Stable Model Semantics for Founded Bounds

Rehan Abdul Aziz, Geoffrey Chu and Peter J. Stuckey

National ICT Australia, Victoria Laboratory,∗
Department of Computing and Information Systems,
University of Melbourne, Australia
Email: raziz@student.unimelb.edu.au, gchu@csse.unimelb.edu.au, pjs@csse.unimelb.edu.au

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Answer Set Programming (ASP) is a powerful form of declarative programming used in areas such as planning or reasoning. ASP solvers enforce *stable model semantics*, which rule out solutions representing certain kinds of circular reasoning. Unfortunately, current ASP solvers are incapable of solving problems involving cyclic dependencies between multiple integer or continuous quantities effectively. In this paper, we generalize the notion of stable models to *bound founded variables* with arbitrary domains, where *bounds* on such variables need to be justified by some rule in the program in order for the model to be stable. We show how to handle significantly more general rule forms where bound founded variables can act as head or body variables, and where head and body variables can be related via complex constraints subject to certain monotonicity requirements. We describe a new unfounded set detection algorithm which allows us to enforce this generalization of the stable model semantics. We also show how these unfounded sets can be explained in order to allow effective conflict-directed clause learning. The new solver merges the best features of CP, SAT and ASP solvers and allows new types of problems to be solved very efficiently.

KEYWORDS: Answer Set Programming, Stable Model Semantics, Finite Domain Solving

1 Introduction

Many problems in the areas of planning or reasoning can be efficiently expressed using Answer Set Programming (ASP) (Gebser et al. 2012; Baral 2003). ASP enforces stable model semantics (Gelfond and Lifschitz 1988) on logic programs, which disallows solutions representing certain kinds of circular reasoning. Current state-of-the-art ASP solvers convert an ASP program into a Boolean representation by *grounding* the variables in the rules to all their allowed values, and using a Boolean Satisfiability (SAT) solver (Marques-Silva and Sakallah 1999; Moskewicz et al. 2001) to solve the resulting Boolean problem (Gebser et al. 2012). Naively, this gives the wrong semantics, as SAT solvers do not enforce the stable model semantics. This is overcome by augmenting the SAT solver with a component that detects *unfounded sets* (Van Gelder et al. 1988) (sets of variables that can only be set true using circular reasoning), and fixes such sets of variables to false (Anger et al. 2006; Gebser et al. 2007). Unfortunately, this approach suffers from the *grounding bottleneck* problem. In ASP programs with finite domain (FD) constraints, grounding the FD constraints will often generate a very large or exponential number of Boolean clauses, causing the SAT-based ASP solver to run out of memory.

Several methods have been proposed to overcome the grounding bottleneck in ASP solvers.

∗ NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

These include translational approaches (Drescher and Walsh 2010; Liu et al. 2012) and approaches in which ASP solvers are combined with Constraint Programming (CP) (Rossi et al. 2006) such that the FD constraints are handled natively by the CP solver without grounding (Gebser et al. 2009; Mellarkod et al. 2008; Baselice et al. 2005). In the latter approaches, each occurrence of an FD constraint c in the logic program is replaced with a Boolean variable b_c. Reified constraints of form: $b_c \leftrightarrow c$ are posted in the CP solver and the domains of the b_c's are channeled between the ASP and CP solvers. A recent approach implements unfounded set detection as a propagator inside a CP solver that prunes unfounded sets from partial models and explains them during search (Aziz et al. 2013). Another solution extends an ASP solver with external propagators that can generate nogoods during search to explain their propagation (Drescher and Walsh 2012). While the methods mentioned above avoid the grounding bottleneck, all of them have two major shortcomings. First, they cannot correctly handle any ASP program where ASP variables appear as arguments in FD constraints in the rules. This is because the unfounded set detection algorithm does not detect any inference loop that goes through external constraints in rule bodies. Secondly, none of these systems can efficiently model problems involving cyclic dependencies between integer or continuous quantities. Due to the possibility of positive feedback loops, a generalization of stable model semantics to integer or continuous variables is required to get sensible (i.e., stable) solutions of such problems.

Consider the simple example of calculating shortest paths over an undirected graph (V,E), where the distance from x to y is $e_{x,y} = e_{y,x}$. Let $sp_{x,y}$ be integer variables representing the length of the shortest path from x to y. A simple declarative model is as follows:

$$\begin{array}{ll} \forall x, & sp_{x,x} \leq 0 \\ \forall x,y, & sp_{x,y} \leq sp_{y,x} \\ \forall x,y,z, & sp_{x,y} \leq e_{x,z} + sp_{z,y} \end{array}$$

Unfortunately, if we give such a model to a CP solver (which uses logical semantics), we can get completely nonsensical solutions such as $sp_{x,y} = 0$ for all x,y. What we need in order to solve this problem properly is some sort of stable model semantics over the *bounds* of the integer variables $sp_{x,y}$. In particular, we want a semantics where the upper bounds of the variables $sp_{x,y}$ need to be justified, i.e., for each value k, unless there is some rule that is forcing $sp_{x,y}$ to be smaller than k, then $sp_{x,y}$ should not be smaller than k. It is possible to do this using the existing ASP language as follows. Let $spub(x,y,d)$ represent whether we can go from x to y in less than or equal to d cost. Then a model such as:

$$\begin{array}{ll} \forall x,y,k, & spub(x,y,k) \leftarrow spub(x,y,k-1) \\ \forall x, & spub(x,x,0). \\ \forall x,y,k, & spub(x,y,k) \leftarrow spub(y,x,k) \\ \forall x,y,r,s, & spub(x,y,r) \leftarrow spub(z,y,s) \wedge r = e_{x,z} + s \end{array}$$

will generate stable models where $spub(x,y,d)$ is true iff d is $\geq$ the correct shortest path value, allowing us to calculate $sp_{x,y}$. However, this is extremely inefficient and suffers from a significant grounding bottleneck if distances are large.

