Reformulation of Global Constraints in Answer Set Programming

Christian Drescher
Vienna University of Technology
Austria

Toby Walsh
NICTA and University of New South Wales
Australia

Abstract
We show that global constraints on finite domains like all-different can be reformulated into answer set programs on which we achieve arc, bound or range consistency. These reformulations offer a number of other advantages beyond providing the power of global propagators to answer set programming. For example, they provide other constraints with access to the state of the propagator by sharing variables. Such sharing can be used to improve propagation between constraints. Experiments with these encodings demonstrate their promise.

Introduction
There are several approaches to representing and solving constraint satisfaction problems: constraint programming (CP; Dechter 2003, Rossi, van Beek, and Walsh 2006), answer set programming (ASP; Baral 2003), propositional satisfiability checking (SAT; Biere et al. 2009), its extension to satisfiability modulo theories (SMT; Nieuwenhuis, Oliveras, and Tinelli 2006), and many more. Each has its particular strengths: for example, CP systems support global constraints, ASP systems permit recursive definitions and offer default negation, whilst SAT solvers often exploit very efficient implementations. In many applications it would often be helpful to exploit the strengths of multiple approaches. Consider the problem of timetabling at a university (Jarvisalo et al. 2009). To model the problem, we need to express the mutual exclusion of events (for instance, we cannot place two events in the same room at the same time). A straightforward representation of such constraint with clauses and rules uses quadratic space. In contrast, global constraints such as all-different typically supported by CP systems can give a much more concise encoding. On the other hand, there are features which are hard to describe in traditional constraint programming, like the temporary unavailability of a particular room. However, this is easy to represent with non-monotonic rules such as those used in ASP. Such rules also provide a flexible mechanism for defining new relations on the basis of existing ones.

Answer set programming has been put forward as a powerful paradigm to solve constraint satisfaction problems. Niemelä (1999) shows that ASP embeds SAT but provides a more expressive framework from a knowledge representation point of view. Moreover, modern ASP solvers compete with the best SAT solvers. An empirical comparison of the performance of ASP and constraint logic programming (CLP; Jaffar and Maher 1994) systems on solving combinatorial problems conducted by Dovier, Formisano, and Pontelli shows ASP encodings to be more compact, more declarative, and highly competitive. However, as some problems are more naturally modelled by using non-propositional constructs, like resources or functions over finite domains, and by using global constraints in particular, there is an increasing desire to handle constraints beyond pure ASP.

One approach to combining ASP and CP is to integrate theory-specific predicates into propositional formulas (motivated by SMT), and to extend the ASP solver’s decision engine with a higher level proof procedure (Baselice, Bonatti, and Gelfond 2005; Melkko and Gelfond 2008). However, the resulting systems have a number of limitations. First, they are tied to particular ASP and CP solvers. Second, the support for global constraints is limited. Third, communication between the ASP and CP solver is restricted. Alternative techniques, such as reformulating constraints into ASP have received little attention. The key contribution of our work is an investigation of reformulation in the context of answer set programming, illustrated by reformulations of the popular all-different constraint. The resulting approach has been implemented in the new preprocessor inca. Empirical evaluation demonstrates its computational potential.

Background
Answer Set Programming A (normal) logic program \Pi over a set of primitive propositions \mathcal{A} is a finite set of rules of the form \textit{a_0 ← a_1, \ldots, a_m, not a_{m+1}, \ldots, not a_n} where 0 \leq m \leq n and \textit{a_i ∈ \mathcal{A}} are atoms for 0 \leq i \leq n. A literal \textit{a} is an atom \textit{a} or its default negation \textit{not a}. For a rule \textit{r}, let head(r) = a_0 be the head of \textit{r} and body(r) = {a_1, \ldots, a_m, not a_{m+1}, \ldots, not a_n} the body of \textit{r}. The set of atoms occurring in a logic program \Pi is denoted by atom(\Pi), and the set of bodies in \Pi is body(\Pi) = \textit{http://www.satcompetition.org}

Background
Answer Set Programming A (normal) logic program \Pi over a set of primitive propositions \mathcal{A} is a finite set of rules of the form \textit{a_0 ← a_1, \ldots, a_m, not a_{m+1}, \ldots, not a_n} where 0 \leq m \leq n and \textit{a_i ∈ \mathcal{A}} are atoms for 0 \leq i \leq n. A literal \textit{a} is an atom \textit{a} or its default negation \textit{not a}. For a rule \textit{r}, let head(r) = a_0 be the head of \textit{r} and body(r) = \textit{http://www.satcompetition.org}
{body(r) | r ∈ II}. For regrouping bodies sharing the same head a, define \(body(a) = \{body(r) | r ∈ II, head(r) = a\}\).

