

Translation-Based Constraint Answer Set Solving

Christian Drescher and Toby Walsh

NICTA and University of New South Wales, Sydney, Australia

Abstract

We solve constraint satisfaction problems through translation to answer set programming (ASP). Our reformulations have the property that unit-propagation in the ASP solver achieves well defined local consistency properties like arc, bound and range consistency. Experiments demonstrate the computational value of this approach.

1 Introduction

Several formalisms have been proposed for representing and solving combinatorial problems: constraint programming (CP; [Rossi *et al.*, 2006]), answer set programming (ASP; [Baral, 2003]), propositional satisfiability checking (SAT; [Biere *et al.*, 2009]), its extension to satisfiability modulo theories (SMT; [Nieuwenhuis *et al.*, 2006]), and many more. Each has its particular strengths: for example, CP systems support global constraints, SAT often exploits very efficient implementations, whilst ASP systems permit recursive definitions and offer default negation. As a non-monotonic reasoning paradigm, ASP is particularly adequate for common-sense reasoning and modelling of dynamic and incomplete knowledge, and was put forward as a powerful paradigm to solve constraint satisfaction problems (CSP) in [Niemelä, 1999]. Moreover, modern ASP solvers have experienced dramatic improvements in their performance [Gebser *et al.*, 2007] and compete with the best SAT solvers. Empirical comparisons with CP have shown that, whilst ASP encodings are often highly competitive and more elaboration tolerant, non-propositional constructs like global constraints are more efficiently handled by CP systems [Dovier *et al.*, 2005].

This led to the integration of CP with ASP in *hybrid* frameworks, most notably constraint answer set programming (CASP; [Gebser *et al.*, 2009b]). Similar to SMT, the key idea of a *hybrid* approach is that theory-specific solvers interact in order to compute solutions to the whole constraint model. However, the elaboration of constraint interdependencies from different solver types is limited by the restricted interface between the ASP and the CP solver.

This paper puts forward a *translation-based* approach rather than a *hybrid* one. In this approach, all parts of the CSP model are mapped into ASP for which highly efficient

solvers are available. We make several contributions to the study of translation into ASP [Drescher and Walsh, 2010]:

- We consider four different but generic encodings: the direct, support, bound and range encoding. Each represents constraints in a different way.
- We provide theoretical results on their propagation strength, i.e., what type of local consistency is achieved by the unit-propagation of an ASP solver.
- We illustrate our approach on the popular ALL-DIFFERENT constraint. This ensures that a set of variables take all different values. Unit-propagation on our encodings can simulate complex propagation algorithms with a similar overall runtime complexity.
- We conduct experiments on CSPLib [Gent and Walsh, 1999], a large problem library widely used for benchmarking by the CP community. Our results demonstrate the competitiveness of this approach.

2 Background

Answer Set Programming As a form of logic programming oriented towards solving CSP, ASP comes with an expressive but simple modelling language. Formally, a *logic program* over a set of primitive propositions \mathcal{A} , $\perp \in \mathcal{A}$, is a finite set of *rules* r of the form

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

where $h, a_i \in \mathcal{A}$ are *atoms*, $1 \leq i \leq n$. A *literal* is an atom a or its default negation $\text{not } a$. The special atom \perp denotes a proposition that is always false. For a rule r , define $\text{head}(r) = h$ and $\text{body}(r) = \{a_1, \dots, a_m, \text{not } a_{m+1}, \dots, \text{not } a_n\}$. Furthermore, let $\text{body}(r)^+ = \{a_1, \dots, a_m\}$ and $\text{body}(r)^- = \{a_{m+1}, \dots, a_n\}$. A rule r with $\text{head}(r) = \perp$ is widely referred to as an *integrity constraint*. The semantics of a logic program is given by its answer sets, which are the key objects of interest in this paradigm. Given a logic program P over \mathcal{A} , a set $X \subseteq \mathcal{A}$ is an *answer set* of P iff X is the \subseteq -minimal model of the *reduct* [Gelfond and Lifschitz, 1988]