As another example consider the following version of the Company Controls problem. Given a set of companies C, and two companies $s,t \in C$, we want company s to gain control of company t at minimum cost. Each company i has m_i shares in total. A company i can gain control of company j if it controls more than 50% of company j's shares either directly (owns shares in j itself) or indirectly (controls a company that owns shares in j). Initially each company $i \in C$ owns $a_{i,j}$ of company j's shares. Company s is going to try to gain control of company t by either buying shares in t directly, or by buying shares of other companies to gain control over

them and then using them to buy more shares to gain control over additional companies, etc., until it controls t. Each company i has a limited amount of cash $budget_i$ to buy shares with. Let c_i represent whether company s controls company i. Note that we assume company s controls itself, so $c_s = true$ by default. Let $b_{i,j}$ be the amount of stocks of company j that company s forces company i to buy. Note that company s can only force company i to buy if it controls company i, so $\neg c_i \rightarrow b_{i,j} = 0$. Suppose the stock of company j has a fixed market price of p_j. Then we can model it as follows:

minimize	$\sum_{j \in C} p_j b_{s,j}$	expenditure of company s
subject to:		
	c_s	company s controls itself by default
	$\leftarrow \neg c_t$	s must control t
$\forall j \in C,$	$c_j \leftarrow \sum_{i \in C} c_i * (a_{i,j} + b_{i,j}) > 0.5 * m_j$	s control j if it control >50% shares
$\forall i, j \in C,$	$\leftarrow b_{i,j} > 0 \wedge \neg c_i$	only controlled companies can buy
$\forall i \in C,$	$\leftarrow \sum_{j \in C} a_{j,i} + b_{j,i} > m_j$	cannot buy more than available
$\forall i \in C,$	$\leftarrow \sum_{j \in C} b_{i,j} > budget_i$	budget constraint

This version of the problem is different from the one used in ASP competitions[1] where buying options are limited to a relatively small number of preset packages. Our version allows companies to buy shares in any increment. If the m_i's are large, grounding our version to an ASP program will quickly cause the solver to run out of memory.

In this paper, we propose a new approach to ASP that allows a wider range of problems to be modeled and solved efficiently. We generalize the notion of stable models to *bound founded variables*. These variables cannot take an arbitrary value from their domain. Instead, their value defaults to the lowest or highest value allowed by the rules in the program depending on whether they are *lb-founded* or *ub-founded* (lower/upper bound founded) variables. Thus the *bounds* on such variables need to be justified by some rule in order for a model to be stable. We show how to handle significantly more general rule forms where bound founded variables can act as head or body variables, and where these variables can be related via complex constraints subject to certain monotonicity requirements. We describe a new unfounded set detection algorithm that allows us to enforce this generalization of stable model semantics. We also show how these unfounded sets can be explained in order to allow effective conflict-directed clause learning.

2 Definitions and Background

Let $\Rightarrow$ denote logical implication and $\Leftrightarrow$ denote logical equivalence. A *constraint satisfaction problem* (CSP) is a tuple (V, C, D), where V is a set of variables, C is a set of constraints, and D is a set of unary domain constraints of the form $\{x \in D(x) \mid x \in V\}$. Each constraint c restricts the values that a set of variables $vars(c) \subseteq V$ can be simultaneously assigned. A *valuation* θ is a function that maps each $x \in V$ to a value in $D(x)$.

A *propagator* p_c for constraint c is a contracting function from domains to domains such that for any D, $c \wedge D \Rightarrow p_c(D)$, i.e., it prunes infeasible variable/value pairs to return a smaller domain. A constraint programming solver branches on the value of variables and performs a fixpoint calculation using the propagators to reduce the size of the current domain. If any variable's domain becomes empty, then that subtree is failed and has no solutions. If all variables are fixed

[1] `https://www.mat.unical.it/aspcomp2011/FinalProblemDescriptions/CompanyControlsOptimize`

and no failure occurs, then we have a solution. An event-based propagation engine (Schulte and Stuckey 2008) uses domain change events such as lower/upper bound change events or value fixing events (*lb_event*(x), *ub_event*(x), *fix_event*(x) resp.) to lazily wake up propagators. Each propagator subscribes to the events that may allow it to prune additional values.

Given a constraint c and a variable argument x, c is *monotonically increasing (resp. decreasing)* w.r.t. x if increasing (resp. decreasing) x's value can never cause c to go from satisfied to unsatisfied. A *HORN-CP* is a CSP where every constraint c is monotonically increasing in at most one of its arguments and is monotonically decreasing in all other arguments. The *minimal solution* to a HORN-CP is a solution θ such that there does not exist another solution θ' with $\forall v \in V, \theta'(v) \leq \theta(v)$. Note that for Boolean variables, $false < true$.

Similarly to how the minimal model for a HORN-SAT instance can be found by running unit propagation on the constraints to fixed point and then setting all unfixed variables to false, the minimal solution for a HORN-CP instance can be found by running bounds consistent propagators on the constraints to fixed point and then setting all variables to their lowest allowed values. If the propagation leads to an empty domain, there are no solutions. Otherwise, we will always get a minimal solution.

Example 1
Consider the constraints $x \geq 0$, $y \geq 4 + x$, $2x \geq y$. This is a HORN-CP program, each constraint monotonically increasing in the first argument. A fixpoint calculation calculates $x \geq 0$, $y \geq 4$, $x \geq 2$, $y \geq 6$, $x \geq 3$, $y \geq 7$, $x \geq 4$, $y \geq 8$. The minimal solution is $x = 4$, $y = 8$. □