The semantics of a logic program is given by its answer sets, being total well-founded models of II. For a formal introduction to ASP, we refer the reader to [Baral 2003].

The semantics of important extensions to logic programs, such as choice rules, integrity, and cardinality constraints, is given through program transformations that introduce additional propositions (cf. Simons, Niemelä, and Soininen 2002). A choice rule allows for the non-deterministic choice over atoms in \(\{a_1, \ldots, a_k\}\) and has the form \(a_0 \leftarrow a_k+1, \ldots, a_m, not a_{m+1}, \ldots, not a_n\). An integrity constraint of the form \(a_1, \ldots, a_m, not a_{m+1}, \ldots, not a_n\) is a short hand for a rule with an unsatisfiable head, and thus forbids its body to be satisfied in any answer set. A cardinality constraint of the form \(k\{\hat{a}_1, \ldots, \hat{a}_n\}\) is interpreted as no \(k\) literals of the set \(\{\hat{a}_1, \ldots, \hat{a}_n\}\) are included in an answer set. Simons, Niemelä, and Soininen provide a transformation that needs just \(O(nk)\) rules, introducing atoms \(l(\hat{a}_i, j)\) to represent the fact that at least \(j\) of the literals with index \(i\), i.e. the literals in \(\{\hat{a}_1, \ldots, \hat{a}_n\}\), are in a particular answer set candidate. Then, the cardinality constraint can be encoded by an integrity constraint \(l(\hat{a}_i, k)\) and the three following rules, where \(1 ≤ i ≤ n\) and \(1 ≤ j ≤ k\):

\[
l(\hat{a}_i, j) ← l(\hat{a}_{i+1}, j) \quad l(\hat{a}_i, j + 1) ← \hat{a}_i, l(\hat{a}_{i+1}, j)
\]

Nogoods of Logic Programs We want to view inferences in ASP as unit-propagation on nogoods. Following Gebser et al. (2007), inferences in ASP rely on atoms and program rules, which can be expressed by using atoms and bodies. Thus, for a program II, the domain of Boolean assignments \(A\) is fixed to \(dom(A) = atom(II) ∪ body(II)\).

Formally, a Boolean assignment \(A\) is a set \(\{σ_1, \ldots, σ_n\}\) of signed literals \(σ_i\) for \(1 ≤ i ≤ n\) of the form \(Ta\) or \(Fa\) where \(a ∈ dom(A)\). \(To\) expresses that \(a\) is assigned true and \(Fa\) that it is false in \(A\). (We omit the attribute Boolean for assignments whenever clear from the context.) The complement of a signed literal \(σ\) is denoted by \(π\), that is \(To = Fa\) and \(F\overline{a} = Ta\). In the context of ASP, a nogood is a set \(\{σ_1, \ldots, σ_n\}\) of signed literals, expressing a constraint violated by any assignment \(A\) such that \(δ ⊆ A\). For a nogood \(δ\), a signed literal \(σ\) \(δ\), and an assignment \(A\), we say that \(δ\) unit and \(π\) unit-resulting if \(δ \setminus A = \{σ\}\). Let \(A^T = \{a ∈ dom(A) | Ta ∈ A\}\) the set of true propositions and \(A^F = \{a ∈ dom(A) | Fa ∈ A\}\) the set of false propositions. A total assignment, that is \(A^T ∪ A^F = dom(A)\), and \(A^F ∪ A^T = \emptyset\), is a solution for a set \(Δ\) of nogoods if \(δ ⊆ A\) for all \(δ \in Δ\).

