# Systems integrating answer set programming and constraint programming

Michael Gelfond and Veena S. Mellarkod and Yuanlin Zhang
{mgelfond,veena.s.mellarkod,yzhang}@cs.ttu.edu

Texas Tech University, USA

**Abstract.** We will demonstrate two systems implementing subclasses of a new language $\mathcal{AC}(\mathcal{C})$. The language $\mathcal{AC}(\mathcal{C})$ not only has the problem modeling power of both Answer Set Prolog (ASP) and Constraint Logic Programming (CLP) but also allows for more efficient inference algorithms combining reasoning techniques from ASP and CLP systems.

## 1 Introduction

Logic programming with answer sets semantics [6], also called Answer Set Prolog (ASP), has proved useful in finding solutions to a variety of programming tasks, ranging from building decision support systems for the Space Shuttle [3] and program configuration [10], to solving problems arising in bio-informatics [4], zoology and linguistics [5]. Though positive, this experience allows to identify a number of problems and inadequacies of the ASP approach to declarative programming. One of them is that for a number of tasks, the existing ASP solvers are not sufficiently efficient. This becomes immediately obvious if the program contains variables ranging over large domains. Even though ASP solvers use intelligent grounding optimization techniques, ground instantiation of such a program can still be huge, which can cause both memory and time problems and make ASP solvers practically useless. When facing variables with large domains, Constraint Logic Programming (CLP) [11, 8] can not only avoid the grounding problems but also provide effective ways (e.g., through techniques developed in Constraint Satisfaction Problems and Linear Programming) to handle the constraints effectively. A natural way is to combine ASP with CLP, which will allow one to benefit from both paradigms. Clearly the combination greatly improves the knowledge representation capacity of CLP thanks to the non-monotonicity of ASP. We first introduce a language $\mathcal{AC}(\mathcal{C})$ with features from both ASP and CLP and then present systems implementing subclasses of this language.

## 2 The language of $\mathcal{AC}(\mathcal{C})$

The new language $\mathcal{AC}(\mathcal{C})$ is parameterized by a constraint domain $\mathcal{C}$. Informally, $\mathcal{C}$ specifies the primitive constraints that can be used in $\mathcal{AC}(\mathcal{C})$. To define the language, we need some terminology. By *sort* we mean a non-empty countable collection of strings over some fixed alphabet. Strings of sort $S_i$ will be referred to as *object constants* of $S_i$. A *sorted signature*, $\Sigma$, is a collection of sorts, properly typed predicate and function symbols, and variables. Each variable, $X$, takes on values from a unique sort denoted by $sort(X)$. When needed we assume that $\Sigma$ contains standard numerical sorts of natural numbers, integers, rational numbers, etc. as well as standard numerical functions and relations such as $+$, $-$, $>$, $<$, etc. *Terms*, *literals*, and *extended literals* of $\Sigma$ are defined as usual.

In $\mathcal{AC}(\mathcal{C})$, the sorts of $\Sigma$ are divided into *regular* and *constraint.* Constraint sorts will be declared by an expression *#csort*. Intuitively a sort is declared to be a constraint sort if it is a large (often numerical) set with primitive constraint relations from $\mathcal{C}$ (e.g., $\leq$) defined between its elements. Grounding *constraint variables*, i.e., variables ranging over constraint sorts, would normally lead to huge grounded program. This is exactly what should be avoided by the $\mathcal{AC}(\mathcal{C})$ solvers. The $\mathcal{AC}(\mathcal{C})$ solvers will only ground variables ranging over regular sorts (*regular variables*).

Predicates of the language are divided into four types: regular, constraint, defined and mixed. *Regular* predicates denote relations among objects of regular sorts; *constraint* predicates denote primitive numerical relations among objects of constraint sorts; *defined* predicates are defined in terms of constraint, regular, and defined predicates; *mixed* predicates denote relations between objects which belong to regular sorts and those which belong to constraint ones. Mixed predicates are *not* defined by the rules of the program and are similar to abducible relations of abductive logic programming.

**Definition 1.** [*Syntax of* $\mathcal{AC}(\mathcal{C})$]
A *standard* $\mathcal{AC}(\mathcal{C})$ *rule* over signature $\Sigma$ is a statement of the form:

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

such that

- if $k > 1$ then $h_1, \ldots, h_k$ are regular literals;
- if $k = 1$ then $h_1$ is a regular or defined literal;
- $l_1, \ldots, l_n$ are arbitrary literals of $\Sigma$.

