A Translational Approach to Constraint Answer Set Solving

CHRISTIAN DRESCHER∗
Vienna University of Technology, Austria

TOBY WALSH
NICTA and University of New South Wales, Australia

submitted 26 January 2010; revised 12 May 2010; accepted 21 March 2010

Abstract

We present a new approach to enhancing Answer Set Programming (ASP) with Constraint Processing techniques which allows for solving interesting Constraint Satisfaction Problems in ASP. We show how constraints on finite domains can be decomposed into logic programs such that unit-propagation achieves arc, bound or range consistency. Experiments with our encodings demonstrate their computational impact.

KEYWORDS: answer set programming, constraint processing, decomposition

1 Introduction

Answer Set Programming (ASP; Baral 2003) has been put forward as a powerful paradigm to solve Constraint Satisfaction Problems (CSP) in (Niemelä 1999). Indeed, ASP has been shown to be a useful in various applications, among them planning (Lifschitz 1999), model checking (Heljanko and Niemelä 2003), and bioinformatics (Baral et al. 2004), and decision support for NASA shuttle controllers (Nogueira et al. 2001). It combines an expressive but simple modelling language with high-performance solving capacities. In fact, modern ASP solvers, such as *clasp* (Gebser et al. 2007a), compete with the best Boolean Satisfiability (SAT; Biere et al. 2009) solvers. An empirical comparison of the performance of ASP and traditional Constraint Logic Programming (CLP; Jaffar and Maher 1994) on solving CSP conducted by Dovier et al. (2005) shows ASP encodings to be more compact, more declarative, and highly competitive, but also revealed shortcomings: non-Boolean constructs, like resources or functions over finite domains, in particular global constraints, are more naturally modelled and efficiently handled by Constraint Processing (CP; Dechter 2003; Rossi et al. 2006) systems.

This led to the integration of CP techniques into ASP. Similar to Satisfiability

∗ Part of this work was performed when Christian Drescher was studying at the New University of Lisbon, Portugal.

Modulo Theories (SMT; Nieuwenhuis et al. 2006), the key idea of an *integrative* approach is to incorporate theory-specific predicates into propositional formulas, and extending an ASP solver's decision engine for a more high-level proof procedure. Recent work on combining ASP with CP was conducted in (Baselice et al. 2005; Mellarkod et al. 2008; Mellarkod and Gelfond 2008) and (Gebser et al. 2009). While Mellarkod and Gelfond both view ASP and CP solvers as blackboxes, Gebser et al. embed a CP solver into an ASP solver adding support for advanced backjumping and conflict-driven learning techniques. Balduccini (2009) and Järvisalo et al. (2009) cut ties to ad-hoc ASP and CP solvers, and principally support global constraints. Dal Palù et al. (2009) put further emphasis on handling constraint variables with large domains, and presented a strategy which only consider parts of the model that actively contribute in supporting constraint answer sets. However, each system has a subset of the following limitations: either they are tied to particular ASP and CP solvers, or the support for global constraints is limited, or communication between the ASP and CP solver is restricted.

This paper introduces a *translational* approach to Constraint Answer Set Solving rather than an integrative one. Motivated by the success of SAT-based constraint solvers, such as the award-winning system *Sugar* (Tamura et al. 2006), we show how to enhance ASP with Constraint Processing techniques through translation to ASP. A first study was conducted in Gebser et al. (2009) with the system *xpanda* for representing multi-valued propositions in ASP. One of the key contributions of our work is an investigation of constraint decomposition techniques in the new field of Constraint Answer Set Programming, illustrated on the popular *all-different* constraint. The resulting approach has been implemented in the new preprocessor *inca*. Empirical evaluation demonstrates its computational potential.

The remainder of this paper is organized as follows. We start by giving the background notions of ASP and Constraint Satisfaction. Various generic ASP encodings of constraints on finite domains and proofs of their properties are given in Section 3. In Section 4, we empirically evaluate our approach and compare to existing research. Section 5 draws conclusions.

2 Background

2.1 Answer Set Programming

A *(normal) logic program* over a set of primitive propositions $\mathcal{A}$ is a finite set of *rules* of the form

$$h \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$$

where $0 \leq m \leq n$ and $h, a_i \in \mathcal{A}$ is an *atom* for $1 \leq i \leq n$. A *literal* $\hat{a}$ is an atom a or its default negation $not\ a$. For a rule r, let $head(r) = h$ be the *head* of r and $body(r) = \{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$ the *body* of r. Furthermore, define $body(r)^+ = \{a_1, \ldots, a_m\}$ and $body(r)^- = \{a_{m+1}, \ldots, a_n\}$. The set of atoms occurring in a logic program Π is denoted by $atom(\Pi)$, and the set of bodies in Π is $body(\Pi) = \{body(r) \mid r \in \Pi\}$. For regrouping bodies sharing the same head a, define $body(a) = \{body(r) \mid r \in \Pi,\ head(r) = a\}$.

The semantics of a program is given by its answer sets. A set $X \subseteq \mathcal{A}$ is an *answer set* of a logic program Π over $\mathcal{A}$, if X is the $\subseteq$-minimal model of the *reduct* (Gelfond and Lifschitz 1988)

$$\Pi^X = \{head(r) \leftarrow body(r)^+ \mid r \in \Pi,\ body(r)^- \cap X = \emptyset\}.$$

The semantics of important extensions to normal logic programs, such as choice rules, integrity and cardinality constraints, is given through program transformations that introduce additional propositions (cf. Simons et al. 2002). A *choice rule* allows for the non-deterministic choice over atoms in $\{h_1, \ldots, h_k\}$ and has the following form:

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$$

An *integrity constraint*

$$\leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$$

is an abbreviation for a rule with an unsatisfiable head, and thus forbids its body to be satisfied in any answer set. A *cardinality constraint*

$$\leftarrow k\{\hat{a}_1, \ldots, \hat{a}_n\}$$

is interpreted as no answer set satisfies k literals of the set $\{\hat{a}_1, \ldots, \hat{a}_n\}$. It can be transformed into $\binom{n}{k}$ integrity constraints r such that $body(r) \subseteq \{\hat{a}_1, \ldots, \hat{a}_n\}$ and $|body(r)| = k$. Simons et al. provide a transformation that needs just $\mathcal{O}(nk)$ rules, introducing atoms $l(\hat{a}_i, j)$ to represent the fact that at least j of the literals with index $\geq i$, i.e. the literals in $\{\hat{a}_i, \ldots, \hat{a}_n\}$, are in a particular answer set candidate. Then, the cardinality constraint can be encoded by an integrity constraint $\leftarrow l(\hat{a}_1, k)$ and the three following rules, where $1 \leq i \leq n$ and $1 \leq j \leq k$:

$$l(\hat{a}_i, j) \leftarrow l(\hat{a}_{i+1}, j) \qquad l(\hat{a}_i, j+1) \leftarrow \hat{a}_i, l(\hat{a}_{i+1}, j) \qquad l(\hat{a}_i, 1) \leftarrow \hat{a}_i$$

Notice that both transformations are modular. Alternatively, modern ASP solvers also incorporate propagators for cardinality constraints that run in $\mathcal{O}(n)$.

