Inductive Definitions in Constraint Programming

Rehan Abdul Aziz **Peter J. Stuckey** **Zoltan Somogyi**

Department of Computing and Information Systems,
The University of Melbourne and National ICT Australia (NICTA)

Email: raziz@student.unimelb.edu.au, pjs@csse.unimelb.edu.au, zs@unimelb.edu.au

Abstract

Constraint programming (CP) and answer set programming (ASP) are two declarative paradigms used to solve combinatorial problems. Many modern solvers for both these paradigms rely on partial or complete Boolean representations of the problem to exploit the extremely efficient techniques that have been developed for solving propositional satisfiability problems. This convergence on a common representation makes it possible to incorporate useful features of CP into ASP and vice versa. There has been significant effort in recent years to integrate CP into ASP, primarily to overcome the *grounding* bottleneck in traditional ASP solvers that exists due to their inability to handle integer variables efficiently. On the other hand, ASP solvers are more efficient than CP systems on problems that involve inductive definitions, such as reachability in a graph. Besides efficiency, ASP syntax is more natural and closer to the mathematical definitions of such concepts. In this paper, we describe an approach that adds support for answer set rules to a CP system, namely the lazy clause generation solver chuffed. This integration also naturally avoids the grounding bottleneck of ASP since constraint solvers natively support finite domain variables. We demonstrate the usefulness of our approach by comparing our new system against two competitors: the state-of-the-art ASP solver clasp, and clingcon, a system that extends clasp with CP capabilities.

Keywords: Answer set programming, constraint programming, stable model semantics, inductive definitions.

1 Introduction

Constraint programming (Rossi et al. 2006) is a declarative programming paradigm that is used to solve a wide range of computationally difficult problems. It allows users to encode a problem as a concise mathematical model, and pass it to a constraint solver that computes solution(s) that satisfy the model. The goal of CP is that users should find writing models as natural and as easy as possible, which requires hiding away all the complexity involved in finding those solutions inside the constraint solvers, where only the solvers' implementers have to see it.

Motivated by the efficient engineering techniques developed in the domain of propositional satisfiability (SAT) solving (Mitchell 2005), a recent highly competitive constraint solving approach, *lazy clause generation* (Ohrimenko et al. 2009, Feydy & Stuckey 2009) builds an on-the-fly Boolean representation of the problem during execution. This keeps the size of the representation small. The propagators record their results of failed searches as Boolean clauses (*nogoods*) so that the solver can later use SAT unit propagation on those clauses to find other instances of that failure elsewhere in the search tree much more quickly.

Constraint modelling languages allow users to succinctly define many natural notions, particularly when using *global constraints*, which can capture the entirety of some substructure of the problem. Global constraints also allow solvers to use efficient specialized reasoning about these substructures. But constraint modelling cannot naturally capture some important constructs, such as transitive closure. (Propositional) definite logic programs do allow the modelling of transitive closure efficiently, because they rely on a least model semantics, which ensures that *positive recursion* in the rules among a set of atoms, i.e. circular support between these atoms to establish each other's truth, is not sufficient to cause an atom to be true. When we extend logic programs to normal logic programs, which allow negative literals in the body, we can extend the least model approach in at least two ways which still maintain this property: the *stable model* (Gelfond & Lifschitz 1988) and the *well-founded model* (Van Gelder et al. 1988).

Answer set programming (ASP) (Baral 2003), based on stable model semantics, is another form of declarative programming. Answer set solvers take as input a normal logic program, usually modelling a combinatorial problem, and calculate its stable models, each of which corresponds to one of the problem's solutions. The incorporation of some of the engineering techniques originally developed for SAT solvers (such as nogood learning) in answer set solvers has resulted in excellent performance (Gebser et al. 2007). The implementation of these techniques relies on translating the normal logic program back to propositional formulas, an approach which was first proposed by Lin & Zhao (2004). Representing the program as its Clark's completion introduces practically no overhead, but detecting *unfounded sets* (Van Gelder et al. 1988), sets of atoms that are supported only by each other but have no *external* support, is far from straightforward. The main reason for this is that a program can potentially have a very large number of unfounded sets (Lifschitz & Razborov 2006). To tackle this, Gebser et al. (2007) use a principle similar to lazy clause generation: they calculate and record unfounded sets *lazily* during the search, as the

need arises.

The effectiveness of the above approach motivates us to incorporate stable model semantics into our modelling language MiniZinc (Nethercote et al. 2007) in order to broaden the scope of problems that we can model; in particular, we are thinking about problems such as reachability, whose mathematical descriptions require induction. Unless specialized graph constraints are used (Dooms et al. 2005, Viegas & Azevedo 2007), these problems cannot be efficiently handled by existing constraint solvers. In this paper, we propose the use of inductive definitions in MiniZinc and empirically demonstrate its usefulness. We describe two implementations of unfounded set calculation as propagators for the lazy clause generator chuffed. These implementations are based on the *source pointer* technique (Simons et al. 2002) combined with either of the unfounded set algorithms described by Anger et al. (2006) and Gebser et al. (2012).

The rest of the paper is organized as follows. Section 2 lays out the theoretical background required for this paper. Section 3 describes, with the help of a running example, how recursive definitions under propositional semantics lack the ability to model certain problems correctly and efficiently, and presents an extension of the MiniZinc modelling language as a solution. Section 4 explains how this extension may be implemented. Section 5 describes one of our two implementations in detail. We evaluate both implementations experimentally in Section 6. We then discuss related work by other authors in Section 7.

2 Background

Constraints and propagators

We consider constraints over a set of variables $\mathcal{V}$. We divide $\mathcal{V}$ into two disjoint sets, namely integer variables $\mathcal{I}_\mathcal{V}$ and Boolean atoms $\mathcal{A}_\mathcal{V}$. A literal is an atom or its negation. A domain D is a mapping: from $\mathcal{I}_\mathcal{V}$ to fixed finite sets of integers, and from $\mathcal{A}_\mathcal{V}$ to sets over $\{\top, \bot\}$. A domain D_1 is *stronger* than a domain D_2, written $D_1 \sqsubseteq D_2$, if $D_1(x) \subseteq D_2(x)$ for all $x \in \mathcal{V}$. A valuation θ is a mapping of variables to a single value in their domains, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. Let $vars$ be a function that returns the set of variables that appear in any expression. We say that a valuation θ is an element of the domain D, written $\theta \in D$, if $\theta(x) \in D(x)$ for all $x \in vars(\theta)$. A valuation θ is *partial* if $vars(\theta) \subset \mathcal{V}$ and *complete* if $vars(\theta) = \mathcal{V}$.

A constraint c is a restriction on the values that a set of variables, represented by $vars(c)$, can be simultaneously assigned. In our setting, a constraint c is associated with one or more *propagators* that operate on $vars(c)$. Propagators for a constraint work by narrowing down the values that the variables of the constraint can take. More formally, a propagator f is a monotonically decreasing function from domains to domains, that is, $f(D) \sqsubseteq D$, and $f(D_1) \sqsubseteq f(D_2)$ if $D_1 \sqsubseteq D_2$. A propagator f for a constraint c is *correct* iff for all possible domains D, and for all solutions θ to c, if $\theta \in D$, then $\theta \in f(D)$.

