Breakfast problem]** We have a scheduling problem, "Prepare coffee and toast. Have them ready within 2 minutes of each other. Brew coffee for 3-5 minutes; toast bread for 2-4 minutes." We start by defining signature, $\Sigma = \{C_r = \{\ start, end, brew, toast,\ S_c = [1..2]\ \}, P_r = \{step, occurs\}, C_c = \{D_c = [0..1439], R_c = [-1439..1439]\}, V_c = \{T_1, T_2\}, F_c = \{-\}, P_c = \{>\}, P_m = \{at\}\}$. Constants "$brew$, $toast$" represents actions 'brewing coffee' and 'toasting bread'. To solve this, we first represent constraints and then we have a small planning module to represent action a_i occurs at some time step S_i. The constraints are as follows.

% Brew coffee for 3 to 5 minutes is represented using two c-rules

$\leftarrow\ occurs(brew, S), at(S, start, T_1), at(S, end, T_2), T_2 - T_1 > 5.$

$\leftarrow\ occurs(brew, S), at(S, start, T_1), at(S, end, T_2), T_1 - T_2 > -3.$

% Toast bread for 2 to 4 minutes is represented by two c-rules

$\leftarrow\ occurs(toast, S), at(S, start, T_1), at(S, end, T_2), T_2 - T_1 > 4.$

$\leftarrow\ occurs(toast, S), at(S, start, T_1), at(S, end, T_2), T_1 - T_2 > -2.$

% Coffee and bread should be ready between 2 minutes of each other

$\leftarrow\ occurs(brew, S_1), occurs(toast, S_2), at(S_1, end, T_1),$
$at(S_2, end, T_2), T_2 - T_1 > 2.$

$\leftarrow\ occurs(brew, S_1), occurs(toast, S_2), at(S_1, end, T_1),$
$at(S_2, end, T_2), T_1 - T_2 > -2.$

% Start time of step 1 is before step 2

$\leftarrow\ at(S_1, start, T_1), at(S_2, start, T_2), S_1 < S_2, T_1 - T_2 > -1.$

% A simple planning module to represent occurrence of actions:

$step(1..2).$

$occurs(brew, S)\ or\ occurs(toast, S)\ \leftarrow\ step(S).$

$\leftarrow\ action(A),\ occurs(A, S_1),\ occurs(A, S_2),\ S_1 \neq S_2.$

The first c-rule is read as: 'If brewing coffee occurs at step S, then duration between start and end of S cannot be more than 5 minutes'. The second c-rule says that 'start and end times for S cannot be less than 3 minutes. The c-atom is written as $T_1 - T_2 > -3$ instead of $T_2 - T_1 < 3$ as the implementation allows only constraints of the form $X - Y > K$. The disjunctions in the head of the rules of the above program can be eliminated using non-disjunctive rules. A solution to the above breakfast scheduling problem can be found by computing answer sets of the program using $\mathcal{AD}solver$. A solution would be to start brewing coffee at 0th minute and end at 3rd minute; start toasting bread at 2nd minute and end at 4th minute. This solution can be extracted from an answer set $\{occurs(brew, 1),\ occurs(toast, 2),\ at(1, start, 0),\ at(1, end, 3),\ at(2, start, 2),\ at(2, end, 4)\} \cup M_c$.

Suppose we would like to schedule an action a such that it occurs either between 3am and 5am or between 7am and 8am. To represent this restriction, we would require a constraint of the form, *if action 'a' occurs at step S and step S occurs at time T, then T cannot be outside intervals [3-5] or [7-8].* We cannot represent this directly in $\mathcal{AC}_0$. Instead we introduce two r-atoms int_1 and int_2 to represent intervals [3-5] and [7-8] respectively. The r-atom int_1 denotes that action a occurs in interval [3-5]. We write a disjunction on int_i to choose the interval and then use int_i to write the constraints. The following example shows the representation of the constraint.