A *consistency restoring rule* (cr-rule) of $\mathcal{AC}(\mathcal{C})$ is a statement of the form:

$$r : l_0 \overset{+}{\leftarrow} l_1,\ \ldots, l_m,\ not\ \ l_{m+1}, \ldots, not\ l_n \tag{2}$$

where $r$ is a term which uniquely denotes the name of the rule and $l_i$'s are regular literals.

An $\mathcal{AC}(\mathcal{C})$ *program* $\Pi$ consists of definitions of sorts of a signature $\Sigma$, declarations of variables - statements of the form $sort(V_1, \ldots, V_k) = sort\_name$, and a collection of standard and consistency restoring $\mathcal{AC}(\mathcal{C})$ rules over $\Sigma$.

Classification of literals of the signature $\Sigma$ of an $\mathcal{AC}(\mathcal{C})$ program $\Pi$ allows partitioning $\Pi$ into three parts:

– *Regular part*, $\Pi_r$, consisting of rules built from regular literals,
– *Defined part*, $\Pi_d$, consisting of rules whose heads are defined literals,
– *Middle part*, $\Pi_m$, consisting of all other rules of $\Pi$.

Intuitively, the regular part corresponds to CR-prolog programs, an extension of ASP programs [2, 1], the defined part corresponds to CLP programs, while the middle part bridges the regular and defined parts.

Elements of $\Pi_r, \Pi_d$ and $\Pi_m$ are called *regular rules*, *defined rules*, and *middle rules* respectively. Note that a standard (ground) $ASP$/CR-Prolog program $\Pi$ can also be viewed as an $\mathcal{AC}(\mathcal{C})$ program in which all the predicates are defined as regular.

We next present the semantics of $\mathcal{AC}(\mathcal{C})$. Let $R$ be a rule of an $\mathcal{AC}(\mathcal{C})$ program $\Pi$ with signature $\Sigma$. A *ground instance* of $R$ is obtained from $R$ by:

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

An $ASP$ program $ground(\Pi)$ consisting of all ground instances of all rules in $\Pi$ is called the *ground instantiation* of $\Pi$.

A consistent set $S$ of ground literals over the signature $\Sigma$ is called a *partial interpretation* of an $\mathcal{AC}(\mathcal{C})$ program $\Pi$ if it satisfies the following conditions:

1. A constraint literal $l \in S$ iff $l$ is true under the intended interpretation of its symbols;
2. For every mixed predicate $m(\bar{X}_r, \bar{Y}_c)$ and every ground instantiation $\bar{t}_r$ of $\bar{X}_r$, there is a unique ground instantiation $\bar{t}_c$ of $\bar{X}_c$ such that $m(\bar{t}_r, \bar{t}_c) \in S$.

We first define semantics for $\mathcal{AC}(\mathcal{C})$ programs without cr-rules.

**Definition 2.** [*Answer sets of $\mathcal{AC}(\mathcal{C})$ programs without cr-rules*]
A partial interpretation $S$ of the signature $\Sigma$ of an $\mathcal{AC}(\mathcal{C})$ program $\Pi$ is called an *answer set* of $\Pi$ if there is a set $M$ of ground mixed literals of $\Sigma$ such that $S$ is an answer set of the $ASP$ program $ground(\Pi) \cup M$.

By $stand(\Pi)$ we denote the collection of standard rules of $\Pi$. By $\alpha(r)$, we denote a standard rule obtained from a cr-rule $r$ by replacing $\overset{+}{\leftarrow}$ by $\leftarrow$; For a set $\mathcal{R}$ of cr-rules, $\alpha(\mathcal{R}) = \{\alpha(r) : r \in \mathcal{R}\}$. A minimal (with respect to set theoretic inclusion) collection $\mathcal{R}$ of cr-rules of $\Pi$ such that $stand(\Pi) \cup \alpha(\mathcal{R})$ is consistent (i.e., has an answer set) is called an *abductive support* of $\Pi$.

**Definition 3.** [*Answer sets of arbitrary $\mathcal{AC}(\mathcal{C})$ programs*]
A set $S$ is called an *answer set* of $\mathcal{AC}(\mathcal{C})$ program $\Pi$ if it is an answer set of program $stand(\Pi) \cup \alpha(\mathcal{R})$ for some abductive support $\mathcal{R}$ of $\Pi$.