2.2 Nogoods

We want to view inferences in ASP as unit-propagation on nogoods. Following Gebser et al. (2007b), inferences in ASP rely on atoms and program rules, which can be expressed by using atoms and bodies. Thus, for a program Π, the *domain* of Boolean assignments $\mathbf{A}$ is fixed to $dom(\mathbf{A}) = atom(\Pi) \cup body(\Pi)$.

Formally, a Boolean *assignment* $\mathbf{A}$ is a set $\{\sigma_1, \ldots, \sigma_n\}$ of *signed literals* σ_i for $1 \leq i \leq n$ of the form $\mathbf{T}a$ or $\mathbf{F}a$ where $a \in dom(\mathbf{A})$. $\mathbf{T}a$ expresses that a is assigned *true* and $\mathbf{F}a$ that it is *false* in $\mathbf{A}$. (We omit the attribute *Boolean* for assignments whenever clear from the context.) The complement of a signed literal σ is denoted by $\overline{\sigma}$, that is $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. In the context of ASP, a *nogood* (Dechter 2003) is a set $\delta = \{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment $\mathbf{A}$ such that $\delta \subseteq \mathbf{A}$. For a nogood δ, a signed literal $\sigma \in \delta$, and an assignment $\mathbf{A}$, we say that δ is *unit* and $\overline{\sigma}$ is *unit-resulting* if

Input: A set ∇ of nogoods, and an assignment $\mathbf{A}$.
Output: An extended assignment, and a status (either *conflict* or *success*).

repeat
  if $\delta \subseteq \mathbf{A}$ for some $\delta \in \nabla$ **then**
    return $(\mathbf{A}, \mathit{conflict})$;
  $\Sigma \leftarrow \{\delta \in \nabla \mid \delta \setminus \mathbf{A} = \{\sigma\}, \overline{\sigma} \notin \mathbf{A}\}$;
  if $\Sigma \neq \emptyset$ **then let** $\sigma \in \delta \setminus \mathbf{A}$ for some $\delta \in \Sigma$ **in**
    $\mathbf{A} \leftarrow \mathbf{A} \cup (\overline{\sigma})$;
until $\Sigma = \emptyset$;
return $(\mathbf{A}, \mathit{success})$;

Fig. 1. The unit-propagation algorithm.

$\delta \setminus \mathbf{A} = \{\sigma\}$. Let $\mathbf{A^T} = \{a \in \mathit{dom}(\mathbf{A}) \mid \mathbf{T}a \in A\}$ the set of true propositions and $\mathbf{A^F} = \{a \in \mathit{dom}(\mathbf{A}) \mid \mathbf{F}a \in A\}$ the set of false propositions. A *total* assignment, that is $\mathbf{A^T} \cup \mathbf{A^F} = \mathit{dom}(\mathbf{A})$ and $\mathbf{A^F} \cup \mathbf{A^T} = \emptyset$, is a *solution* for a set Δ of nogoods if $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta$.

As shown in Lee (2005), the answer sets of a logic program Π correspond to the models of the completion of Π that satisfy the loop formulas of all non-empty subsets of $\mathit{atom}(\Pi)$. For $\beta = \{a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n\} \in \mathit{body}(\Pi)$, define

$$\Delta_\beta = \left\{ \begin{array}{l} \{\mathbf{T}a_1, \ldots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \ldots \mathbf{F}a_n, \mathbf{F}\beta\}, \\ \{\mathbf{F}a_1, \mathbf{T}\beta\}, \ldots, \{\mathbf{F}a_m, \mathbf{T}\beta\}, \{\mathbf{T}a_{m+1}, \mathbf{T}\beta\}, \ldots, \{\mathbf{T}a_n, \mathbf{T}\beta\} \end{array} \right\}.$$

Intuitively, the nogoods in Δ_β enforce the truth of body β iff all its literals are satisfied. For an atom $a \in \mathit{atom}(\Pi)$ with $\mathit{body}(a) = \{\beta_1, \ldots, \beta_k\}$, let

$$\Delta_a = \left\{ \begin{array}{l} \{\mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k, \mathbf{T}a\}, \\ \{\mathbf{T}\beta_1, \mathbf{F}a\}, \ldots, \{\mathbf{T}\beta_k, \mathbf{F}a\} \end{array} \right\}.$$

Then, the solutions for $\Delta_\Pi = \bigcup_{\beta \in \mathit{body}(\Pi)} \Delta_\beta \cup \bigcup_{a \in \mathit{atom}(\Pi)} \Delta_a$ correspond to the models of the completion of Π. Loop formulas, expressed in the set of nogoods Λ_Π, have to be added to establish full correspondence to the answer sets of Π. Typically, solutions for $\Delta_\Pi \cup \Lambda_\Pi$ are computed by applying *Conflict-Driven Nogood Learning* (CDNL; Gebser et al. 2007b). This combines search and propagation by recursively assigning the value of a proposition and using unit-propagation (Fig. 1) to determine logical consequences of an assignment (Mitchell 2005).

Example 1
Consider the set of nogoods $\nabla = \{\{\mathbf{T}a_1, \mathbf{F}a_2, \mathbf{T}a_3, \mathbf{T}a_4\}, \{\mathbf{F}a_1, \mathbf{T}a_4\}, \{\mathbf{F}a_3, \mathbf{T}a_4\}\}$ and the assignment $\mathbf{A} = \{\mathbf{T}a_4\}$. Unit-propagation extends $\mathbf{A}$ by $\{\mathbf{T}a_1, \mathbf{T}a_2, \mathbf{T}a_3\}$.