A CP problem is a pair (C, D) consisting of a set of constraints C and a domain D. A constraint programming solver solves (C, D) by interleaving propagation with choice. It applies all propagators F for constraints C to the current domain D, and it does so repeatedly until no propagator makes a change (i.e. until a fixpoint is reached). If the final domain D' represents failure ($D(x) = \emptyset$ for some x) then it backtracks to try another choice. If all variables have at least one element in their domains, but some have two or more, then the solver needs to make a choice by splitting the domain of one of these variables into two parts. This *labelling step* results in two subproblems (C, D'') and (C, D'''), which the solver then solves recursively. If all variables have exactly one value in their domains, then there are no choices left to be made, and the domain is actually a valuation. Whether that valuation satisfies all the constraints can be trivially checked, although this check is unnecessary if the propagators of all constraints are guaranteed to find any failures of those constraints. In practice, solvers use event-driven scheduling of propagators and priority mechanisms to try to reach fixpoints as quickly as possible (Schulte & Stuckey 2008).

(Propositional) Normal logic programs

In our proposed system, we divide the set of atomic variables $\mathcal{A}_\mathcal{V}$ into two disjoint subsets: the set of *default* variables $\mathcal{D}_\mathcal{V}$, and the set of non-default variables $\mathcal{N}_\mathcal{V}$. A *normal rule* r has the form:

$$a \leftarrow p_1, \ldots, p_j, {\sim}n_1, \ldots, {\sim}n_k$$

where $a \in \mathcal{D}_\mathcal{V}$ and $\{p_1, \ldots, p_j, n_1, \ldots, n_k\} \subseteq \mathcal{A}_\mathcal{V}$. We say that a is the *head* of r, written r_H, and $\{p_1, \ldots, p_j, {\sim}n_1, \ldots, {\sim}n_k\}$ is the *body* of r, written r_B. To allow us to represent the truth or falsity of each rule body, we have the *bodyRep* function, which maps each rule body to a new *body atom* $b \in \mathcal{N}_\mathcal{V}$. We also have functions that return the positive and negative atoms in each rule body: if $bodyRep(r_B) = b$, then $pos(b) = \{p_1, \ldots, p_j\}$ and $neg(b) = \{n_1, \ldots, n_k\}$. We call the set of positive default literals of the body $b^+ = \{p \mid p \in pos(b), p \in \mathcal{D}_\mathcal{V}\}$.

A normal logic program (NLP) is a set of normal rules. We consider an NLP $\mathcal{P}$ as a constraint. We define the following functions for a default atom a and a body atom b. The set of all body atoms in the program is $bodies(\mathcal{P}) = \{bodyRep(r_B) \mid r \in \mathcal{P}\}$; the set of bodies of the rules whose head is a is $body(a) = \{bodyRep(r_B) \mid r \in \mathcal{P}, r_H = a\}$; the set of heads supported by b is $supHead(b) = \{r_H \mid r \in \mathcal{P}, bodyRep(r_B) = b\}$; and the set of body atoms in whose positive parts $a \in \mathcal{D}_\mathcal{V}$ appears is $posInBody(a) = \{c \mid c \in bodies(\mathcal{P}), a \in c^+\}$.

We use the concept of *positive body-head dependency graph* as defined in (Gebser & Schaub 2005). It is a directed graph $(\mathcal{D}_\mathcal{V} \cup bodies(\mathcal{P}), E(\mathcal{P}))$ where $E(\mathcal{P}) = \{(a, b) \mid a \in \mathcal{D}_\mathcal{V}, b \in posInBody(a)\} \cup \{(b, a) \mid b \in bodies(\mathcal{P}), a \in supHead(b)\}$.

We associate each strongly connected component of the graph with a number, and we map every atom and body literal in the component to this number through a function *scc*.

Our implementation of stable model semantics relies on the translation of logic programs into propositional theories. Given a default atom $a \in \mathcal{D}_\mathcal{V}$, the Clark completion (Clark 1978) of its definition, $Comp(a)$, is the formula $a \leftrightarrow \vee_{b \in body(a)} b$. The completion of $\mathcal{P}$ is the formula:

$$Comp(\mathcal{P}) = \bigwedge_{x \in \mathcal{D}_\mathcal{V}} Comp(x)$$

The following formula ensures that all body literals are equal to the conjunction of their literals:

$$Body(\mathcal{P}) = \bigwedge_{b \in bodies(\mathcal{P})} (b \leftrightarrow \bigwedge_{p \in pos(b)} p \wedge \bigwedge_{n \in neg(b)} \neg n)$$

We refer to the conjunctive normal form (CNF) of the formula $Comp(P) \wedge Body(P)$ as $Clauses(P)$.

Given a valuation θ, a set $U \subseteq \mathcal{D}_\mathcal{V}$ is *unfounded* with respect to θ iff for every rule $r \in \mathcal{P}$ and $bodyRep(r_B) = b$:

1. $r_H \notin U$ or
2. $\theta(p) = \bot$ for some $p \in pos(b)$ or $\theta(n) = \top$ for some $n \in neg(b)$ or
3. $b^+ \cap U \neq \emptyset$.

Basically, a set is unfounded if every default variable in it depends on some other variable in it being true, but none of them have *external support*, i.e. none of them can be proven true without depending on other default variables in the set. This is expressed most directly by the third alternative. The previous alternatives cover two different uninteresting cases: rules whose heads are not in the potentially unfounded set we are testing, and rules whose bodies are known to be false.

Finally, we say that a complete valuation θ satisfies P, or that θ is a *constraint stable model* of P iff $\theta \models Clauses(P)$ and there is no $U \subseteq \mathcal{D}_\mathcal{V}$ such that U is an unfounded set with respect to θ.

Constraint stable models can also be defined through use of *constraint reducts* (Gebser et al. 2009). Given a valuation θ such that $vars(\theta) \supseteq \mathcal{N}_\mathcal{V}$ and $vars(\theta) \cap \mathcal{D}_\mathcal{V} = \emptyset$, the constraint reduct of $\mathcal{P}$ with respect to θ, written $\mathcal{P}^\theta$, is a version of $\mathcal{P}$ that has no non-default variables. The reduct is computed by first removing the rules whose bodies have one or more literals that are not satisfied by θ, and then removing satisfied non-default atoms from the rule bodies. A complete valuation θ' that extends θ is a constraint stable model of $\mathcal{P}$ if θ' is a stable model of the reduced program $\mathcal{P}^\theta$.

3 Inductive definitions

The goal of this section is to demonstrate the usefulness of inductive definitions in constraint modelling languages with the help of an example. Consider the *Connected Dominating Set* problem.[1] Given a graph `G` defined by `n` nodes numbered `1..n` and a (symmetric) adjacency 2D array `edge` where `edge[i,j] = edge[j,i] = true` iff the nodes `i` and `j` are adjacent. A connected dominating set `D` is a subset of `1..n` such that for each node `i`, either `i` or one of its neighbours is in `D`, and the set `D` is connected (each node in `D` can reach other nodes in `D` by a path of edges both of whose endpoints are in `D`). The problem is to find a dominating set with at most `k` nodes for a given graph.