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

Intuitively, a rule r of the form above can be seen as a condition on the answer sets of a logic program, stating that if a_1, \dots, a_m are in the answer set and none of a_{m+1}, \dots, a_n is included, then h must be in the set. We also consider extensions to logic programs, such as choice rules and cardinality

rules. A *choice rule* of the form

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

allows for the nondeterministic choice over atoms in $\{h_1, \dots, h_k\}$. A *cardinality rule* of the form

$$h \leftarrow k\{a_1, \dots, a_m, \text{not } a_{m+1}, \dots, \text{not } a_n\}$$

infers h if k or more literals in the set $\{a_1, \dots, a_m, \text{not } a_{m+1}, \dots, \text{not } a_n\}$ are satisfied. The semantics of choice rules and cardinality rules is given through program transformations (cf. [Simons *et al.*, 2002]). Note that aggregations and other forms of set constructions are also common in ASP. However, we will limit ourselves to the above concepts as they are expressive enough for what follows. Also note that, although answer set semantics is propositional, atoms in \mathcal{A} can be constructed from a first-order signature. The logic program over \mathcal{A} is then obtained by a *grounding* process, systematically substituting all occurrences of first-order variables with terms formed by function symbols and constants given through the signature. The task of ASP systems is to compute answer sets for logic programs. A successful framework is conflict-driven nogood learning (CDNL; [Gebser *et al.*, 2007]). It reflects conditions from program rules in a set of *nogoods*, and describes ASP inference as unit-propagation on nogoods to determine logical consequences.

Constraint Satisfaction and Consistency We want to use ASP to model and solve CSP. Formally, a CSP is a triple (V, D, C) where V is a finite set of *variables*, each $v \in V$ has an associated finite *domain* $\text{dom}(v) \in D$, and C is a set of constraints. A *constraint* c is a pair (R_S, S) where R_S is a k -ary relation, denoted $\text{range}(c)$, on the variables in $S \in V^k$, denoted $\text{scope}(c)$. Given a (*constraint variable*) *assignment* $A : V \rightarrow \bigcup_{v \in V} \text{dom}(v)$, for a constraint c with $\text{scope}(c) = S = (v_1, \dots, v_k)$ define $A(S) = (A(v_1), \dots, A(v_k))$ and call c *satisfied* if $A(S) \in \text{range}(c)$. Define the set of constraints satisfied by A as $\text{sat}_C(A) = \{c \mid A(\text{scope}(c)) \in \text{range}(c), c \in C\}$. A *binary* constraint c has $|\text{scope}(c)| = 2$. For instance, the constraint $v_1 \neq v_2$ ensures that v_1 and v_2 take different values. An n -ary constraint c has parametrised scope. For instance, ALL-DIFFERENT ensures that a set of variables, $|\text{scope}(c)| = n$, take all different values. As any non-binary constraint, this can be *decomposed* into binary constraints, i.e., $O(n^2)$ constraints $v_i \neq v_j$ for $i < j$. However, as we shall see in the following, such reformulation can hinder inference.

An assignment A is a *solution* to a CSP iff it satisfies all constraints in C . Typically, CP systems use backtracking search to explore assignments in a search tree. In a search tree, each node represents an assignment to some variables, child nodes are obtained by selecting an unassigned variable and having a child node for each possible value for this variable, and the root node is empty. Every time a variable is assigned a value, *constraint propagation* is executed, pruning the set of values for the other variables, i.e., enforcing a certain type of local consistency such as arc, bound, range, or domain consistency. A binary constraint c is *arc consistent* iff a variable $v_1 \in \text{scope}(c)$ is assigned any value $d_1 \in \text{dom}(v_1)$, there exists a compatible value $d_2 \in \text{dom}(v_2)$