2.3 Constraint Satisfaction and Consistency

A *Constraint Satisfaction Problem* is a triple (V, D, C) where V is a set of *variables* $V = \{v_1, \ldots, v_n\}$, D is a set of finite *domains* $D = \{D_1, \ldots, D_n\}$ such that each variable v_i has an associated domain $\mathit{dom}(v_i) = D_i$, and C is a set of *constraints*. Following Rossi et al. (2006), a constraint c is a pair (R_S, S) where R_S is a k-ary

relation on the variables in $S \subseteq V^k$, called the *scope* of c. In other words, R_S is a subset of the Cartesian product of the domains of the variables in S. To access the relation and the scope of c define $\mathit{range}(c) = R_S$ and $\mathit{scope}(c) = S$. For a *(constraint variable) assignment* $A : V \to \bigcup_{v \in V} \mathit{dom}(v)$ and a constraint $c = (R_S, S)$ with $S = (v_1, \ldots, v_k)$, define $A(S) = (A(v_1), \ldots, A(v_k))$, and call c *satisfied* if $A(S) \in \mathit{range}(c)$. A binary constraint c has $|\mathit{scope}(c)| = 2$. For example, $v_1 \neq v_2$ ensures that v_1 and v_2 take different values. A global (or n-ary) constraint c has parametrized scope. For example, the *all-different* constraint ensures that a set of variables, $\{v_1, \ldots, v_n\}$ take all different values. This can be decomposed into $O(n^2)$ binary constraints, $v_i \neq v_j$ for $i < j$. However, as we shall see, such decomposition can hinder inference. An assignment A is a *solution* for a CSP iff it satisfies all constraints in C.

Constraint solvers typically use *backtracking search* to explore the space of partial assignments, and prune it by applying propagation algorithms that enforce a *local consistency* property on the constraints after each assignment. A binary constraint c is called *arc consistent* iff when a variable $v_1 \in \mathit{scope}(c)$ is assigned any value $d_1 \in \mathit{dom}(v_1)$, there exists a consistent value $d_2 \in \mathit{dom}(v_2)$ for the other variable v_2. An n-ary constraint c is *hyper-arc consistent* or *domain consistent* iff when a variable $v_i \in \mathit{scope}(c)$ is assigned any value $d_i \in \mathit{dom}(v_i)$, there exist compatible values in the domains of all the other variables $d_j \in \mathit{dom}(v_j)$ for all $1 \leq j \leq n$, $j \neq i$ such that $(d_1, \ldots, d_n) \in \mathit{range}(c)$.

The concepts of bound and range consistency are defined for constraints on ordered intervals. Let $\mathit{min}(D_i)$ and $\mathit{max}(D_i)$ be the minimum value and maximum value of the domain D_i. A constraint c is *bound consistent* iff when a variable v_i is assigned $d_i \in \{\mathit{min}(\mathit{dom}(v_i)), \mathit{max}(\mathit{dom}(v_i))\}$ (i.e. the minimum or maximum value in its domain), there exist compatible values between the minimum and maximum domain value for all the other variables in the scope of the constraint. Such an assignment is called a *bound support*. A constraint is *range consistent* iff when a variable is assigned any value in its domain, there exists a bound support. Notice that range consistency is in between domain and bound consistency, where domain consistency is the strongest of the four formalisms.

2.4 Constraint Answer Set Programming

Following Gebser et al. (2009), a *constraint logic program* Π is defined as logic programs over an extended alphabet distinguishing regular and constraint atoms, denoted by $\mathcal{A}$ and $\mathcal{C}$, respectively, such that $\mathit{head}(r) \in \mathcal{A}$ for each $r \in \Pi$. Constraint atoms are identified with constraints via a function $\gamma : \mathcal{C} \to C$, and furthermore, define $\gamma(C') = \{\gamma(c) \mid c \in C'\}$ for $C' \subseteq C$. For a (constraint variable) assignment A define the set of constraints satisfied by A as $\mathit{sat}_C(A) = \{c \mid A(\mathit{scope}(c)) \in \mathit{range}(c),\ c \in C\}$, and the *constraint reduct* as

$$\Pi^A = \{\mathit{head}(r) \leftarrow \mathit{body}(r)|_{\mathcal{A}} \mid r \in \Pi, \\ \gamma(\mathit{body}(r)^+|_{\mathcal{C}}) \subseteq \mathit{sat}_C(A),\ \gamma(\mathit{body}(r)^-|_{\mathcal{C}}) \cap \mathit{sat}_C(A) = \emptyset\}.$$

Then, a set $X \subseteq \mathcal{A}$ is a *constraint answer set* of Π with respect to A, if X is an answer set of Π^A.

In the translational approach to Constraint Answer Set Solving, a constraint logic program is compiled into a (normal) logic program by adding an ASP decomposition of all constraints comprised in the constraint logic program. The constraint answer sets can then be obtained by applying the same algorithms as for calculating answer sets, e.g. CDNL. Since all variables will be shared between constraints, nogood learning techniques as in CDNL exploit constraint interdependencies. This can improve propagation between constraints.

3 Encoding Constraint Answer Set Programs

In this section we explain how to translate constraint logic programs with multi-valued propositions into a (normal) logic program. There are a number of choices of how to encode constraints on multi-valued propositions, e.g. a constraint variable v, taking values out of a pre-defined finite domain, $dom(v)$. In what follows, we assume $dom(v) = [1, d]$ for all $v \in V$ to save the reader from multiple superscripts.

3.1 Direct Encoding

A popular choice is called the *direct encoding* (Walsh 2000). In the direct encoding, a propositional variable $e(v, i)$, representing $v = i$, is introduced for each value i that can be assigned to the constraint variable v. Intuitively, the proposition $e(v, i)$ is true if v takes the value i, and false if v takes a value different from i. For each v, the truth-assignments of atoms $e(v, i)$ are encoded by a choice rule (1). Furthermore, there is an integrity constraint (2) to ensure that v takes at least one value, and a cardinality constraint (3) that ensures that v takes at most one value.

$$
\begin{array}{ll}
(1) & \{e(v,1), \ldots, e(v,d)\} \leftarrow \\
(2) & \leftarrow \mathit{not}\ e(v,1), \ldots, \mathit{not}\ e(v,d) \\
(3) & \leftarrow 2\ \{e(v,1), \ldots, e(v,d)\}
\end{array}
$$

In the direct encoding, each forbidden combination of values in a constraint is expressed by an integrity constraint. On the other hand, when a relation is represented by allowed combinations of values, all forbidden combinations have to be deduced and translated to integrity constraints. Unfortunately, the direct encoding of constraints hinders propagation:

Theorem 1 (*Walsh 2000*)
Enforcing arc consistency on the binary decomposition of the original constraint prunes more values from the variables domain than unit-propagation on its direct encoding.

3.2 Support Encoding

The *support encoding* has been proposed to tackle this weakness (Gent 2002). A *support* for a constraint variable v to take the value i across a constraint c is the

set of values $\{i_1, \ldots, i_m\} \subseteq dom(v')$ of another variable in $v' \in scope(c) \setminus \{v\}$ which allow $v = i$, and can be encoded as follows, extending (1–3):

$$\leftarrow e(v, i), not\ e(v', i_1), \ldots, not\ e(v', i_m)$$

This integrity constraint can be read as whenever $v = i$, then at least one of its supports must hold. In the support encoding, for each constraint c there is one support for each pair of distinct variables $v, v' \in scope(c)$, and for each value i.