3 Bound Founded Answer Set Programming

We now define *bound founded answer set programs* (BFASPs). A BFASP P is a tuple $P = (\mathcal{N} \cup \mathcal{F}, R)$, where $\mathcal{N}$ are standard (CP) variables, $\mathcal{F}$ are founded variables, and R is a set of *rule constraints*. Variables can either be Boolean, integer, or continuous. Founded variables can be divided into two categories: lb-founded and ub-founded. Therefore, the complete type of a variable is determined by two factors: whether it is Boolean, integer, or continuous and whether it is lb-founded, ub-founded or standard. The stable model semantics requires that the lower bounds (resp. upper bounds) of lb-founded (resp. ub-founded) variables are justified by some rule constraint in order to be valid in a stable model. This means that an lb-founded Boolean variable is the same as a variable in ASP, i.e., in the absence of any rule that forces it to be true, the variable becomes false. It is intuitive to think of an lb-founded integer x with domain $[L \dots U]$ as a set of ASP variables $[x \geq L], [x \geq L+1], \dots, [x \geq U], [x \geq U+1]$ where $[x \geq v]$ represents the truth value of $x \geq v$, linked by the following ASP rules:

$$\begin{array}{lll} & [x \geq L]. & \\ \forall v \in [L..U-1], & [x \geq v] & \leftarrow [x \geq v+1] \\ & & \leftarrow [x \geq U+1] \end{array}$$

The value of x is then the largest value v such that $[x \geq v]$ is true but $[x \geq v+1]$ is false. Similarly, an lb-founded continuous variable can be thought of as an infinite set of ASP variables $[x \geq v]$ where $\forall v < v', [x \geq v] \leftarrow [x \geq v']$, and ub-founded variables are analogous, but with ASP variables representing $x \leq v$ instead. Of course, this grounding is precisely what we want to avoid in this paper.

Each rule constraint r is a pair $(cons(r), head(r))$ where $cons(r)$ is a constraint defined by an *arbitrary logical expression* over any standard or founded variable in the problem, and $head(r) \in \mathcal{F}$ is called the *head variable* of r. We now define the stable model semantics for a BFASP P that is valid (defined below). We first consider the case where we only have standard or lb-founded

variables. Given a full assignment θ to the variables in $\mathscr{N} \cup \mathscr{F}$, the *CP reduct* of P w.r.t. θ, written P^θ, is a HORN-CP constructed as follows. We start with P^θ empty. We add all variables in $\mathscr{F}$ to P^θ. For each rule constraint $r = (c(y, x_1, \ldots, x_n), y)$, for each x_i where x_i is not in $\mathscr{F}$, or for which c is not decreasing w.r.t. x_i, we substitute the value of $\theta(x_i)$ for x_i in c. If the modified constraint c is not a tautology, we add it to P^θ. A solution θ to P is a stable solution iff its restriction to the variables $\mathscr{F}$ is the minimal solution of P^θ.

Theorem 1
The CP reduct defined above generalizes the well-known Gelfond-Lifschitz (GL) reduct (Gelfond and Lifschitz 1988) used to transform normal logic programs to positive ASP programs, and is exactly equivalent to the GL reduct in the special case where P is a standard ASP program with normal rules over Boolean variables.

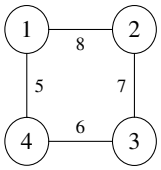
Proof
Consider a normal rule $a \leftarrow p_1 \wedge \ldots \wedge p_n, \sim q_1 \wedge \ldots \wedge \sim q_m$. In BFASP format, this would be a rule constraint $(a \leftarrow p_1 \wedge \ldots \wedge p_n \wedge \neg q_1 \wedge \ldots \wedge \neg q_m, a)$. The constraint is increasing in a and the q_i's, and is decreasing in the p_i's. Given any solution θ, the GL reduct is constructed by removing all rules where one of the q_i is true in θ, and then removing all negative literals from all rules. Consider what happens when constructing the CP reduct. The constraint is not decreasing w.r.t. the q_i's, so we substitute the value of $\theta(q_i)$ for q_i for each i into the constraint. If any of the q_i is true in θ, then after substituting the values in, c becomes a tautology so we do not add it to P^θ, which is the same as removing the rule in the GL reduct. If all the q_i's are false, then after substituting the values in, the constraint becomes $a \leftarrow p_1 \wedge \ldots \wedge p_n$, which we add to the CP-reduct. This is again the same as in the GL reduct where we remove all negative literals. □

We extend the definition of CP reduct and stable model semantics to ub-founded variables as follows: for each ub-founded variable x in the program, create an lb-founded variable x' and replace every occurrence of x in P with $-x'$, and then map the stable models of the modified program to the original program via $\theta(x) = -\theta(x')$.

Example 2
Consider the program P with three lb-founded integers variables a, b, c with domains $[-10 \ldots 10]$ and two rule constraints: $r1 = (a \geq b - c + 7, a)$ and $r2 = (c \geq 5)$. For the assignment $\theta = \{a = -8, b = -10, c = 5\}$, P^θ has two rules: $(a \geq b - 5 + 7, a)$ and $(c \geq 5)$. The minimal model of P^θ is θ, and hence θ is a stable model of P. □

The *dependency graph* of a BFASP has nodes $\mathscr{N} \cup \mathscr{F}$ and edges $\{(head(r), x) \mid r \in R, x \in var(cons(r)), cons(r)$ not increasing in $x\}$. This generalizes the dependency graph defined for normal logic programs in (Anger et al. 2006). We assign a unique number *id* to every strongly connected component of this graph and map every variable in the component to this number through a function *scc* that preserves the topological order of the components. Next, we say that a BFASP is valid iff there is no rule $r \in R$ and a variable $x \in var(cons(r))$ s.t. $cons(r)$ is non-monotonic in x, and $scc(head(r)) = scc(x)$. This is a broad class of programs that includes all normal logic and linear programs. Our stable model semantics for BFASP guarantees that for any valid BFASP, stable models represent non-circular derivations. For a non-valid BFASP, the semantics defined above is insufficient for that purpose.

Example 3
Consider a program with lb-founded integers variables a, b and rule constraints: $r1 = (a \geq |b|, a)$, $r2 = (b \geq 2a - 6, b)$. The dependency graph has edges (a, b) and (b, a) due to $r1, r2$ respectively. Since the constraint $a \geq |b|$ is non-monotonic in b, and $scc(a) = scc(b)$, this is not a valid BFASP. □




```
n = 4;
e = 4;
start = [1, 2, 3, 4];
end =   [2, 3, 4, 1];
dist =  [8, 7, 6, 5];
```

Fig. 1. A graph of roads and distances, and its representation for the model.

