Integrating Answer Set Programming and Constraint Logic Programming

Veena S. Mellarkod and Michael Gelfond and Yuanlin Zhang
{veena.s.mellarkod,mgelfond,yzhang}@cs.ttu.edu

Texas Tech University

Dedicated to Victor Marek on his 65th birthday

Abstract

We introduce a knowledge representation language $\mathcal{AC}(\mathcal{C})$ extending the syntax and semantics of ASP and CR-Prolog, give some examples of its use, and present an algorithm, $\mathcal{AC}solver$, for computing answer sets of $\mathcal{AC}(\mathcal{C})$ programs. The algorithm does not require full grounding of a program and combines "classical" ASP solving methods with constraint logic programming techniques and CR-Prolog based abduction. The $\mathcal{AC}(\mathcal{C})$ based approach often allows to solve problems which are impossible to solve by more traditional ASP solving techniques. We believe that further investigation of the language and development of more efficient and reliable solvers for its programs can help to substantially expand the domain of applicability of the answer set programming paradigm.

1 Introduction

The work presented in this paper is aimed at further development of declarative programming paradigm based on Answer Set Prolog (ASP) [19, 7] and its extensions. The language has roots in research on non-monotonic logic and semantics of default negation of Prolog (for more details, see [27]). An ASP program Π is a collection of rules of the form

$$l_1 \; or \; \dots or \; l_k \leftarrow l_{k+1}, \dots, l_n, not \; l_{n+1}, \dots, not \; l_m \tag{1}$$

where l's are *literals* (statements of the form $p(\bar{t})$ and $\neg p(\bar{t})$ over some signature Σ, where $\bar{t}$ denotes a vector of terms). The expression on the left hand side of $\leftarrow$ is called the *head* of the rule; that on the right hand side is called the rule's *body*. Note that both the body and the head of the rule can be empty. If the body of a rule is empty then the $\leftarrow$ is omitted and the rule is referred to as a *fact*. Often we will use rules with empty heads which are called *denials*. Such a rule, $\leftarrow body$, is viewed as a shorthand for a rule $h \leftarrow body, not \; h$ where h is an atom not occurring anywhere else in the program. Connectives *or* and *not* are referred to as *epistemic disjunction* and *default negation* respectively; $\neg$ is often referred to as *classical* or *strong* negation. An ASP program Π can be viewed as a specification for the sets of beliefs to be held by a rational reasoner associated with Π. Such sets, called *answer sets* of Π, are represented by collections of ground literals (i.e., literals containing no variables). A rule (1) is viewed as a constraint which says that if literals $l_{k+1}, \dots, l_n$ belong to an answer set A of Π and none of the literals $l_{n+1}, \dots, l_m$ belong to A then A must contain at least one of the literals $l_1, \dots, l_k$. To form answer sets of Π, the reasoner must satisfy Π's rules together with the *rationality principle* which says: "*Believe nothing you are not forced to believe.*"

Given a computational problem P, an ASP programmer

- Expresses information relevant to the problem in the language of ASP;
- Reduces P to a query Q requesting computation of (parts of) answer sets of Π;
- Uses inference engine, i.e., a collection of reasoning algorithms, to solve Q.

There is a number of inference engines currently available to an ASP programmer. If the corresponding program does not contain disjunction, classical negation or rules with empty heads and is acyclic [1], i.e., only allows naturally terminating recursion, then the classical SLDNF-resolution of Prolog [13] and its variants [12] or fix-point computations of deductive databases (possibly augmented by constraint solving algorithms as in [42, 21, 30]) can be used to answer the query Q. Presently, there are multiple applications of solving various computational problems using these methods. In the last decade we have witnessed the coming of age of inference engines aimed at computing the answer sets of Answer Set Prolog programs [34, 35, 23, 15, 17, 20]. These engines are often referred to as *answer set solvers*. Normally they start their work with grounding the program, i.e., instantiating its variables by ground terms. The resulting program has the same answer sets as the original one but is essentially propositional. The grounding techniques employed by answer set solvers are rather sophisticated. Among other things they utilize algorithms from deductive databases, and require a good understanding of the relationship between various semantics of logic programming. The answer sets of the grounded program are often computed using substantially modified and expanded satisfiability checking algorithms. Another approach reduces the computation of answer sets to (possibly multiple) calls to existing satisfiability solvers [2, 20, 25].