for the other variable v_2 . An n -ary constraint c is *domain consistent* iff a variable $v_i \in \text{scope}(c) = \{v_1, \dots, v_n\}$ is assigned any value $d_i \in \text{dom}(v_i)$, there exist compatible values in the domains of all the other variables $d_j \in \text{dom}(v_j)$, $1 \leq j \leq n$, $j \neq i$. Bound and range consistency are defined for constraints over finite intervals. A constraint c is *bound consistent* iff a variable v_i is assigned $d_i \in \{\min(\text{dom}(v_i)), \max(\text{dom}(v_i))\}$ there exist consistent values between the minimum and maximum domain value for all the other variables in the scope of the constraint, called a *bound support*. A constraint is *range consistent* iff a variable is assigned any value in its domain, there exists a bound support. Range consistency is in between domain and bound consistency, where domain consistency is the strongest of the three local consistency properties.

Constraint Answer Set Programming Constraint logic programming naturally merges CP and logic programming, while preserving the advantages of either approach to modelling and solving CSP. Formally, a *constraint logic program* is a logic program P over an alphabet distinguishing regular atoms \mathcal{A} and constraint atoms \mathcal{C} , such that $\text{head}(r) \in \mathcal{A}$ for each $r \in P$ [Gebser *et al.*, 2009b]. A function $\gamma : \mathcal{C} \rightarrow C$ associates constraint atoms with constraints. (The set C stems from the definition of CSP.) For sets of constraints $C' \subseteq C$ define $\gamma(C') = \{\gamma(c) \mid c \in C'\}$. Given a constraint logic program P over \mathcal{A} and \mathcal{C} , and an assignment A , a set $X \subseteq \mathcal{A}$ is a *constraint answer set* of P with respect to A iff X is an answer set of the *constraint reduct* [Gebser *et al.*, 2009b]:

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

The idea in our translation-based approach to constraint answer set solving is to compile a constraint logic program into a (normal) logic program by adding an ASP reformulation of constraint variables and all constraints that appear in the constraint logic program. This allows us to apply CDNL to compute constraint answer sets. A key advantage is that nogood learning techniques can exploit constraint interdependencies since all variables will be shared between constraints. This can improve propagation between constraints. Our reformulations also provides a propagator for the negation of a constraint.

3 Reformulating CASP into ASP

We now present four ASP encodings for variables and constraints over finite domains. All constraints c are reified via atoms $\text{sat}(c)$, and $\text{violate}(c)$, indicating whether c is satisfied or violated, respectively. To ensure consistency, i.e., either $\text{sat}(c)$ or $\text{violate}(c)$ is in an answer set, we post

$$\text{sat}(c) \leftarrow \text{not } \text{violate}(c) \\ \text{violate}(c) \leftarrow \text{not } \text{sat}(c)$$

for every constraint c . Other representations, e.g., using choice rules, are also possible. To save the reader from multiple superscripts, in the following, we will assume $\text{dom}(v) = [1, d]$ for all $v \in V$.

Direct Encoding A straightforward encoding is the *direct encoding* in which an atom $e(v, i)$ is introduced for each constraint variable v and each value i from their domain, representing $v = i$. Intuitively, $e(v, i)$ is in an answer set if v takes the value i , and it is not if v takes a value different from i . For each v , possible assignments are encoded by a choice rule (1). Furthermore, we specify that v takes at least one value (2) and that it takes at most one value (3).

$$\{e(v, 1), \dots, e(v, d)\} \leftarrow \quad (1)$$

$$\perp \leftarrow \text{not } e(v, 1), \dots, \text{not } e(v, d) \quad (2)$$

$$\perp \leftarrow 2 \{e(v, 1), \dots, e(v, d)\} \quad (3)$$

A constraint c is encoded as forbidden combination of values, i.e., if $v_1 = d_1, v_2 = d_2, \dots, v_n = d_n$ is such a forbidden combination then we encode

$$\text{violate}(c) \leftarrow e(v_1, d_1), e(v_2, d_2), \dots, e(v_n, d_n).$$

Unfortunately, the direct encoding hinders propagation:

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

The support encoding has been proposed in the domain of SAT to tackle this weakness [Gent, 2002].

