Representing Constraint Satisfaction Problems in Answer Set Programming

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 answer set programming (ASP) and constraint programming, in which ASP is viewed as a specification language for constraint satisfaction problems. ASP programs are written in such a way that their answer sets encode the desired constraint satisfaction problems; the solutions of those problems are then found using constraint satisfaction techniques. 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 the practical applicability of the approach to industrial-size problems.

1 Introduction

Answer Set Programming (ASP) [1,2,3] is a declarative programming paradigm with roots in the research on non-monotonic logic and on the semantics of default negation of Prolog.

In recent years, ASP has been applied successfully to solving complex problems (e.g. [4,5]), and the underlying language has been extended in various directions to broaden its applicability even further (e.g. [6,7]).

Particular interest has been recently devoted to the integration of ASP with Constraint Logic Programming (CLP) (see [8,9] and the *clingcon* system[1]), aimed at combining the ease of knowledge representation of ASP with the powerful support for numerical computations of CLP. Such approaches are based on an extension of the ASP language, and on the use of answer set and constraint solvers modified to 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 limit for the practical applicability of the approach is the lack of specific support for global constraints. Without global constraints, applications' performance is often heavily impacted by the combinatorial

[1] http://www.cs.uni-potsdam.de/clingcon/

explosion of the underlying search space, even for relatively small (compared to the intended application domain) problem instances.

In this paper we describe an approach for integrating ASP and constraint programming, in which ASP is viewed as a specification language for constraint satisfaction problems. ASP 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 state-of-the-art solvers when tackling industrial-size problems.

The paper is organized as follows. We start by giving background notions of ASP and constraint satisfaction. In Section 3, we describe our encoding of constraint satisfaction problems in ASP and define its semantics. In Section 4 we explain how to compute the solutions to the constraint problems encoded by the answer sets of ASP programs. Section 5 compares our approach with existing research on integrating ASP and CLP. In Section 6, we draw conclusions.

2 Background

The syntax and semantics of ASP are defined as follows. Let Σ be a signature containing constant, function and predicate symbols. Terms and atoms are formed as usual. A literal is either an atom a or its strong (also called classical or epistemic) negation $\neg a$. The sets of atoms and literals formed from Σ are denoted by $atoms(\Sigma)$ and $literals(\Sigma)$ respectively.

A *rule* is a statement of the form:[2]

$$h \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n \tag{1}$$

where h and l_i's are literals and *not* is the so-called *default negation*. The intuitive meaning of the rule 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. We call h the *head* of the rule, and $\{l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n\}$ the *body* of the rule. Given a rule r, we denote its head and body by $head(r)$ and $body(r)$ respectively.

A *program* is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of rules over Σ. Often we denote programs by just the second element of the pair, and let the signature be defined implicitly. In that case, the signature of Π is denoted by $\Sigma(\Pi)$.

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

A set A of literals is *consistent* if no two complementary literals, a and $\neg a$, belong to A. A literal l is *satisfied* by a consistent set of literals A if $l \in A$. In this case, we write $A \models l$. If l is not satisfied by A, we write $A \not\models l$. A set $\{l_1, \ldots, l_k\}$ of literals is satisfied by a set of literals A ($A \models \{l_1, \ldots, l_k\}$) if each l_i is satisfied by A.

Programs not containing default negation are called *definite*. A consistent set of literals A is *closed* under a definite program Π if, for every rule of the form (1) such that the body of the rule is satisfied by A, the head belongs to A.

Definition 1. *A consistent set of literals A is an* answer set *of definite program Π if A is closed under all the rules of Π and A is set-theoretically minimal among the sets closed under all the rules of Π.*

The *reduct* of a program Π with respect to a set of literals A, denoted by Π^A, is the program obtained from Π by deleting:

- Every rule, r, such that $l \in A$ for some expression of the form not l from the body for r;
- All expressions of the form not l from the bodies of the remaining rules.