Example 7. "action a should be performed in between intervals [3-5] am or [7-8] am". Let r-atoms int_1 and int_2 represent intervals [3-5] and [7-8] respectively and int_i is true when action a occurs in interval int_i. To keep it simple, let us suppose that action a occurs at some step say 1. We need to assign time for this step. Atom $at(0, T)$ denotes time of step 0 and represents start time for our problem 12 am.

$occurs(a, 1)$.

% action 'a' occurs in interval int_1 or int_2

int_1 *or* int_2.

% 'If a occurs at step S and int_1 is true, then S should be between [3-5]', is encoded using two c-rules

$\leftarrow\ int_1, occurs(a, S), at(0, T_1), at(S, T_2), T_1 - T_2 > -3$

$\leftarrow\ int_1, occurs(a, S), at(0, T_1), at(S, T_2), T_2 - T_1 > 5$

% 'If a occurs at step S and int_2 is true, then S should be between [7-8]', is encoded using two c-rules

$\leftarrow\ int_2, occurs(a, S), at(0, T_1), at(S, T_2), T_1 - T_2 > -7$

$\leftarrow\ int_2, occurs(a, S), at(0, T_1), at(S, T_2), T_2 - T_1 > 8$

An answer set for this program would be $\{occurs(a, 1),\ int_2,\ at(0, 0),\ at(1, 7)\} \cup M_c$, where a occurs at 7 am. The following example is from [8], we show that we can represent the problem and answer some of the questions asked in the example. Though, syntax of $\mathcal{AC}_0$ does not allow choice rules and cardinality constraints [11], $\mathcal{AD}solver$ built on top of *lparse* and *Surya* allows these type of rules in its input language. We use choice rules in the following example.

Example 8. [8] **[Carpool]** John goes to work either by car (30-40 mins), or by bus (at least 60 mins). Fred goes to work either by car (20-30 mins), or in a car pool (40-50 mins). Today John left home between 7:10 and 7:20, and Fred arrived between 8:00 and 8:10. We also know that John arrived at work about 10-20 mins after Fred left home. We wish to answer queries such as: "Is the information in the story consistent?","Is it possible that John took the bus, and Fred used the carpool?", "What are the possible times at which Fred left home?".

```
%% John goes to work either by car or by bus. (a choice rule)
1{ j_by_car, j_by_bus }1.
%% Fred goes to work either in car or by car pool
1{ f_by_car, f_by_cpool }1.

%% define 'time' as csort and 'at' as a mixed predicate
#csort time(0..1440).
timepoint(start_time; start_john; end_john; start_fred; end_fred).
#mixed at(timepoint, time).

%% "It takes John 30 to 40 minutes by car"
:- j_by_car, at(start_john,T1), at(end_john,T2), T2 - T1 > 40.
:- j_by_car, at(start_john,T1), at(end_john,T2), T1 - T2 >-30.

%% "It takes John atleast 60 minutes by bus"
:- j_by_bus, at(start_john,T1), at(end_john,T2), T1 - T2 >-60.
```

```
%% We view the start time as 7am, that is 0 minutes = 7am
%% Today John left home between 7:10 and 7:20
:- at(start_john,T), at(start_time,T0), T0 - T > -10.
:- at(start_john,T), at(start_time,T0), T - T0 > 20.
```

The other informations in the example are represented by similar c-rules. Now let us look at answering each of the questions in the problem.

```
Question (1) Is the information in story consistent?
To answer this question, we find answer sets of the program.
Answer Set: j_by_car f_by_cpool at(start_fred,20) at(end_fred,60)
            at(start_time,0) at(start_john,10) at(end_john,40)
```