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

\[
\Delta_β = \{Ta_1, \ldots, Ta_m, Fa_{m+1}, \ldots, Fa_n, Fβ\}, \{Fa_1, Tβ\}, \ldots, \{Fa_m, Tβ\}, \{Ta_{m+1}, Tβ\}, \ldots, \{Ta_n, Tβ\}\}
\]

Intuitively, the nogoods in \(Δ_β\) enforce the truth of body \(β\), as all its literals are satisfied. For an atom \(a \in atom(II)\) with \(body(a) = \{β_1, \ldots, β_k\}\), let

\[
Δ_a = \{\{Fβ_1, \ldots, Fβ_k, Ta\}, \{Tβ_1, Fa_1, \ldots, \{Tβ_k, Fa_k\}\} \}
\]

Then, the solutions for \(Δ_II = \bigcup_{β ∈ body(II)} Δ_β \bigcup \bigcup_{a ∈ atom(II)} Δ_a\) correspond to the models of the completion of II. Loop formulas, expressed in the set of nogoods \(Δ_II\), have to be added to establish full correspondence to the answer sets of II. Typically, solutions for \(Δ_II \bigcup Δ_II\) are computed by applying conflict-driven nogood learning (CDNCL: Gebser et al. 2007). This combines search and propagation by recursively assigning the value of a proposition and using unit-propagation to determine logical consequences of an assignment (Mitchell 2005).

Constraint Satisfaction Problem The classic definition of a constraint satisfaction problem is as follows (cf. Rossi, van Beek, and Walsh 2006). 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 \(dom(v_i) = D_i\), and \(C\) is a set of constraints. A constraint \(c\) is a pair \((R, S)\) where \(RS\) is a k-ary relation on the variables in \(S ⊆ V^k\), called the scope of \(c\). In other words, \(RS\) 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 \(range(c) = RS\) and \(scope(c) = S\). For a (constraint variable) assignment \((A) : V → \bigcup_{v ∈ V} dom(v)\) and a constraint \(c = (R, S)\) with \(S = \{v_1, \ldots, v_k\}\), define \(A(S) = (A(v_1), \ldots, A(v_k))\), and call \(c\) satisfied if \(A(S) ∈ range(c)\). Given this, define the set of constraints satisfied by \(A\) as \(satc(A) = \{c | A(scope(c)) ∈ range(c), c ∈ C\}\).

A binary constraint \(c\) has \(|scope(c)| = 2\). For example, \(v_1 ≠ v_2\) 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, \(\{v_1, \ldots, v_k\}\) take all different values. This can be decomposed into \(O(n^2)\) binary constraints, \(v_i ≠ v_j\). However, as we shall see, such reformulation can hinder inference. An assignment \(A\) is a solution iff it satisfies all constraints in \(C\).

Constraint solvers typically use backtracking search to explore the space of partial assignments. Various heuristics affecting, for instance, the variable selection criteria and the ordering of the attempted values, can be used to guide the search. Each time a variable is assigned a value, a deterministic propagation stage is executed, pruning the set of values to be attempted for the other variables, i.e., enforcing a certain type of local consistency.

A binary constraint \(c\) is called arc consistent iff when a variable \(v_1 ∈ scope(c)\) is assigned any value \(d_1 ∈ dom(v_1)\), there exists a consistent value \(d_2 ∈ dom(v_2)\) for the other variable \(v_2\). An n-ary constraint \(c\) is hyper-arc consistent or domain consistent iff when a variable \(v_1 ∈ scope(c)\) is assigned any value \(d_1 ∈ dom(v_1)\), there exist compatible values in the domains of all the other variables \(d_j ∈ dom(v_j)\)
for all $1 \leq j \leq n$, $j \neq i$ such that $(d_1, \ldots, d_n) \in \text{range}(c)$.

Relational consistency (Dechter and van Beek 1997) extends the concept of local consistency. I.e. a constraint $c$ is relationally $k$-arc consistent if any consistent assignment of a $k$-elementary subset of variables from $\text{scope}(c)$ extends to a consistent assignment of all variables in $\text{scope}(c)$.

The concepts of bound and range consistency are defined for constraints on ordered intervals. Let $\min(D_i)$ and $\max(D_i)$ 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 $d_i \in \{\min(\text{dom}(v_i)), \max(\text{dom}(v_i))\}$ (i.e. the minimum or maximum value in its domain), there exist compatible values between the minimum and maximum domain 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 three formalisms.