We are now ready to define the notion of answer set of a program.

Definition 2. *A consistent set of literals A is an* answer set *of program Π if it is an answer set of the reduct Π^A.*

To simplify the programming task, variables are often allowed to occur in ASP programs. A rule containing variables (called a *non-ground* rule) is then viewed as a shorthand for the set of its *ground instances*, obtained by replacing the variables in it by all the possible ground terms. Similarly, a non-ground program is viewed as a shorthand for the program consisting of the ground instances of its rules.

Let us now turn our attention to Constraint Programming. In this paper we follow the traditional definition of constraint satisfaction problem. The one that follows is adapted from [10]. 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.[3] 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.

[3] Strictly speaking, the use of the same index i across sets X and D in the above definition of the set of domains would require X and D to be ordered. However, as the definition of CSP is insensitive to the particular ordering chosen, we follow the approach, common in the literature on constraint satisfaction, of simply considering X and D sets and abusing notation slightly in the definition of CSP.

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. A *global constraint* is a constraint that captures a relation between a non-fixed number of variables [11], 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, and in line with the CLP Schema [12,13], in this paper we assume that a CSP includes the specification of the intended constraint domain.

3 Representing constraint problems in ASP

Our approach consists in writing ASP 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 ASP using the following three types of statements.

– 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 states that the CSP is over the specified constraint domain, thereby fixing an interpretation for the intensionally specified constraints.

– 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 informally states that the domain of x is $[l, u]$.[4]

– 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

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

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.[5] For example, if 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 $r'(a,3), r'(b,1), r'(c,2)$ (that is, by tuples $\langle a,3\rangle$, $\langle b,1\rangle$, $\langle c,2\rangle$), the expression $[r'/2]$ denotes the sequence $\langle 3,1,2\rangle$.

Example 1. The following sets of statements encode simple CSPs:

$$
\begin{array}{l}
A_1 = \\
\{cspdomain(\mathit{fd}), \\
\ cspvar(v(1),1,3), cspvar(v(2),2,5), cspvar(v(3),1,4), \\
\ required(v(1)+v(2) \leq 4), required(v(2)-v(3)>1), \\
\ required(sum([v/1]) \geq 4)\}
\end{array}
$$

$$
\begin{array}{l}
A_2 = \\
\{cspdomain(\mathit{fd}), \\
\ cspvar(start(j1),1,100), cspvar(start(j2),25,100), \\
\ cspvar(start(j3),30,80), cspvar(start(j4),45,150), \\
\ required(serialized([start/1],[duration/2]))\}
\end{array}
$$

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 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.

[5] The choice of a particular order is due to the fact that global constraints that accept multiple lists often expect the elements in the same position throughout the lists to be in a certain relation. More sophisticated techniques for the specification of lists are possible, but, according to our analysis of the use of global constraints in constraint satisfaction, this method should work well in most cases.

Example 2. The following is not a well-formed CSP definition:

$$\{cspdomain(\mathit{fd}), cspvar(x, 1, 2), required(x < y)\}.$$

On the other hand, the sets of atoms from Example 1 are well-formed CSP definitions.

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. 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. Constraint 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 3. Set A_1 from Example 1 defines the CSP $\langle X_1, D_1, C_1\rangle$:

- $X_1 = \{v(1), v(2), v(3)\}$
- $D_1 = \{\{1, 2, 3\}, \{2, 3, 4, 5\}, \{1, 2, 3, 4\}\}$
- $C_1 = \left\{ \begin{array}{l} v(1) + v(2) \leq 4,\ v(2) - v(3) > 1, \\ sum(v(1), v(2), v(3)) \geq 4 \end{array} \right\}$

Consider A_2 from Example 1 and

$$\begin{array}{l} I = \{duration(j1, 20), duration(j2, 10), \\ \qquad duration(j3, 50), duration(j4, 60)\}. \end{array}$$