Support Encoding We now encode support information for assignments rather than the encoding of conflicts. For each possible assignment to a variable one of its supports must hold, that is, the set of values for the other variable which allow this assignment. Formally, a *support* for a constraint variable v to take the value i across a constraint c is the set of values $\{i_1, \dots, i_m\} \subseteq \text{dom}(v')$ of another variable in $v' \in \text{scope}(c) \setminus \{v\}$ which allow $v = i$, and can be encoded in the following rule, based on (1–3):

$$\text{violate}(c) \leftarrow e(v, i), \text{not } e(v', i_1), \dots, \text{not } e(v', i_m).$$

It can be read as whenever $v = i$, then at least one of its supports must hold, otherwise the constraint is violated. 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 have used program transformation [Simons *et al.*, 2002] in [Drescher and Walsh, 2010] to reformulate ALL-DIFFERENT straightforwardly according to our support encoding into $\mathcal{O}(d)$ cardinality rules:

$$\text{violate}(c) \leftarrow 2 \{e(v_1, i), \dots, e(v_n, i)\} \quad (4)$$

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

Range Encoding In the *range encoding*, we represent that a variable can take values from an interval $v \in [l, u]$, i.e., a value between l and u (inclusive). An atom $r(v, l, u)$ is introduced for each v and $[l, u] \subseteq [1, d]$. For each range $[l, u]$, the following $\mathcal{O}(nd^2)$ rules encode $v \in [l, u]$ whenever $v \notin$

$[1, l - 1]$ and $v \notin [u + 1, d]$, and enforce a consistent set of ranges, i.e., $v \in [l, u]$ implies $v \in [l - 1, u]$ and $v \in [l, u + 1]$:

$$r(v, l, u) \leftarrow \text{not } r(v, l - 1), \text{not } r(v, u + 1, d) \quad (5)$$

$$\perp \leftarrow r(v, l - 1, u), \text{not } r(v, l, u) \quad (6)$$

$$\perp \leftarrow r(v, l, u + 1), \text{not } r(v, l, u) \quad (7)$$

Constraints are encoded into integrity constraints representing conflict regions $v_1 \in [l_1, u_1], \dots, v_n \in [l_n, u_n]$:

$$\text{violate}(c) \leftarrow r(v_1, l_1, u_1), \dots, r(v_n, l_n, u_n)$$

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

An efficient propagator for ALL-DIFFERENT enforces range consistency by pruning Hall intervals [Leconte, 1996]. A Hall interval of size k completely contains the domains of k variables, formally, $|\{v \mid \text{dom}(v) \subseteq [l, u]\}| = u - l + 1$. 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 (cf. [Bessière *et al.*, 2009a]). We encode ALL-DIFFERENT such that no interval $[l, u]$ can contain more variables than its size:

$$\text{violate}(c) \leftarrow u - l + 2 \{r(v_1, l, u), \dots, r(v_n, l, u)\}. \quad (8)$$

This simple reformulation can simulate a complex propagation algorithm like the one in [Leconte, 1996] with a similar overall complexity.

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

Bound Encoding In our *bound encoding*, similar to the *order encoding* [Tamura *et al.*, 2006], an atom $b(v, i)$ is introduced for each variable v and value i to represent that v is bounded by i , i.e., $v \leq i$. For each v , possible assignments are encoded by a choice rule (9). To ensure a consistent set of bounds, (10) encodes that $v \leq i$ implies $v \leq i + 1$. Finally, (11) encodes $v \leq d$, i.e., some value must be assigned to v .