The above answer set corresponds to John using the car and Fred using the car pool. John starts at 7:10 am and reaches at 7:40 am. Fred starts at 7:20 am and reaches at 8:00 am. The information is consistent since the program has an answer set. The time taken by $\mathcal{AD}solver$ to find an answer set was 0.065 secs of which 0.018 secs was used by $\mathcal{P}ground_d$.

```
Question(2) Is it possible that John took the bus and Fred
used carpool? To answer this question, we add the following
knowledge to our program and compute answer sets.
j_by_bus.
f_by_cpool.
There are no answer sets for this new program.
```

Therefore, according to the story, it is not possible for John to take a bus and Fred to use a carpool and have the story consistent. The time taken by $\mathcal{AD}solver$ was 0.029 secs of which $\mathcal{P}ground_d$ took 0.018 secs.

```
Question (3) What are the possible times that Fred left home?
To answer this question, we need to find the interval of time
when Fred can leave home and still have the story consistent.
```

This answer cannot be found using $\mathcal{AD}solver$, as the underlying constraint solver built cannot answer these type of interval questions.

The temporal constraints from the above problems are examples of simple and disjunctive temporal constraints [8]. Using cr-rules in $\mathcal{AC}_0$ we can represent important information like, "an event e may happen but it is very rare". Such information is very useful in default reasoning. Combining such information together with c-rules allows us to represent qualitative soft constraints [13] like, "an event e may happen but it is very rare; if event e happens then ignore constraint c". The following example is an extension of Example 8 and shows the representation of qualitative soft temporal constraints.

Example 9. Consider example 8, we remove information that "John arrived at work about 10-20 mins after Fred left home" and extend the story as follows: It is desirable for Fred to arrive atleast 20 mins before John.

```
%% Fred desires to arrive atleast 20 mins before John.
:- at(end_fred,T1), at(end_john,T2), not is_late, T1-T2 >-20.
%% CR-rule r1: We may possibly believe that Fred is late
r1: is_late +-.
```

For the newly added information, we get two models where Fred arrives before John in each of them.

```
Answer set(1):j_by_bus f_by_car at(end_john,100) at(end_fred,60)
            at(start_time,0) at(start_john,20) at(start_fred,30)
Answer set(2):j_by_bus f_by_cpool at(end_john,80) at(end_fred,60)
             at(start_time,0) at(start_john,20) at(start_fred,20)
```

To compute the two models, $\mathcal{AD}solver$ took 0.064 seconds of which 0.019 seconds were used for grounding and loading. Now we would like to expand our story, "We come to know that Fred's car is broken and therefore, he cannot use it". We add the following rule to the program.

```
:- f_by_car.
```

For the new program, we get one model where John travels by bus and Fred uses the carpool and still reaches before John.

```
Answer set:j_by_bus f_by_cpool at(end_john,80) at(end_fred,60)
          at(start_time,0) at(start_john,20) at(start_fred,20)
```

$\mathcal{AD}solver$ took 0.053 seconds to compute the model. Suppose we know that John used his car today. Will Fred arrive atleast 20 mins before John as desired? For this, we add the following rule to the program.

```
j_by_car.
```

There is no model where Fred arrives 20 minutes before John and the cr-rule was fired to give the following answer set.

```
Answer set: j_by_car f_by_cpool is_late at(start_fred,20)
at(end_fred,60) at(start_time,0) at(start_john,20) at(end_john,60)
```

Fred is late and cannot arrive 20 minutes before John as desired. $\mathcal{AD}solver$ took 0.052 seconds to compute the model.

The examples show that $\mathcal{AC}_0$ allows a natural representation of simple temporal constraints, disjunctive temporal constraints and qualitative soft constraints. The implemented solver is faster than a standard ASP solver when domains of constraint variables are large. The language of CR-Prolog also allows preferences on the cr-rules [4]. Given two cr-rules r_1 and r_2, the statement $prefer(r_1, r_2)$ allows preference to cr-rule r_1 when compared to cr-rule r_2. CR-Prolog allows static and dynamic preferences. The language $\mathcal{AC}_0$ does not allow preferences but $\mathcal{AC}_0$ syntax can be easily extended to allow CR-Prolog style preferences and the semantics would be a natural extension of CR-Prolog. Though language $\mathcal{AC}_0$ does not allow preferences, the solver $\mathcal{AD}solver$ which is built using the meta layer of CR-Prolog solver, allows preferences. So, we can express soft qualitative temporal constraints with preferences which is used in constraint programming [13].