A MiniZinc (Nethercote et al. 2007) model for this problem excluding the connectedness condition is:

```
int: k;                       % size limit
int: n;                       % nodes in G
set of int: N = 1..n;         % node set
array[N,N] of bool: edge; % edges in G

array[N] of var bool: d;  % is member of D?

constraint sum(i in N)(bool2int(d[i])) <= k;
constraint forall(i in N)
  d[i] \/ exists(j in N where
    edge[i,j])(d[j]);
```

[1] See `http://dtai.cs.kuleuven.be/events/ASP-competition/Benchmarks/ConnectedDSet.shtml`

But modelling the connectedness condition in CP is very difficult. We may imagine we can model this by saying each node in `D` is adjacent to another node in `D`.

```
constraint forall(i in N)
  (d[i] -> exists(j in N, j != i)
    (d[j] /\ edge[i,j]);
```

This model is incorrect, for example for the symmetric completion of the asymmetric graph with edges `{(1,2),(2,3),(3,4),(4,5),(4,6),(5,6)}` and limit 4, it has a solution `D = {1,2,4,5}` which is dominating but not connected.

We need to define a base case for connectedness, and reason about reachability from there in terms of distance: `reach[n,s]` means we can reach node `n` from base node `min_idx` (the node in `D` with least index) in `s` or fewer steps. (The `var N` indicates that `min_idx` must belong to the previously defined set `N`.)

```
var N: min_idx = min(i in N)
  (i + bool2int(not d[i])*n);
array[N,0..n-1] of var bool: reach;
constraint forall(i in N)
  (reach[i,0] <-> i == min_idx);
constraint forall(i in N)
  (forall(s in 0..n-2)
    (reach[i,s+1] <-> (reach[i,s] \/
       exists(j in N where edge[i,j])
         (d[i] /\ reach[j,s]))));
constraint forall(i in N)
  (d[i] -> reach[i,n-1]);
```

The model defines only `min_idx` as reachable with 0 steps, and node `i` is reachable in `s+1` steps (`reach[i,s+1]`) if it was reachable previously or if it is in `d` and there is an adjacent node `j` reachable in `s` steps. The model is correct giving two answers `D = {2,3,4,5}` and `D = {2,3,4,6}`. This model is very expensive, requiring `n*n` Boolean variables to define the final connected set.

Lets consider the ASP model for the same problem, shown here in gringo syntax:

```
% select the dominating set
{ dom(U) : vtx(U) }.

% dominating set condition
in(V) :- edge(U,V), dom(U).
in(V) :- dom(V).
:- vtx(U), not in(U).

% connectivity constraints
reach(U) :- dom(U),
  not dom(V) : vtx(V) : V < U.
reach(V) :- reach(U), dom(V), edge(U,V).
:- dom(U), not reach(U).

% size bound
:- not { dom(U) : vtx(U) } K, bound(K).
```

where the input is `vtx(i)` where `i` is in `1..n`, `edge(i,j)` whenever `edge(i,j)`, and `bound(k)` for limit `k`. The size constraint is expressed in negation, while the dominating set is expressed more obscurely. The base case that relies on defining minimum index computation is arguably more transparent. The biggest difference is the reachability condition which is much more succinct and much more efficient.

The advantage of the ASP model is that it makes use of the *inductive* interpretation of the rules for transitive closure. The solution `D = {1,2,4,5}` is

not generated because the transitive closure computation cannot generate `reach(4)` or `reach(5)` from `reach(1)`.

In order to incorporate this succinct modelling, and to take advantage of the efficient solving approaches for this, we extend MiniZinc with *inductively defined predicates*.

For the running example we use an *inductive definition* of the predicate `reach` as follows

```
idpredicate reach(N: i) = i == min_idx \/
  exists(j in N where edge[i,j])
     (d[i] /\ reach(j));
```

and add the constraint

```
constraint forall(i in N)(d[i] -> reach(i));
```

Inductively defined predicates are allowed to have only fixed arguments; that is, their arguments cannot be decision variables. They can use arbitrary MiniZinc in the bodies with the restriction that they cannot introduce new decision variables, and inductively defined literals cannot appear inside non-Boolean constraint expressions. The example above uses the existing decision variables `min_idx` and `d` in the body.

In MiniZinc we assume that inductively defined predicates have an (extended) stable model semantics (Gelfond & Lifschitz 1988). At the moment, we have not defined the stable models of inductive definitions. Instead, we talk about the constraint stable models of an equivalent set of normal rules. What the extended stable models of the inductive definitions should be, and how the translation of the inductive predicate into a set of normal rules should work, are both directions for our future research. We comment further on this topic in Section 8.

4 Mapping inductive definitions to FlatZinc

The first step in solving a MiniZinc model is having the translator program mzn2fzn map it to FlatZinc, a lower level language that is easier for solvers to understand and implement. In this section, we show what this translation has to do for MiniZinc models that contain inductive definitions.

The first task of the translation is identifying the default variables used by inductively defined predicates, and defining them properly. The FlatZinc we want to generate for a default variable named `dv` is:

```
var bool: dv;
default_variable(dv);
```

The first line defines `dv` as a Boolean variable, while the second tells the solver that this variable is a default variable, and not an ordinary Boolean variable. This line defines a predicate we added to FlatZinc:

```
predicate default_variable(var bool: v);
```

Each use of this predicate declares a default Boolean variable. For our running example from previous section, we want to generate this FlatZinc:

```
array[N] of var bool: reach;
default_variable(reach[1]);
...
default_variable(reach[n]);
```

The second task of the translator is replacing the inductively defined predicate for each default variable with a set of normal logic rules that have the same constraint stable models as that predicate.

To make this possible, we have extended FlatZinc with a predicate that represents normal rule definitions:

```
predicate normal_rule(var bool: head,
  array[int] of var bool: pos_atoms,
  array[int] of var bool: neg_atoms);
```

As the names suggest, `pos_atoms` and `neg_atoms` contain the atoms appearing in positive and negative literals in the rule respectively, so the generic normal rule r shown in Section 2 would be represented as

$$\texttt{normal\_rule}(a, [p_1, \ldots, p_j], [n_1, \ldots, n_k])$$

The `head` must be a default variable, but the positive and negative literals have no such restriction.

MiniZinc supports quantifiers, and the inductive definition of `reach` in the previous section included a quantifier. FlatZinc does not allow quantifiers, so we must eliminate them during translation. This requires knowing the data over which such predicates operate. For `reach`, this data is the `edge` predicate. If the `edge` predicate contains the three facts `{(1,2),(2,1),(2,2)}`, representing a small graph with two nodes and three edges, we want to translate `reach` into these constraints:

```
normal_rule(reach[1],[min_idx==1],[]);
normal_rule(reach[2],[min_idx==2],[]);
normal_rule(reach[1],[d[1],reach[2]],[]);
normal_rule(reach[2],[d[2],reach[1]],[]);
normal_rule(reach[2],[d[2],reach[2]],[]);
```

The first step in this translation is the replacement of existential quantifiers in predicate bodies with disjunctions. The second step is the replacement of disjunctions in bodies with two or more rules. This particular translation is essentially a form of the Lloyd-Topor transformation (Lloyd & Topor 1984). While that is the translation we want to do, we have not yet implemented either the translation, or the recognition of `idpredicate` definitions in MiniZinc. Yet to test the effectiveness of our system, we need *some* way to generate FlatZinc code that implements inductive definitions. To do this, we have exploited mzn2fzn's existing ability to expand out quantifications. We have simply added the `normal_rule` predicate to MiniZinc as well as FlatZinc.