4 A Language for Expressing BFASPs

Since BFASPs allow more complex rules over head or body variables, existing ASP languages are insufficient to express BFASPs. We extend the well-known CP modelling language MINIZINC (Nethercote et al. 2007) to express BFASPs. The extension is quite minimal. First, we define two new keywords "lbfvar" and "ubfvar" which can be used in place of the "var" keyword to indicate that a variable is lb-founded and ub-founded respectively. Secondly, we annotate every rule constraint with its head variables via "head(y)". Note that n-ary logical predicates in ASP are modelled as n-dimensional arrays of lb-founded Boolean variables.

Example 4
The shortest path program can be modelled as follows:

```
int: n;        % number of nodes
int: e;        % number of edges
set of int: Node = 1..n;
set of int: Edge = 1..e;
array [Edge] of int: start;     % start node of edge
array [Edge] of int: end;       % end node of edge
array [Edge] of int: dist;      % distance of edge

array [Node,Node] of ubfvar int: sp;      % shortest path from node x to node y

constraint forall (x, y in Node) (sp[x,y] <= sp[y,x] :: head(sp[x,y]));
constraint forall (x in Node) (sp[x,x] <= 0 :: head(sp[x,x]));
constraint forall (e in Edge, y in Node) (
        (sp[start[e], y] <= dist[e] + sp[end[e], y] :: head(sp[start[e], y])) /\
        (sp[end[e], y] <= dist[e] + sp[start[e], y] :: head(sp[end[e], y])));
solve satisfy;
```

The graph shown in Figure 1 is modelled with the instance data shown on the right. Under the logical semantics used by standard CP solvers, an assignment such as $\forall x,y,\theta(sp[x,y]) = 0$ is a solution to this P, but does not give the shortest paths between the nodes. Under stable model semantics however, this is not a solution. Calculating the minimal model of the reduct P^θ gives $sp[1,2] = 8$, $sp[1,3] = 11$, $sp[1,4] = 5$, $sp[2,3] = 7$, $sp[2,4] = 13$, $sp[3,4] = 6$, which does not match θ, so θ is not a stable model. It is easy to see that the only stable model for this problem is given by the minimal model of P^θ, which corresponds to the correct shortest path values. □

5 Enforcing Stable Model Semantics in BFASPs

To simplify the presentation, from this point on, we will assume that every ub-founded variable in the program has been replaced by an lb-founded variable as described in Section 3. A critical component of most current state of the art ASP solvers is an unfounded set detection algorithm. Detecting unfounded sets during search allows the solver to prune off unstable partial assignments early on and to ensure that the final solution is stable. For BFASPs, we have to generalize the concept of unfounded sets of propositional variables to unfounded sets of bounds. Given a set B of bounds of the form $[y \geq v]$ where $y \in \mathscr{F}$, B is an unfounded set of bounds w.r.t. to the current domain if given the current domain, no bound in the set can be established from its rules

without relying on some other bound in the set B. In an analogous way to the propositional case, if we find an unfounded set of bounds during search, then no stable model in that subtree can have any of those bounds true, so all of those bounds can be set false. We now present an unfounded set algorithm for BFASPs. This algorithm generalizes the unfounded set algorithm for the propositional case described in (Simons et al. 2002), which uses *source pointers* and utilizes *scc* to optimize the unfounded set detection.

As a preprocessing step, for all rule constraints $r = (c(y, x_1, \ldots, x_n), y)$, we convert them into the form $(y \geq f_r(x_1, \ldots, x_n), y)$, such that $c(y, x_1, \ldots, x_n) \Leftrightarrow y \geq f_r(x_1, \ldots, x_n)$. Such a function f_r always exists if c is increasing in y. Let $body(r) = \{x_1, \ldots, x_n\}$, and $body^+(r)$ and $body^-(r)$ be the subsets of $\{x \mid x \in body(r), scc(x) = scc(y)\}$ in which f_r is monotonically increasing and decreasing respectively. Let $body^0(r) = \{x \mid x \in body(r), scc(x) \neq scc(y)\}$. Let $rules(y) = \{r \mid head(r) = y\}$. Let $int\_rules(y) = \{r \mid head(r) = y, body^+(r) \neq \emptyset\}$. Let $ext\_rules(y) = rules(y) \setminus int\_rules(y)$. We also initialize a static data structure *EventToRule* consisting of pairs (e, r) where e is a domain change event that might cause r to stop justifying a bound on y. That is, for each x_i we add $(ub\_event(x_i), r)$, $(lb\_event(x_i), r)$ or $(change\_event(x), r)$ depending on whether f_r is increasing in x_i, decreasing x_i, or neither respectively.

The solver maintains a *justification graph JustGraph*, which consists of a set of triples (b, r, s), where b is a bound that is justified by some rule constraint r, and s is a set of other bounds with variables in the same level whose justification was used to infer that b might be justified. We define several functions of the current justification graph *JustGraph* and the current domain D. For brevity, we leave out the implicit arguments of *JustGraph* and D. Let $lb(x)$, $ub(x)$ and $val(x)$ give the current lower bound, upper bound and value respectively of x in D. $val(x)$ is undefined if x is not fixed in D. Let $jb(x) = max\{v \mid \exists r, \exists s, (x \geq v, r, s) \in JustGraph\}$. Let $ext\_jb(x) = max\{v \mid \exists r \in ext\_rules(x), \exists s, (x \geq v, r, s) \in JustGraph\}$.