**Encoding Global Constraints in ASP**

In this section we explain how to reformulate multi-valued variables and constraints on finite domains into a logic program under answer set semantics. In what follows, we assume $\text{dom}(v) = [1, d]$ for all $v \in V$ to save the reader from multiple superscripts.

**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.

$$\{e(v, 1), \ldots, e(v, d)\} \leftarrow e(v, 1), \ldots, e(v, d)$$  \hspace{1cm} (1)

$$\leftarrow \text{not } e(v, 1), \ldots, \text{not } e(v, d)$$  \hspace{1cm} (2)

$$\leftarrow 2 \{e(v, 1), \ldots, e(v, d)\}$$  \hspace{1cm} (3)

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** 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.

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

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

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 $v, v' \in \text{scope}(c)$, and for each value $i$.

**Theorem 2** 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 can be reduced from the definition by using the equivalence covered by (2–3) to

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

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 $O(d)$ cardinality constraints:

$$\leftarrow 2 \{e(v_1, i), \ldots, e(v_n, i)\}$$  \hspace{1cm} (4)

**Corollary 2.1** Unit-propagation on (1–4) enforces arc consistency on the binary decomposition of the global all-different constraint in $O(nd^2)$ down any branch of the search tree.

**k-support Encoding** The support encoding can be generalized to the k-support encoding (Bessière, Hebrard, and Walsh 2003) representing supports on subsets of $\text{scope}(c)$ for an assignment of another $k$-elementary subset of variables in $\text{scope}(c)$. More formal, a $k$-support $S$ for an assignment $A$ of $k$ variables from $\text{scope}(c)$, say $v_1 = i_1, \ldots, v_k = i_k$, is an assignment $v'_1 = i'_1, \ldots, v'_l = i'_l$ such that $\{v'_1, \ldots, v'_l\} \subseteq \text{scope}(c) \setminus \{v_1, \ldots, v_k\}$ which allows $A$. We introduce a support-variable $s$, that evaluates to true iff $S$ holds:

$$s \leftarrow e(v'_1, i_1), \ldots, e(v'_l, i'_l)$$

Furthermore, let $\{S_1, \ldots, S_m\}$ be the set of all $k$-supports of $A$. A $k$-support rule for $A$ is defined as

$$\leftarrow e(v_1, d_1), \ldots, e(v_k, d_k), \text{not } s_1, \ldots, \text{not } s_m$$

meaning that as long as $A$ holds then at least one of its $k$-supports $S_1, \ldots, S_m$ must hold. In the $k$-support encoding, for each constraint $c$ there is one $k$-support rule for each assignment $A$ of $k$ variables from $\text{scope}(c)$.

**Theorem 3** Unit-propagation on the $k$-support encoding enforces relational $k$-arc consistency on the original constraint.


**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 \( O(nd^2) \) rules encode \( v \in [l, u] \) whenever it is safe to assume that \( v \not\in [1, l - 1] \) and \( v \not\in [u + 1, d] \), and enforce a consistent set of ranges such that \( v \in [l, u] \) \( \Rightarrow v \in [l - 1, u] \wedge v \in [l, u + 1] \):

\[
\begin{align*}
   r(v, l, u) &\iff \neg r(v, l, u - 1), \neg r(v, u + 1, d) \quad (5) \\
   \iff r(v, l, u - 1), \neg r(v, l, u) \quad (6) \\
   \iff r(v, l, u + 1), \neg r(v, l, u) \quad (7)
\end{align*}
\]

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:

\[
\begin{align*}
   \iff r(v_1, l_1, u_1), \ldots, r(v_n, l_n, u_n)
\end{align*}
\]

**Theorem 4** Unit-propagation on the range encoding ensures range consistency on the original constraint.

A propagator for the global all-different constraint that enforces range consistency pruning Hall intervals has been proposed by [Leconte (1996)] and encoded to SAT by [Bessière et al. (2009)]. An interval \([l, u]\) is a Hall interval if \( |\{v \mid \text{dom}(v) \subseteq [l, u]\}| = u - l + 1 \). 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. The following reformulation 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.

\[
\begin{align*}
   \iff u - l + 2 \{r(v_1, l, u), \ldots, r(v_n, l, u)\} \quad (8)
\end{align*}
\]

This simple reformulation can simulate a complex propagation algorithm like [Leconte] with a similar overall complexity of reasoning.