$$\{b(v, 1), \dots, b(v, d)\} \leftarrow \quad (9)$$

$$\perp \leftarrow b(v, i), \text{not } b(v, i + 1) \quad (10)$$

$$\perp \leftarrow \text{not } b(v, d) \quad (11)$$

Similar to the range encoding, we represent conflict regions $l_1 < v_1 \leq u_1, \dots, l_n < v_n \leq u_n$ as below

$$\text{violate}(c) \leftarrow b(v_1, u_1), \dots, b(v_n, u_n), \\ \text{not } b(v_1, l_1), \dots, \text{not } b(v_n, l_n).$$

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

In order to achieve a reformulation of ALL-DIFFERENT that can only prune bounds, the bound encoding for variables is linked to (8) as follows:

$$r(v, l, u) \leftarrow \text{not } b(v, l - 1), b(v, u) \quad (12)$$

$$\perp \leftarrow r(v, l, u), b(v, l - 1) \quad (13)$$

$$\perp \leftarrow r(v, l, u), \text{not } b(v, u) \quad (14)$$

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

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

Table 1: Runtime results in seconds for pigeon hole problems.

4 Experiments

We have conducted experiments on hard combinatorial problems modelled with ALL-DIFFERENT constraints that stem from CSPLib [Gent and Walsh, 1999]. Experiments consider different options in our translation-based approach to constraint answer set solving. We denote the support encoding by S , the bound encoding by B , and the range encoding by R . To explore the impact of small Hall intervals, we also tried B_k and R_k , an encoding with only those cardinality rules (8) for which $u - l + 1 \leq k$. The consistency achieved by B_k and R_k may be weaker than bound and range consistency, respectively, when $k < n$. We also include the hybrid CASP systems *clingcon* (0.1.2), and *ezcsp* (1.6.9) in our empirical analysis. While *clingcon* extends the ASP system *clingo* (2.0.2) with the CP solver *gecode* (2.2.0), *ezcsp* combines the grounder *gringo* (2.0.3) and ASP solver *clasp* (1.3.0) with *sicstus* (4.0.8) as CP solver. (Note that the system *clingo* combines the grounder *gringo* and ASP solver *clasp* in a monolithic way.) To provide a representative comparison with *clingcon* and *ezcsp*, we have applied *clingo* (2.0.3) to the encodings in our translation-based approach. To compare the performance of constraint answer set solvers against traditional CP, 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.

Pigeon Hole Problems The famous *pigeon hole problem* is to show that it is not possible to assign n pigeons to $n-1$ holes if each pigeon must be assigned a distinct hole. As can be seen from the results shown in Table 1, our bound and range encodings perform significantly faster compared to weaker encodings and the other options using filtering algorithms for the ALL-DIFFERENT constraint that achieve arc consistency on its binary decomposition. However, as can be expected on such problems, detecting large Hall intervals is essential.

Quasigroup Completion A *quasigroup* is an algebraic structure over n elements and can be represented by an $n \times n$ -multiplication table such that each element in the structure occurs exactly once in each row and each column of the table. The *quasigroup completion problem* is to show whether a partially filled table can be completed to a multiplication table of a quasigroup. We have included models for *gecode* that enforce bound and domain consistency on ALL-DIFFERENT, denoted *gecode_B* and *gecode_D*, respectively, in our experiments. Table 2 gives the runtime for solving QCP of size

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

Table 2: Average times over 100 runs on quasigroup completion problems. Timeouts, if any, are given in parenthesis.

$n = 20$. The left-most column gives the ratio of preassigned entries. The results demonstrate phase transition behaviour in the systems *ezcsp*, *clingcon*, *gecode*, and *gecode_B*, while our ASP encodings and *gecode_D* (not shown) solve all problems within seconds. We conclude that learning constraint interdependencies as in our approach (using CDNL) is sufficient to tackle quasigroup completion, i.e., specialised algorithms that enforce domain consistency are not necessary.