We assume that for each rule r, we can evaluate the function f_r when all its arguments are fixed. Let $sjb(r)$ (strongest justified bound) be a function that returns a value such that no bound stronger than $y \geq sjb(r)$ can be justified by r given the current domain and justified bounds. We require that, when all the variables in $body(r)$ are fixed, $sjb(r)$ must return the smallest value with the above property. A naive way to implement $sjb(r)$ is as follows. If any of the variables in $body^0(r)$ are not yet fixed, return $sjb(r) = \infty$. Otherwise, return $sjb(r) = f_r(\gamma(x_1), \ldots, \gamma(x_n))$, where $\gamma(x) = ub(x)$ if $x \in body^+(r)$ and $\gamma(x) = lb(x)$ if $x \in body^-(r)$ and $\gamma(x) = val(x)$ if $x \in body^0(r)$. A better implementation will return non-infinite $sjb(r)$ values even when not all the variables in $body^0(r)$ are fixed yet. This can be done by leveraging the propagators in a CP solver, but for lack of space, we do not go into details. To allow clause learning, we also need to explain unfounded sets, so we need a function $expl(r)$ that returns a set of literals that explain why no bound stronger than $y \geq sjb(r)$ can be justified. For the naive implementation of $sjb(r)$ above, when not all vars in $body^0(r)$ are fixed, we return $expl(r) = \emptyset$. Otherwise, we return $expl(r) = \{x \leq ub(x) \mid x \in body^+(r)\} \cup \{x \geq lb(x) \mid x \in body^-(r)\} \cup \{x = val(x) \mid x \in body^0(r)\}$.

We post a foundedness propagator for each (non-empty) SCC, which is responsible for the variables in the component. The priority of each propagator is the same as the *id* of that component. The propagators support four methods: propagate(), processEvents(), getUnfoundedSet() and getExpl(). The pseudocode is given in Figure 2. The function propagate() first calls processEvents() to process any domain change events and dejustify any bounds whose justification has been made invalid by the domain changes. Next, if there is any bound in the current domains of any variable x with $scc(x) = id$ that is not justified, it calls getUnfoundedSet() to either rejustify the bounds, or find an unfounded set. If an unfounded set is found, then it sets the bounds in the unfounded set *false* with the explanation from the call getExpl(). If there are any

```
propagate()
  processEvents()
  HasUnjust = {y | scc(y) = id, jb(y) < max(D(y))}
  if(HasUnjust ≠ ∅)
    Unfounded = getUnfoundedSet()
    if(Unfounded ≠ ∅)
      E = getExpl(Unfounded)
      for(y ∈ Unfounded)
        Propagate y ≤ jb(y) with explanation E
  if(HasUnjust ≠ ∅) requeuePropagator()

processEvents()
  Dejustified = ∅
  LostExt = ∅
  for(e ∈ DomEvents)
    for(r s.t. (e,r) ∈ EventToRule)
      for((y ≥ k, r', s) ∈ JustGraph s.t. r = r')
        if(k = ext_jb(y)) LostExt = LostExt ∪ {y}
        JustGraph = JustGraph − {(y ≥ k, r', s)}
        Dejustified = Dejustified ∪ {y ≥ k}
  for(y ∈ LostExt)
    r = argmax_{r' ∈ ext_rule(y))} sjb(r')
    JustGraph = JustGraph ∪ {(y ≥ sjb(r), r, ∅)}
    Dejustified = Dejustified − {y ≥ sjb(r)}
  while(∃b ∈ Dejustified)
    Dejustified = Dejustified − {b}
    for((bound, rule, supports) ∈ JustGraph s.t. b ∈ supports)
      JustGraph = JustGraph − {(bound, rule, supports)}
      Dejustified = Dejustified ∪ {bound}

getUnfoundedSet()
  let z be such that z ∈ HasUnjust
  HasUnjust = HasUnjust − {z}
  Unfounded = Unfounded ∪ {z}
  UnprocRules = int_rules(z)
  while(∃r ∈ UnprocRules)
    UnprocRules = UnprocRules − {r}
    y = head(r)
    nb = min(sjb(r), max(D(y)))
    if(nb > jb(y))
      JustGraph = JustGraph ∪
        {(y ≥ nb, r, {x ≥ jb(x) | x ∈ body+(r)})}
      UnprocRules = UnprocRules ∪
        {r' | r' ∈ R, y ∈ body+(r'), head(r') ∈ Unfounded}
    if(nb = max(D(y)))
      Unfounded = Unfounded − {y}
    else
      for(l ∈ expl(r))
        if(D ⊭ l)
          HasUnjust = HasUnjust − {var(l)}
          Unfounded = Unfounded ∪ {var(l)}
          UnprocRules = UnprocRules ∪ int_rules(var(l))
  return Unfounded

getExpl(Unfounded)
  E = ∅
  for(y ∈ Unfounded)
    for(r ∈ rules(y))
      E = E ∪ expl(r)
  E = E − {y ≤ v | y ∈ Unfounded, v ≥ jb(y)}
  return E
```

Fig. 2. Pseudocode for unfounded set detection algorithm

unprocessed variables left, the propagator requeues itself in the propagation queue so that higher priority propagators can run before this propagator is run again.

The function processEvents() crawls through the existing justification graph to find out what bounds lose their justification due to new domain changes. The function getUnfoundedSet() does a fix-point calculation using the *sjb*() to find out what bounds can be rejustified. The function getExpl(*Unfounded*) resolves the explanations for why each bound cannot be justified in order to derive an explanation for the whole unfounded set. See the pseudocode for more details.

The algorithm is initialized as follows. At the root node, $JustGraph = \emptyset$. We wake up all foundedness propagators. On the first call to each foundedness propagator at the root node, we initialize *LostExt* to all the variables in that level instead of the usual $\emptyset$. *JustGraph* is a trailed data structure, so upon backtracking, it is reverted to its previous value.