Theorem 2 (*Gent 2002*)
Unit-propagation on the support encoding enforces arc consistency on the binary decomposition of the original constraint.

We illustrate this approach on an encoding of the global *all-different* constraint. For variables v, v' and value i it is defined by the following $\mathcal{O}(n^2 d)$ integrity constraints:

$$\leftarrow e(v, i), not\ e(v', 1), \ldots, not\ e(v', i-1), not\ e(v', i+1), \ldots, not\ e(v', d)$$

To keep the encoding small, we make use of the following equivalence (e)

$$e(v', i) \equiv not\ e(v', 1), \ldots, not\ e(v', i-1), not\ e(v', i+1), \ldots, not\ e(v', d)$$

covered by (2–3) and get

$$\leftarrow e(v, i), e(v', i).$$

Observe, that this is also the direct encoding of the binary decomposition of the global *all-different* constraint. However, this observation does not hold in general for all constraints. As discussed in the Background section of this paper, we can express above condition as $\mathcal{O}(d)$ cardinality constraints:

$$(4) \quad \leftarrow 2\ \{e(v_1, i), \ldots, e(v_n, i)\}$$

Corollary 1
Unit-propagation on (1–4) enforces arc consistency on the binary decomposition of the global *all-different* constraint in $\mathcal{O}(nd^2)$ down any branch of the search tree.

Proof
From the definition of cardinality constraints, (4) ensure that for all distinct $v, v' \in scope(c)$ any value i is not taken by both v and v'. These integrity constraints correspond to the support encoding of the global *all-different* constraint since (2–3) cover the equivalence (e). By Theorem 2, unit-propagation on this support encoding enforces arc consistency on the binary decomposition of the *all-different* constraint.

For each of the n variables, there are $\mathcal{O}(d)$ nogoods resulting from (1–3) that can be woken $\mathcal{O}(d)$ times down any branch of the search tree. Each propagation requires $\mathcal{O}(1)$ time. Rules (1–3) therefore take $\mathcal{O}(nd^2)$ down any branch of the search to propagate. There are $\mathcal{O}(nd)$ nogoods resulting from (4) that each take $\mathcal{O}(1)$ time to propagate down any branch of the search tree. The total running time is given by $\mathcal{O}(nd^2) + \mathcal{O}(nd) = \mathcal{O}(nd^2)$. □

3.3 Range Encoding

In the *range encoding*, a propositional variable $r(v, l, u)$ is introduced for all $[l, u] \subseteq [1, d]$ to represent whether the value of v is between l and u. For each range $[l, u]$, the following $\mathcal{O}(nd^2)$ rules encode $v \in [l, u]$ whenever it is safe to assume that $v \notin [1, l-1]$ and $v \notin [u+1, d]$, and enforce a consistent set of ranges such that $v \in [l, u] \Rightarrow v \in [l-1, u] \wedge v \in [l, u+1]$:

$$
\begin{array}{lll}
(5) & r(v,l,u) \leftarrow & not\ r(v,1,l-1), not\ r(v,u+1,d) \\
(6) & \leftarrow & r(v,l-1,u), not\ r(v,l,u) \\
(7) & \leftarrow & r(v,l,u+1), not\ r(v,l,u)
\end{array}
$$

Constraints are encoded into integrity constraints representing conflict regions. When the combination $v_1 \in [l_1, u_1], \ldots, v_n \in [l_n, u_n]$ violates the constraint, the following rule is added:

$$\leftarrow r(v_1, l_1, u_1), \ldots, r(v_n, l_n, u_n)$$

Theorem 3
Unit-propagation on the range encoding enforces range consistency on the original constraint.

Proof
Suppose we have a set of ranges on the domains of the constraint variables in which no unit-propagation is possible and no domain is empty. Consider any constraint variables $v_1, \ldots, v_i, \ldots, v_n$ and value d_i such that there is no bound support in $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots v_n$ for $v_i = d_i$, i.e. there are no compatible values for the other variables v_j distinct from v_i where $v_j \in [l_j, u_j]$. Hence, all instantiations such that $v_i = d_i$ are in a conflict region $v_1 \in [l_1, u_1] \subseteq [l'_1, u'_1], \ldots, v_n \in [l_n, u_n] \subseteq [l'_n, u'_n]$. For each v_j we have $\mathbf{T}r(v_j, l_j, u_j)$ representing $v_j \in [l_j, u_j]$. Then, the binary nogoods $\{\mathbf{T}r(v_j, l_j, u_j), \mathbf{F}r(v_j, l_j - 1, u_j)\}$ and $\{\mathbf{T}r(v_j, l_j, u_j), \mathbf{F}r(v_j, l_j, u_j + 1)\}$ resulting from (6) and (7) are unit, and eventually we get $\mathbf{T}r(v_j, l'_j, u'_j)$ for $[l_j, u_j] \subseteq [l'_j, u'_j]$. But then the nogood $\{\mathbf{T}r(v_1, l'_1, u'_1), \ldots, \mathbf{T}r(v_n, l'_n, u'_n)\}$ encoding the conflict region is unit and forces $\mathbf{F}r(v_i, l'_i, u'_i)$ representing $v_i \notin [l'_i, u'_i]$. By nogoods resulting from (6) and (7) we get d_i is not in the domain of v_i, and the domains are bound consistent as required. Since at least one value must be in each domain, encoded in (5), we have a set of non-empty domains which are range consistent. □

A propagator for the global *all-different* constraint that enforces range consistency pruning Hall intervals has been proposed in Leconte (1996) and encoded to SAT in Bessière et al. (2009). An interval $[l, u]$ is a *Hall interval* iff $|\{v \mid dom(v) \subseteq [l, u]\}| = u - l + 1$. In other words, a Hall interval of size k completely contains the domains of k variables. Observe that in any bound support, the variables whose domains are contained in the Hall interval consume all values within the Hall interval, whilst any other variable must find their support outside the Hall interval.

Example 2
Consider the global *all-different* constraint over the variables $\{v_1, v_2, v_3, v_4\}$ with $dom(v_1) = \{2,3\}$, $dom(v_2) = \{1,2,4\}$, $dom(v_3) = \{2,3\}$, $dom(v_4) = \{1,2,3,4\}$. $[2,3]$ is a Hall interval of size 2 as the domain of 2 variables, v_1 and v_3, is completely contained in it. Therefore we can remove $[2,3]$ from the domains of all the other variables. This leaves v_2 and v_4 with a domain containing values 1 and 4.

