A Preliminary Report on Integrating of Answer Set and Constraint Solving

Sabrina Baselice[1] and Piero Bonatti[1] and M. Gelfond[2]

[1] Dipartimento di Scienze Fisiche, Università "Federico II"
Complesso Universitario di Monte Sant'Angelo
Via Cinthia, Napoli, Italy
{baselice,bonatti}@na.infn.it
[2] Texas Tech University
Department of Computer Science
Lubbock, TX, USA
mgelfond,@cs.ttu.edu

Abstract. Despite all efforts on intelligent grounding, state-of-the-art answer set solvers still have huge memory requirements, because they compute the ground instantiation of the input program before the actual reasoning starts. This prevents ASP to be effective on several classes of problems. In this paper we integrate answer set generation and constraint solving to reduce the memory requirements for a class of multi-sorted *logic programs with cardinality constraints*. We prove some theoretical results, introduce a provably sound and complete algorithm, and report experimental results showing that our approach can solve problem instances with significantly larger domains.

1 Introduction

Answer set solvers are proving to be competitive with other reasoners on several benchmarks [10, 11], and are being used successfully as planners and plan verifiers in the RCS/USA Advisor system [1, 9], a decision support system for NASA shuttle controllers (http://krlab.cs.ttu.edu/~marcy/RCS/).

Still, state-of-the-art answer set solvers have a major limitation: they use huge amounts of memory, because the ground instantiation of the input program must be computed before the actual reasoning starts. This problem is mitigated to some extent through intelligent grounding techniques that partially evaluate program rules when possible, thereby deleting some rule instances that are surely not applicable. However, this technique is not effective enough on some classes of programs, including several programs for reasoning about actions and change.

In this paper we integrate answer set generation and constraint solving to reduce the memory requirements for a class of multi-sorted *logic programs with cardinality constraints* [10, 11] whose signature can be partitioned into: (i) a set of so-called *regular predicates* over domains whose size can be handled by a standard answer set solver; (ii) a set of *constrained* predicates that can be handled by a constraint solver in a way that does not require grounding (so larger domains can be allowed here); (iii) a set

of predicates—called *mixed predicates*—that create a "bridge" between the above two partitions.

Then reasoning can be implemented by having an answer set solver interact with a constraint solver. A critical aspect is the form that the definitions of mixed predicates may take. If they were completely general, then that part of the program would be just as hard to reason with as unrestricted programs because mixed predicates may range over arbitrary domains. Accordingly, the framework introduced in this paper supports restricted definitions for mixed predicates, that can be either functions from "regular" to "large" domains (strong semantics) or slightly weaker mappings where each combination of "regular" values must be associated to at least one vector of values from "large" domains (weak semantics).

We study the relationships between strong and weak semantics, and introduce an algorithm for computing the strong semantics efficiently under the simplifying assumption that mixed predicates do not occur in the scope of negation. Moreover, we report experimental results providing preliminary evidence that our approach can solve problem instances with significantly larger domains. In this first paper we focus only on the comparison with a standard answer set programming approach.

The paper is organized as follows. The next section is devoted to preliminaries. Then, in Section 3, we introduce the class of programs we deal with, and prove some of their theoretical properties. The algorithm for reasoning on these programs is described and proved to be correct and complete in Section 4. Section 6 reports the experiments and Section 7 concludes the paper with a final discussion and possible directions for future work.

2 Preliminaries

We adopt a sorted first-order language based on a given signature Σ. Letters x, y, z range over variables, a, b, c range over constant symbols, letters f, g, h over function symbols, and letters p, q, r over predicate symbols. Let $\mathcal{S}$ be a finite set of *sorts*. And assume a *sort specification* is given, that is, a function sort mapping:

- each constant c onto a set $\mathsf{sort}(c) \subseteq \mathcal{S}$;
- each variable x onto a (single) sort $\mathsf{sort}(x) \in \mathcal{S}$;
- each n-ary function symbol f onto a tuple $\mathsf{sort}(f) = \langle S_1, \ldots, S_{n+1} \rangle \in \mathcal{S}^{n+1}$;
- each n-ary predicate symbol p onto a tuple $\mathsf{sort}(p) = \langle S_1, \ldots, S_n \rangle \in \mathcal{S}^n$.

Note that sorts may overlap because constants may be associated to two or more sorts.

Example 1. A sort *steps*, modelling plan steps, may contain the integer constants in the interval $[0, 10]$, while a sort *time*, modelling time points, may contain the integer constants in $[0, 600000]$.

