CR-Prolog as a Specification Language for Constraint Satisfaction Problems

Marcello Balduccini

Intelligent Systems, OCTO
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com

Abstract In this paper we describe an approach for integrating CR-Prolog and constraint programming, in which CR-Prolog is viewed as a specification language for constraint satisfaction problems. Differently from other methods of integrating ASP and constraint programming, our approach has the advantage of allowing the use of off-the-shelf, unmodified ASP solvers and constraint solvers, and of global constraints, which substantially increases practical applicability.

1 Introduction

Particular interest has been recently devoted to the integration of Answer Set Programming (ASP) [1] with Constraint Logic Programming (CLP) (see [2,3]), aimed at combining the ease of knowledge representation of ASP with the powerful support for numerical computations of CLP. Such approaches are mostly based on extending the ASP language and on using answer set and constraint solvers modified so that they can work together. Although the combination of ASP and CLP showed substantial performance improvements over ASP alone, the restriction of using ad-hoc ASP and CLP solvers limits the practical applicability of the approach. In fact, programmers can no longer select the solvers that best fit their needs (most notably, SMODELS, DLV, SWI-Prolog and SICStus Prolog), as is instead commonly done in ASP. Another limitation is the general lack of specific support for global constraints. Without global constraints, applications' performance is often heavily impacted by the combinatorial explosion of the underlying search space.

In [4] we have presented a method for integrating ASP and constraint programming. In this paper we extend the approach and integrate constraint programming with an extension of ASP, called CR-Prolog [5]. CR-Prolog introduces in ASP the notion of consistency restoring rule, which is particularly useful to represent unlikely events, less-desired choices, etc. Our technique consists in viewing CR-Prolog as a specification language for constraint satisfaction problems. CR-Prolog programs are written in such a way that their answer sets encode the desired constraint satisfaction problems; the solutions to those problems are found using constraint satisfaction techniques. Both the answer sets and the solutions to the constraint problems can be computed with arbitrary off-the-shelf solvers, as long as a (relatively simple) translation procedure is defined from the ASP encoding of the constraint problems to the input language of the

constraint solver selected. Moreover, our approach allows the use of the global constraints available in the selected constraint solver. Compared to the other approaches to the integration of ASP and CLP, our technique allows programmers to exploit the full power of the underlying state-of-the-art solvers when tackling industrial-size problems. Finally, although space restrictions prevent us from discussing it here, our experiments have also shown that our technique produces programs that are arguably more compact and easy to understand than those written in CLP alone, but with comparable performance.

2 Background

The syntax of CR-Prolog is defined as follows. A *regular rule* is a statement of the form[1] $h \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$, where h and l_i's are literals (defined as usual). The intuitive meaning of the statment is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, has to believe h. A *consistency-restoring rule* (or *cr-rule*) is a statement of the form: $r : h \stackrel{+}{\leftarrow} l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$, where r, called the cr-rule's *name*, is a (possibly compound) term uniquely identifying the cr-rule. The intuitive reading of the statement is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, *may possibly* believe h. The implicit assumption is that this possibility is used as little as possible, only when the reasoner cannot otherwise form a non-contradictory set of beliefs. A preference order on the use of cr-rules can also be given by means of atoms of the form $prefer(r_1, r_2)$ [5]. By rule we mean a regular rule or a cr-rule. Given rule ρ, we call $\{l_1, \ldots, \text{not } l_n\}$ its *body* ($body(\rho)$). Given cr-rule name r, $body(r)$ denotes the body of the corresponding cr-rule. A *program* is a set of rules. As usual, a non-ground program is viewed as a shorthand for the program consisting of the ground instances of its rules. Given a program Π, the *regular part* of Π is the set of its regular rules, and is denoted by $reg(\Pi)$. The set of its cr-rules is denoted by $cr(\Pi)$. The semantics of CR-Prolog can be found in [5].

Let us now turn our attention to Constraint Programming. The definition of constraint satisfaction problem that follows is adapted from [6]. A *Constraint Satisfaction Problem (CSP)* is a triple $\langle X, D, C \rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ is a set of domains, such that D_i is the domain of variable x_i (i.e. the set of possible values that the variable can be assigned), and C is a set of constraints. Each constraint $c \in C$ is a pair $c = \langle \sigma, \rho \rangle$ where σ is a list of variables and ρ is a subset of the Cartesian product of the domains of such variables. An *assignment* is a pair $\langle x_i, a \rangle$, where $a \in D_i$, whose intuitive meaning is that variable x_i is assigned value a. A *compound assignment* is a set of assignments to distinct variables from X. A *complete assignment* is a compound assignment to all the variables in X. A constraint $\langle \sigma, \rho \rangle$ specifies the acceptable assignments for the variables from σ. We say that such assignments *satisfy* the constraint. A *solution* to a CSP $\langle X, D, C \rangle$ is a complete assignment satisfying every constraint from C. Constraints can be represented either *extensionally*, by specifying the pair $\langle \sigma, \rho \rangle$, or *intensionally*, by specifying an expression involving variables, such as $x < y$. In this paper we focus on constraints represented intensionally.