Set $A_2 \cup I$ defines the CSP $\langle X_2, D_2, C_2\rangle$:

- $X_2 = \{start(j1), start(j2), start(j3), start(j4)\}$
- $D_2 = \{\{1, 2, \ldots, 100\}, \{25, \ldots, 100\}, \ldots\}$
- $C_2 =$
  $\{serialized([start(j1), start(j2), start(j3), start(j4)],$
  $[20, 10, 50, 60])\}$

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 3. *A pair $\langle A, \alpha \rangle$ is an extended answer set of program Π iff A is an answer set of Π and α is a solution to the CSP defined by A.*

Example 4. Consider set A_1 from Example 1. An extended answer set of A_1 is:

$$\langle A_1, \{(v(1), 1), (v(2), 3), (v(3), 1)\} \rangle.$$

Example 5. Consider the program:

$$P_1 = \begin{cases} index(1).\ index(2).\ index(3).\ index(4). \\ \\ cspdomain(\mathit{fd}). \\ \\ cspvar(v(I), 1, 10) \leftarrow index(I). \\ \\ required(v(I1) - v(I2) \geq 3) \leftarrow \\ \qquad index(I1),\ index(I2), \\ \qquad I2 = I1 + 1. \end{cases}$$

An extended answer set of P_1 is:

$$\begin{array}{l} \langle \{index(1), \ldots, index(4), cspdomain(\mathit{fd}), \\ \quad cspvar(v(1), 1, 10), \ldots, cspvar(v(4), 1, 10), \\ \quad required(v(1) - v(2) \geq 3), \ldots, \\ \quad required(v(3) - v(4) \geq 3)\}, \\ \{(v(1), 10), (v(2), 7), (v(3), 4), (v(4), 1)\} \rangle \end{array}$$

Example 6. Consider the riddle:
"There are either 2 or 3 brothers in the Smith family. There is a 3 year difference between one brother and the next (in order of age). The age of the eldest brother is twice the age of the youngest. The youngest is at least 6 years old."

A program, P_2, that finds the solutions to the riddle is:

% There are either 2 or 3 brothers in the Smith family.
$num\_brothers(2) \leftarrow$ not $num\_brothers(3)$.
$num\_brothers(3) \leftarrow$ not $num\_brothers(2)$.

$index(1).index(2).index(3).$

$is\_brother(B) \leftarrow$
$\quad index(B), index(N),$
$\quad num\_brothers(N),$
$\quad B \leq N.$

$eldest\_brother(1).$

$$
\begin{array}{l}
youngest\_brother(B) \leftarrow \\
\qquad index(B), \\
\qquad num\_brothers(B).
\end{array}
$$

$$cspdomain(\mathit{fd}).$$

$$cspvar(age(B), 1, 80) \leftarrow index(B), is\_brother(B).$$

% 3 year difference between one brother and the next.

$$
\begin{array}{l}
required(age(B1) - age(B2) = 3)) \leftarrow \\
\qquad index(B1), index(B2), \\
\qquad is\_brother(B1), is\_brother(B2), \\
\qquad B2 = B1 + 1.
\end{array}
$$

% The eldest brother is twice as old as the youngest.

$$
\begin{array}{l}
required(age(BE) = age(BY) * 2) \leftarrow \\
\qquad index(BE), index(BY), \\
\qquad eldest\_brother(BE), \\
\qquad youngest\_brother(BY).
\end{array}
$$

% The youngest is at least 6 years old.

$$
\begin{array}{l}
required(age(BY) \geq 6) \leftarrow \\
\qquad index(BY), \\
\qquad youngest\_brother(BY).
\end{array}
$$

An extended answer set of P_2 is:

$$
\begin{array}{l}
\langle \{num\_brothers(3), \\
\quad cspvar(age(1), 1, 80), \ldots, cspvar(age(3), 1, 80), \ldots\}, \\
\quad \{(age(1), 12), (age(2), 9), (age(3), 6)\}\rangle,
\end{array}
$$

which states that there are 3 brothers, of age 12, 9, and 6 respectively. Notice that there is no extended answer set containing $num\_brothers(2)$.