All the other terms have a unique sort. Intuitively, in $\mathsf{sort}(f)$, S_i is the sort of the i-th argument of f ($1 \leq i \leq n$) and S_{n+1} is the sort of the output. Similarly, in $\mathsf{sort}(p)$, S_i is the sort of the i-th argument of predicate p ($1 \leq i \leq n$).

Terms and atoms are defined accordingly. Each variable x with $\mathsf{sort}(x) = S$ and each constant c such that $S \in \mathsf{sort}(c)$ are terms of sort S. Each expression $f(t_1, \ldots, t_n)$

such that $\mathsf{sort}(f) = \langle S_1, \ldots, S_n, S \rangle$ and each t_i is a term of sort S_i is a term of sort S. Nothing else is a term. We write $t : s$ to state that term t belongs to sort s.

All expressions $p(t_1, \ldots, t_n)$ such that $\mathsf{sort}(p) = \langle S_1, \ldots, S_n \rangle$ and each t_i is a term of sort S_i are atoms. Literals are either atoms (positive literals) or expressions of the form `not` A where A is an atom (negative literals).

A variable substitution over $\{x_1, \ldots, x_n\}$ is a function mapping each variable x_i onto a term of $\mathsf{sort}(x_i)$. The notions of instance and ground instantiation are defined as usual from the above notion of (typed) substitution. The ground instantiation of a set of expressions E will be denoted by $\mathsf{ground}(E)$.

Given a logic program P consisting of *normal rules* $A \leftarrow L$ and *denials* $\leftarrow L$, where L is a collection of literals, the *stable models* of P [5] are defined as follows.

We first need a notion of program *reduct* P^I, where I is a set of ground atoms. The reduct P^I is obtained from $\mathsf{ground}(P)$ by removing:

- all the rules and constraints with a literal `not` B in their body, s.t. $B \in I$;
- all negative literals from the remaining rules and constraints.

Note that P^I is a set of Horn clauses. Therefore, if P^I is consistent, then it has a unique minimal Herbrand model, that will be denoted by $\mathsf{lm}(P^I)$.

Now I is a *stable model* of P if and only if $I = \mathsf{lm}(P^I)$.

The most popular answer set frameworks are based on the above notions of program and semantics, and extensions thereof. Answer sets are identified with stable models; each answer set represents a possible solution to the given problem instance (programs may have no stable models, as well as multiple stable models). One important extension consists of *cardinality constraints* [10, 11], that in their simplest version are expressions of the form

$$l\{A\}u$$

where A is an atom, l and u are integers. Roughly speaking, $l\{A\}u$ forces the answer sets of the given program to contain a number n of instances of A, such that $l \leq n \leq u$ (u may be omitted in case there is no upper bound). The complete framework is more general. It allows for cardinality constraints in rule bodies and *weight constraints*, that generalize cardinality constraints and allow programmers to express preferences and optimization criteria on problem solutions. For a general and precise definition of cardinality and weight constraints, the reader is referred to [10, 11]. They are fully supported by SMODELS.

3 Constrained Programs

The sorts of constrained programs are partitioned into *regular* and *constrained* sorts. Intuitively, regular sorts are small enough to be handled by standard answer set solvers, while constrained sorts are large enough to require reasoners that do not instantiate the corresponding variables.

Variables and constants are called *regular* or *constrained* according to their sorts. A function f is regular (resp. constrained) if all the sorts in $\mathsf{sort}(f)$ are regular (resp. constrained). Function f is *mixed* if $\mathsf{sort}(f)$ comprises both regular and constrained sorts. Predicate symbols are classified in a similar way.

In this paper we assume that the output sort of all functions is a constrained sort. The reason is that most answer set solvers do not (yet) support function symbols, while constraint solvers do (functions are typically standard arithmetic functions).

According to the above classification, signature Σ is partitioned into Σ_r, Σ_c and Σ_m, where *r*, *c* and *m* stand for *regular*, *constrained* and *mixed* respectively.

The atoms over Σ_r, Σ_c, and Σ_m are referred to as *r*-atoms, *c*-atoms, and *m*-atoms respectively. Similarly for literals. The parameters of an *m*-atom whose sorts are constrained (regular) will be often referred to as *c*-parameters (*r*-parameters).

We assume that *c*-predicates have a predefined interpretation, and that the equality predicate is a *c*-predicate. The intended interpretation of *c*-predicates will be represented by a set of ground atoms M_c (the set of all true ground *c*-atoms).