To generate the FlatZinc that we would want generated from

```
idpredicate reach(N: i) = i == min_idx \/
  exists(j in N where edge[i,j])
    (d[i] /\ reach(j));
```

a MiniZinc user can now write these constraints:

```
constraint forall(i in N)
  (normal_rule(reach[i],[i==min_idx],[]));

constraint forall(i,j in N where edge[i,j])
  (normal_rule(reach[i],[d[i],reach[j]],[]));
```

Basically, until we implement our full translation, we require users to expand out existential quantifiers and disjunctions for themselves, although as we have shown above, this can be conveniently done with the MiniZinc generator `forall`.

5 Implementation

Before we describe our implementation in detail, let us sketch briefly how propagation works in chuffed. A propagator can subscribe to an event e, written *Subscribe*(e). When the event e takes place, the *WakeUp* function of the propagator is called. At this point, the propagator can *Queue* up for propagation.

Since a single event can wake up more than one propagator, each propagator has a priority; any woken propagators are added to the queue in priority order. The *Propagate* function of a propagator is called after all the higher priority propagators have finished. The code of the *Propagate* function can choose to *Requeue* itself after it has done some work, if it wants higher priority propagators to run before it does some more work.

For unfounded set calculation, we have implemented two approaches taken from existing literature. The first one is based on the approach outlined by Gebser et al. (2007), which is the combination of smodels' source pointer technique (Simons et al. 2002) with the unfounded set computation algorithm described by Anger et al. (2006). The second approach follows Gebser et al. (2012) in combining the source pointer technique with a different unfounded set computation algorithm (described in that paper). We call our implementation of the first approach anger, and the second one gebser, after the authors of their unfounded set algorithms.

Computing unfounded sets is inherently more expensive than most other propagators. We therefore want to invoke the unfounded set propagator as rarely as possible, which requires its priority to be low. This low priority is required for another reason as well: the algorithms used by the unfounded set propagator need to run *after* unit propagation has finished, so that they have access to a consistent valuation of all the Boolean solver variables. If they do not, then the work that they do is likely to turn out to be wasted.

In the rest of this section, we will describe the first approach in detail, and then briefly outline the second approach. For all the algorithms used in this section, we assume that they access the valuation θ.

Initial calculations

When the solver is initialized, prior to any propagation and search, we calculate $Clauses(\mathcal{P})$ and record its clauses. We could also optimize $Clauses(\mathcal{P})$ by exploiting any equivalences present in it (Gebser et al. 2008), but we do not (yet) do so. For example, if one atom a is known to be exactly equivalent to a body b, because a is defined by only one clause represented by b, we can consistently use one constraint variable for both a and b. Doing so reduces both the number of variables the solver needs to manage, and eliminates the propagation steps that would otherwise be needed to keep them consistent.

We then calculate the strongly connected components of the body-head graph implicit in $\mathcal{P}$. We number each component, and assign each default atom and body the number of the component it is in. The only atoms of interest are those whose components contain more than one atom, since only they can ever participate in an unfounded set. The calculation of the SCCs allows us to distinguish between these *cyclic* atoms, and all other atoms, which are *acyclic*.

Establishing source pointers

Both our unfounded set detection algorithms are based on the idea of source pointers. Each cyclic default atom has a *source*, which is a non-false body b such that atoms in b^+ are not unfounded. As long as the source of an atom is non-false, the atom has evidence of not being unfounded. If the source of an atom becomes false, then we must look for another source for it; if we cannot find one, then the atom is part of an unfounded set.

Algorithm 1 *EstablishSourcePointers*()

```
1: for each atom a do source(a) ← ⊥
2: for each body b do
3:     if θ(b) ≠ ⊥ then ct(b) ← |b⁺| else ct(b) ← ∞
4:     if ct(b) = 0 then
5:         if θ(b) = ⊤ then
6:             MustBeQ.add(b)
7:         else
8:             MayBeQ.add(b)
9: while MustBeQ ≠ ∅ do
10:    b ← MustBeQ.pop()
11:    for a ∈ supHead(b) : source(a) = ⊥ do
12:        source(a) ← ⊤
13:        for c ∈ posInBody(a) do
14:            ct(c) ← ct(c) − 1
15:            if ct(c) = 0 then
16:                if θ(c) = ⊤ then
17:                    MustBeQ.add(c)
18:                else
19:                    MayBeQ.add(c)
20: while MayBeQ ≠ ∅ do
21:    b ← MayBeQ.pop()
22:    for a ∈ supHead(b) : source(a) = ⊥ do
23:        source(a) ← b, Subscribe(b = ⊥)
24:        for c ∈ posInBody(a) do
25:            ct(c) ← ct(c) − 1
26:            if ct(c) = 0 then MayBeQ.add(c)
27: for each atom a : source(a) = ⊥ do θ(a) = ⊥
```

We initialize the source pointers of default variables before beginning search. Our initialization, shown in Algorithm 1, partitions the set of default atoms into three disjoint sets: *MustBeTrue*, the set of atoms that are true in every constraint stable model of $\mathcal{P}$; *MayBeTrue*, the atoms that can be true in some constraint stable model; and *CantBeTrue*, atoms that cannot be true in any constraint stable model. Atoms in *MustBeTrue* cannot be part of any unfounded set, and the unfounded atoms in *CantBeTrue* can be set to false at this early stage. Only atoms in *MayBeTrue* actually require source pointers; we record the source "pointers" of atoms in *MustBeTrue* as $\top$, and the source pointers of atoms in *CantBeTrue* as $\bot$.

Algorithm 1 describes a bottom-up calculation which is similar to the Dowling-Gallier algorithm (Dowling & Gallier 1984). The algorithm keeps two queues of bodies, *MustBeQ* and *MayBeQ*, that we use to incrementally build *MustBeTrue* and *MayBeTrue* respectively. If for some $b \in bodies(\mathcal{P})$, $\theta(b) = \top$ and $b^+ \subseteq MustBeTrue$, then we add body b to *MustBeQ*. Otherwise, if $\theta(b) \neq \bot$ and $b^+ \subseteq MustBeTrue \cup MayBeTrue$, then we add b to *MayBeQ*. Since the heads of a body b in *MayBeQ* can become true due to b, we set their sources to b. Whenever we assign the id of a solver variable b to be the source of another variable a, we make the propagator subscribe to the event $b = \bot$, since if b becomes false, the propagator must determine a new source for a (or construct an unfounded set from a if one exists).

The algorithm works by keeping a count $ct(b)$ for each body b. Before the end of the first while loop, $ct(b)$ represents the number of atoms in b^+ that we need to find in *MustBeTrue* before we can put the heads supported by b into *MustBeTrue*. After the end of the first while loop, when there is no possibility left of finding any atoms that must be true, $ct(b)$ represents the number of atoms in b^+ that we need to find in *MayBeTrue* before we can put the heads supported by b into *MayBeTrue*.

At the end, we set all the atoms that do not have a source to false, since these atoms that cannot be true in any constraint stable model of the program.