Example 5

Consider the problem of building a network of roads subject to monetary constraints. Each segment of road is optional and has a certain build cost, and we want to find the road design that gives the best total travel times. We can model this with a slight alteration to the shortest path model in Example 4. We remove the last constraint and the solve goal and add:

```
array [Edge] of int: cost;                % cost to build road
array [Edge] of var bool: b;              % build road or not
int: budget;                              % max cost of roads built

constraint forall (e in Edge, y in Node) (
    (sp[start[e], y] <= dist[e] + sp[end[e], y] <- b[e] :: head(sp[start[e], y])) /\
    (sp[end[e], y] <= dist[e] + sp[start[e], y] <- b[e] :: head(sp[end[e], y])));
constraint sum (e in Edge) (cost[e] * bool2int(b[e])) <= budget;

solve minimize sum (i, j in Node where i < j) (sp[i,j]);
```

Consider the instance from Example 4 again but where we add the following lines representing the build cost and budget to the data file:

```
cost = [50, 30, 20, 60];
budget = 100;
```

Since the b variables do not depend on anything else, the SCC algorithm will set them to level 0. The set of sp variables are cyclically dependent on each other, so the SCC algorithm will set them to level 1. The first set of constraints in the model force $sp[x,y] = sp[y,x]$ and we will not mention them any further. At the root node, the second set of rules give $sp[x,x] \leq 0$ for all x with empty justifications. Depending on the evaluation order, different rules in the third set may justify the same bound. We will just pick any one of them in this example. The third set of rules give: $sp[1,2] \leq 8$, $sp[2,3] \leq 7$, $sp[3,4] \leq 6$, $sp[4,1] \leq 5$ with empty justifications, $sp[1,3] \leq 11$ with justification $\{sp[3,4] \leq 6\}$, and $sp[2,4] \leq 13$ with justification $\{sp[3,4] \leq 6\}$. No stronger upper bounds can be justified, thus the foundedness propagator forces $sp[1,2] \geq 8$, $sp[2,3] \geq 7$, $sp[3,4] \geq 6$, $sp[4,1] \geq 5$, $sp[1,3] \geq 11$, $sp[2,4] \geq 13$ with explanation $\emptyset$. Note that what unfounded set detection is doing is basically to calculate the most optimistic values of the shortest paths given the current set of decisions, and then set those values as the lower bounds of the $sp[x,y]$.

Suppose search tries $b[1] = \mathit{true}$. Propagation forces $sp[1,2] = 8$. The budget constraint forces $b[4] = \mathit{false}$. This domain change event triggers the rules $sp[4,1] \leq 5 \leftarrow b[4]$ and $sp[1,3] <= 5 + sp[3,4] \leftarrow b[4]$ to stop justifying bounds on their heads, so $sp[4,1] \leq 5$ and $sp[1,3] \leq 11$ become dejustified. The third set of constraints now allow us to rejustify a bound of $sp[1,3] \leq 15$ with justification $\{sp[2,3] \leq 7\}$, followed by $sp[4,1] \leq 21$ with justification $\{sp[1,3] \leq 15\}$. No stronger bounds can be justified, thus the foundedness propagator forces $sp[1,3] \geq 15$ and $sp[4,1] \geq 21$ with explanation $\{\neg b[4]\}$.

Suppose then search tries $b[2] = \mathit{true}$. Propagation forces $sp[2,3] = 7$ and $sp[1,3] = 15$. Suppose search then tries $b[3] = \mathit{true}$. Propagation forces $sp[3,4] = 6$, $sp[2,4] = 13$ and $sp[4,1] = 21$. This gives a stable model with objective value: $6 + 7 + 8 + 13 + 15 + 21 = 70$. Suppose we backtracked and tried $b[3] = \mathit{false}$ instead. This domain changed event causes $sp[3,4] \leq 6$ and $sp[4,1] \leq 21$ to become dejustified. Since $sp[3,4] \leq 6$ was used to support $sp[2,4] \leq 13$, that also becomes dejustified. No bounds can be rejustified, thus the foundedness propagator forces $sp[2,4] \geq \infty$, $sp[3,4] \geq \infty$ and $sp[1,4] \geq \infty$ with explanation $\{\neg b[4], \neg b[3]\}$, i.e., there is no path from a, b or c to d. This stable model has an objective value of ∞. □

6 Experimental Results

We compare our implementation with state of the art ASP system GRINGO (version `3.0.4`) + CLASP (version `2.0.6`), which we call CL+GR in this section. We refer to our implementation as CHUFFED. At present it only supports bound founded integers and Booleans. All experiments were run 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. Every instance was run 10 times, and each timing (that is not —) in the tables in this section is the median of these 10 times. All times are given in seconds. For all experiments, the timeout for grounding and flattening was 5 minutes, and timeout for solving was 10 minutes.[2] We ran our experiments on non-expert encodings of the following problems:

ShortPath: Finding the shortest path from a source node to a destination node in a directed graph of N nodes and E edges.

RoadCon: The road construction problem as defined in Example 5, on an undirected graph of N nodes, where D is the maximum distance between two points and C is the maximum cost to build a road segment.

CompCon: Company controls problem as defined in the introduction where C is the number of

[2] All instances and encodings used are available in GRINGO and MINIZINC format at: `http://ww2.cs.mu.oz.au/~pjs/bound_founded/`

companies and S is the number of shares each company has.
UtilPol: Utilitarian policies: Suppose a government wants to decide which policies to enact in order to maximize the happiness of its citizens. Enacting each policy incurs a certain cost. Additionally, the happiness of a citizen depends on the chosen policies as well as the happiness of other citizens. A citizen's happiness can be increased (or decreased) by a certain amount if some other citizen's happiness is above a certain threshold. More formally, the input consists of P policies, C citizens, cost of each policy p ($cost_p$), total available budget (b), utility of policy p for citizen c ($util_{c,p}$), change in happiness of citizen c due to happiness of another citizen c' ($rel_{c,c'}$) and the threshold after which this change applies ($t_{c,c'}$). We can represent the happiness of a citizen c by the lb-founded integer hap_c, and the decision whether or not to enact a policy p by a Boolean variable en_p. The happiness of a citizen c is given by the following rule constraint:

$$(hap_c \geq \sum_p util_{c,p} en_p + \sum_{c'} rel_{c,c'}(hap_{c'} \geq t_{c,c'}), \{hap_c\})$$

The objective is to maximize the sum of happiness of all citizens by choosing a set of policies to enact given a budget.

We chose these problems since they contain founded bounds on integers. The results from our experiments are presented in Tables 1 to 4 in the following format: each row represents the results for a particular instance. The performance of each system is given by several columns. The first column gives the grounding (gr) or flattening (flat) time. If the system failed to complete this phase, then a — is put in all its columns. The second column gives the best value produced by the system (omitted in Table 1 which deals with satisfiability). The third column tells whether the system provably found the optimal value. If a system did not produce any answer for an instance, then a — is put for this field. The final column gives the actual time that the solver took to solve the instance, the value is 600 if the solver did not complete before timeout or ran out of memory.