Regular predicates can be defined with normal programs, as in standard ASP. The definitions of mixed predicates are restricted, instead. Let an atom be *free* if its arguments are all pairwise distinct variables. For all free atoms A we write $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$ to state that the *r*-variables (resp. *c*-variables) of A are those in $\boldsymbol{x}_r$ (resp. $\boldsymbol{x}_c$). We denote with $A(\boldsymbol{a}, \boldsymbol{b})$ the instance of A such that $\boldsymbol{x}_r$ is replaced by $\boldsymbol{a}$ and $\boldsymbol{x}_c$ with $\boldsymbol{b}$.

In this paper we deal with two possible semantics of mixed predicates.[3] Under the *weak semantics*, for all free mixed atoms $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$ there is an implicit axiom

$$\forall \boldsymbol{x}_r \exists \boldsymbol{x}_c . A(\boldsymbol{x}_r, \boldsymbol{x}_c)\,, \tag{1}$$

that can be expressed by including into the program a cardinality constraint $1\{A(\boldsymbol{a}, \boldsymbol{x}_c)\}$ for each sequence of ground arguments $\boldsymbol{a}$ of the appropriate type and length.[4]

Under the *strong semantics*, for all free mixed atoms $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$ there is an implicit axiom

$$\forall \boldsymbol{x}_r \exists ! \boldsymbol{x}_c . A(\boldsymbol{x}_r, \boldsymbol{x}_c)\,, \tag{2}$$

that can be encoded in a similar way with a suitable set of cardinality constraints like $1\{A(\boldsymbol{a}, \boldsymbol{x}_c)\}1$.

Moreover, constrained programs may contain constraints that relate all kinds of predicates (regular, constrained, and mixed).

Definition 1

1. *A* regular rule *(r-rule) is a rule of the form $A \leftarrow B$ or $\leftarrow B$ where A is an r-atom and B is a collection of r-literals.*
2. *A (proper)* constraint *is a rule of the form $\leftarrow B$ where B is a collection of arbitrary literals, including at least one nonregular literal.*
3. *A* constrained program, *P, is the union of a set of regular rules, R(P), and a set of constraints, C(P).*

Example 2. In our running example (a planning and scheduling problem) we have two regular sorts: *step* (representing plan steps) and *action*. We write $step : 0..10$ to state that the constants *c* with $step \in \mathsf{sort}(c)$ are those in the integer interval $[0, 10]$. Analogously,

[3] A more general approach is described in the final discussion.

[4] In SMODELS this can be done with a single rule having a cardinality constraint in the head. A similar remark applies to the encoding of (2). We refer the reader to [10, 11] for more details.

we may write $action : a_1, \ldots, a_n$ to enumerate all possible actions. The regular signature Σ_r contains only one relation o over $action \times step$. Intuitively, $o(A, S)$ means that action A occurs at step S. The regular part $R(P)$ contains n rules that force at least one action to be executed at each step. For $i = 1, \ldots, n$:

$$o(a_i, S) \leftarrow \texttt{not}\ o(a_1, S), \ldots, \texttt{not}\ o(a_{i-1}, S), \texttt{not}\ o(a_{i+1}, S), \ldots, \texttt{not}\ o(a_n, S).$$

Moreover, $R(P)$ contains a denial that forbids concurrent actions:

$$\leftarrow o(A_1, S), o(A_2, S), \texttt{not}\ eq(A_1, A_2).$$
$$eq(X, X).$$

The constraint signature Σ_c comprises the sort $time : 0..600000$ with the standard arithmetic functions: $+, -, ||$ etc., and relations: $>, \geq$, etc.

The mixed signature Σ_m comprises a relation $time(S, T)$ associating each plan step S to at least one time point T under the weak semantics (exactly one under the strong semantics).

The following constraints $C(P)$ ensure that time is assigned to steps monotonically and that each step is associated to exactly one time point (the latter is needed only under the weak semantics);

$$\leftarrow time(S1, T1), time(S2, T2), S1 < S2, T1 \geq T2.$$
$$\leftarrow time(S, T1), time(S, T2), T1 \neq T2.$$

Moreover, one can specify a minimal duration for each action, e.g., 3 time units for a_1

$$\leftarrow o(a_1, S1), time(S1, T1), o(A2, S2), time(S2, T2), |T2 - T1| < 3\,. \qquad (3)$$

Formally, the semantics of constrained programs is a specialization of the stable model semantics for logic programs with weight constraints, taking into account the intended interpretation M_c of Σ_c and the implicit semantics of mixed predicates.

We first need a generalization of the program *reduct* P^I, where P is now a constrained program and I a set of ground atoms. The reduct P^I is obtained from $\mathsf{ground}(P)$ by removing:

- all the rules and constraints with a literal $\texttt{not}\ B$ in their body, s.t. $B \in I \cup M_c$;
- all rules and constraints with a c-atom A in their body, such that $A \notin M_c$;
- all negative literals and c-atoms from the remaining rules and constraints.