[1] For simplicity we focus on non-disjunctive programs. Our results extend to disjunctive programs in a natural way.

A *global constraint* is a constraint that captures a relation between a non-fixed number of variables [7], such as $sum(x, y, z) < w$ and *all_different*$(x_1, \ldots, x_k)$. One should notice that the mapping of an intensional constraint specification into a pair $\langle \sigma, \rho \rangle$ depends on the *constraint domain*. For example, the expression $1 \leq x < 2$ corresponds to the constraint $\langle \langle x \rangle, \{\langle 1 \rangle\} \rangle$ if the finite domain is considered, while it corresponds to $\langle \langle x \rangle, \{\langle v \rangle \mid v \in [1, 2)\} \rangle$ in a continuous domain. For this reason, in this paper we assume that a CSP includes the specification of the intended constraint domain.

3 Encoding Constraint Problems in CR-Prolog

Our approach consists in writing CR-Prolog programs whose answer sets encode the desired constraint satisfaction problems (CSPs). The solutions to the CSPs are then computed using constraint satisfaction techniques.

CSPs are encoded in CR-Prolog using the following three types of statements: (1) a constraint domain declaration is a statement of the form $cspdomain(\mathcal{D})$, where $\mathcal{D}$ is a constraint domain such as fd, q, or r; informally, the statement says that the CSP is over the specified constraint domain, thereby fixing an interpretation for the intensionally specified constraints; (2) a constraint variable declaration is a statement of the form $cspvar(x, l, u)$, where x is a ground term denoting a variable of the CSP (CSP variable or constraint variable for short), and l and u are numbers from the constraint domain; the statement says that the domain of x is $[l, u]$;[2] (3) a constraint statement is a statement of the form $required(\gamma)$, where γ is an expression that intensionally represents a constraint on (some of) the variables specified by the $cspvar$ statements; intuitively the statement says that the constraint intensionally represented by γ is required to be satisfied by any solution to the CSP. For the purpose of specifying global constraints, we allow γ to contain expressions of the form $[\delta/k]$. If δ is a function symbol, the expression intuitively denotes the sequence of all variables formed from function symbol δ and with arity k, ordered lexicographically. For example, given CSP variables $v(1), v(2), v(3)$, $[v/1]$ denotes the sequence $\langle v(1), v(2), v(3) \rangle$. If δ is a relation symbol and $k \geq 1$, the expression intuitively denotes the sequence $\langle e_1, e_2, \ldots, e_n \rangle$ where e_i is the last element of the i^{th} k-tuple satisfying relation δ, according to the lexicographic ordering of such tuples. For example, given a relation r' defined by $r'(a, 3), r'(b, 1), r'(c, 2)$, the expression $[r'/2]$ denotes the sequence $\langle 3, 1, 2 \rangle$.

Example 1. A simple CSP is encoded by $A_1 = \{cspdomain(\mathit{fd}),\ cspvar(v(1), 1, 3),\ cspvar(v(2), 2, 5),\ cspvar(v(3), 1, 4),\ required(v(1) + v(2) \leq 4),\ required(v(2) - v(3) > 1),\ required(sum([v/1]) \geq 4)\}$.

In the rest of this paper, we consider signatures that contain: relations $cspdomain$, $cspvar$, $required$; constant symbols for the constraint domains $\mathcal{FD}$, $\mathcal{Q}$, and $\mathcal{R}$; suitable symbols for the variables, functions and relations used in the CSP; the numerical constants needed to encode the CSP.

Let A be a set of atoms formed from relations $cspdomain$, $cspvar$, and $required$. We say that A is a *well-formed CSP definition* if: A contains exactly one constraint

[2] As an alternative, the domain of the variables could also be specified using constraints. We use a separate statement for similarity with CLP languages.