Table 1 presents comparison of CL+GR, a CP solver CPX and CHUFFED on the shortest path *ShortPath* benchmark. The CP encoding views the shortest path problem as a combinatorial problem, where the solver constructs the shortest path by choosing a set of edges. CL+GR only produces a solution for the smallest instance, and fails to ground the other instances in the allocated time. The comparison between CHUFFED and CPX is more interesting. On smaller instances, performance of both systems is comparable but on bigger instances, the flattening plus the solving time of the combinatorial encoding increases significantly. We do not compare with CPX in the remaining experiments since it requires modelling the cyclic dependencies inductively which quickly blows up.

Table 2 compares the performance of CL+GR and CHUFFED on the road construction *RoadCon* benchmark. CL+GR starts to break down on the second instance but CHUFFED continues to give answers for instances that contain as many as 60 nodes. Table 3 presents results on the company controls benchmark *CompCon*. Grounding is not the bottleneck for this benchmark, but the solving time of CLASP increases significantly as the number of stocks and number of companies are increased. For CHUFFED, increasing the number of companies makes the problem significantly harder but the effect of increase in the number of stocks is relatively milder as compared to CLASP. CHUFFED solves an instance that has 30 companies and fails on an instance in which there are 50 companies. Table 4 presents results for utilitarian policies problem *UtilPol*. As in previous benchmarks, CHUFFED comprehensively outperforms CL+GR. The domains of *rel* and *util* variables were set to $0 \ldots 5$ in all instances. Any increase in the sizes of domains of these variables has no effect on the performance of CHUFFED but has a significant effect on the performance of CL+GR. The right of Table 4 shows the sizes of the grounded program and the running times of CLASP as the domains of variables are scaled by the number given. For a tenfold

		CL+GR			CHUFFED			CPX		
N	E	gr	comp	time	flat	comp	time	flat	comp	time
20	10	6.00	Y	2.66	0.01	Y	0.00	0.03	Y	0.06
50	20	—	—	—	0.05	Y	0.01	0.37	Y	0.36
100	30	—	—	—	0.25	Y	0.12	3.12	N	600
250	100	—	—	—	1.00	Y	1.16	28.11	N	600
600	200	—	—	—	2.50	Y	8.97	169.08	N	600
800	200	—	—	—	12.79	N	600	—	—	—

Table 1. *ShortPath*

			CL+GR				CHUFFED			
N	D	C	gr	opt	comp	time	flat	opt	comp	time
5	10	8	0.43	24	Y	0.65	0	24	Y	0.00
9	11	13	111.79	—	N	600	0.02	123	Y	0.06
25	45	40	—	—	—	—	0.66	2629	Y	3.46
45	25	32	—	—	—	—	4.91	2174531	N	600
60	30	25	—	—	—	—	13.15	6613781	N	600

Table 2. *RoadCon*

		CL+GR				CHUFFED			
S	C	gr	opt	comp	time	flat	opt	comp	time
10	5	0.00	102	Y	0.00	0.00	102	Y	0.00
50	10	0.05	702	N	600	0.02	234	Y	0.14
100	30	0.97	14701	N	600	0.22	408	Y	5.99
250	30	2.65	—	N	600	0.21	75	Y	35.52
500	50	16.63	—	N	600	0.72	—	N	600

Table 3. *CompCon*

increase in the domain size, the size of the ground program increases from 748 kilobytes to 337 megabytes. The running time also increases significantly until timing out on the last 3 instances. In contrast CHUFFED requires 3.6kB and less than 0.005 seconds for all instances.

7 Related Work

ASP and many of its major extensions are subsumed by BFASP. ASP with choice rules, weighted sum, max, min etc, (see (Calimeri et al. 2013) for a recent standardization of the ASP language) are easily expressible as BFASPs. Similar, Constraint Answer Set Programming (CASP) (Gebser et al. 2009) and even Fuzzy ASP (FASP) (Nieuwenborgh et al. 2006; Blondeel et al. 2013) can all be expressed directly as BFASP instances.

In CASP, all rules are of the form: $a \leftarrow p_1 \wedge \ldots \wedge p_n \wedge \neg n_1 \wedge \ldots \wedge \neg n_m \wedge c_1(\bar{x}) \wedge \ldots \wedge c_l(\bar{x})$ where p_i, n_i are ASP variables and c_i are FD constraints over standard integer variables. We can convert it to a BFASP by adding the head annotation "head(a)" to each CASP rule. The stable model semantics of this BFASP are identical to that of CASP defined in (Gebser et al. 2009).

Fuzzy ASP can be converted into BFASPs with the same stable model semantics. In FASP, the model must specify how each logical connective acts on the value of its arguments. However, all the commonly used options allow the FASP to be directly translated into BFASP. For example, depending on whether we use the Lukasiewicz, Godel-Dummett, or product norm, a FASP rule: $x \leftarrow p[1] \wedge \ldots \wedge p[k] \wedge \neg n[1] \wedge \ldots \wedge \neg n[m]$ translates into one of the following BFASP rules:

```
x >= sum(i in 1..k)(p[i]) + sum(i in 1..m)(1-n[i]) - (k + m - 1) :: head(x)
```

		CL+GR				CHUFFED			
C	P	gr	opt	comp	time	flat	opt	comp	time
3	5	0.07	34	Y	.36	0.01	34	Y	0.00
5	6	66.64	—	N	600	0.00	680	Y	0.00
20	25	89.10	—	N	600	.06	886	Y	208.64
90	90	—	—	—	—	1.40	12628	N	600
250	230	—	—	—	—	13.74	—	N	600

Scale	gr kB	CL+GR
1	748	0.36
2	2816	1.25
3	8252	9.39
4	18741	37.83
5	36288	114.22
6	67570	260.39
7	113475	498.09
8	170287	600
9	243608	600
10	336386	600

Table 4. *UtilPol comparison, and scaling behaviour on the smallest instance*

```
x >= min(i in 1..k)(p[i]) <- forall(i in 1..m)(not n[i]) :: head(x)
x >= product(i in 1..k)(p[i]) <- forall(i in 1..m)(not n[i]) :: head(x)
```

It may initially appear that BFASP is very similar to FASP. However, BFASP is a significant generalization. It is far more generic in terms of what the head variable can represent (not just fuzzy Booleans but arbitrary values in $\mathbb{R}$) and it allows a wider range of constraints in the rules (not just logical connectives). Furthermore, our implementation can handle non-linear rules such as those produced by the product norm (which the MIP based method of (Janssen et al. 2008) cannot handle), and we have an unfounded set detection algorithm for BFASP that allows us to prune unstable partial models during search (which the MIP-based method of (Janssen et al. 2008) also cannot do).