Example 10. This example shows the representation of qualitative soft temporal constraints with preferences. Let us use example 9. We remove information that "John arrived at work about 10-20 mins after Fred left home and Fred arrived between 8:00 and 8:10" and extend the story as follows: It is desirable for Fred to arrive atleast 20 mins before John. If possible, Fred desires to start from home after 7:30am. We also know that Fred's car is broken and John used his car today.

```
%% If possible, Fred desires to leave after 7:30am
:- at(start_time,T1), at(start_fred,T2), not start_early,
   T1 - T2 > -30.
%% CR-rule r2: sometimes, Fred may need to start early.
r2: start_early +-.
```

The above rules along with other rules from examples 9 and 8 represent the information in the story. We get two answer sets where cr-rules were used in both.

```
Answer set (1):    j_by_car f_by_cpool is_late
            at(start_john,20) at(start_fred,30)
            at(end_john,60) at(end_fred,70) at(start_time,0)
Answer set (2): j_by_car f_by_cpool start_early
            at(start_john,20) at(start_fred,0)
            at(end_john,60) at(end_fred,40) at(start_time,0)
```

Now we add new preference information that "Fred prefers coming before John than starting late from home". we represent the preference as follows:

```
% Prefer starting early to reaching late
prefer(r2,r1).
```

Now, we get only one model:

```
Answer set: j_by_car f_by_cpool start_early
            at(start_john,20) at(end_john,60)
            at(start_fred,0) at(end_fred,40) at(start_time,0)
```

The other model is not preferred when compared to this one and therefore is not returned. $\mathcal{AD}solver$ computed the answer set in 0.13 seconds.

The above example clearly shows the use of preferences from CR-Prolog along with c-rules gives a natural representation of qualitative soft constraints with preferences. Similarly, we can use cr-rules, cr-preferences and c-rules together to represent disjunctive soft temporal constraints and disjunctive soft temporal constraints with preferences which are also useful for scheduling problems.
Another investigation we are concerned with is whether $\mathcal{AC}_0$ can be used for complex planning and scheduling problems. Also, whether we can use $\mathcal{AD}solver$ to compute answer sets in realistic time for these problems. To test this, we have used the system USA-Advisor[6] , a decision support system for the Reaction Control System (RCS) of the Space Shuttle.
The RCS has primary responsibility for maneuvering the aircraft while it is in space. It consists of fuel and oxidizer tanks, valves and other

plumbing needed to provide propellant to the maneuvering jets of the shuttle. It also includes electronic circuitry: both to control the valves in the fuel lines and to prepare the jets to receive firing commands. Overall the system is rather complex, on that it includes 12 tanks, 44 jets, 66 valves, 33 switches, and around 160 computer commands (computer-generated signals). The RCS can be viewed, in a simplified form, as a directed graph whose nodes are tanks, jets and pipe junctions, and whose arcs are labeled by valves. For a jet to be ready to fire, oxidizer and fuel propellants need to flow through the nodes (tanks, junctions) and valves which are open and reach the jet. A node is pressurized when fuel or oxidizer reaches the node.
The system can be used for checking plans, planning and diagnosis. To test our solver, we have expanded the system to allow explicit representation of time to perform some scheduling. We use it to solve planning and scheduling tasks. We will illustrate our extension by the following example.