**Corollary 4.1** Unit-propagation on (5–8) enforces range consistency on the global all-different constraint in \( O(nd^3) \) down any branch of the search tree.

A hybrid that links the range encoding of \( v \) to its direct representation extends the range encoding as follows, for each \( i \in \text{dom}(v) \):

\[
\begin{align*}
   e(v, i) &\iff r(v, i, i) \\
   &\iff e(v, i), \neg r(v, i, i)
\end{align*}
\]

These rules encode the equivalence \( v = i \iff v \in [i, i] \).

**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 the value of \( v \) is bounded by \( i \). That is, \( v \leq i \) if \( b(v, i) \) is assigned \( true \), and \( v > i \) if \( b(v, i) \) is assigned \( false \). 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) \( \forall i \in [1, d - 1] \). Another integrity constraint (11) encodes \( v \leq d \), that at least one value must be assigned to \( v \):

\[
\begin{align*}
   \{b(v, 1), \ldots, b(v, d)\} &\iff b(v, i), \neg b(v, i + 1) \quad (10) \\
   \iff \neg b(v, d) \quad (11)
\end{align*}
\]

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:

\[
\begin{align*}
   \iff b(v_1, u_1), \ldots, b(v_n, u_n), \neg b(v_1, l_1), \ldots, \neg b(v_n, l_n)
\end{align*}
\]

**Theorem 5** Unit-propagation on the bound encoding enforces bound consistency on the original constraint.

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:

\[
\begin{align*}
   r(v, l, u) &\iff \neg b(v, l - 1), b(v, u) \quad (12) \\
   \iff r(v, l, u), b(v, l - 1) \quad (13) \\
   \iff r(v, l, u), \neg b(v, u) \quad (14)
\end{align*}
\]

**Corollary 5.1** Unit-propagation on (8–14) enforces bound consistency on the global all-different constraint in \( O(nd^2) \) down any branch of the search tree.

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.

To access the value of \( v \), the bound encoding can be extended to a hybrid by adding the following rules to the bound encoding for each \( i \in [1, d] \):

\[
\begin{align*}
   e(v, i) &\iff b(v, i), \neg b(v, i - 1) \\
   \iff e(v, i), \neg b(v, i) \\
   \iff e(v, i), b(v, i - 1)
\end{align*}
\]

The first rule enforces \( e(v, i) \) to be true if possible values for \( v \) are bound to the singleton \( i \), i.e., \( v \leq i \) and \( v \not\leq i - 1 \) are in the assignment. On the other hand, the condition \( v = i \Rightarrow v \leq i \wedge v \not\leq i - 1 \) is represented as integrity constraints.

**Non-ground Logic Programs** Although our semantics is propositional, atoms in \( A \) and can be constructed from a first-order signature \( \Sigma = (F, V, P) \), where \( F \) is a set of function symbols (including constant symbols), \( V \) is a denumerable collection of first-order variables, and \( P \) is a set of predicate symbols. The logic program over \( A \) is then obtained by a grounding process, systematically substituting all occurrences of variables \( V \) by terms in \( T(F) \), where \( T(F) \) denotes the set of all ground terms over \( F \). Atoms in \( A \) are formed from predicate symbols \( P \) and terms in \( T(F) \).
Experiments

To evaluate these reformulations, we conducted experiments on encodings 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 reformulation of permutation extends (4) by

\[ \leftarrow \text{not } e(v_1, i), \ldots, \text{not } e(v_n, i) \]

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

\[ \leftarrow d - u + l \{ \text{not } r(v_1, l, u), \ldots, \text{not } r(v_n, l, u) \} \]

This can increase propagation. Our reformulations have been implemented within the prototypical preprocessor inp4 which compiles an (extended) logic programs with high-level statements for global constraints, constraint variables, 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 in different settings using different reformulations. 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 pure CP system gecode\( ^4 \) (3.2.0), and the integrated system ezcsp\( ^5 \) (1.6.9; Balduccini 2009) in our empirical analysis. The latter combines the grounder gringo (2.0.3) and ASP solver clasp (1.3.0) with sicstus\( ^6 \) (4.0.8) as a constraint solver. Since inca is a pure preprocessor, we select the ASP system clingo (2.0.3) as its backend to provide a representative comparison with ezcsp. Note that clingo stands for clasp on gringo and combines both systems in a monolithic way. 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.