Note that P^I is a set of Horn clauses also under the generalized definition. Therefore, if P^I is consistent, then it has a unique minimal Herbrand model $\mathsf{lm}(P^I)$. Like the standard notion of reduct, P^I results from the evaluation of negative literals against I. Moreover, the generalized notion evaluates all the constrained literals w.r.t. their intended semantics M_c.

Definition 1. *A* weak answer set *of a constrained program P is a set of ground atoms $M = M_r \cup M_m$ satisfying the following conditions:*

AS1 *M_r is a set of r-atoms and M_m is a set of m-atoms;*
AS2 *$R(P)^{M_r}$ is consistent and $M_r = \mathsf{lm}(R(P)^{M_r})$;*
AS3 *each constraint $(\leftarrow \boldsymbol{L}) \in \mathsf{ground}(C(P))$ contains a literal L_i false in M;*
AS4 *for each free m-atom $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$, and for each vector of r-constants $\boldsymbol{a}$ of the appropriate length, M_m contains at least one instance of $A(\boldsymbol{a}, \boldsymbol{x}_c)$.*

A strong answer set *of a constrained program P is a weak answer set $M = M_r \cup M_m$ satisfying the following additional condition:*

AS5 *for each free m-atom $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$, and for each vector of r-constants $\boldsymbol{a}$ of the appropriate length, M_m contains* at most *one instance of $A(\boldsymbol{a}, \boldsymbol{x}_c)$.*

Note that AS2 basically states that M_r is a stable model of the regular part of P.

Remark 1. We might have alternatively specified the semantics of a constrained program P as the stable models of the program obtained by extending P with M_c and with the cardinality constraints that encode (1) and (2). Then AS1-AS5 might have been proved as theorems. This requires an extension of the *splitting set theorem* [7]. The details have been worked out in [2] and are omitted here due to space limitations.

Theorem 2 (Strong vs. Weak semantics). *Let P be a constrained program in which m-atoms never occur in the scope of negation. For each weak answer set M of P, there exists a strong answer set M' of P such that $M' \subseteq M$ and $M \setminus M'$ is a set of m-atoms.*

Note that the assumption on negative m-atoms is satisfied by our running example.

Corollary 1. *Under the hypothesis of Theorem 2, the strong answer sets of P are the minimal weak answer sets of P.*

Corollary 2. *Under the hypothesis of Theorem 2, the strong and weak skeptical semantics of P (i.e., the intersection of the strong, resp. weak answer sets) coincide.*

In the light of the above corollaries, we shall focus on the strong semantics, which is a way of computing a "representative" class of answer sets.

4 Computing strong answer sets

In this section we introduce a nondeterministic algorithm for computing strong answer sets. The actual implementation used in the experiments is derived from the nondeterministic algorithm by adding backtracking. The algorithm we introduce can be applied to constrained programs where mixed predicates have only positive occurrences. More general approaches require further work (cf. Section 7).

Our algorithm computes *strong kernels*, that is, compact representations of a (potentially large) set of strong answer sets.

Definition 2.

1. *A* strong completion *of a set of ground atoms I is a set $I \cup J$ such that:*
   - *J is a set of ground m-atoms;*

   - *for each free m-atom $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$ and each vector of r-constants $\boldsymbol{a}$ of the appropriate length, $I \cup J$ contains exactly one instance of $A(\boldsymbol{a}, \boldsymbol{x}_c)$.*
2. *A* strong kernel *of a constrained program P is a set of ground atoms K with at least one strong completion, and such that all the strong completions of K are strong answer sets of P.*

In general, K is the intersection of exponentially many strong answer sets of P. Since *all* strong completions of K are strong answer sets, it is trivial to generate any particular answer set including K, given K itself.

The algorithm that integrates answer set solving and constraint solving is formulated in terms of a generic answer set solver and a generic constraint solver. The former, called ASGEN, takes as input a regular program P and a set of ground literals S. Intuitively, ASGEN is an incremental solver, and S is the previous partial attempt at constructing an answer set for P. The solver may either fail to further extend S to an answer set of P, or it may return a refined attempt S'. So we assume that ASGEN enjoys of following formal properties:

1. $\text{ASGEN}(P, S)$ returns either NULL or a set S' of ground literals consistent with P.
2. If $\text{ASGEN}(P, S)$ returns a set S' then $S \subset S'$.
3. If $\text{ASGEN}(P, S)$ returns a complete set S' then S' is an answer set of P; here, by *complete* we mean that each ground literal occurs in S', either positively or negatively.
4. ASGEN is nondeterministically complete, that is for each answer set S of P there exists an integer $n \geq 0$ s.t. at least one computation of $\text{ASGEN}^n(P, \emptyset)$ returns S.