Quasigroup Existence The *quasigroup existence problem* is to determine the existence of certain interesting classes of quasigroups with some additional properties ([Fujita *et al.*, 1993]). The properties are represented by axioms #1 – #7 in the direct encoding. In *ezcsp* and *gecode*, we additionally use constructive disjunction. Their logic programming equivalent are integrity constraints, exploited in the options S , B_k , R_k and *clingcon*. As for *ezcsp* and *clingcon* on benchmark classes #1 to #4, our results presented in Table 3 suggest that both constructive disjunction and integrity constraints have a similar behaviour. However, our encodings benefit again from learning constraint interdependencies, resulting in runtimes that outperform all other systems including *gecode* on the hardest problems.

Graceful Graphs A labelling of the nodes in a graph (V, E) is *graceful* if it assigns a unique label from the integers in $[0, |E|]$ such that, when each edge is labelled with the distance between its nodes' labels, the resulting edge labels are all different. The *graceful graph problem* is to determine the existence of such a labelling. We use auxiliary variables for edge labels. Their relation to node labels is represented in the direct encoding which weakens the overall consistency. Table 4 shows our results for *double wheel graphs*, i.e., graphs composed of two copies of a cycle with n nodes, each connected to a central hub. Our encodings compete with *ezcsp* and outperform the other systems, whilst the support encoding performs better than bound and range encodings. We observe some variability in the results for B_k and R_k , e.g., for $n = 8$ the options B_1 and B solve the problem within the time limit but B_3 does not, although B_3 contains B_1 . We explain this variability by

#	n	S	B_1	B_3	B	R	$ezcsp$	$clingcon$	$gecode$
1	7	1.7	1.7	1.7	1.7	1.6	65.0	189.8	0.6
1	8	19.0	5.9	4.7	19.8	4.7	—	—	—
1	9	—	139.4	152.0	234.6	466.9	—	—	—
2	7	1.7	1.7	1.7	1.8	1.8	46.1	1.5	1.2
2	8	46.6	9.6	10.6	37.7	14.8	—	—	—
2	9	—	246.0	55.7	88.3	213.4	—	—	—
3	7	0.2	0.2	0.2	0.3	0.3	3.2	1.0	0.0
3	8	0.4	0.4	0.5	0.5	0.5	4.3	9.0	0.2
3	9	10.2	7.4	9.5	16.5	12.8	—	—	18.2
4	7	0.2	0.2	0.2	0.3	0.3	2.8	0.7	0.1
4	8	0.5	0.6	0.7	0.9	0.7	27.9	36.8	0.3
4	9	1.3	1.0	2.1	3.0	0.9	442.1	288.8	3.7
5	10	1.6	1.5	1.6	1.9	1.6	—	—	0.2
5	11	2.1	2.2	2.4	3.4	2.4	—	—	0.8
5	12	27.0	6.2	9.1	12.4	10.4	—	—	16.4
6	10	1.2	1.4	1.5	1.8	1.5	10.5	—	0.1
6	11	2.7	2.8	4.0	4.2	4.8	125.5	—	1.2
6	12	32.0	12.9	25.6	36.4	50.6	—	—	24.6
7	8	0.4	0.4	0.4	0.6	0.5	1.1	—	0.1
7	9	0.7	1.0	1.2	1.7	1.4	9.1	—	0.9
7	10	6.7	3.2	5.2	8.0	4.6	—	—	22.0

Table 3: Results in seconds for quasigroup existence.

n	S	B_1	B_3	B	R	$ezcsp$	$clingcon$	$gecode$
4	1.3	2.0	1.5	3.2	2.5	0.6	0.1	0.1
5	4.5	5.0	4.5	13.5	31.4	1.0	2.0	0.1
6	7.2	11.0	17.6	47.7	110.2	1.2	—	7.2
7	23.8	28.3	67.9	227.9	432.9	18.0	—	—
8	48.4	68.4	—	207.8	356.8	4.3	—	—
9	82.8	106.5	200.4	486.6	227.4	390.5	—	—

Table 4: Results in seconds for graceful graph problems.

the lookback-based branching heuristic used by *clingo* being misled by the extra variables introduced in B_k and R_k . This is inherent to a growing size of the encoding.