We use the following example to illustrate the concepts in the new language.

*Example 1.* [$\mathcal{AC}(\mathcal{C})$ programs and their answer sets]
Consider a domain with an action $a$ and a fluent $f$. The action can be executed only at time between 0:10am and 0:20am or between 2:00am and 3:00am.

We use a regular sort $step = \{0, 1\}$ to denote steps of a trajectory of the acting agent, and a constraint sort $time = \{0..1000\}$ to denote the actual time (say in minutes). The relation $at(S, T)$, holds iff step $S$ of the trajectory is executed at time $T$. It is a typical example of mixed relation. The defined relation $acceptable(T)$ holds iff action $a$ can be executed at time $T$.

A program P with the corresponding sorts, variables, mixed relation and defined relation is given below.

% sorts

$time = \{0..1000\}$.
$step = \{0..1\}$.
$action = \{a\}$.
$fluent = \{f\}$.

% variable declarations

$sort(T) = time$.
$sort(S, S') = step$.

% constraint relation $\leq$ defined on *time*, and declarations

$\#csort(time)$.
$\#defined\ acceptable\_time(time)$.
$\#mixed\ at(step, time)$.
$\#regular\ occurs(action, step)$.
$\#regular\ holds(fluent, step)$.
$\#regular\ next(step, step)$.

(To make our program executable with our implementations, we should use the *lparse* notation $time(0..1000)$, $step(0..1)$, $action(a)$, and $fluent(f)$ for sorts, and $\#domain\ time(T)$ and $\#domain\ step(S, S')$ for variable declarations.)

$acceptable\_time(T) \leftarrow 10 \leq T \leq 20$.
$acceptable\_time(T) \leftarrow 120 \leq T \leq 180$.
$\neg occurs(A, S) \leftarrow at(S, T),\ not\ acceptable\_time(T)$.
$next(1, 0)$.
$holds(f, S') \leftarrow occurs(a, S),\ next(S', S)$.
$occurs(a, 0)$.

Let $I = [10, 20] \cup [100, 120]$ and consider $t_0, t_1, t_2 \in time$ such that $t_0, t_1 \in I$ and $t_2 \notin I$. Let $A_1$ be a collection of atoms consisting of the specification of sorts, $step(0), step(1), action(a)$, etc, and atoms $at(0, t_0), at(1, t_1), next(1, 0), occurs(a, 0)$, $holds(f, 1)$, $acceptable\_time(t)$ for every $t \in I$. Let $A_2 = (A_1 \setminus \{at(1, t_1)\}) \cup \{at(1, t_2), \neg occurs(a, 1)\}$. It is not difficult to check that $A_1$ and $A_2$ are answer sets of $P$.

Now let us consider a program $P'$ obtained from $P$ by replacing the rule

$holds(f, S') \leftarrow occurs(a, S),\ next(S', S).$

of $P$ by

$holds(f, S') \leftarrow occurs(a, S),\ next(S', S),\ not\ ab(S, S').$

and by adding rules $\neg holds(f, 1).$ and $ab(S, S') \overset{+}{\leftarrow}.$

It is not difficult to check that answer sets $A_1'$ and $A_2'$ of $P'$ are obtained from answer sets $A_1$ and $A_2$ of $P$ by replacing $holds(f, 1)$ by $\neg holds(f, 1)$ and $ab(0, 1)$.

From this example, one can see that problems involving both planning and scheduling components can be easily modeled by $\mathcal{AC}(\mathcal{C})$.

## 3 Systems implementing subclasses of $\mathcal{AC}(\mathcal{C})$

We have implemented two systems – $\mathcal{AD}solver$ and $\mathcal{AC}solver$ – for some subclasses of $\mathcal{AC}(\mathcal{C})$. The first restricts $\mathcal{AC}(\mathcal{C})$ programs parameterized by difference constraints as follows: 1) the regular part allows no disjunction; 2) the middle part allows only denials whose bodies contain at most one primitive constraint; and 3) the defined part is empty. A difference constraint is of the form $X - Y > c$ where $X, Y$ are variables, and $c$ is a constant number. The second restricts $\mathcal{AC}(\mathcal{C})$ programs parameterized by linear inequalities over real numbers by 1) allowing no disjunction in the regular part; 2) allowing no mixed predicate or regular literals in the rules with defined predicates as heads; and 3) requiring that the defined part has a unique answer set.