As usually, when we write $\text{ASGEN}^n(P, \emptyset)$ we mean:

$$\text{ASGEN}^0(P, \emptyset) = \emptyset$$
$$\text{ASGEN}^n(P, \emptyset) = \text{ASGEN}(P, \text{ASGEN}^{n-1}(P, \emptyset)).$$

Note that this formulation is compatible with virtually any strategy for interleaving the answer set construction and constraint solving. Note also that as a special case, ASGEN may immediately return complete sets (upon success) like SMODELS.

The only requirements on the constraint solver are that it should be sound and nondeterministically complete for each set of c-clauses χ. In other words, all substitutions σ returned by the constraint solver should be solutions of χ (i.e., $\chi\sigma$ should be satisfiable), and for each solution σ of χ, there should be a computation that returns σ.

The constraint solver is applied to a partially evaluated version of the constraints. To specify the partial evaluation procedure we need some auxiliary notation.

For each constraint $c = \leftarrow B$, we denote by $\mathsf{reg}(c)$, $\mathsf{con}(c)$, and $\mathsf{mix}(c)$, respectively, the collections of regular, constrained and mixed literals belonging to B.

We say that a substitution γ is *r-grounding* iff γ replaces each r-variable with a ground r-term and leaves the other variables unchanged.

Definition 3. *The* partial r-evaluation *of a set of constraints C w.r.t. a set of ground literals S, denoted by* $\mathsf{PE}(C, S)$*, is defined by*

$$\mathsf{PE}(C, S) = \{(\leftarrow \mathsf{mix}(c), \mathsf{con}(c))\gamma \mid c \in C,\ \gamma \textit{ r-grounding, and } \mathsf{reg}(c)\gamma \subseteq S\}.$$

Note that the members of $\mathsf{PE}(C, S)$ contain no r-atoms and no r-variables, because the former have been simplified away and the latter have been replaced with r-constants. Note also that in this process some constraints may disappear, as $\mathsf{reg}(c)$ may match no literals in S. Intuitively, S is to be provided by the answer set solver.

The constraint processing algorithm applies to a *normalized* version of $\mathsf{PE}(C, S)$, denoted by $\mathsf{PE}^n(C, S)$, satisfying the following properties:

N1 No m-literal occurring in $\mathsf{PE}^n(C, S)$ contains two or more occurrences of the same variable;

Moreover, for all free m-atoms $A(\boldsymbol{x}_r, \boldsymbol{x}_c)$,

N2 If both $A(\boldsymbol{a}, \boldsymbol{y}_c)$ and $A(\boldsymbol{a}, \boldsymbol{z}_c)$ occur in $\mathsf{PE}^n(C, S)$, then $\boldsymbol{y}_c = \boldsymbol{z}_c$.
N3 If both $A(\boldsymbol{a}, \boldsymbol{y}_c)$ and $A(\boldsymbol{b}, \boldsymbol{z}_c)$ occur in $\mathsf{PE}^n(C, S)$ and $\boldsymbol{a} \neq \boldsymbol{b}$, then $\boldsymbol{y}_c$ and $\boldsymbol{z}_c$ have no variables in common.

Note that condition N2 is the opposite of the classic standardization apart approach. N2 and N3 together require the vectors of c-variables to be in one-to-one correspondence with the vectors of regular arguments. Condition N1 can be fulfilled by introducing equations $x_i = x_j$ in $\mathsf{con}(c)$ when needed. Condition N2 and N3 can be fulfilled by variable renaming.

Example 3. In the running example, whenever S contains the pair $o(a_1, 1)$, $o(a_i, 2)$, constraint (3) yields the partially evaluated constraint

$$\leftarrow \mathit{time}(1, T1), \mathit{time}(2, T2), |T2 - T1| < 3.$$

After normalization, and assuming this particular constraint has not been modified, for all the atoms $\mathit{time}(1, x)$ occurring in $\mathsf{PE}^n(C(P), S)$, we have $x = T1$. In this way—roughly speaking—any solution to the constraints is forced to fulfil the property (2) of strong semantics.

We are now ready to prove soundness and completeness for Algorithm 1.