5 Related Work

Most previous work integrates CP techniques into ASP to avoid huge ground instantiations given through logic programs with first-order variables over large domains. An ASP system was extended in [Baselice *et al.*, 2005; Mellarkod and Gelfond, 2008; Mellarkod *et al.*, 2008] such that it does not require full grounding, since variables and limitations on their domains can be handled in the CP solver. A similar approach presented in [Dal Palù *et al.*, 2009] employs the CP solver to compute also the answer sets. Although these hybrid strategies potentially eliminate the bottleneck that is inherent to the translation-based approach, they view ASP and CP solvers as blackboxes which do not match the performance of state-of-the-art SMT solvers. In particular, they do not make use of conflict-driven learning and back-jumping techniques. This gap was closed by the approach taken in [Gebser *et al.*, 2009b] following the one by SMT solvers in letting the ASP solver deal with the propositional structure of the

logic program, while a CP solver addresses the constraints. Apart from extending the unit-propagation of an ASP solver through constraint propagation, it deals with the elaboration of reasons for atoms derived by constraint propagation within conflict resolution. The elaboration of conflict information from constraint propagators, however, is limited since constraint propagators lack support for this feature (they would have to keep an implication graph to record reasons for each propagation step). Hence, the conflict resolution process cannot exploit constraint interdependencies. A different hybrid approach to solving CASP is presented in [Balduccini, 2009], where an answer set of a logic program with constraint atoms encodes a desired CSP which, in turn, is handled by a CP system. A more general framework using multiple declarative paradigms to specify CSP is proposed in [Järvisalo *et al.*, 2009]. Either approach, however, restricts communication between different solver types in order to compute solutions to the whole CASP model, e.g., they also do not incorporate conflict-driven learning and back-jumping techniques.

In a translation-based approach, all parts of the model are mapped into a single constraint language for which highly efficient off-the-shelf solvers are available. Hence, related work has mostly focussed on the translation of constraints to SAT (cf. [Walsh, 2000; Gent, 2002]). Translation into ASP, however, can be more general than translation into SAT: Every nogood can be syntactically represented by a clause, but other ASP constructs are also possible, such as cardinality and weight constraints [Simons *et al.*, 2002]. ASP was put forward as a novel paradigm for modelling and solving CSP in [Niemelä, 1999], where straightforward encodings to represent generic constraints via either allowed or forbidden combination of values has been presented. Preliminary work on translating CASP into ASP was conducted in [Gebser *et al.*, 2009a], but they did not consider what level of consistency was achieved by their translation.

Decompositions of ALL-DIFFERENT into simple arithmetic constraints such that bound and range consistency can be achieved were proposed in [Bessière *et al.*, 2009a]. There is no polynomial-sized decomposition that achieves domain consistency [Bessière *et al.*, 2009b].

6 Conclusions

We have shown that constraint answer set programming is a promising approach to representing and solving combinatorial problems that naturally merges CP and ASP, while preserving the advantages of both paradigms. We have presented a translation-based approach to constraint answer set solving. In particular, we have proposed various generic ASP encodings for constraints on finite domains such that the unit-propagation of an ASP solver achieves a certain type of local consistency. We have formulated our techniques as a preprocessor that can be applied to existing ASP systems without changing their source code. This allows for programmers to select the solver that best fit their needs. An empirical evaluation of the computational impact on benchmarks from CP has shown our approach outperforming CP and hybrid CASP systems on most instances. As a key advantage we have identified that CDNL exploits constraint interdependencies which

can improve propagation between constraints.

Future work concerns the combination of our translation-based approach with a hybrid CASP system centred around lazy nogood generation (cf. lazy clause generation in [Ohrimenko *et al.*, 2009]) to combine the advantages of either approach. We will also explore the different choices that arise from this combination.

Acknowledgements NICTA is funded by the Department of Broadband, Communications and the Digital Economy, and the Australian Research Council.