4 Computing Extended Answer Sets

To compute the extended answer sets of a program, we combine the use of answer set solvers and constraint solvers. The algorithm is as follows:

Algorithm Alg_1
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 $A \in \mathcal{A}$:
   (a) Select a constraint solver $solve_{\mathcal{D}}$ for the constraint domain $\mathcal{D}$ specified by the constraint domain declaration from A.
   (b) Translate the CSP definition from A into an encoding $\chi_A^{\mathcal{D}}$ suitable for $solve_{\mathcal{D}}$.

(c) Let $\mathcal{S} = \{\alpha_1, \ldots, \alpha_k\}$ be the set of solutions returned by $solve_{\mathcal{D}}(\chi_A^{\mathcal{D}})$.
(d) For each $\alpha \in \mathcal{S}$, $\mathcal{E} := \mathcal{E} \cup \langle A, \alpha \rangle$.

4. Return $\mathcal{E}$.

As can be seen from step (3b), the algorithm relies on the correctness of the translation from the CSP definition to the encoding for the constraint solver. More precisely:

Definition 4. *A translation algorithm $Trans$ from CSP definitions to encodings suitable for a constraint solver $solve$ is* correct *if α is a solution to the CSP defined by A iff α is one of the answers returned by $solve(Trans(A))$.*

The following theorems deal with the soundness and completeness of Alg_1. Their proofs are not difficult, and are omitted to save space.

Theorem 1. *Let Π be a program and $Trans$ be a translation algorithm as above. If $\langle A, \alpha \rangle \in Alg_1(\Pi)$ and $Trans$ is correct, then $\langle A, \alpha \rangle$ is an extended answer set of Π.*

Theorem 2. *Let Π be a program and $Trans$ be a translation algorithm as above. If $\langle A, \alpha \rangle$ is an extended answer set of Π, $Trans$ is correct, and the solver selected for A at step (3a) of the algorithm is complete for $\chi_A^{\mathcal{D}}$, then $\langle A, \alpha \rangle \in Alg_1(\Pi)$.*

A convenient way to compute the solutions of the CSPs at step (3c) is to use the constraint solvers embedded in CLP systems. Therefore, we describe an algorithm to translate from a CSP definition to a CLP program. The translation algorithm assumes that the constraint variables that occur in the CSP definition being translated are *legal ground terms* for the CLP system, or that a suitable mapping to legal terms has implicitly taken place, and that the CLP system can handle all the constraint domains of interest. The algorithm is based on the CLP Schema [12,13].

Algorithm ψ
Input: a CSP definition A
Output: a CLP program P

1. $P := \emptyset$
2. $\nu := \emptyset$ { *CLP variables for the encoding of the CSP* }
3. $\theta := \emptyset$ { *body of the top-level clause of P* }
4. Retrieve atom $cspdomain(\mathcal{D})$ from A.
5. Add to P a directive:[6]

$$:- use\_module(library(cs))$$

   where cs is a suitable constraint solver for constraint domain $\mathcal{D}$ (e.g. clpfd, clpr).
6. For each $cspvar(x, l, u) \in A$:
   (a) $\nu := \nu \cup \{V_x\}$, where V_x is a fresh CLP variable.
   (b) $\theta := \theta \cup \{V_x \geq l, V_x \leq u\}$.
7. For each $required(\gamma_1) \in A$:
   (a) Obtain γ_2 from γ_1 by replacing every expression of the form $[f/k]$ in γ_1, where f is a function symbol, with the list of all the CSP variables of the form $f(t_1, t_2, \ldots, t_k)$ declared in A, ordered lexicographically.

[6] We use the syntax of SICStus [14]. The translation for other CLP systems is similar.