domain declaration; the same CSP variable does not occur in two or more constraint variable declarations of A; every CSP variable that occurs in a constraint statement from A also occurs in a constraint variable declaration from A. Let A be a well-formed CSP definition. The CSP *defined* by A is the triple $\langle X, D, C\rangle$ such that: $X = \{x_1, x_2, \ldots, x_k\}$ is the set of all CSP variables from the constraint variable declarations in A; $D = \{D_1, D_2, \ldots, D_k\}$ is the set of domains of the variables from X, where the domain D_i of variable x_i is given by arguments l and u of the constraint variable declaration of x_i in A, and consists of the segment between l and u in the constraint domain specified by the constraint domain declaration from A; C is a set containing a constraint γ' for each constraint statement $required(\gamma)$ of A, where γ' is obtained by: (1) replacing the expressions of the form $[f/k]$, where f is a function symbol, by the list of variables from X formed by f and of arity k, ordered lexicographically; (2) replacing the expressions of the form $[r/k]$, where r is a relation symbol and $k \geq 1$, by the sequence $\langle e_1, \ldots, e_n\rangle$, where, for each i, $r(t_1, t_2, \ldots, t_{k-1}, e_i)$ is the i^{th} element of the sequence, ordered lexicographically, of atoms from A formed by relation r; (3) interpreting the resulting intensionally specified constraint w.r.t. the constraint domain specified by the constraint domain declaration from A.

Example 2. Set A_1 from Example 1 defines the CSP:

$$\langle\{v(1), v(2), v(3)\}, \left\{\begin{array}{c}\{1,2,3\},\{2,3,4,5\},\\ \{1,2,3,4\}\end{array}\right\}, \left\{\begin{array}{c} v(1)+v(2)\leq 4,\ v(2)-v(3)>1,\\ sum(v(1),v(2),v(3))\geq 4\end{array}\right\}\rangle.$$

Let A be a set of literals. We say that A *contains* a well-formed CSP definition if the set of atoms from A formed by relations $cspdomain$, $cspvar$, and $required$ is a well-formed CSP definition. We also say that a CSP is defined by a set of literals A if it is defined by the well-formed CSP definition contained in A. Notice that, if a set A of literals does not contain a well-formed CSP definition, A does not define any CSP. For simplicity, in the rest of the discussion we omit the term "well-formed" and simply talk about CSP definitions.

Definition 1.

- *A pair $\langle A, \alpha\rangle$ is an* extended answer set *of cr-rule free program Π iff A is an answer set of Π and α is a solution to the CSP defined by A.*
- *Let Π be a program, and R be a set of names of cr-rules from Π, and $prefer^*$ be the transitive closure of relation $prefer$. $\mathcal{V} = \langle A, R\rangle$ is an* extended view *of Π if: (1) A is an extended answer set of $reg(\Pi) \cup \theta(R)$; (2) for every r_1, r_2, if $S \models$ prefer$^*(r_1, r_2)$, then $\{r_1, r_2\} \not\subseteq R$; (3) for every r in R, $body(r)$ is satisfied by S.*
- *An extended view $\mathcal{V}$ is an* extended candidate answer set *of Π if, for every view $\mathcal{V}'$ of Π, $\mathcal{V}'$ does not dominate $\mathcal{V}$.*[3]
- *A is an* extended answer set *of Π if: (1) there exists a set R of names of cr-rules from Π such that $\langle A, R\rangle$ is a candidate answer set of Π, and (2) for every extended candidate answer set $\langle A', R'\rangle$ of Π, $R' \not\subset R$.*

[3] The notion of dominance extends to extended views in a natural way.

Example 3. Consider set A_1 from Example 1. An extended answer set of A_1 is $\langle A_1, \{(v(1),1),(v(2),3),(v(3),1)\}\rangle$. Consider program P_1 below and a corresponding extended answer set. Notice that the cr-rule is used to say that the sum of the CSP variables should be less than 20 *if at all possible*.

$$P_1 = \begin{cases} i(1).\ \ldots i(4).\ \ cspdomain(\mathit{fd}). \\ cspvar(v(I),1,10) \leftarrow i(I). \\ required(v(I1) - v(I2) \geq 3) \leftarrow \\ \qquad i(I1),\ i(I2),\ I2 = I1 + 1. \\ required(sum([v/1]) \geq 20) \leftarrow \\ \qquad \text{not } can\_violate. \\ r:\ can\_violate \stackrel{+}{\leftarrow}. \end{cases}$$

Extended answer set:
$\langle \{i(1),\ldots,i(4), cspdomain(\mathit{fd}),$
$cspvar(v(1),1,10),\ldots,$
$cspvar(v(4),1,10),$
$required(v(1)-v(2)\geq 3),\ldots,$
$required(v(3)-v(4)\geq 3)\},$
$\{(v(1),10),(v(2),7),(v(3),4),(v(4),1)\}\rangle$

To compute the extended answer sets of a cr-rule free program, we combine the use of answer set solvers and constraint solvers (see Algorithm 1). As discussed in [4], step (5) of Algorithm 1 relies on the correctness of the translation from the CSP definition to the encoding for the constraint solver. Soundness and completeness results for the algorithm can be found in [4]. The extended answer sets of arbitrary CR-Prolog programs can be computed by extending the CRMODELS algorithm from [8]. The complete algorithm is shown below (see Algorithm 2). We refer the reader to [8] for the definition of operators γ_i, τ, λ, ν and hr. To compute extended answer sets, the original algorithm is modified to use a new function $\epsilon_1(\Pi)$, which returns an arbitrary element of $\epsilon(\Pi)$ (and replaces function α_1 as used in the original algorithm). Soundness and completeness of the algorithm follow from soundness and completeness of Algorithm 1 and from the results in [8].