References

- [Balduccini, 2009] M. Balduccini. Representing constraint satisfaction problems in answer set programming. In *Proceedings of ICLP'09, ASPOCP'09 Workshop*, 2009.
- [Baral, 2003] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
- [Baselice *et al.*, 2005] S. Baselice, P. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In *Proceedings of ICLP'05*, pages 52–66. Springer, 2005.
- [Bessière *et al.*, 2009a] C. Bessière, G. Katsirelos, N. Narodytska, C.-G. Quimper, and T. Walsh. Decompositions of all different, global cardinality and related constraints. In *Proceedings of IJCAI'09*. AAAI Press/The MIT Press, 2009.
- [Bessière *et al.*, 2009b] C. Bessière, G. Katsirelos, N. Narodytska, and T. Walsh. Circuit complexity and decompositions of global constraints. In *Proceedings of IJCAI'09*, pages 412–418, 2009.
- [Biere *et al.*, 2009] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.
- [Dal Palù *et al.*, 2009] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Answer set programming with constraints using lazy grounding. In *Proceedings of ICLP'09*, pages 115–129. Springer, 2009.
- [Dovier *et al.*, 2005] A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP solutions to NP-complete problems. In *Proceedings of ICLP'05*, pages 67–82. Springer, 2005.
- [Drescher and Walsh, 2010] C. Drescher and T. Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming*, 10(4-6):465–480, 2010.
- [Fujita *et al.*, 1993] M. Fujita, J. K. Slaney, and F. Bennett. Automatic generation of some results in finite algebra. In *Proceedings of IJCAI'93*, pages 52–59. Morgan Kaufmann Publishers, 1993.
- [Gebser *et al.*, 2007] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of IJCAI'07*, pages 386–392. AAAI Press/MIT Press, 2007.
- [Gebser *et al.*, 2009a] M. Gebser, H. Hinrichs, T. Schaub, and S. Thiele. xpanda: A (simple) preprocessor for adding multi-valued propositions to ASP. In *Proceedings of WLP'09*, 2009.
- [Gebser *et al.*, 2009b] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *Proceedings of ICLP'09*, pages 235–249. Springer, 2009.
- [Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP'88*, pages 1070–1080. The MIT Press, 1988.
- [Gent and Walsh, 1999] I. P. Gent and T. Walsh. CSPLIB: A benchmark library for constraints. In *Proceedings of CP'99*, pages 480–481. Springer, 1999.
- [Gent, 2002] I. P. Gent. Arc consistency in SAT. In *Proceedings of ECAI'02*, pages 121–125. IOS Press, 2002.
- [Järvisalo *et al.*, 2009] M. Järvisalo, E. Oikarinen, T. Janhunen, and I. Niemelä. A module-based framework for multi-language constraint modeling. In *Proceedings of LPNMR'09*, pages 155–169. Springer, 2009.
- [Leconte, 1996] M. Leconte. A bounds-based reduction scheme for constraints of difference. In *CP'96, Workshop on Constraint-based Reasoning*, 1996.
- [Mellarkod and Gelfond, 2008] V. Mellarkod and M. Gelfond. Integrating answer set reasoning with constraint solving techniques. In *Proceedings of FLOPS'08*, pages 15–31. Springer, 2008.
- [Mellarkod *et al.*, 2008] V. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):251–287, 2008.
- [Niemelä, 1999] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
- [Nieuwenhuis *et al.*, 2006] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. 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, 2006.
- [Ohrimenko *et al.*, 2009] O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
- [Rossi *et al.*, 2006] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.
- [Simons *et al.*, 2002] P. Simons, I. Niemelä, and T. Soinen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.
- [Tamura *et al.*, 2006] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. In *Proceedings of CP'06*, pages 590–603. Springer, 2006.
- [Walsh, 2000] T. Walsh. SAT v CSP. In *Proceedings of CP'00*, pages 441–456. Springer, 2000.