(b) Obtain γ_3 from γ_2 by replacing every expression of the form $[r/k]$ in γ_2, where r is a relation symbol and $k \geq 1$, by the list $[e_1, \ldots, e_n]$, where, for each i, $r(t_1, t_2, \ldots, t_{k-1}, e_i)$ is the i^{th} element of the list, ordered lexicographically, of atoms from A formed by r.
(c) Obtain γ_4 from γ_3 by replacing every occurrence of a CSP variable x in γ_3 by the corresponding V_x from ν.
(d) $\theta := \theta \cup \{\gamma_4\}$.
8. $\theta := \theta \cup \{labeling(\nu)\}$.[7]
9. $\lambda := \{(x, V_x) \mid x \in \nu\}$.
10. $P := P \cup \{solve(\lambda) :- \theta\}$.
11. Return P.

Example 7. Consider the CSP definition

$$A_3 = \begin{cases} cspdomain(\mathsf{fd}). \\ cspvar(x, 1, 5).\ cspvar(y, 1, 5). \\ required(x < y). \end{cases}$$

Its translation into CLP is:

$$\psi(A_3) = \begin{cases} :- use\_module(library(\mathsf{clpfd})). \\ \\ solve([(x, V_x), (y, V_y)]) :- \\ \qquad V_x \geq 1, V_x \leq 5, \\ \qquad V_y \geq 1, V_y \leq 5, \\ \qquad V_x < V_y, \\ \qquad labeling([V_x, V_y]). \end{cases}$$

Example 8. The CSP definition

$$A_4 = \begin{cases} cspdomain(\mathsf{fd}). \\ cspvar(var(a), 1, 5).\ cspvar(var(b), 2, 8). \\ duration(var(a), 4).\ duration(var(b), 2). \\ required(serialized([var/1], [duration/2])). \end{cases}$$

is translated by ψ into:

$$\psi(A_4) = \begin{cases} :- use\_module(library(\mathsf{clpfd})). \\ \\ solve([(var(a), V_x), (var(b), V_y)]) :- \\ \qquad V_x \geq 1, V_x \leq 5, \\ \qquad V_y \geq 2, V_y \leq 8, \\ \qquad serialized([V_x, V_y], [4, 2]), \\ \qquad labeling([V_x, V_y]). \end{cases}$$

The following theorem ensures the correctness of the translation.

[7] To simplify the notation, we allow sets to occur in CLP programs, although, strictly speaking, they would have to be encoded as Prolog lists.

Theorem 3. *The translation algorithm ψ is* correct.

Proof. *(Sketch)*
According to Definition 4, we have to prove that, if A is a CSP definition, then α is a solution to the CSP defined by A iff there exists a derivation from goal $solve(S)$ in $\psi(A)$ that succeeds with substitution $S|_\alpha$.

Left-to-right. Let α be a solution to the CSP defined by A and let us prove that there exists a derivation from goal $solve(S)$ in $\psi(A)$ that succeeds with substitution $S|_\alpha$. Because the value assigned by α to each variable x is by definition within the domain of the variable, the conditions added to the body of the clause for $solve$ in step (6b) are satisfied. Also, by definition α satisfies every constraint from the CSP defined by A. Hence, it is not difficult to see that the conditions added to the body of the clause in step (7d) are satisfied. Because all the conditions are satisfied by α, the call to predicate $labeling$ is bound to succeed, and the derivation is indeed successful.

Right-to-left. Now let α be such that goal $solve(S)$ in $\psi(A)$ succeeds with substitution $S|_\alpha$, and let us prove that α is a solution to the CSP defined by A. First of all, notice that, if goal $solve(S)$ succeeds with substitution $S|_\alpha$, then all the conditions in the body of the clause for $solve$ must be satisfied. Since the call to $labeling$ succeeds, all the V_x variables are instantiated in α. Hence, α is a compound assignment, and moreover it is a complete assignment by construction. Because the conditions added by step (6b) are all satisfied, all the variables are guaranteed to have values within the respective domains as specified in A. Finally, because the conditions added by step (7d) are all satisfied, it is not difficult to conclude that the constraints from A are satisfied by α. Hence, α is a solution to the CSP defined by A.