Algorithm 2 *WakeUp*($b = \bot$)

```
1: if first wakeup on current search tree branch then
2:     U ← ∅, P ← ∅
3: for each atom a : source(a) = b do
4:     if θ(a) ≠ ⊥ then
5:         P.add(a)
6:         Queue()
```

Algorithm 3 *Propagate*()

```
1: U ← U \ {a ∈ D_V | θ(a) = ⊥}
2: P ← P \ {a ∈ D_V | θ(a) = ⊥}
3: while U = ∅ do
4:     if P = ∅ then return
5:     a ← P.pop()
6:     if θ(a) ≠ ⊥ then
7:         if ∃b ∈ body(a) : θ(b) ≠ ⊥ and scc(a) ≠ scc(b) then
8:             source(a) ← b, Subscribe(b = ⊥)
9:         else
10:            UnfoundedSet(a)
11: a ← U.remove()
12: if θ(a) ≠ ⊤ then Requeue()
13: θ(a) = ⊥
14: add loop_nogood(U, a)
```

The *WakeUp* and *Propagate* functions

EstablishSourcePointers() subscribes our propagator to events that record the source of a default variable becoming false. When such events happen, the subscription system will call the *WakeUp* function in Algorithm 2, which delegates most of its work to the *Propagate* function in Algorithm 3. These two functions jointly manage two global variables: U, which contains atoms that form an unfounded set, and P, which contains atoms that are pending an unfounded check. These variables are global because they must retain their values across all the propagation invocations in a propagation step between two consecutive labelling steps. We set both variables to be empty the first time we get control after a labelling step.

When an invocation of *WakeUp* tells us that a body b is false, we add the atoms supported by b to the pending queue P unless they are already known to be false. However, we do not process the pending queue immediately; we let higher priority propagators run first, to allow them to tighten the current valuation as much as possible before we process the pending queue in our own low priority *Propagate* function.

The *Propagate* function starts by removing all the atoms that have become false from both U and P. (Other propagators with higher priorities can set an atom in P to false *after* the *WakeUp* that put that atom in P.)

If U is not empty, we remove an atom a from U, set it to false, add its *loop nogood* (see below) to the set of learned constraints, and requeue the propagator to allow propagators of higher priority efficiently propagate the effects of setting a to false. This may or may not fix the values of all the atoms in the updated U. While it does not, each invocation of *Propagate* will set another unfounded atom to false.

For a given set of default atoms $U \subseteq \mathcal{D}_\mathcal{V}$, we denote the set of *external bodies* as $EB(U) = \{b \mid b \in bodies(\mathcal{P}), supHead(b) \cap U \neq \emptyset, b^+ \cap U = \emptyset\}$. The loop nogood of a set U with respect to an atom $a \in U$ is $loop\_nogood(U, a) = (\neg a \vee b_1 \vee \ldots \vee b_n)$ where $EB(U) = \{b_1, \ldots, b_n\}$ (Gebser et al. 2007). This captures the idea that an atom in a set cannot be true unless one of the external bodies of the set is true.

When there are no more known-to-be-unfounded atoms left, we look for atoms in the pending queue that can be part of an unfounded set. If the pending queue is empty, then there cannot be any more unfounded sets, and we are done. If there is an atom a in P, we test whether it is supported by an external body, a body b in a lower SCC. If it is, then a is not unfounded. If it isn't, then it is possible that a is part of an unfounded set, and we invoke *UnfoundedSet* to check if a can be extended to an unfounded set. If the call fails and leaves $U = \emptyset$, we try again with a different member of P. If it succeeds, we handle the newly-made unfounded set the same way as we handle unfounded sets that already exist when *Propagate* is invoked.

Unfounded set calculation

Algorithm 4 is based on the unfounded set algorithm by Anger et al. (2006). Its key local data structure is *Unexp*, which contains the bodies that may contain external support for some of the atoms in U. A body variable b can support an atom a only if it represents the body of one of the rules of a, it is not false, and it does not contain any atoms that are themselves unfounded. If b is in a lower SCC than a, then we take its valuation as a given; if it is not false, then it supports a and therefore a cannot be declared unfounded; if it is false, then it does not support a. If it is in the same SCC as a, then b may or may not support a; we need to find out which. That is why we put into *Unexp* the set of bodies that may support the atoms in U. To help us to do this, we define the function $maysupport(a)$ as $\{b \mid b \in body(a), \theta(b) \neq \bot, b^+ \cap U = \emptyset, scc(b) = scc(a)\}$. The algorithm uses two other data structures, *SAtoms* and *SBodies* (*S* is for *supported* in this context): *SAtoms* contains atoms that have been proven to be not unfounded, while *SBodies* contains externally supported bodies. A non-false body b is externally supported if every atom in b^+ is either in *SAtoms*, or belongs to a different component.

The algorithm processes the unexplored bodies in *Unexp* one by one. It looks at the positive default atoms in each such body. Those that are in *SAtoms* or in a lower SCC are known to support b; the others are not. We compute $nks(b, curscc, SAtoms)$ as the set of not-known-to-be-supporting atoms in b: $nks(b, curscc, SAtoms) = \{p \mid p \in b^+ : p \notin SAtoms$ and $scc(p) = curscc\}$

If this is not empty, then then we need to test the atoms in it to see whether or not they actually do support b. If the test on Line 10 succeeds for an atom p, then we have found a source for p. The algorithm records this source. It then removes p from P and adds it to the set of supported atoms. If the test on Line 10 fails, then the algorithm makes p part of the unfounded set U. The definition of *Unexp* says that bodies whose positive atoms are in U must not be in it; on line 17 we remove from it the bodies that would now violate that invariant. To allow later iterations of the outermost loop to check whether p can be supported via other bodies, we then add those possible bodies to *Unexp*. All these changes may have reduced the set of not-known-to-be-supporting atoms to the empty set, which is why we compute that set again.

If the set of not-known-to-be-supporting atoms is empty, either originally or after being recomputed, then we know b is externally supported (Line 20). This means that all atoms in U that have a rule whose body is represented by b are now supported by b. We compute R as the set of these atoms, and we record b as their source. We also remove them from U and P, and add them to *SAtoms*. Adding them to

Algorithm 4 *UnfoundedSet*(a)

```
1: curscc ← scc(a)
2: U ← {a}
3: Unexp ← maysupport(a)
4: SAtoms ← ∅, SBodies ← ∅
5: while Unexp ≠ ∅ do
6:    b ← Unexp.pop()
7:    if nks(b, curscc, SAtoms) ≠ ∅ then
8:       [b is not externally supported]
9:       for p ∈ nks(b, curscc, SAtoms) do
10:         if ∃c ∈ body(p) : θ(c) ≠ ⊥ and
            (scc(c) ≠ curscc or c ∈ SBodies) then
11:            if scc(source(p)) = curscc then
12:               source(p) ← c, Subscribe(c = ⊥)
13:            if P.contains(p) then P.remove(p)
14:            SAtoms.add(p)
15:         else
16:            U.add(p)
17:            Unexp ← Unexp \ {d | d ∈ Unexp, p ∈ d+}
18:            Unexp ← Unexp ∪ maysupport(p)
19:   if nks(b, curscc, SAtoms) = ∅ then
20:      [b is externally supported]
21:      SBodies.add(b)
22:      R ← {r | r ∈ U, b ∈ body(r)}
23:      for r ∈ R do
24:         source(r) = b
25:      Subscribe(b = ⊥)
26:      while R ≠ ∅ do
27:         r ← R.pop()
28:         U.remove(r)
29:         P.remove(r)
30:         SAtoms.add(r)
31:         for j ∈ posInBody(r) : θ(j) ≠ ⊥ and
            ∀t ∈ j+, (t ∈ SAtoms or scc(t) ≠ curscc) do
32:            SBodies.add(j)
33:            for a ∈ supHead(j) ∩ U do
34:               source(a) ← j, Subscribe(j = ⊥)
35:               R.add(a)
36:      Unexp ← ⋃p∈U maysupport(p)
```

SAtoms may make more bodies qualify for membership of *SBodies*, which in turn may provide external support for more atoms. We put any such atoms into R as well, and we keep going until everything in R has been processed. Once we have removed as many atoms as possible from U and have reached a fixpoint, we reinitialize *Unexp* based on the final value of U.