Theorem 3. *If a non-failed run of Algorithm 1 returns a set of literals K, then K is a strong kernel of P.*

Theorem 4. *For each strong answer set M of P there exists a run of Algorithm 1 that returns a strong kernel $K \subseteq M$.*

5 The CASP prototype

The CASP prototype is a simplified implementation of Algorithm 1, based on the answer set solver SMODELS [8]. CASP is meant to be an exploratory prototype, built with off-the-shelf components. While this strategy accelerated prototype deployment, it prevented us from exploiting the potential interleaving of answer set solving and constraint solving, supported by Algorithm 1. In this first prototype, the answer set solver always returns a complete answer set, so the loop in Algorithm 1 makes always one iteration.

Algorithm 1
CASPSOLVER (P)

1: **Inputs:** $P = R(P) \cup C(P)$: a constrained program with no negative m-literals.
2: **Outputs:** either a strong kernel of P or FAIL
3: **begin**
4: $S := \emptyset$;
5: **loop**
6: $S :=$ ASGEN($R(P), S$);
7: **if** $S =$ NULL **then**
8: FAIL;
9: **else**
10: $C := \mathsf{PE}^n(C(P), S)$;
11: **if** $\bigwedge_{c \in C} \neg\mathsf{con}(c)$ has no solution **then**
12: FAIL;
13: **else if** S is complete **then**
14: **choose** a solution σ of $\bigwedge_{c \in C} \neg\mathsf{con}(c)$;
15: Let $M(C)$ be the set of mixed literals in C;
16: **return** $S \cup M(C)\sigma$;
17: **end**

Let P be the input program. When P has a strong answer set, CASP returns a strong kernel for P, plus auxiliary information useful for analyzing the behavior of the system including the number of atoms, conjunctions, disjunctions, and variables occurring in $\bigwedge_{c \in C} \neg\mathsf{con}(c)$.

CASP consists of a script CASPSCRIPT that first runs the answer set solver on $R(P)$. Then for each answer set S of $R(P)$, CASPSCRIPT calls a GNU Prolog constraint logic program with finite domains, that implements steps 10-16 of Algorithm 1. In case of failure (step 12), CASPSCRIPT does not always fail; if $R(P)$ has more stable models, CASPSCRIPT feeds the next one to the Prolog module.

The finite domain (FD) constraint solver of GNU Prolog is an instance of the Constraint Logic Programming scheme introduced by Jaffar and Lassez in 1987 [6] and is based on the CLP(FD) framework [4]. Constraints are defined on FD variables and solved by means of arc-consistency (AC) techniques [12]. Arc consistency is not a complete inference mechanism; it ensures only that all solutions (if any) are in the current variable domains. In general, some variable assignments over the current domains are not solutions. Therefore, a final solution generation and checking phase is needed. In many cases, though, the domains produced by arc consistency are tight enough to speed up significantly the computation of solutions.

6 Experimental Results

We experimented with a few variants of the constrained program illustrated in the examples. Of course, this can only be regarded as a preliminary evaluation. Still, the example we choose is of significant interest. Programs similar to our running example have been used in the USA Advisor project, related to NASA missions [1, 9], and for protocol verification [3]. In both cases memory requirements happened to cause problems.

We did not insist much on the performance of the answer set solver, because there exists a rich body of literature on experimental evaluations and benchmarking of SMODELS. We focused on the performance of the constraint solver as $\bigwedge_{c \in C} \neg \mathsf{con}(c)$ and the number of disjunctions occurring in it grow.

The tests have been run on a Pentium(R) M processor 1.5GHz, with 1Mb cache and 512Mb core memory.

Fig. 1. SMODELS program

Regular part :

```
step(0..1).
action(1..2).
1{o(A,S) : action(A)}1 :- step(S).
```

Costrained part :

```
time(0..600000).
1{time(S,T) : time(T)}1 :- step(S).
:- o(A1,S1), time(S1,T1), o(A2,S2), time(S2,T2), abs(T1 - T2) < 3, neq(S1,S2),
   time(T1), time(T2), step(S1), step(S2), action(A1), action(A2).
:- time(S1,T1), time(S2,T2), S1 < S2, T1 >= T2, time(T1), time(T2), step(S1), step(S2).
:- time(S,T1), time(S,T2), neq(T1,T2), time(T1), time(T2), step(S).
```

Recall that the example has two regular sorts, *action* and *step*, and one constrained sort *time*. We started by encoding the planning and scheduling problem as the SMODELS program with weight constraints [10, 11] in Figure 1. In particular, the implicit semantics of mixed predicates has been encoded with the weight constraint

$$1\{\mathtt{time(S,T)} : \mathtt{time(T)}\}1 :- \mathtt{step(S)}. \tag{4}$$

This constraint says that for all steps S there exists exactly one time point T satisfying $\mathtt{time}(S,T)$.