Q.E.D.

5 Related Work

The *clingcon* system[8] 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. As the system is very recent, too little documentation is currently available about the system for a thorough analysis.

The approach proposed by Mellarkod, Gelfond and Zhang [15,9] is based on an extension, $\mathcal{AC}(\mathcal{C})$, of the syntax and semantics of ASP and 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 body of the rules. Such atoms are treated as abducibles, and their truth determined by solving a suitable CSP. For example, an $\mathcal{AC}(\mathcal{C})$ program solving the same problem as the CSP defined in

[8] http://www.cs.uni-potsdam.de/clingcon/

Example 5 is:

$$
\begin{array}{l}
val(1).\ val(2).\ \ldots\ val(10).\\
\#csort(val).\\
\\
index(1).\ index(2).\ index(3).\ index(4).\\
var(v(I)) \leftarrow index(I).\\
\#mixed\ has\_value(var, val).\\
\\
\leftarrow V1 - V2 < 3,\\
\quad has\_value(v(I1), V1), has\_value(v(I2), V2),\\
\quad index(I1), index(I2), I2 = I1 + 1.
\end{array}
$$

Because constraint-related atoms (e.g. $V1 - V2 < 3$ above) are allowed to occur in the body of the rules, in a sense this approach allows feeding the results of solving CSPs back into the ASP computation, allowing for further inference. For example in [9] the authors show an ASP program containing the rules:

$$
\begin{array}{l}
acceptable\_time(T) \leftarrow 10 \leq T \leq 20.\\
acceptable\_time(T) \leftarrow 100 \leq T \leq 120.\\
\\
\neg occurs(A, S) \leftarrow at(S, T), \text{not } acceptable\_time(T).
\end{array}
$$

where the application of (one of the ground instances of) the last rule depends on the solutions to the constraints in the first two rules.

In practice, though, many problems can be solved by first generating a partial answer using (non-extended) ASP, and then completing the answer by solving a CSP. *It is remarkable that most of the examples of use of $\mathcal{AC}(\mathcal{C})$ from [9] are structured in this way.*

Problems that combine planning and scheduling are particularly good candidates for this approach. In Example 2 from [9] the authors solve such a problem by dividing the program in a planning and a scheduling part. The planning part is written with the usual ASP planning techniques, while the scheduling part introduces the assignment of (wall-clock) times to the steps of the plan and imposes suitable constraints. The problem can be easily solved using our approach by replacing the scheduling part from [9] with the following rules:

$$
\begin{array}{l}
cspdomain(\mathit{fd}).\\
\\
cspvar(at(S), 0, 1440).\\
\\
required(at(S0) \leq at(S1)) \leftarrow\\
\qquad next(S1, S0).\\
\\
required(at(S) - at(0) \leq 60) \leftarrow\\
\qquad goal(S).\\
\\
required(at(S0) - at(S1) \leq -20) \leftarrow\\
\qquad next(S1, S0),\\
\qquad occurs(go\_to(john, home), S0),\\
\qquad holds(at\_loc(john, \mathit{office}), S0).
\end{array}
$$

In fact, there is a whole subclass of $\mathcal{AC}(\mathcal{C})$ programs that can be automatically translated into "equivalent" ASP programs encoding suitable CSPs, as we show next.

Let us consider the class of $\mathcal{AC}_0$ programs without consistency restoring rules (from now on simply called $\mathcal{AC}_0$ programs). We will show that these programs can be translated into ASP programs whose extended answer sets correspond to the answer sets of the original programs.