The programming methodology based on the use of ASP solvers was originally advocated in [28, 33]. It proved to be useful for finding solutions to a variety of programming tasks, ranging from building decision support systems for the Space Shuttle [6] and product configuration [39], to solving problems arising in bio-informatics [8], zoology and linguistics [10]. Though positive, this experience allowed to identify a number of problems and inadequacies of the ASP approach to declarative programming.

First it became clear that for a number of tasks which require the use of ASP solvers these solvers are not sufficiently efficient. This becomes immediately obvious if the program contains variables ranging over large domains. Even though ASP solvers use intelligent grounding optimization techniques, ground instantiations of such a program can still be huge, which can cause both memory and time problems and make ASP solvers practically useless. A partial solution to this problem is suggested in [9, 32] where the language of ASP and its reasoning mechanism were extended to partially avoid grounding of variables ranging over the large domains and to replace such grounding with the use of constraint solving techniques. Specifically, [9] introduces the syntax and semantics of such a language and gives a simple algorithm for computing answer sets of its programs. A substantially more efficient incremental algorithm combining ASP with a specific constraint domain of difference constraints is proposed and implemented in [32]. This work substantially expands the scope of applicability of the ASP paradigm. In this paper we further expand this work by designing a more powerful extension $\mathcal{AC}(\mathcal{C})$ of ASP, which combines ASP with Constraint Logic Programming (CLP). We also give an algorithm, $\mathcal{AC}solver$, for computing answer sets of programs in the new language. The algorithm combines "classical" ASP solving methods with constraint satisfaction techniques and SLDNF resolution. We are currently working on the solver implementing this algorithm.

The second difficulty of using ASP for a number of applications was related to insufficient expressive power of the language. For instance, in a typical diagnostic task one often needs to explain unusual behavior of a system manifested by the incoherence of an ASP program encoding its normal behavior. This requires the ability to naturally mix the computation of answer sets of a program with some form of abductive reasoning. We were not able to find a way to utilize the existing abductive logic programming systems for such tasks, and opted for an introduction of a new language, CR-Prolog [5, 4], which is capable of expressing rare events that are ignored during a normal computation and only used if needed to restore coherence of the program. Consider, for instance, a program Π_0 consisting of regular ASP rules

$\neg p \leftarrow not\ p$
$q \leftarrow \neg p$

which say that p is normally believed to be false, and that if p is believed to be false then q must be believed to be true. The program has a unique answer set $\{\neg p, q\}$. Now let us expand Π_0 by a coherence restoring rule (cr-rule)

$p \overset{+}{\leftarrow}$

which says that p is possible but so rare that it can be ignored during the reasoning process unless it is needed for restoring coherence. The resulting program Π_1 still has one answer set $\{\neg p, q\}$. The cr-rule above remains unused. The situation changes if we expand Π_1 by a new fact

$\neg q$

Since regular rules of the new program Π_2 are incoherent, i.e., the program consisting of these rules has no answer set, the reasoner associated with the program is forced to use the cr-rule. The resulting answer set is $\{p, \neg q\}$.

The expressive power and reasoning ability of CR-Prolog proved to be useful in many situations beyond diagnostic reasoning. CR-Prolog was also successfully used in planning to produce higher quality plans than regular ASP [3], for reasoning about intentions, reasoning with weak constraints [11] a la DLV, etc. So we expand $\mathcal{AC}(\mathcal{C})$ by cr-rules and show an example of their use adding CR-Prolog abduction to the plethora of reasoning techniques discussed above.

In the next section of the paper we define the syntax and semantics of our language, $\mathcal{AC}(\mathcal{C})$. Section 3 contains an algorithm for computing answer sets of a large subclass of $\mathcal{AC}(\mathcal{C})$ programs and the corresponding soundness results. (The proofs of the theorems can be found in the Appendix.) Section 4 contains examples illustrating the methodology of knowledge representation and reasoning in $\mathcal{AC}(\mathcal{C})$. We end by a short conclusion.