Second approach

Our second implementation, gebser, differs from our first, anger, only in its use of a different unfounded set algorithm. We have taken that algorithm directly from (Gebser et al. 2012), so here we just give its outline. The algorithm uses the concept of a *scope*, which is an upper bound on U. The algorithm computes the scope by starting with P, and extending it through a fixpoint algorithm. It then computes U by restricting the scope to a single SCC.

6 Experiments

We benchmarked our implementations anger and gebser against two competing systems. The first is a combination of clasp (version `2.0.6`) and gringo (version `3.0.4`), which we call cl+gr in our tables for brevity. The second is clingcon (Gebser et al. 2009) (version `2.0.0-beta`), which is an extension of clasp with CP capabilities. We ran all the benchmarks on a Lenovo model 3000 G530 notebook with a 2.1 GHz Core 2 Duo T6500 CPU and 3 GB of memory running Ubuntu 12.04. We repeated each experiment with a timeout five times, and each experiment without a timeout twice; the results we present are their averages.

		Solved	Opt	AvgPct
RoutingMin	cl+gr	N/A	N/A	N/A
	clingcon	19	3	69.1
	gebser	19	8	33.5
	anger	18	9	33.9
RoutingMax	cl+gr	N/A	N/A	N/A
	clingcon	22	0	100.0
	gebser	27	0	42.2
	anger	27	0	37.4

Table 1: Results for routing on 34 instances

We ran two sets of benchmarks. The first set consists of different instances of two routing problems, which are slightly modified versions of the models used in the experiments by Liu et al. (2012). Our reason for selecting these two problems is that they involve not just reachability, but also variables with large finite domains. The two problems differ only in their objective; they use the same data representation and impose the same set of constraints. Each instance of these problems is specified by

- a weighted directed graph (V, E, w) where $w : E \mapsto \mathbb{N}$,
- a source node $s \in V$,
- a set of destination nodes $D \subseteq V \setminus \{s\}$, and
- a deadline for each destination $f : D \mapsto \mathbb{N}$.

Their solutions consist of two parts:

- a cycle-free route $(r_0, r_1, \ldots, r_k)$ where $r_0 = s$, $(r_i, r_{i+1}) \in E$ for all $i \in \{1, ..., k-1\}$, and for each $d \in D$, $d = r_i$ for some $i \in \{1, ..., k\}$, and
- a time assignment $t : V \mapsto \mathbb{N}$ such that $t(r_0) = 0$, $t(r_{i+1}) \geq t(r_i) + w(r_i, r_{i+1})$ for all $i \in \{1, ..., k-1\}$, and for each $d \in D$, $t(d) \leq f(d)$.

For the *RoutingMin* problem, the objective is minimizing the total delay $\sum_{d \in D}(f(d) - t(d))$. For the *RoutingMax* problem, the objective is maximizing the total delay.

Table 1 presents our results on 34 instances each of RoutingMin and RoutingMax. The sizes of the graphs in those instances range from 21 to 87 nodes. The Solved column gives the number of instances for which the named solver computed a result (which may or may not be optimal) within the timeout period, which was one minute. The Opt column gives the number of these instances for which the solver not only computed the optimal result, but also proved it to be optimal.

Since the sizes of the instances vary significantly, the minimum and maximum values of the total delay differ greatly as well. Averages of the delays are therefore not an appropriate representation of the overall quality of the solutions from a solver. Therefore we express the quality of each solution as a percentage of the maximum delay computed by any solver on the relevant problem instance. If all the solvers compute a delay of 0, we score all solvers as 0% for RoutingMin and as 100% for RoutingMax. The AvgPct column shows the average of these percentages for the

	cl+gr	flat	anger		gebser	
			o/a	prop	o/a	prop
WR (S/8)	894.49	10.22	166.74	95.39	456.71	7.58
WR (U/7)	24.14	8.89	41.20	28.44	53.06	1.29
GP (S/7)	9.69	2.88	656.69	4.13	659.27	2.86
GP (U/6)	43.02	0.95	221.38	9.76	243.08	3.54
CDS (S/7)	26.80	0.39	74.98	44.53	27.24	0.50
CDS (U/8)	1618.22	0.64	566.47	318.65	395.84	4.86
MG (S/15)	2.56	32.95	43.04	42.14	0.86	0.03

Table 2: ASP problems, geometric restart, no timeout

	cl+gr		flat	anger		gebser	
WR (S/8)	148.21	(5/1)	9.86	149.12	(4/1)	96.74	(5/1)
WR (U/7)	101.68	(5/1)	11.68	36.68	(0/0)	49.10	(0/0)
GP (S/7)	39.24	(0/0)	2.58	21.36	(0/0)	46.52	(0/0)
GP (U/6)	169.22	(5/1)	0.91	123.96	(3/1)	121.72	(0/0)
CDS (S/7)	185.09	(10/2)	0.41	102.79	(5/1)	127.88	(5/1)
CDS (U/8)	274.77	(10/2)	0.69	321.20	(20/4)	314.56	(20/4)
MG (S/15)	5.07	(0/0)	52.86	49.95	(5/1)	1.37	(0/0)

Table 3: ASP problems, Luby restart, with timeout

instances for which *all* the solvers get a solution. All the solvers were run with a slow restart strategy that used a Luby sequence (Luby et al. 1993) with a restart base of 400.

Due to the large domains involved, grounding is very inefficient, which causes cl+gr to run out of memory on all our test instances, even the smallest. For RoutingMin, all the solvers solve roughly the same number of instances, but anger and gebser get optimal solutions on almost three times as many instances, and the average quality of their solutions is also better by about a factor of 2. (For minimization, better performance is represented by smaller percentages, while for maximization, it is represented by larger ones.) For the instances of RoutingMax that all the solvers can solve, clingcon invariably generates the best solutions. However, clingcon generates solutions for substantially fewer instances than anger and gebser, showing that it is not as robust.

Our second set of benchmarks is a selection of problems taken from the second ASP competition [2]: Wire Routing (WR), Graph Partitioning (GP), Connected Dominating Set (CDS), and Maze Generation (MG). For each of these problems, we report on their satisfiable (S) and unsatisfiable (U) instances in separate rows of both Tables 2 and 3; the numbers after the forward slashes give the number of instances in each category. Table 2 shows results for a setup in which all the solvers were run using the default restart strategy of clasp (geometric restart, with the restart threshold starting at 100 conflicts, multiplied by 1.5 after each restart) and without time limits, while Table 3 shows the results when all the solvers were run with a Luby sequence restart strategy with restart base 10, and with a timeout of 10 minutes.