The following decomposition of the global *all-different* constraint will permit us to achieve range consistency via unit propagation. It ensures that no interval $[l,u]$ can contain more variables than its size.

$$(8) \quad \leftarrow u - l + 2\ \{r(v_1, l, u), \ldots, r(v_n, l, u)\}$$

This simple decomposition can simulate a complex propagation algorithm like Leconte's with a similar overall complexity of reasoning.

Corollary 2
Unit-propagation on (5–8) enforces range consistency on the global *all-different* constraint in $\mathcal{O}(nd^3)$ down any branch of the search tree.

Proof
Clearly, the cardinality constraints (8) reflect all conflict regions such that no Hall interval $[l,u]$ can contain $u - l + 2$ variables, that are more variables than its size. Hence, (8) is a range encoding of the global *all-different* constraint. By Theorem 3, unit-propagation on this encoding enforces range consistency on the global *all-different* constraint.

There are $\mathcal{O}(nd^2)$ nogoods resulting from (5–7) that can be woken $\mathcal{O}(d)$ times down any branch of the search tree. Each propagation requires $\mathcal{O}(1)$ time. Rules (5–7) therefore take $\mathcal{O}(nd^3)$ down any branch of the search to propagate. There are $\mathcal{O}(nd^2)$ nogoods resulting from (8) that each take $\mathcal{O}(1)$ time to propagate down any branch of the search tree. The total running time is given by $\mathcal{O}(nd^3) + \mathcal{O}(nd^2) = \mathcal{O}(nd^3)$. □

3.4 Bound Encoding

A last encoding is called the *bound encoding* (Crawford and Baker 1994). In the bound encoding, a propositional variable $b(v,i)$ is introduced for each value i to represent that the value of v is bounded by i. That is, $v \leq i$ if $\mathbf{T}b(v,i)$, and $v > i$ if $\mathbf{F}b(v,i)$. Similar to the direct encoding, for each v, the truth-assignments of atoms $b(v,i)$ are encoded by a choice rule (9). In order to ensure that assignments represent a consistent set of bounds, the condition $v \leq i \Rightarrow v \leq i+1$ is posted as integrity constraints (10). Another integrity constraint (11) encodes $v \leq d$, that at least one value must be assigned to v:

$$(9) \quad \{b(v,1), \ldots, b(v,d)\} \leftarrow$$
$$(10) \quad \leftarrow b(v,i), not\ b(v,i+1) \qquad \forall i \in [1, d-1]$$
$$(11) \quad \leftarrow not\ b(v,d)$$

Constraints are encoded into integrity constraints representing conflict regions similar to the range encoding. When all combinations in the region

$$l_1 < v_1 \leq u_1, \ldots, l_n < v_n \leq u_n$$

violate a constraint, the following rule is added:

$$\leftarrow b(v_1, u_1), \ldots, b(v_n, u_n), not\ b(v_1, l_1), \ldots, not\ b(v_n, l_n)$$

Theorem 4
Unit-propagation on the bound encoding enforces bound consistency on the original constraint.

Proof
Suppose we have a set of bounds on the domains of the constraint variables in which no unit-propagation is possible and no domain is empty. Consider any constraint variable v_i such that if v_i is assigned its minimum domain value $l_i{+}1$ or its maximum domain value u_i there are no compatible values of the other constraint variables $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ between their minimum $l_1{+}1, \ldots, l_{i-1}{+}1, l_{i+1}{+}1, \ldots, l_n{+}1$ and their maximum domain values $u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n$, respectively. First, we analyse the case $v_i = u_i$, that is, all instantiations such that $v_i = u_i$ are in a conflict region $l'_1 \leq l_1 < v_1 \leq u_1 \leq u'_1, \ldots, l'_n \leq l_n < v_n \leq u_n \leq u'_n$. For each v_j we have $\mathbf{F}b(v_j, l_j)$ and $\mathbf{T}b(v_j, u_j)$, representing $v_j > l_j$ and $v_j \leq u_j$. Then, the binary nogoods $\{\mathbf{T}b(v_j, l_j - 1), \mathbf{F}b(v_j, l_j)\}$ and $\{\mathbf{T}b(v_j, u_j), \mathbf{F}b(v_j, u_j + 1)\}$ resulting from (10) are unit, and eventually we get $\mathbf{F}b(v_j, l'_j)$ for $l'_j \leq l_j$ as well as $\mathbf{T}b(v_j, u'_j)$ for $u'_j \geq u_j$. But then the nogood $\{\mathbf{F}b(v_1, l'_1), \mathbf{T}b(v_1, u'_1), \ldots, \mathbf{F}b(v_n, l'_n), \mathbf{T}b(v_n, u'_n)\}$ encoding the conflict region is unit and forces $\mathbf{T}b(v_i, l'_i)$ representing $v_i \leq l'_i$. Since $l'_i < u_i$ and by the nogoods resulting from (10) we get u_i is not in the domain of v_i.

The second case, where v_i is assigned its minimum domain value $l_i + 1$, is symmetric, and we conclude that the domains are bound consistent as required. Since at least one value must be in each domain, resulting from (11), we have a set of non-empty domains which are bound consistent. □

In order to get a representation of the global *all-different* constraint that can only prune bounds, the bound encoding for variables is linked to (8) as follows:

$$\begin{array}{ll} (12) & r(v, l, u) \leftarrow not\ b(v, l - 1), b(v, u) \\ (13) & \leftarrow r(v, l, u), b(v, l - 1) \\ (14) & \leftarrow r(v, l, u), not\ b(v, u) \end{array}$$

Corollary 3
Unit-propagation on (8–14) enforces bound consistency on the global *all-different* constraint in $\mathcal{O}(nd^2)$ down any branch of the search tree.

Proof
This result follows from Corollary 2 and Theorem 4. Observe that the decompositions for range and bound consistency both encode the same conflict regions.

For each of the n variables, there are $\mathcal{O}(d)$ nogoods resulting from (9–11) that can be woken $\mathcal{O}(d)$ times down any branch of the search tree. Each propagation requires $\mathcal{O}(1)$ time. Rules (9–11) therefore take $\mathcal{O}(nd^2)$ down any branch of the search to propagate. Furthermore, there are $\mathcal{O}(nd^2)$ nogoods resulting from (8) and (12–14) that each take $\mathcal{O}(1)$ time to propagate down any branch of the search tree. The total running time is given by $\mathcal{O}(nd^2)$. □

Note that an upper bound h can be posted on the size of Hall intervals. The resulting encoding with only those cardinality constraints (5) for which $u - l + 1 \leq h$ detects Hall intervals of size at most h, and therefore enforces a weaker level of consistency.