We start by defining a translation from $\mathcal{AC}_0$ programs to ASP programs. For simplicity, we assume that all mixed predicates in Π have a single constraint parameter (extending to multiple constraint parameters is not difficult). Given a mixed atom $m(t_1, t_2, \ldots, t_{k-1}, t_k)$ where t_i's are terms, we call *functional representation* of the mixed atom the expression $m(t_1, t_2, \ldots, t_{k-1})$. For example, the functional representation of $at(S0, T0)$ is $at(S0)$.

Let Π be an $\mathcal{AC}_0$ program. The ASP-translation, Π', of Π is obtained from Π by:

- Adding a fact $cspdomain(\mathcal{D})$, where $\mathcal{D}$ is an appropriate constraint domain.
- Replacing each declaration $\#mixed\ m(p_1, p_2, \ldots, p_{k-1}, p_k)$ by a rule:

$$cspvar(m(X_1, X_2, \ldots, X_{k-1}), l, u) \leftarrow \\ p_1(X_1), p_2(X_2), \ldots, p_{k-1}(X_{k-1}).$$

  where l and u are the lower and upper bounds of constraint sort p_k and X_i's are variables.[9]
- Replacing each denial in the middle part of Π of the form:

$$\leftarrow \Gamma, c \tag{2}$$

  where c is the constraint literal, by a rule $required(\neg c') \leftarrow \Gamma'$, where c' is obtained by replacing every variable in c by the functional representation of the corresponding mixed atom and Γ' is the set of regular literals from Γ. Notice that $\neg c'$ can be typically simplified by replacing the comparison symbol in c' appropriately. For example, the denial, d_1:

$$\leftarrow goal(S), at(0, T1), at(S, T2), T2 - T1 > 60$$

  is replaced by the rule:

$$required(at(S) - at(0) \leq 60) \leftarrow goal(S).$$

In the rest of the discussion, the expression $c(d)$ will denote the constraint, c, from the body of a denial d of the form (2). For example, given denial d_1 above, $c(d_1)$ is $T2 - T1 > 60$.

Notice that the semantics of $\mathcal{AC}_0$, differently from that of ASP programs, is defined for possibly non-ground programs. In fact, because of the intended use of constraint atoms,

[9] The bounds can be easily extracted from the definition of sort p_k in Π. If the domain does not consist of a single interval, extra constraints can be added to the CSP definition, but we will assume a single interval for simplicity.

one would typically expect variables to be used for the constraint parameters of mixed literals and the corresponding arguments in the constraints. Therefore, *we assume that in Π variables are used for the constraint parameters of mixed predicates*.

We say that a *partial-ground rule* is a rule where the only variables occur as constraint parameters of the mixed predicates and as arguments of the constraint atoms. The *partial grounding* of a rule r is obtained by grounding all the variables of r, except those that occur as constraint parameters of mixed predicates. The set of partial-ground rules obtained this way is denoted by $pground(r)$.

We say that a ground denial d of the form (2) is *constraint-blocked* w.r.t. set of ground literals A if Γ is satisfied by A. The following is an important property of constraint-blocked ground denials.

Proposition 1. *Let Π be an $\mathcal{AC}_0$ program. For every answer set A of Π and every ground mixed-part denial d, if d is constraint-blocked w.r.t. A, then its constraint $c(d)$ is not satisfied.*

We say that a partial-ground denial d of the form (2) is *constraint-blocked* w.r.t. A if some ground instance of d is constraint-blocked w.r.t. A.