Pigeon Hole Problem  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 reformulations are faster compared to weaker encodings (see Table 1). It appears that sicstus\( ^7 \) and gecode\'s default configuration uses filtering algorithms for the global all-different constraint achieve arc consistency on its binary decomposition. However, on such problems, detecting large Hall intervals is essential.

Latin Squares  A Latin square is an \( n \times n \)-table filled with \( n \) different elements such that each element occurs exactly once in each row and each column of the table. The Latin square puzzle (LSP) is to determine whether a partially filled table can be completed in such a way that a Latin square is obtained. Randomly generated LSP has been proposed as a benchmark domain for CP systems by Gomes and Selman since it combines the features of purely random problems and highly structured problems. Table 2 compares the runtime for solving LSP problems of size \( 20 \times 20 \) where the first column gives the percentage of preassigned values. We included gecode with algorithms that enforce bound and domain consistency, denoted as gecode\( _B \) and gecode\( _D \), while our Boolean encodings and gecode\( _D \) solve all problems within seconds. Interestingly, learning constraint interdependencies as in our approach is sufficient to tackle LSP. In fact, most of the time for \( S, B_k, R_k \) is spent on grounding, but not for solving the actual problem.

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 in CP [Petrie and Smith (2003), using auxiliary variables \( d(v,w) \) for edge labels. We represent the equivalence \( d(v,w) = |f(v)-f(w)| \) in the direct encoding which weakens the overall consistency. Our exper-

---

3http://potassco.sourceforge.net/ provides the systems clasp, clingo, gringo, inca, and the benchmark set
4http://www.gecode.org/
5http://krlab.cs.ttu.edu/~marcy/ezcsp/
6http://www.sics.se/sicstus/

| \( n \) | \( S \) | \( B \) | \( R \) | ezcsp | gecode | gecode\( _B \) |
|---|---|---|---|---|---|---|
| 10 | 5 | <1 | <1 | <1 | 1 | <1 |
| 11 | 46 | 1 | <1 | 2 | 17 | 9 |
| 12 | 105 | 4 | <1 | 3 | 184 | 104 |
| 13 | — | 25 | <1 | 30 | — | — |
| 14 | — | 125 | <1 | 197 | — | — |
| 15 | — | — | <1 | — | — | — |
| 16 | — | — | <1 | — | — | — |

Table 1: Runtime results in seconds for PHP.

| \( n \) | \( S \) | \( B \) | \( R \) | ezcsp | gecode | gecode\( _B \) |
|---|---|---|---|---|---|---|
| 10 | 3 | 5 | 8 | 7 | 30 (7) | 2 (4) | <1 (1) |
| 20 | 2 | 5 | 8 | 7 | 21 (20) | 5 (4) | <1 (3) |
| 30 | 2 | 5 | 8 | 7 | 10 (30) | 3 (13) | 1 (5) |
| 35 | 2 | 5 | 8 | 7 | 22 (24) | 14 (13) | 6 (7) |
| 40 | 2 | 5 | 8 | 7 | 52 (29) | 12 (20) | 6 (9) |
| 45 | 2 | 5 | 8 | 7 | 36 (35) | 18 (25) | 6 (13) |
| 50 | 2 | 5 | 8 | 7 | 36 (50) | 25 (32) | 6 (18) |
| 55 | 2 | 4 | 8 | 7 | 61 (51) | 20 (41) | 31 (29) |
| 60 | 2 | 4 | 8 | 7 | 60 (63) | 36 (51) | 27 (35) |
| 70 | 2 | 4 | 7 | 6 | 70 (66) | 28 (45) | 17 (27) |
| 80 | 2 | 4 | 7 | 5 | 16 (18) | 17 (13) | 7 (7) |
| 90 | 2 | 4 | 7 | 5 | 1 | <1 (1) | 3 |

Table 2: Average times over 100 runs on LSP. Timeouts are given in parenthesis, if any.
Table 3: Runtime results in seconds for GGP.