The numbers in the slots of Table 2 all represent an average runtime, in seconds, over all the problem instances represented by the row. The cl+gr column gives the average time taken by cl+gr to solve those instances. The flat column gives the average time taken to flatten the MiniZinc model to FlatZinc. The anger and gebser overall (o/a) columns give the average time taken to solve the resulting FlatZinc models using the anger and gebser variants of our implementation. The anger and gebser prop columns give the average times taken by our propagator within those overall solution times.

The numbers in time slots of Table 3 also represent average execution times; the execution time will be the timeout time (600 seconds) if the solver does not complete before then. Table 3 omits propagation times to make room for the numbers in parentheses after each average execution time. These represent respectively the number of runs on which the given solver failed to produce a solution before the timeout, and the number of instances to which those runs belong. (For example, 4/1 means that of the five runs on a problem instance, one produced a solution, but four did not.) Neither table has a column for clingcon, since the purpose of Tables 2 and 3 is to compare gebser and anger with a native ASP solver on pure Boolean problems. The results of running these problems on clingcon would be the same as the results of cl+gr, since these problems have nothing to do with the difference between cl+gr and clingcon, namely the finite domain extension present in clingcon.

The overall results in Table 2 are mixed. On WR/U, GP/S, GP/U and (due to flattening) MG/S, cl+gr clearly beats both anger and gebser. On WR/S and CDS/U, both anger and gebser clearly beat cl+gr. On CDS/S, cl+gr clearly beats anger, but edges out gebser by just a whisker. The overall winner on these tests is clearly cl+gr.

However, this picture changes when we switch our attention to Table 3. Even after including flattening time, gebser is faster than cl+gr on four of the seven problem sets (WR/S, WR/U, GP/U, CDS/S), and it is slower on only three (GP/S, CDS/U and MG/S). It is also more robust, failing to find a solution on only six problem instances, compared to seven for cl+gr. Our other system anger is less robust, failing to find solutions on eight problem instances, though for two of these, it did solve them on *some* runs. However, to compensate for this, anger is the fastest system on three problem sets (WR/U, GP/S, CDS/S).

The propagation time columns in Table 2 show that anger spends a lot more time on unfounded set propagation than gebser. In some cases, such as WR/S, this pays off handsomely, in the form of more effective pruning of the search space. In some other cases, such as CDS/U, anger spends less time outside the propagator than gebser, so anger seems to get better pruning, but not enough to pay back the extra cost of the propagator itself. And on most problems in Table 2, the extra cost of its propagator does not even help anger get better pruning. This suggests that we should investigate whether one can blend the two unfounded set calculation algorithms in order to achieve the pruning power of anger (Anger et al. 2006), or something close to it, at an efficiency closer to that of gebser (Gebser et al. 2012).

7 Related work

The closest modelling system to our approach is the IDP system (Wittocx et al. 2008*b*) which extends classical logic with the use of inductive definitions. Like our proposed extension of MiniZinc, IDP allows arbitrary first order formulae in the rule bodies of its inductive definitions while most ASP systems allow only normal rules. Unlike ASP solvers which apply the closed world assumption (an atom that cannot be derived is assumed to be false) to entire programs, our system and IDP can localize it to only a certain part of the program: default atoms for us, and *definitional atoms* for IDP. IDP handles constraints by

[2] See http://dtai.cs.kuleuven.be/events/ASP-competition/SubmittedBenchmarks.shtml

grounding, just as a traditional ASP system.

There has been significantly more effort in recent years to integrate CP into ASP systems than to integrate ASP into CP systems. The principal advantage of ASP over CP systems is the ability to use recursive definitions, particularly to model transitive closure. On the other hand, pure answer set solvers have a serious efficiency problem when dealing with problems that involve finite domain variables. This is why most research in this area has focused on integrating efficient finite domain handling in ASP systems, resulting in a new domain of research called *constraint answer set solving* (Drescher 2010).

The systems described by Baselice et al. (2005), Mellarkod & Gelfond (2008), Mellarkod et al. (2008) view both answer set and constraint solvers as black boxes, and their frameworks do not allow the incorporation of modern engineering techniques such as nogood learning and advanced backjumping. The clingcon system (Gebser et al. 2009), while it implements nogood learning and backjumping, still treats the CP solver as an *oracle* that does not explain its propagation to the ASP solver, and works around this shortcoming by using an indirect method to record nogoods generated by propagation done by the CP solver. All these systems have to incur some overheads for communication between the ASP and CP solvers.

The approach described by Drescher & Walsh (2010) avoids this overhead by translating the CP part of the problem into ASP rules, and achieves its efficiency through unit propagation on these rules; that paper also gives their translation of the *alldifferent* global constraint. One shortcoming of this approach is its reliance on an *a priori* ASP decomposition of global constraints; the example of the performance gains achieved by techniques such as lazy clause generation strongly suggests that such decompositions should be done lazily. Their more recent paper (Drescher & Walsh 2012) overcomes this shortcoming by allowing lazy nogood learning from external propagators. The resulting system is close to what we have implemented, and shows promising performance in comparison with clingcon. The mingo system described by Liu et al. (2012) does translation in the other direction: it translates an ASP program (extended with integer and real variables) to a mixed integer program.

The unique feature of constraint languages and solvers that distinguishes them from other declarative systems like ASP, SMT, and SAT is the use of global constraints, and the existence of extremely efficient propagators to solve these constraints. Other solvers usually rely on a single propagation method such as unit propagation. Specialized propagation techniques for global constraints, such as the one given by Schutt et al. (2011) for the *cumulative* constraint, allow much stronger and more efficient propagation than approaches using decomposition and unit propagation.

8 Conclusion

We have shown a method for adding answer set programming capabilities to the general purpose constraint programming language MiniZinc. The resulting system is much better at solving combined ASP/CP problems than existing systems, and we hope that our examples have convinced readers that such problems can be expressed more *naturally* in the syntax of MiniZinc than in the syntax of ASP languages. MiniZinc is also more flexible: it can express constraints on non-Boolean variables (such as integers, floats and sets); it can express complex Boolean expressions more naturally, *and* (with the exception of disjunctions in heads) it can express all ASP extensions, including weight constraints, choice rules, cardinality constraints, integrity constraints, and aggregates such as sum, count, min and max.

We have shown two implementations of our extensions to MiniZinc. Our benchmark results show that both these systems can solve combined ASP/CP problems that cannot be solved by cl+gr, even though its ASP component, clasp, won the last two competitions for pure ASP solvers. We have also shown that our system is competitive with clingcon, an extension of clasp, on such problems, being better on some tasks, worse on others.

We have several directions for future work. We will start by implementing our proposal for the inductive predicate syntax in MiniZinc, which should allow programmers to model problems more naturally, without manually grounding normal rules. We intend to investigate different ways to map these predicates to FlatZinc. We will look at the approaches used by the grounder of the IDP system (Wittocx et al. 2008*a*) and the transformation to ASP rules described by Mariën et al. (2004). Adopting some of these approaches may require moving from the stable model semantics to the well-founded semantics; the jury is still out on which users find more natural. We plan to investigate moving the grounding phase entirely to runtime, as proposed by the lazy grounding scheme of Palù et al. (2009) and the lazy model expansion scheme of De Cat et al. (2012). We also intend to look into incorporating other efficient features of ASP, such as preprocessing (Gebser et al. 2008). Finally, we intend to see whether we can construct an unfounded set algorithm that combines the efficiency of Gebser et al. (2012) with the more effective pruning of Anger et al. (2006).