Theorem 4. *Every $\mathcal{AC}_0$ program Π can be translated into an ASP program whose extended answer sets are in one-to-one correspondence with the answer sets of Π.*

Proof. *Because of space restrictions, we omit the complete proof. However, we believe that it is useful to show the mapping that the proof is based upon.*

Let Π be an $\mathcal{AC}_0$ program and Π' be its translation, as described above. We define a mapping μ_Π from a set A of literals from the signature of Π into a pair $\langle A', \alpha \rangle$, where A' is a set of literals and α is an association of values to CSP variables. The mapping is defined as follows:

- *For each ground mixed atom $m(t_1, t_2, \ldots, t_{k-1}, t_k)$ from A, α maps the CSP variable denoted by $m(t_1, t_2, \ldots, t_{k-1})$ to value t_k.*
- *$A' \supseteq (A \setminus mixed(\Pi))$.*
- *A' includes an atom $cspdomain(\mathcal{D})$, where $\mathcal{D}$ is an appropriate constraint domain.*
- *For every ground mixed atom $m(t_1, \ldots, t_{k-1}, t_k) \in A$, A' includes an atom $cspvar(m(t_1, \ldots, t_{k-1}), l, u)$ where l and u are the lower and upper bounds of the constraint sort of the last argument of predicate m.*
- *For every denial d from the middle part of Π and every partial-grounding d^* of d that is constraint-blocked w.r.t. A, A' includes the atom $required(\neg c')$, where c' is obtained from the constraint atom c of d^* by replacing every variable in c by the functional representation of the corresponding mixed atom from $body(d^*)$.*

6 Conclusions

In this paper we have shown how ASP and constraint satisfaction can be integrated by viewing ASP as a specification language for constraint satisfaction problems. This approach allows a useful integration of the two paradigms without the need to extend

the language of ASP, without the need for ad-hoc solvers, and with support for global constraints. The last two features seem particularly appealing for the development of industrial-size applications. The paper also contains results showing that an important subclass of $\mathcal{AC}(\mathcal{C})$ programs can be automatically rewritten using our method.

Although space restrictions prevented us from discussing it here, our experiments have also shown that our technique produces programs that are significantly more compact and easy to understand than similar programs written in CLP alone, but with comparable performance. We plan to discuss this further in an extended paper.

References

1. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of ICLP-88. (1988) 1070–1080
2. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing (1991) 365–385
3. Marek, V.W., Truszczynski, M. The Logic Programming Paradigm: a 25-Year Perspective. In: Stable models and an alternative logic programming paradigm. Springer Verlag, Berlin (1999) 375–398
4. Balduccini, M., Gelfond, M., Nogueira, M.: Answer Set Based Design of Knowledge Systems. Annals of Mathematics and Artificial Intelligence (2006)
5. Baral, C., Chancellor, K., Tran, N., Joy, A., Berens, M.: A Knowledge Based Approach for Representing and Reasoning About Cell Signalling Networks. In: Proceedings of the European Conference on Computational Biology, Supplement on Bioinformatics. (2004) 15–22
6. 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
7. Brewka, G., Niemela, I., Syrjanen, T.: Logic Programs wirh Ordered Disjunction. **20**(2) (2004) 335–357
8. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an Integration of Answer Set and Constraint Solving. In: Proceedings of ICLP 2005. (2005)
9. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating Answer Set Programming and Constraint Logic Programming. Annals of Mathematics and Artificial Intelligence (2008) (to appear).
10. Smith, B.M.: 11. Handbook of Constraint Programming. In: Modelling. Elsevier (2006) 377–406
11. Katriel, I., van Hoeve, W.J.: 6. Handbook of Constraint Programming. In: Global Constraints. Elsevier (2006) 169–208
12. Jaffar, J., Lassez, J.L.: Constraint Logic Programming. In: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, ACM Press (Jan 1987) 111–119
13. Marriott, K., Stuckey, P.J., Wallace, M.: 12. Handbook of Constraint Programming. In: Constraint Logic Programming. Elsevier (2006) 409–452
14. SICStus: Sicstus Prolog Web Site (2008) http://www.sics.se/isl/sicstuswww/site/.
15. Mellarkod, V.S., Gelfond, M.: Enhancing ASP Systems for Planning with Temporal Constraints. In: LPNMR 2007. (May 2007) 309–314