Example 11. **[Planning and scheduling in USA-Advisor]** Assume that after a node N gets pressurized it takes around 5 seconds for the oxidizer propellant to get stabilized at N and 10 seconds for fuel propellant to get stabilized. Further, we cannot open a valve V which links N1 to N2, (link(N1,N2,V)), until N1 has been stabilized. We would like to assign real times to the time steps given in the program such that this constraint is satisfied. Also, can we answer questions like: can the whole manuver take less than 30 secs?
$\Sigma = \Sigma_{old} \cup \{P_r = \{otank, ftank, got\_opened, got\_pressurized\},\ P_m = \{at\}, C_c = \{D_c = [0..400], R_c = [-400..400]\}, F_c = \{-\}, P_c = \{>\}\}$. Atoms $otank(X)$ and $ftank(X)$ denote that X is a oxidizer tank and fuel tank respectively. Fluent $got\_opened(V, S)$ is true when valve V was closed at step $S-1$ and got opened at step S. Fluent $got\_pressurized(N, X, S)$ is true when node N is not pressurized at step $S-1$ and is pressurized at step S by tank X. Atom $at(S, T)$ is read as 'step S is performed at time T', where S is a regular variable with domain 0 to plan length; T is a constraint variable with domain [0..400] seconds. The new program contains all rules from original advisor, and new rules describing the scheduling constraints. The first rule is from USA-Advisor, followed by some new rules. The second rule shows the connection between original program and new one.
% Tank node N_1 is pressurized by tank X if it is connected by an open valve to a node which is pressurized by tank X of sub-system R

$$h(pressurized\_by(N_1, X), S) \leftarrow step(S), tank\_of(N_1, R), h(in\_state(V, open), S),\ link(N_2, N_1, V), tank\_of(X, R),\ h(pressurized\_by(N_2, X), S).$$

% node gets pressurized when it was not pressurized at S and pressurized at S+1.

$$got\_pressurized(N, X, S+1) \leftarrow link(N_1, N, V),\ tank\_of(X, R), not\ h(pressurized\_by(N, X), S), h(pressurized\_by(N, X), S+1).$$

% A valve V linking N_1 to N_2 cannot be opened unless N_1 is stabilized.
% If N_1 is pressurized by oxidizer tank, N_1 takes 5 seconds to stabilize.

$\leftarrow$ $link(N_1, N_2, V), got\_pressurized(N_1, X, S_1), S_1 < S_2, otank(X),$
$got\_opened(V, S_2), at(S_1, T_1), at(S_2, T_2), T_1 - T_2 > -5$
% If N_1 is pressurized by fuel tank, N_1 takes 10 seconds to stabilize.
$\leftarrow$ $link(N_1, N_2, V), got\_pressurized(N_1, X, S_1), S_1 < S_2, ftank(X),$
$got\_opened(V, S_2), at(S_1, T_1), at(S_2, T_2), T_1 - T_2 > -5$
% time should be increasingly assigned to steps
$\leftarrow$ $S_1 < S_2, at(S_1, T_1), at(S_2, T_2), T_1 - T_2 > -1$
% The jets of a system should be ready to fire by 30 seconds
$\leftarrow$ $system(R), goal(S, R), at(0, T_1), at(S, T_2), T_2 - T_1 > 30$

$\mathcal{AD}solver$ was tested using USA-Advisor extension example 11. We tested the solver on 450 auto-generated instances. The files used were "rcs1, plan, heuristics, problem-base" [3], an instance file and scheduling constraints file (see example 11). The files and instances can be found at [3]. Due to space limitations, timing results of only 300 instances are shown in Figure 1. The instances of the left (right) figure are the first 50 instances from folder 'instances/instances-auto /ins' (instances/instances-auto /ins-4). Each instance is run to compute answer sets to find plans of length n=3, n=4 and n=5. The timing results shown is the time taken for $\mathcal{AD}solver$ to compute a single answer set or return $false$ to denote no plan for the specified plan length (n) exists.