4 Experiments

To evaluate our decompositions, we conducted experiments on encodings[1] of CSP containing *all-different* and *permutation* constraints. The global *permutation* constraint is a special case of *all-different* when the number of variables is equal to the number of all their possible values. A decomposition of *permutation* extends (4) by

$$\leftarrow \mathit{not}\ e(v_1, i), \ldots, \mathit{not}\ e(v_n, i)$$

or (8) by the following rule where $1 \leq l \leq u \leq k$:

$$\leftarrow d - u + l\ \{\mathit{not}\ r(v_1, l, u), \ldots, \mathit{not}\ r(v_n, l, u)\}$$

This can increase propagation. Our translational approach to Constraint Answer Set Solving has been implemented within the prototypical preprocessor *inca*. Although our semantics is propositional, *inca* compiles constraint logic programs with first-order variables, function symbols, and aggregates, etc. in linear time and space, such that the logic program can be obtained by a *grounding* process. Experiments consider *inca*[1] in different settings using different decompositions. We denote the support encoding of the global constraints by S, the bound encoding of the global constraints by B, and the range encoding of the global constraints by R. To explore the impact of small Hall intervals, we also tried B_k and R_k, an encoding of the global constraints with only those cardinality constraints (8) for which $u - l + 1 \leq k$. The consistency achieved by B_k and R_k is therefore weaker than full bound and range consistency, respectively.

We also include the integrated systems *clingcon*[1] (0.1.2), and *ezcsp*[2] (1.6.9) in our empirical analysis. While *clingcon* extends the ASP system *clingo*[1] (2.0.2) with the generic constraint solver *gecode*[3] (2.2.0), *ezcsp* combines the grounder *gringo*[1] (2.0.3) and ASP solver *clasp*[1] (1.3.0) with *sicstus*[4] (4.0.8) as a constraint solver.

[1] `http://potassco.sourceforge.net/`
[2] `http://krlab.cs.ttu.edu/~marcy/ezcsp/`
[3] `http://www.gecode.org/`
[4] `http://www.sics.se/sicstus/`

Table 1. Data on time and space for selected translations.

			S		B		R	
	n	time	atoms	rules	atoms	rules	atoms	rules
	12	0.1	236	151	1,829	1,907	1,061	2,423
PHP	14	0.1	304	177	2,876	2,981	1,630	3,877
	16	0.1	380	203	4,263	4,399	2,375	5,823
	8	0.3	655	36,525	1,297	41,565	3,217	38,237
*QG*5	10	0.9	1,219	109,066	2,421	118,946	7,121	111,446
	12	2.3	2,039	267,643	4,057	284,755	13,849	270,067
	4	0.6	564	74,945	941	78,848	4,135	81,174
DW_n	6	3.0	1,130	363,115	1,983	373,002	12,581	381,724
	8	9.2	1,888	1,122,549	3,409	1,142,132	28,355	1,163,810

Note that *clingo* stands for *clasp* on *gringo* and combines both systems in a monolithic way. Since *inca* is a pure preprocessor, we select the ASP system *clingo* (2.0.3) as its backend to provide a representative comparison with *clingcon* and *ezcsp*. The behaviour of *xpanda* is simulated by setting S and, therefore, is not considered in our study. We also do not separate time spend on grounding and solving the problem, since the grounder's share of the overall runtime is generally insignificant on our benchmarks.

To compare the performance of Constraint Answer Set solvers against pure CP systems, we also report results of *gecode* (3.2.0). Its heuristic for variable selection was set to a smallest domain as in *clingcon*. All experiments were run on a 2.00 GHz PC under Linux. We report results in seconds, where each run was limited to 600 s time and 1 GB RAM.

Space Complexity. Data on the size of selected translations shown in Table 1 confirms our theoretical results. For *PHP* and DW_n instances (description follows), the number of atoms in the support (bound/range) encoding is bounded by $\mathcal{O}(n^2)$ ($\mathcal{O}(n^3)$). The number of rules is $\mathcal{O}(n)$ ($\mathcal{O}(n^3)$) for *PHP*, $\mathcal{O}(n^3)$ ($\mathcal{O}(n^3)$) for DW_n due to constraints represented in the direct encoding. An $n \times n$ table is modelled in $QG5$, raising the number of atoms to $\mathcal{O}(n^3)$ ($\mathcal{O}(n^4)$), and rules to $\mathcal{O}(n^4)$ ($\mathcal{O}(n^4)$).

4.1 Pigeon Hole Problems

The *Pigeon Hole Problem* (PHP) is to show that it is impossible to put n pigeons into $n-1$ holes if each pigeon must be put into a distinct hole. Clearly, our bound and range decompositions are faster compared to weaker encodings (see Table 2). On such problems, detecting large Hall intervals is essential.

4.2 Quasigroup Completion

A *quasigroup* is an algebraic structure $(Q, \cdot)$, where Q is a set and $\cdot$ is a binary operation on Q such that for every pair of elements $a, b \in Q$ there exist unique

Table 2. Runtime results in seconds for PHP.

n	S	B_1	B_2	B_3	B	R_3	R	*ezcsp*	*clingcon*	*gecode*
10	5.4	0.7	0.5	0.1	**0.0**	0.2	**0.0**	1.8	1.4	0.9
11	46.5	3.5	1.5	1.0	**0.0**	1.9	**0.0**	16.7	15.2	9.0
12	105.0	14.8	7.1	3.9	**0.0**	2.6	0.1	183.9	172.5	104.1
13	—	91.4	68.6	25.4	0.1	30.4	**0.0**	—	—	—
14	—	—	350.1	125.0	**0.0**	196.9	0.1	—	—	—
15	—	—	—	—	**0.1**	—	**0.1**	—	—	—
16	—	—	—	—	**0.1**	—	**0.1**	—	—	—

Table 3. Average times over 100 runs on QCP. Timeouts are given in parenthesis.