Algorithm 1: ϵ

Input: Program Π
Output: The set of extended answer sets of Π

1 $\mathcal{E} := \emptyset$
2 Let $\mathcal{A}$ be the set of answer sets of Π containing a CSP definition.
3 **for each** $A \in \mathcal{A}$ **do**
4 Select solver $solve_{\mathcal{D}}$ for constraint domain $\mathcal{D}$ as specified by $cspdomain(\mathcal{D}) \in A$.
5 Translate the CSP definition from A into an encoding $\chi_A^{\mathcal{D}}$ suitable for $solve_{\mathcal{D}}$.
6 Let $\mathcal{S} = \{\alpha_1, \ldots, \alpha_k\}$ be the set of solutions returned by $solve_{\mathcal{D}}(\chi_A^{\mathcal{D}})$.
7 **for each** $\alpha \in \mathcal{S}$ **do** $\mathcal{E} := \mathcal{E} \cup \langle A, \alpha\rangle$.
8 **end**
9 **return** $\mathcal{E}$

4 Related Work

The *clingcon* system [3] integrates the answer set solver Clingo and the constraint solver Gecode. The system thus differs significantly from ours in that programmers cannot arbitrarily select the most suitable ASP and constraint solvers for the task at hand.

The approach proposed in [2] is based on an extension, called $\mathcal{AC}(\mathcal{C})$, of CR-Prolog, allowing the use of CSP-style constraints in the body of the rules. The assignment of values to the constraint variables is denoted by means of special atoms occurring in the

Algorithm 2: CRMODELS-CSP

Input: A CR-Prolog program Π
Output: The extended answer sets of Π

$C := \emptyset$; $\mathcal{A} := \emptyset$; $i := 0$
while $i \leq |cr(\Pi)|$ **do**
  $C' := \emptyset$
  repeat
    if $\gamma_i(\Pi) \cup C$ *is inconsistent* **then** $M := \bot$
    else
      $\langle M, \alpha \rangle := \epsilon_1(\gamma_i(\Pi) \cup C)$
      if $\tau(M, \Pi)$ *is inconsistent* **then**
        $\mathcal{A} := \mathcal{A} \cup \{\langle M \cap \Sigma(\Pi), \alpha \rangle\}$
        $C' := C' \cup \{ \leftarrow \lambda(M \cap atoms(appl, hr(\Pi))). \}$
      end
      $C := C \cup \{ \leftarrow \lambda(M), \nu(M). \}$
    end
  until $M = \bot$
  $C := C \cup C'$; $i := i + 1$ *[now consider views obtained with one more cr-rule]*
end
return $\mathcal{A}$

body of the rules. Such atoms are treated as abducibles, and their truth determined by solving a suitable CSP. The following result connects our approach and $\mathcal{AC}(\mathcal{C})$.

Theorem 1. *An $\mathcal{AC}_0$ program Π can be translated into a CR-Prolog program whose extended answer sets are in one-to-one correspondence with the answer sets of Π.*

References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing (1991) 365–385
2. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating Answer Set Programming and Constraint Logic Programming. Annals of Mathematics and Artificial Intelligence (2008)
3. Gebser, M., Ostrowski, M., Schaub, T.: Constraint Answer Set Solving. In: 25th International Conference on Logic Programming (ICLP09). Volume 5649. (2009)
4. Balduccini, M.: Representing Constraint Satisfaction Problems in Answer Set Programming. In: ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09). (2009)
5. Balduccini, M., Gelfond, M.: Logic Programs with Consistency-Restoring Rules. In Doherty, P., McCarthy, J., Williams, M.A., eds.: International Symposium on Logical Formalization of Commonsense Reasoning. AAAI 2003 Spring Symposium Series (Mar 2003) 9–18
6. Smith, B.M.: 11. Modelling. Foundations of Artificial Intelligence. In: Handbook of Constraint Programming. Elsevier (2006) 377–406
7. Katriel, I., van Hoeve, W.J.: 6. Global Constraints. Foundations of Artificial Intelligence. In: Handbook of Constraint Programming. Elsevier (2006) 169–208
8. Balduccini, M.: CR-MODELS: An Inference Engine for CR-Prolog. In: LPNMR 2007. (May 2007) 18–30