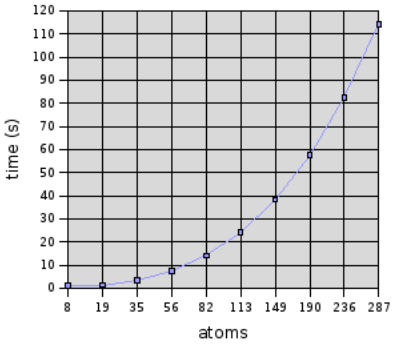
Sort *time* is the interval of integers $[0 - 600000]$. These values are determined by the following requirement: scheduling should cover plans at least one week long with the granularity of seconds.

With 2 actions and 2 steps, the front-end of SMODELS (*lparse*), responsible of the ground instantiation of the program and its simplification, did not terminate within 95 minutes and was killed (the main reasoning process was never reached). On the same program (without weight constraints, which are implicit in the strong semantics) CASP solves up to 10 steps in about 30 seconds. If the time domain is increased to 6 million points, then *lparse* crashes (probably because of exceeding memory needs), while CASP solves up to 10 steps in less than 2 minutes.

The details of the experiment with 6 million time points are given in Figure 2. Column *step* represents the corresponding regular sort, the fields *atoms*, *var*, *conj*, and *disj*, respectively, show the number of atoms, variables, conjunctions and disjunctions of the formula $\bigwedge_{c \in C} \neg \mathsf{con}(c)$ fed to the constraint solver. Field *attempts* is related to the number of backtracks; it counts the number of stable models of the regular part fed into the Prolog module before the first strong kernel is found. Finally, column *Smodels* reports the time needed by Smodels to compute the stable models of the regular part, and column *time* shows the overall time needed to produce the first strong kernel.

Fig. 2. test-1 results

step	*atoms*	*var*	*conj*	*disj*	*attempts*	*Smodels*	*time*
{0,1}	8	2	5	2	1	0m0.016s	0m1.259s
{0,...,2}	19	3	12	6	1	0m0.007s	0m1.363s
{0,...,3}	35	4	22	12	1	0m0.007s	0m3.379s
{0,...,4}	56	5	35	20	1	0m0.012s	0m7.440s
{0,...,5}	82	6	51	30	1	0m0.092s	0m14.091s
{0,...,6}	113	7	70	42	1	0m0.112s	0m24.253s
{0,...,7}	149	8	92	56	1	0m0.165s	0m38.430s
{0,...,8}	190	9	117	72	1	0m0.356s	0m57.580s
{0,...,9}	236	10	145	90	1	0m0.758s	1m22.542s
{0,...,10}	287	11	176	110	1	0m1.309s	1m54.056s
SORT: action={1,2} – time={0,...,6.000.000}							



The results with 600,000 time points are reported in Figure 3. In this experiment constraints are trivial. Basically, they only assign a minimal length to each action execution, so they are always satisfiable, for all action sequences chosen by the answer set solver, and without any backtracking.

Now, if we make constraints more difficult by posing upper bounds on the entire plan execution (so that constraints cannot be trivially satisfied and some backtracking is needed), we obtain the results illustrated in Figure 4. The time needed for constraint solving significantly increases. In future work, it will be interesting to explore different constraint solution strategies on a wider selection of examples.

7 Conclusions

Preliminary experimental results show that the integration of answer set programming and constraint solving techniques may significantly enhance the applicability range of ASP. A simple planning and scheduling problem can be naturally formulated and solved, while one of the most powerful state-of-the-art answer set solvers cannot even reach the main reasoning phase. Our method shares with constraint logic programming frameworks the ability of returning answers that may be compact representations of

Fig. 3. test-2 results

step	*atoms*	*var*	*conj*	*disj*	*attempts*	*Smodels*	*time*
{0,1}	12	2	9	2	1	0.011s	0.306s
{0,...,2}	28	3	21	6	1	0.019s	0.579s
{0,...,3}	51	4	38	12	1	0.009s	1.157s
{0,...,4}	81	5	60	20	1	0.012s	1.654s
{0,...,5}	118	6	87	30	1	0.020s	3.416s
{0,...,6}	162	7	119	42	1	0.039s	6.184s
{0,...,7}	213	8	156	56	1	0.166s	10.175s
{0,...,8}	271	9	198	72	1	0.364s	15.811s
{0,...,9}	336	10	245	90	1	0.461s	23.151s
{0,...,10}	408	11	297	110	1	1.308s	32.949s
SORT: action={1,2} -- time={0,...,600.000}							