In designing algorithms for computing the answer sets of (restricted) $\mathcal{AC}(\mathcal{C})$ programs, we are only interested in a reduced answer set containing the regular literals and mixed literals but not the defined literals. One reason to ignore defined literals is that defined literals from an answer set are uniquely determined by the primitive constraints and regular and mixed literals in the set. Every answer set contains the same set of defined literals because the defined part has a unique answer set. Another reason is that in many applications, we are only interested in when a defined predicate can be satisfied instead of computing all the defined literals in an answer set. To find a reduced answer set, we ground only the regular variables in a program. Using the partially grounded program, we then enumerate the possibilities of the regular literals with the help of search space pruning by the definition of answer set (e.g., unit propagation in DPLL and atmost() operator in SMODELS). Intuitively, the mixed predicate associates constraint variables with regular terms. In the process of enumeration, the middle rules "produce" constraints on the constraint variables. For instance, consider the middle rule $\leftarrow at(0, X), at(1, Y), X > Y$ where $at(0, X)$ is read as the step 0 (in a plan) occurs at real time $X$. Since we have to include one copy of each mixed atom (in terms of the regular terms in the atom, see the definition of partial interpretation) in an answer set, to satisfy the middle rule, the constraint $X \leq Y$ (the negation of $X > Y$) has to be satisfied. In other words, the mixed literals "define" variables while the middle rules "post" constraints on these variables. A constraint solver is employed to solve the constraints produced by middle rules during the enumeration. The constraint solver is desirable to be incremental in answering the satisfiability of the newly produced constraint(s) and in removing constraints that are withdrawn due to backtracking in enumerating regular literals.

Program `lparse` is employed to ground the regular variables for both $\mathcal{AD}solver$ and $\mathcal{AC}solver$.

### 3.1 $\mathcal{AD}solver$

The constraint solver of $\mathcal{AD}solver$ to process the difference constraints employs an incremental algorithm [9] that has a complexity of $O(m+n\log n)$ for a newly produced constraint and of constant time for removing a constraint. $m$ and $n$ are the number of constraints and variables (accumulated so far) respectively.

Difference constraints can be used to represent a significant class of problems in temporal reasoning (and thus in planning and scheduling problems). To test $\mathcal{AD}solver$ in realistic applications, we extend the planning component of USA-Advisor[3] – a decision support system for the reaction control system (RCS) of the space shuttle – by adding a scheduling component.

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.

To maneuver the aircraft, the planning component needs to generate a sequence of actions to make the oxidizer and fuel propellants flow through the nodes (tanks, junctions) and valves which are open and reach the jet. The scheduling component is to find a schedule of the actions generated from the planning component. Consider the following example. A node is pressurized when fuel or oxidizer reaches the node. 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. Time steps of the program should be assigned actual time (in seconds) satisfying these constraints. We should be able to answer questions like: can a particular maneuver be performed in less than 30 seconds?

In the scheduling component, we expand the signature of USA-Advisor by constraint sort time = [0..400] and mixed predicate $at(S,T)$ which is read as step $S$ is performed at time $T$. In addition we need relations $otank(X)$ and $ftank(X)$ which hold if $X$ is an oxidizer tank and fuel tank respectively. Fluent $got\_opened(V,S)$ is true when valve $V$ is closed at step $S-1$ and gets opened at step $S$. Fluent $got\_pressurized(N,X,S)$ is true when node $N$ is not pressurized

| Instances | Plan length | Plan existence | CPU time (s) |
|---|---|---|---|
| $I_1$ | 3 | No | 2.7 |
| $I_2$ | 4 | No | 650.9 |
| $I_3$ | 4 | No | 278.8 |
| $I_4$ | 4 | Yes | 407.0 |
| $I_5$ | 4 | Yes | 865.9 |
| $I_6$ | 3 | No | 269.4 |