%	S	B_3	B	R_3	R	*ezcsp*	*clingcon*	*gecode*	$gecode_{BC}$
10	2.6	5.0	8.2	6.0	7.3	29.6 (7)	9.7 (4)	2.2 (4)	0.5 (1)
20	2.4	5.0	8.0	6.2	7.2	21.3 (20)	6.2 (5)	5.0 (4)	0.9 (3)
30	2.3	4.8	7.9	6.1	7.1	10.3 (30)	12.9 (13)	2.9 (13)	1.1 (5)
35	2.3	4.8	7.9	6.1	7.0	21.6 (24)	11.2 (17)	14.1 (13)	6.2 (7)
40	2.3	4.7	7.8	6.0	6.9	51.6 (29)	23.1 (22)	11.7 (20)	5.7 (9)
45	2.3	4.7	7.8	5.9	6.8	36.3 (35)	14.7 (28)	17.7 (25)	6.3 (13)
50	2.3	4.6	7.7	5.9	6.8	36.1 (50)	21.2 (37)	25.1 (32)	6.3 (18)
55	2.3	4.5	7.6	5.8	6.7	61.4 (51)	24.4 (44)	19.6 (41)	30.9 (29)
60	2.2	4.4	7.5	5.6	6.6	60.2 (63)	31.4 (56)	36.0 (51)	27.2 (35)
70	2.2	4.2	7.1	5.1	6.0	70.0 (66)	30.2 (50)	28.0 (45)	17.0 (27)
80	2.1	4.0	6.7	4.7	5.5	16.2 (18)	4.2 (18)	17.2 (13)	7.0 (7)
90	2.1	4.0	6.7	4.7	5.5	1.4	2.6 (1)	0.4 (1)	3.2

elements $x, y \in Q$ which solve the equations $a \cdot x = b$ and $y \cdot a = b$. The *order* n of a quasigroup is defined by the number of elements in Q. A quasigroup can be represented by an $n \times n$-multiplication table, where for each pair a, b the table gives the result of $a \cdot b$, and it defines a *Latin square*. This means that each element of Q occurs exactly once in each row and each column of the table. The *Quasigroup Completion Problem* (QCP) is to determine whether a partially filled table can be completed in such a way that a multiplication table of a quasigroup is obtained. Randomly generated QCP has been proposed as a benchmark domain for CP systems by Gomes and Selman (1997) since it combines the features of purely random problems and highly structured problems. Table 3 compares the runtime for solving QCP problems of size 20×20 where the first column gives the percentage of preassigned values. We included *gecode* with algorithms that enforce bound and domain consistency, denoted as $gecode_{BC}$ and $gecode_{DC}$ (not shown), in the experiments. Our analysis exhibits phase transition behaviour of the systems *ezcsp*, *clingcon*, *gecode*, and $gecode_{BC}$, while our Boolean encodings and $gecode_{DC}$ solve all problems within seconds. Interestingly, learning constraint interdependencies as in our approach is sufficient to tackle QCP. In fact, most of the time for S, B_k, R_k is spent on grounding, but not for solving the actual problem.

Table 4. Runtime results in seconds for QEP.

	n	S	B_1	B_3	B	R_3	R	*ezcsp*	*clingcon*	*gecode*
QG1	7	1.7	1.7	1.7	1.7	1.7	1.6	65.0	189.8	**0.6**
	8	19.0	5.9	**4.7**	19.8	6.4	**4.7**	—	—	—
	9	—	139.4	152.0	234.6	**27.6**	466.9	—	—	—
QG2	7	1.7	1.7	1.7	1.8	1.7	1.8	46.1	1.5	**1.2**
	8	46.6	**9.6**	10.6	37.7	11.7	14.8	—	—	—
	9	—	246.0	**55.7**	88.3	119.7	213.4	—	—	—
QG3	7	0.2	0.2	0.2	0.3	0.2	0.3	3.2	1.0	**0.0**
	8	0.4	0.4	0.5	0.5	0.5	0.5	4.3	9.0	**0.2**
	9	10.2	**7.4**	9.5	16.5	11.0	12.8	—	—	18.2
QG4	7	0.2	0.2	0.2	0.3	0.3	0.3	2.8	0.7	**0.1**
	8	0.5	0.6	0.7	0.9	0.8	0.7	27.9	36.8	**0.3**
	9	1.3	**1.0**	2.1	3.0	1.1	0.9	442.1	288.8	3.7
QG5	8	0.4	0.4	0.4	0.5	0.4	0.4	6.9	5.3	**0.0**
	9	0.7	0.8	0.8	0.9	0.8	0.8	249.2	—	**0.0**
	10	1.6	1.5	1.6	1.9	1.6	1.6	—	—	**0.2**
	11	2.1	2.2	2.4	3.4	2.8	2.4	—	—	**0.8**
	12	27.0	**6.2**	9.1	12.4	8.4	10.4	—	—	16.4
QG6	8	0.4	0.4	0.5	0.5	0.5	0.4	0.8	—	**0.0**
	9	0.7	0.7	0.8	0.9	0.8	0.8	1.2	—	**0.0**
	10	1.2	1.4	1.5	1.8	1.6	1.5	10.5	—	**0.1**
	11	2.7	2.8	4.0	4.2	3.9	4.8	125.5	—	**1.2**
	12	32.0	**12.9**	25.6	36.4	25.7	50.6	—	—	24.6
QG7	8	0.4	0.4	0.4	0.6	0.5	0.5	1.1	—	**0.1**
	9	**0.7**	1.0	1.2	1.7	1.2	1.4	9.1	—	0.9
	10	6.7	**3.2**	5.2	8.0	4.7	4.6	—	—	22.0

4.3 Quasigroup Existence

The *Quasigroup Existence Problem* (QEP) is to determine the existence of certain interesting classes of quasigroups. We follow Fujita et al. (1993) and look at problems QG1 to QG7 that were target to open questions in finite mathematics. We represent them in the direct encoding which weakens the overall consistency. Furthermore, we add the axiom $a \cdot n \geq a - 1$ where n is the order of the desired quasigroup, to avoid some symmetries in search space. We also assume quasigroups to be *idempotent*, that means $a \cdot a = a$. QEP has been proposed as a benchmark domain for CP systems in Gent and Walsh (1999). All axioms have been modelled in *ezcsp* and *gecode* using constructive disjunction, and in S, B_k, R_k and *clingcon* using integrity constraints. Table 4 demonstrates that both constructive disjunction and integrity constraints have a similar behaviour, as for *ezcsp* and *clingcon* on benchmark classes QG1 to QG4. On harder instances, conflict-driven learning appears to be too costly for *clingcon*. Additional experiments revealed that *clingcon* without learning performs like *ezcsp*. On the other hand, our decompositions benefit

Table 5. Runtime results in seconds for GGP.

DW_n	S	B_1	B_3	B	R_3	R	*ezcsp*	*clingcon*	*gecode*
3	11.4	3.8	5.7	8.7	6.0	10.4	6.5	66.9	**1.8**
4	1.3	2.0	1.5	3.2	3.0	2.5	0.6	**0.1**	**0.1**
5	4.5	5.0	4.5	13.5	12.5	31.4	1.0	2.0	**0.1**
6	7.2	11.0	17.6	47.7	21.3	110.2	**1.2**	—	7.2
7	23.8	28.3	67.9	227.9	60.0	432.9	**18.0**	—	—
8	48.4	68.4	—	207.8	58.4	356.8	**4.3**	—	—
9	**82.8**	106.5	200.4	486.6	227.4	—	390.5	—	—

from learning constraint interdependencies, resulting in runtimes that outperform all other systems including *gecode* on the hardest problems.