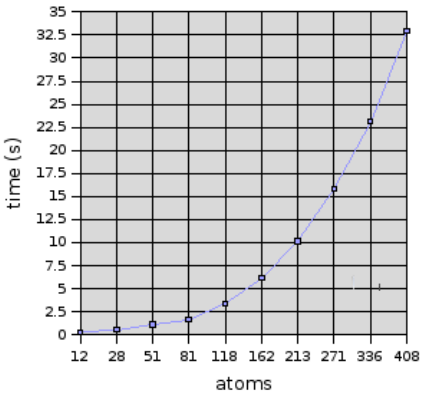


Fig. 4. test-3 results

step	*atoms*	*var*	*conj*	*disj*	*attempts*	*Smodels*	*time*
{0,1}	10	2	8	1	1	0m0.097s	0m0.216s
{0,...,2}	22	3	18	3	1	0m0.053s	0m0.117s
{0,...,3}	39	4	32	6	1	0m0.057s	0m0.131s
{0,...,4}	61	5	50	10	1	0m0.070s	0m0.181s
{0,...,5}	88	6	72	15	1	0m0.098s	0m0.209s
{0,...,6}	120	7	98	21	1	0m0.203s	0m0.245s
{0,...,7}	157	8	128	28	2	0m0.171s	0m45.349s
{0,...,8}	199	9	162	36	3	0m0.364s	3m54.919s
{0,...,9}	246	10	200	45	8	0m0.556s	35m11.777s
SORT: action={1,2} -- time={0,...,600.000}							

exponentially many distinct problem solutions, each of which can be easily extracted from the answer.

This work can be extended along several directions. First of all we are looking for more classes of examples of practical interest to extend our experimentation.

A second line of research concerns the interplay of the two solvers. A tighter integration of answer set generation and constraint solving may anticipate inconsistency detection, thereby improving failure handling. It would be interesting to explore dependency-directed forms of backtracking. Such a refined system should be compared through benchmarking to planners and schedulers based on different logics and reasoning methods (for a collection of pointers to such approaches, see `http://www.aaai.org/AITopics/html/planning.html`).

We mentioned that constrained programs are basically a subclass of weight constraint programs. It may be possible to extend the class of weight constraints supported by our approach, e.g., by using different bounds (e.g., mixing weak and strong semantics), and by dropping the requirement that for all free m-atoms A and all vector of r-constants $\boldsymbol{a}$, answer sets must contain at least one instance of $A(\boldsymbol{a}, \boldsymbol{x}_c)$. Many of our results can be adapted under the assumption that for all distinct weight constraints $l_1\{A_1\}u_1$ and $l_2\{A_2\}u_2$ in a program, A_1 and A_2 are not unifiable.

Moreover, it would be nice to support negative mixed literals. Unfortunately, our approach cannot be easily adapted; the solutions we have explored so far require blind grounding over constrained domains, which is exactly what should be avoided.

Acknowledgments Work partially supported by the EU working group WASP (5FP), IST-2001-37004 and ARDA contract.

References

1. M. Balduccini, M. Gelfond, R. Watson, and M. Nogueira. The USA-Advisor: A case study in answer set planning. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001*, volume 2173 of *Lecture Notes in Computer Science*, pages 439–442. Springer, 2001.
2. S. Baselice. Integrazione di tecniche di Answer Set Programming e Constraint Solving. Tesi di laurea, Università degli studi di Napoli Federico II, Naples, Italy, October 2004.
3. L. Carlucci Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Trans. Comput. Logic*, 2(4):542–580, 2001.
4. P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
5. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th ICLP*, pages 1070–1080. MIT Press, 1988.
6. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–582, May/July 1994.
7. V. Lifschitz and H. Turner. Splitting a Logic Program. In *Proceedings of the 12th International Conference on Logic Programming, Kanagawa 1995*, MIT Press Series Logic Program, pages 581–595. MIT Press, 1995.
8. I. Niemelä and P. Simons. Smodels — an implementation of the stable model and well-founded semantics for normal lp. In *Logic Programming and Nonmonotonic Reasoning, 4th*

International Conference, LPNMR'97, Proceedings, volume 1265 of *LNCS*, pages 421–430. Springer, 1997.
9. M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog decision support system for the Space Shuttle. In *Practical Aspects of Declarative Languages, Third International Symposium, PADL 2001*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2001.
10. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.
11. T. Syrjänen. Cardinality constraint programs. In *JELIA*, pages 187–199, 2004.
12. C. Teng, P. Van Hentenryck, and Y. Deville. A generic arc-consistency algorithm and its specializations, June 11 1992.