**Table 1.** Experimental results of the extended USA-Advisor on random instances. The instances correspond to the files in the original USA-Advisor package as follows: $I_1$ – instances-auto/ins/instance_001, $I_2$ – instances-auto/ins-4/instance_008, $I_3$ – instances-auto/ins-8/instance_009, $I_4$ – instances-monica/ins-3-0/instance_0012, $I_5$ – instances-monica/ins-3-0/instance_011, $I_6$ – instances-monica/ins-3-0/instance_034.

at step $S-1$ and is pressurized at step $S$ by tank $X$. Among other rules, we have, for instance, the following rules which involve typical temporal constraints.

$$\begin{aligned}\leftarrow\ & link(N_1, N_2, V), got\_pressurized(N_1, X, S_1), S_1 < S_2, otank(X),\\ & got\_opened(V, S_2), at(S_1, T_1), at(S_2, T_2), T_1 - T_2 > -5.\end{aligned}$$

It says that a valve $V$ which links $N_1$ to another node can not be opened until $N_1$ is stabilized, i.e., the time between $N_1$ being pressurized (by an oxidizer tank) and $V$ being opened should be longer than 5 seconds. The rule below requires that the jets of a system $R$ should be ready to fire ($goal(S, R)$) in 30 seconds.

$$\leftarrow\ system(R), goal(S, R), at(0, T_1), at(S, T_2), T_2 - T_1 > 30.$$

We have tested $\mathcal{AD}solver$ using the extended USA-Advisor on the random instances of the initial situations and target maneuvers[1]. We list some of the sample results in Table 1. The experiments were carried out on a DELL PowerEdge 1850 (two 3.6GHz Intel Xeon CPUs) with Linux.

It is worth noting that for the standard translation of our program into a regular ASP program, the grounder `lparse1.1.1` can't ground the simplest program instance $I_1$ in a day. The latest version of grounder `gringo`[2] takes an hour to ground $I_1$, producing a file of size 16 Gbytes, but the answer set solver `clasp` can not produce any results in 30 hours.

## 3.2 $\mathcal{AC}$*solver*

$\mathcal{AC}solver$ allows defined part in an $\mathcal{AC}(\mathcal{C})$ program. To compute an answer set, during the enumeration process, the middle rules may produce queries, i.e., defined atoms, that have to be satisfied. In implementing the system, we use CLP(R) system [7] to answer the satisfiability of a newly produced query. We

[1] `http://www.krlab.cs.ttu.edu/Software/Download/rcs/`

[2] Unreleased version obtained from the author in Aug 2008.

have tested some examples involving both default reasoning and constraints on $\mathcal{AC}solver$, it can compute an answer set in a reasonable amount of time. Currently, a significant effort is made to develop a new version of $\mathcal{AC}solver$ where the focus is on an algorithm (stilled based on the CLP(R) implementation) to answer and remove queries incrementally.

## 4 Conclusion

In the demonstration, we will run both $\mathcal{AD}solver$ and $\mathcal{AC}solver$ on a number of interesting examples including some planning and scheduling programs, disjunctive temporal constraints and explanations. The test cases will be made available online.

## References

1. Marcello Balduccini. CR-MODELS: An inference engine for CR-Prolog. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'07)*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 18–30. Springer, 2007.
2. Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. In Patrick Doherty, John McCarthy, and Mary-Anne Williams, editors, *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, pages 9–18, Mar 2003.
3. Marcello Balduccini, Michael Gelfond, and Monica Nogueira. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, 47:183–219, 2006.
4. Chitta Baral, Karen Chancellor, Nam Tran, Nhan Tran, Anna Joy, and Michael Berens. A knowledge based approach for representing and reasoning about cell signalling networks. In *Proceedings of European Conference on Computational Biology, Supplement on Bioinformatics*, pages 15–22, 2004.
5. Daniel R. Brooks, Esra Erdem, James W. Minett, and Donald Ringe. Character-based cladistics and answer set programming. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages*, pages 37–51, 2005.
6. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.
7. J. Jaffar, S. Michaylov, P. J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.
8. Joxan Jaffar and M. J. Maher. Constraint Logic Programming. *Journal of Logic Programming*, 19/20:503–581, 1994.
9. G. Ramalingam, Junehwa Song, Leo Joskowicz, and Raymond E. Miller. Solving systems of difference constraints incrementally. *Algorithmica*, 23(3):261–275, 1999.
10. Timo Soininen and Ilkka Niemella. Developing a declarative rule language for applications in product configuration. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages*, pages 305–319, 1998.
11. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.