Acknowledgments: NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy, and by the Australian Research Council. We are grateful to Geoffrey Chu for helping us to understand chuffed, to Guohua Liu for providing us with clingcon encodings of routing problems, and to the Potassco mailing list users for their timely replies to our queries about clingcon.

References

Anger, C., Gebser, M. & Schaub, T. (2006), Approaching the core of unfounded sets, *in* 'Proceedings of the International Workshop on Nonmonotonic Reasoning', pp. 58–66.

Baral, C. (2003), *Knowledge representation, reasoning and declarative problem solving*, Cambridge University Press.

Baselice, S., Bonatti, P. A. & Gelfond, M. (2005), Towards an integration of answer set and constraint solving, *in* 'Proceedings of the 21st International Conference on Logic Programming', Sitges, Spain, pp. 52–66.

Clark, K. L. (1978), Negation as failure, *in* H. Gallaire & J. Minker, eds, 'Logic and Data Bases', Plenum Press, NY, pp. 127–138.

De Cat, B., Denecker, M. & Stuckey, P. (2012), Lazy model expansion by incremental grounding,

in 'Technical Communications of the 28th International Conference on Logic Programming', Budapest, Hungary.

Dooms, G., Deville, Y. & Dupont, P. (2005), CP(graph): introducing a graph computation domain in constraint programming, *in* 'In proceedings of the 11th International Conference on Principles and Practice of Constraint Programming', Sitges, Spain, pp. 211–225.

Dowling, W. & Gallier, J. (1984), 'Linear time algorithms for testing the satisfiability of propositional Horn formulae', *Journal of Logic Programming* **1**, 267–284.

Drescher, C. (2010), Constraint answer set programming systems, *in* 'Technical Communications of the 26th International Conference on Logic Programming', Dagstuhl, Germany, pp. 255–264.

Drescher, C. & Walsh, T. (2010), 'A translational approach to constraint answer set solving', *CoRR* **abs/1007.4114**.

Drescher, C. & Walsh, T. (2012), Answer set solving with lazy nogood neneration, *in* 'Technical Communications of the 28th International Conference on Logic Programming', Budapest, Hungary.

Feydy, T. & Stuckey, P. J. (2009), Lazy clause generation reengineered, *in* 'Proceedings of the 15th International Conference on the Principles and Practice of Constraint Programming', Lisbon, Portugal, pp. 352–366.

Gebser, M., Kaufmann, B., Neumann, A. & Schaub, T. (2007), Conflict-driven answer set solving, *in* 'Proceedings of the 20th International Joint Conference on Artificial Intelligence', Hyderabad, India, p. 386.

Gebser, M., Kaufmann, B., Neumann, A. & Schaub, T. (2008), Advanced preprocessing for answer set solving, *in* 'Proceedings of the 18th European Conference on Artificial Intelligence', Patras, Greece, pp. 15–19.

Gebser, M., Kaufmann, B. & Schaub, T. (2012), 'Conflict-driven answer set solving: From theory to practice', *Artif. Intell* **187**, 52–89.

Gebser, M., Ostrowski, M. & Schaub, T. (2009), Constraint answer set solving, *in* 'Proceedings of the 25th 25th International Conference on Logic Programming', Pasadena, CA, pp. 235–249.

Gebser, M. & Schaub, T. (2005), Loops: Relevant or redundant?, *in* 'LPNMR', pp. 53–65.

Gelfond, M. & Lifschitz, V. (1988), The stable model semantics for logic programming, *in* 'ICLP/SLP', pp. 1070–1080.

Lifschitz & Razborov (2006), 'Why are there so many loop formulas?', *ACM Transactions on Computational Logic* **7**(2), 261–268.

Lin, F. & Zhao, Y. (2004), 'ASSAT: computing answer sets of a logic program by SAT solvers', *Artificial Intelligence* **157**(1-2), 115–137.

Liu, G., Janhunen, T. & Niemela, I. (2012), Answer set programming via mixed integer programming, *in* 'Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning', pp. 32–42.

Lloyd, J. W. & Topor, R. W. (1984), 'Making Prolog more expressive', *Journal of Logic Programming* **1**(3), 225–240.

Luby, M., Sinclair, A. & Zuckerman, D. (1993), 'Optimal speedup of Las Vegas algorithms', *Information Processing Letters* **47**, 173–180.

Mariën, M., Gilis, D. & Denecker, M. (2004), On the relation between ID-logic and answer set programming, *in* 'JELIA', pp. 108–120.

Mellarkod, V. S. & Gelfond, M. (2008), Integrating answer set reasoning with constraint solving techniques, *in* 'Proceedings of the 9th International Symposium on Functional and Logic Programming', Ise, Japan, pp. 15–31.

Mellarkod, V. S., Gelfond, M. & Zhang, Y. (2008), 'Integrating answer set programming and constraint logic programming', *Ann. Math. Artif. Intell* **53**(1-4), 251–287.

Mitchell, D. G. (2005), 'A SAT solver primer', *Bulletin of the EATCS* **85**, 112–132.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J. & Tack, G. (2007), MiniZinc: Towards a standard CP modelling language, *in* 'Proceedings of the 13th International Conference on the Principles and Practice of Constraint Programming', Providence, RI, pp. 529–543.

Ohrimenko, O., Stuckey, P. J. & Codish, M. (2009), 'Propagation via lazy clause generation', *Constraints* **14**(3), 357–391.

Palù, A. D., Dovier, A., Pontelli, E. & Rossi, G. (2009), Answer set programming with constraints using lazy grounding, *in* 'ICLP', pp. 115–129.

Rossi, F., Beek, P. v. & Walsh, T. (2006), *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, Elsevier Science, New York, NY.

Schulte, C. & Stuckey, P. (2008), 'Efficient constraint propagation engines', *ACM Transactions on Programming Languages and Systems* **31**(1), Article No. 2.

Schutt, A., Feydy, T., Stuckey, P. J. & Wallace, M. G. (2011), 'Explaining the cumulative propagator', *Constraints* **16**(3), 250–282.

Simons, P., Niemelä, I. & Soininen, T. (2002), 'Extending and implementing the stable model semantics', *Artificial Intelligence* **138**(1–2), 181–234.

Van Gelder, A., Ross, K. A. & Schlipf, J. S. (1988), Unfounded sets and well-founded semantics for general logic programs, *in* 'Proceedings of the ACM Symposium on Principles of Database Systems', pp. 221–230.

Viegas, R. D. & Azevedo, F. (2007), GRASPER, *in* 'Proceedings of the 13th Portuguese Conference on Artificial Intelligence', Guimarães, Portugal, pp. 633–644.

Wittocx, J., Mariën, M. & Denecker, M. (2008*a*), GIDL: A Grounder for FO, *in* 'Proceedings of the Twelfth International Workshop on Non-Monotonic Reasoning', pp. 189–198.

Wittocx, J., Mariën, M. & Denecker, M. (2008*b*), The IDP system: a model expansion system for an extension of classical logic, *in* 'LaSh', pp. 153–165.