4.4 Graceful Graphs

A labelling f of the nodes of a graph (V, E) is *graceful* if f assigns a unique label $f(v)$ from $\{0, 1, \ldots, |E|\}$ to each node $v \in V$ such that, when each edge $(v, w) \in E$ is assigned the label $|f(v) - f(w)|$, the resulting edge labels are distinct. The problem of determining the existence of a graceful labelling of a graph (GGP) has been modelled as a CSP in Petrie and Smith (2003). Our experiments consider double-wheel graphs DW_n composed by two copies of a cycle with n vertices, each connected to a central hub. Table 5 shows that our encodings compete with *ezcsp* and outperform the other comparable systems, where the support encoding performs better than bound and range encodings. In most cases, the branching heuristic used in our approach appears to be misled by the extra variables introduced in B_k and R_k. That explains some of the variability in the runtimes.

5 Conclusions

We have provided a new translation-based approach to incorporating Constraint Processing into Answer Set Programming. In particular, we investigated various generic ASP decompositions for constraints on finite domains and proved which level of consistency unit-propagation achieves on them. Our techniques were formulated as preprocessing and can be applied to any ASP system without changing its source code, which allows for programmers to select the solvers that best fit their needs. We have empirically evaluated their performance on benchmarks from CP and found them outperforming integrated Constraint Answer Set Programming systems as well as pure CP solvers. As a key advantage of our novel approach we identified CDNL, exploiting constraint interdependencies which can improve propagation between constraints. Future work concerns a comparison to Niemelä's encoding (Niemelä 1999; You and Hou 2004), and encodings of further global constraints useful in Constraint Answer Set Programming.

Acknowledgements. We are grateful to Martin Gebser and Torsten Schaub for useful discussions on the subject of this paper.

References

Balduccini, M. 2009. CR-prolog as a specification language for constraint satisfaction problems. In *Proceedings of LPNMR'09*. Springer, 402–408.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Baral, C., Chancellor, K., Tran, N., Tran, N., Joy, A., and Berens, M. 2004. A knowledge based approach for representing and reasoning about signaling networks. In *Proceedings of ISMB/ECCB'04*. 15–22.

Baselice, S., Bonatti, P., and Gelfond, M. 2005. Towards an integration of answer set and constraint solving. In *Proceedings of ICLP'05*. Springer, 52–66.

Bessière, C., Katsirelos, G., Narodytska, N., Quimper, C.-G., and Walsh, T. 2009. Decompositions of all different, global cardinality and related constraints. In *Proceedings of IJCAI'09*. AAAI Press/The MIT Press.

Biere, A., Heule, M., van Maaren, H., and Walsh, T., Eds. 2009. *Handbook of Satisfiability*. IOS Press.

Crawford, J. M. and Baker, A. B. 1994. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of AAAI'94*. 1092–1097.

Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. 2009. Answer set programming with constraints using lazy grounding. In *Proceedings of ICLP'09*. Springer, 115–129.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.

Dovier, A., Formisano, A., and Pontelli, E. 2005. A comparison of CLP(FD) and ASP solutions to NP-complete problems. In *Proceedings of ICLP'05*. Springer, 67–82.

Fujita, M., Slaney, J. K., and Bennett, F. 1993. Automatic generation of some results in finite algebra. In *Proceedings of IJCAI'93*. Morgan Kaufmann Publishers, 52–59.

Gebser, M., Hinrichs, H., Schaub, T., and Thiele, S. 2009. xpanda: A (simple) preprocessor for adding multi-valued propositions to ASP. In *Proceedings of WLP'09*.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. 2007a. clasp: A conflict-driven answer set solver. In *Proceedings of LPNMR'07*. Springer, 260–265.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. 2007b. Conflict-driven answer set solving. In *Proceedings of IJCAI'07*. AAAI Press/The MIT Press, 386–392.

Gebser, M., Ostrowski, M., and Schaub, T. 2009. Constraint answer set solving. In *Proceedings of ICLP'09*. Springer, 235–249.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of ICLP'88*. The MIT Press, 1070–1080.

Gent, I. P. 2002. Arc consistency in SAT. In *Proceedings of ECAI'02*. IOS Press, 121–125.

Gent, I. P. and Walsh, T. 1999. CSPLIB: A benchmark library for constraints. In *Proceedings of CP'99*. Springer, 480–481.

Gomes, C. P. and Selman, B. 1997. Problem structure in the presence of perturbations. In *Proceedings of AAAI'97*. AAAI Press, 221–226.

Heljanko, K. and Niemelä, I. 2003. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming 3,* 4-5, 519–550.

Jaffar, J. and Maher, M. J. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Järvisalo, M., Oikarinen, E., Janhunen, T., and Niemelä, I. 2009. A module-based framework for multi-language constraint modeling. In *Proceedings of LPNMR'09*. Springer, 155–169.

Leconte, M. 1996. A bounds-based reduction scheme for constraints of difference. In *CP'96, Second International Workshop on Constraint-based Reasoning*.

Lee, J. 2005. A model-theoretic counterpart of loop formulas. In *Proceedings of IJCAI'05*. Professional Book Center, 503–508.

Lifschitz, V. 1999. Answer set planning. In *Proceedings of ICLP'99*. The MIT Press, 23–37.

Mellarkod, V. and Gelfond, M. 2008. Integrating answer set reasoning with constraint solving techniques. In *Proceedings of FLOPS'08*. Springer, 15–31.

Mellarkod, V., Gelfond, M., and Zhang, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence 53,* 1-4, 251–287.

Mitchell, D. 2005. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science 85*, 112–133.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25,* 3-4, 241–273.

Nieuwenhuis, R., Oliveras, A., and Tinelli, C. 2006. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM 53,* 6, 937–977.

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., and Barry, M. 2001. An A-prolog decision support system for the space shuttle. In *Proceedings of PADL'01*. Springer, 169–183.

Petrie, K. E. and Smith, B. M. 2003. Symmetry breaking in graceful graphs. In *Proceedings of CP'03*. Springer, 930–934.

Rossi, F., van Beek, P., and Walsh, T., Eds. 2006. *Handbook of Constraint Programming*. Elsevier.

Simons, P., Niemelä, I., and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1-2, 181–234.

Tamura, N., Taga, A., Kitagawa, S., and Banbara, M. 2006. Compiling finite linear CSP into SAT. In *Proceedings of CP'06*. Springer, 590–603.

Walsh, T. 2000. SAT v CSP. In *Proceedings of CP'00*. Springer, 441–456.

You, J.-H. and Hou, G. 2004. Arc-consistency + unit propagation = lookahead. In *Proceedings of ICLP'04*. 314–328.