There is another body of work in *partial order programming* (Osorio and Jayaraman 1996) that is similar to BFASPs. In this work, the authors also look at logic programs with rules of form $y \geq f(expr) \leftarrow conditions$. However, they only define the semantics of a limited subclass of such programs. The cases they consider, in our terminology, are as follows. Case 1, they consider any variable in *expr* or *conditions* as standard variables only and apply the standard logical semantics (Section 4.1 of (Osorio and Jayaraman 1996)). This is completely different from the stable model semantics and will give the wrong answers for a simple shortest path program like Example 4. Case 2, they consider only *hierarchical* programs where every variable in *expr* or *conditions* belongs to a previous SCC (Prop 5.1 of (Osorio and Jayaraman 1996)). This is insufficient to model shortest path, as all those variables are in the same SCC. Case 3, they allow non-hierarchical programs, but same SCC variables can only interact through increasing monotonic functions, i.e., f is increasing in all arguments that are in the same SCC as y. This allows shortest path to be modelled, but is still far less expressive than the stable model semantics defined in this paper. Our semantics allows f to be decreasing in arguments from the same hierarchical level, and thus we truly generalize negation by failure to continuous domains, giving the generalization of stable model semantics to continuous variables.

8 Conclusion

We have generalized the stable model semantics to cover bound founded variables and shown how to enforce these semantics on a wide range of problems involving rules with bound founded variables in the head or body. The new solver seamlessly integrates the best features of CP, SAT and ASP. It can use the highly efficient global propagators and finite domain variables from CP, the powerful clause learning from SAT, and the modeling power of stable model semantics from ASP. This new technology allows a wide range of problems to be modeled and solved much more effectively than before.

References

ANGER, C., GEBSER, M., AND SCHAUB, T. 2006. Approaching the core of unfounded sets. In *Proceedings of the International Workshop on Nonmonotonic Reasoning*. Clausthal University of Technology, Institute for Informatics, 58–66.

AZIZ, R. A., STUCKEY, P. J., AND SOMOGYI, Z. 2013. Inductive definitions in constraint programming. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference*, B. Thomas, Ed. CRPIT, vol. 135. ACS, 41–50.

BARAL, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

BASELICE, S., BONATTI, P. A., AND GELFOND, M. 2005. Towards an integration of answer set and constraint solving. In *Proceedings of the 21st International Conference on Logic Programming*. ICLP'05. Springer-Verlag, 52–66.

BLONDEEL, M., SCHOCKAERT, S., VERMEIR, D., AND DE COCK, M. 2013. Fuzzy answer set programming: An introduction. In *Soft Computing: State of the Art Theory and Novel Applications*. Springer, 209–222.

CALIMERI, F., FABER, W., GEBSER, M., IANNI, G., KAMINSKI, R., KRENNWALLNER, T., LEONE, N., RICCA, F., AND SCHAUB, T. 2013. Asp-core-2 input language format. https://www.mat.unical.it/aspcomp2013/ASPStandardization.

DRESCHER, C. AND WALSH, T. 2010. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming 10,* 4-6, 465–480.

DRESCHER, C. AND WALSH, T. 2012. Answer set solving with lazy nogood neneration. In *Technical Communications of the 28th International Conference on Logic Programming*. 188–200.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*. MIT Press, 386.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artif. Intell 187*, 52–89.

GEBSER, M., OSTROWSKI, M., AND SCHAUB, T. 2009. Constraint answer set solving. In *Proceedings of the 25th International Conference on Logic Programming*. Springer, 235–249.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming*. MIT Press, 1070–1080.

JANSSEN, J., HEYMANS, S., VERMEIR, D., AND COCK, M. D. 2008. Compiling fuzzy answer set programs to fuzzy propositional theories. In *Proceedings of the 24th International Conference on Logic Programming*. Springer Berlin Heidelberg, 362–376.

LIU, G., JANHUNEN, T., AND NIEMELA, I. 2012. Answer set programming via mixed integer programming. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning*. AAAI Press, 32–42.

MARQUES-SILVA, J. P. AND SAKALLAH, K. A. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers 48,* 5, 506–521.

MELLARKOD, V. S., GELFOND, M., AND ZHANG, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence 53,* 1-4, 251–287.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference*. ACM, 530–535.

NETHERCOTE, N., STUCKEY, P. J., BECKET, R., BRAND, S., DUCK, G. J., AND 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*. Vol. 4741. Springer, 529–543.

NIEUWENBORGH, D. V., COCK, M. D., AND VERMEIR, D. 2006. Fuzzy answer set programming. In *Proceedings of Logics in Artificial Intelligence, 10th European Conference, JELIA 2006*. Springer Berlin Heidelberg, 359–372.

OSORIO, M. AND JAYARAMAN, B. 1996. Aggregation and well-founded semantics. In *Proceedings of Non-Monotonic Extensions of Logic Programming, NMELP '96*. Springer-Verlag, 71–90.

ROSSI, F., BEEK, P. V., AND WALSH, T. 2006. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science, New York, NY.

SCHULTE, C. AND STUCKEY, P. J. 2008. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems 31,* 1, Article No. 2.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1–2, 181–234.

VAN GELDER, A., ROSS, K. A., AND 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*. ACM, 221–230.