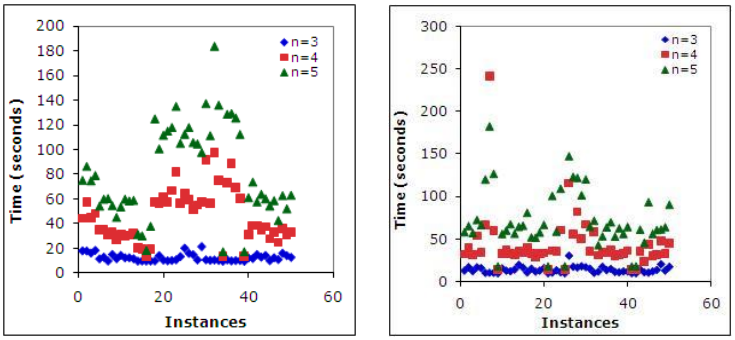

Fig. 1. $\mathcal{AD}solver$ Timing Results on Planning and Scheduling in USA-Advisor

The results show that $\mathcal{AD}solver$ could compute answer sets for most of the instances tried in less than two minutes. There was one instance not shown in the figure (from ins-4, n=4) that took around 3359 seconds to find that there was no plan, this was the only instance that took so long. The number of rules (partially ground) for instances with n=3 was approximately 95,000 rules. The domain of time variables was 0..400 seconds. The USA-Advisor example 11 can be transformed to a regular ASP program. ASP solvers [10, 11, 6] were not able to compute answer sets, the grounder *lparse* they use returned a malloc error because of huge memory requirements.

5 Conclusions

This paper introduces a collection $\mathcal{V}(\mathcal{C})$ of languages parameterized over a class $\mathcal{C}$ of constraints. We study an instance $\mathcal{AC}_0$ of the resulting language where $\mathcal{C}$ is a collection of constraints of the form $X - Y > k$. We design and implement an algorithm for computing the answer sets of a class of $\mathcal{AC}_0$ programs. The algorithm computes answer sets from partial ground programs and tightly couples answer set reasoning mechanisms with constraint solving techniques. This makes it possible to declaratively solve problems which could not be solved by pure ASP or by pure constraint solvers. The use of the language and efficiency of the solver is demonstrated.

References

1. Adsolver. http://www.cs.ttu.edu/∼mellarko/adsolver.html.
2. GNU Prolog. http://www.gprolog.org.
3. Rcs. http://www.krlab.cs.ttu.edu/Software/Download/rcs/.
4. M. Balduccini. *Answer Set Based Design of Highly Autonomous, Rational Agents.* PhD thesis, Texas Tech University, Dec 2005.
5. M. Balduccini. CR-models: An inference engine for CR-prolog. In *Logic Programming and Nonmonotonic Reasoning*, May 2007.
6. M. Balduccini, M. Gelfond, and M. Nogueira. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, 2006.
7. S. Baselice, P. A. Bonatti, and Michael Gelfond. Towards an integration of answer set and constraint solving. In *In Proceedings of ICLP*, pages 52–66, 2005.
8. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.
9. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.
10. Veena S. Mellarkod. Optimizing the computation of stable models using merged rules. Master's thesis, Texas Tech University, May 2002.
11. Ilkka Niemela and Patrik Simons. *Extending the Smodels System with Cardinality and Weight Constraints*, pages 491–521. Logic-Based Artificial Intelligence. Kluwer Academic Publishers, 2000.
12. Martha E. Pollack and Nicola Muscettola. Temporal and resource reasoning for planning, scheduling and execution. *Tutorial Forum Notes, AAAI06*, Jul 2006.
13. F. Rossi, A. Sperduti, K. Venable, L. Khatib, P. Morris, and R. Morris. Learning and solving soft temporal constraints: An experimental study, 2002.
14. Tommi Syrjanen. Implementation of logical grounding for logic programs with stable model semantics. Technical Report 18, Digital Systems Laboratory, Helsinki University of Technology, 1998.