| \( n \) | \( S \) | \( B_1 \) | \( B_2 \) | \( B \) | \( R \) | \( ezcsp \) | \( gecode \) |
|-------|-----|------|------|-----|-----|-------|-------|
| 3     | 11  | 4    | 6    | 9   | 6   | 6     | 2     |
| 4     | 1   | 2    | 1    | 3   | 3   | \(<1\) | \(<1\) |
| 5     | 4   | 5    | 4    | 13  | 12  | 1     | \(<1\) |
| 6     | 7   | 11   | 18   | 48  | 21  | 1     | 7     |
| 7     | 24  | 28   | 68   | 228 | 60  | 18    | —     |
| 8     | 48  | 68   | —    | 208 | 58  | 4     | —     |
| 9     | 83  | 106  | 200  | 487 | —   | 390   | —     |

imements consider double-wheel graphs \( DW_n \) composed by two copies of a cycle with \( n \) vertices, each connected to a central hub. Table 3 shows that our encodings compete with \( ezcsp \) and outperform \( gecode \), 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.

Conclusions

We have reformulated global and other constraints into answer set programs. In particular, we have investigated various generic ASP encodings 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 ASP solver that best fit their needs. We have empirically evaluated the performance of such an approach on benchmarks from CP and found that such reformulations outperform integrated ASP(CP) systems as well as pure CP solvers. Our future works includes the reformulation of other useful global constraints into answer set programming like the regular constraint, as well as global constraints like \( lex \) which are very useful for symmetry breaking.

References

[2003] Balduccini, M. 2003. CR-prolog as a specification language for constraint satisfaction problems. In Proc. of LPNMR’03, 402–408. Springer.

[2005] Gent, I. P. 2005. A SAT solver primer. In Proc. of FLOPS’05, 15–31. Springer.

[2006] Mitchell, D. 2005. A model-theoretic counterpart of loop formulas. In Proc. of IJCAI’05, 503–508. Professional Book Center.

[1994] Crawford, J. M., and Baker, A. B. 1994. Experimental results on the application of satisfiability algorithms to scheduling problems. In Proc. of AAAI’94, 1092–1097.

[1997] Dechter, R., and van Beek, P. 1997. Local and global relational consistency. Theory of Computer Science 173(1):283–308.

[2003] Balduccini, M. 2009. CR-prolog as a specification language for constraint satisfaction problems. In Proc. of LPNMR’03, 402–408. Springer.

[2005] Dovier, A.; Formisano, A.; and Pontelli, E. 2005. A comparison of CLP(FD) and ASP solutions to NP-complete problems. In Proc. of ICLP’05, 67–82. Springer.

[2007] Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. Conflict-driven answer set solving. In Proc. of IJCAI’07, 386–392. AAAI Press/The MIT Press.

[2009] Gebser, M.; Ostrowski, M.; and Schaub, T. 2009. Constraint answer set solving. In Proc. of ICLP’09, 235–249. Springer.

[2002] Gent, I. P. 2002. Arc consistency in SAT. In Proc. of ECAI’02, 121–125. IOS Press.

[1997] Gomes, C. P., and Selman, B. 1997. Problem structure in the presence of perturbations. In Proc. of AAAI’97, 221–226. AAAI Press.

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

[2009] Järvisalo, M.; Oikarinen, E.; Janhunen, T.; and Niemelä, I. 2009. A module-based framework for multilanguage constraint modeling. In Proc. of LPNMR’09, 155–169. Springer.

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

[2005] Lee, J. 2005. A model-theoretic counterpart of loop formulas. In Proc. of IJCAI’05, 503–508. Professional Book Center.

[2008] Mellarkod, V., and Gelfond, M. 2008. Integrating answer set reasoning with constraint solving techniques. In Proc. of FLOPS’08, 15–31. Springer.

[1999] 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.

[2006] 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.

[2003] Petrie, K. E., and Smith, B. M. 2003. Symmetry breaking in graceful graphs. In Proc. of CP’03, 930–934. Springer.

[2006] Rossi, F.; van Beek, P.; and Walsh, T., eds. 2006. Handbook of Constraint Programming. Elsevier.
[2002] Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2):181–234.

[2000] Walsh, T. 2000. SAT v CSP. In Proc. of CP’00, 441–456. Springer.