2 Syntax and Semantics of $\mathcal{AC}(\mathcal{C})$

2.1 Answer Set Prolog

Recall that terms, literals, and rules of program Π with signature Σ are called *ground* if they contain no variables and no symbols for arithmetic functions. A program is called *ground* if all its rules are ground. In this section we briefly review the semantics of ground programs of Answer Set Prolog.

Consistent sets of ground literals over Σ, containing all ground arithmetic literals which are true under the standard interpretation of their symbols, are called *partial interpretations* of Σ. Expressions l and $not\ l$ where l is a literal are called *extended literals* (e-literals). We say that l is *true* in a partial interpretation S if $l \in S$; $not\ l$ is *true* in S if $l \notin S$; disjunction $(l_1\ or\ \ldots or\ l_k)$ is *true* in S if at least one of its members is *true* in S; S *satisfies a logic programming rule* (1) if the head of the rule is true in S or at least one extended literal of the rule's body is not true in S.

The answer set semantics [18] of a logic program Π with signature Σ assigns to Π a collection of *answer sets* – partial interpretations of Σ corresponding to the possible sets of beliefs which can be built by a rational reasoner on the basis of the rules of Π and the rationality principle. The precise definition of answer sets will be first given for ground programs whose rules do not contain default negation. Let Π be such a program and let S be a partial interpretation of signature Σ of Π.

Definition 1 [*Answer set – part one*]
A partial interpretation S of Σ is an *answer set* of Π if S is minimal (in the sense of set-theoretic inclusion) among the partial interpretations of Σ satisfying the rules of Π.

(Note that the rationality principle is captured in this definition by the minimality requirement.)

To extend the definition of answer sets to arbitrary programs, take any program Π, and let S be a partial interpretation of its signature Σ. The *reduct*, Π^S, of Π relative to S is the set of rules

$$l_1 \ or \ \ldots or \ l_k \leftarrow l_{k+1}, \ldots, l_m$$

for all rules (1) in Π such that $\{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$. Thus Π^S is a program without default negation.

Definition 2 [*Answer set – part two*]
A partial interpretation S of Σ is an answer set of Π if S is an answer set of Π^S.

(Here the rationality principle is captured by the fix-point condition above.) A program is called *coherent* if it has an answer set. In what follows we refer to answer sets defined by Definitions 1 and 2 as *ASP answer sets.*

2.2 The language $\mathcal{AC}(\mathcal{C})$

Now we will describe the syntax and informal semantics of the language $\mathcal{AC}(\mathcal{C})$. First let us recall some necessary terminology.

By *sort* we mean a non-empty countable collection of strings over some fixed alphabet. Strings of sort S_i will be referred to as *object constants* of S_i. A *sorted signature*, Σ, is a collection of sorts, properly typed predicate and function symbols, and variables. Each variable, X, takes on values from a unique sort denoted by $sort(X)$. When needed we assume that Σ contains standard numerical sorts of natural numbers, integers, rational numbers, etc. as well as standard numerical functions and relations such as $+$, $-$, $>$, $<$

etc. *Terms*, *literals*, and *extended literals* of Σ are defined as usual. *Rules* of the language, which will be defined below, are similar to rules of CR-Prolog and may contain variables. In the standard ASP/CR-Prolog semantics a rule with variables is viewed as a shorthand for a collection of its ground instantiations. The $\mathcal{AC}(\mathcal{C})$ interpretation of rules with variables is different and allows the construction of solvers which will not require a complete grounding of the program. To achieve this goal we first expand the language of ASP by

- Dividing sorts of Σ into *regular* and *constraint*. Constraint sorts will be declared by an expression $\#csort$. For instance, a sort $time = \{0..1000\}$ of integers between 0 and 1000 can be declared to be a constraint sort by statement

  $\#csort(time)$.

  Intuitively a sort is declared to be a constraint sort if it is a large (often numerical) set with primitive constraint relations from $\mathcal{C}$ (e.g., $\leq$) defined among its elements. Grounding *constraint variables*, i.e., variables ranging over constraint sorts, would normally lead to huge grounded program. This is exactly what should be avoided by the $\mathcal{AC}(\mathcal{C})$ solvers. The $\mathcal{AC}(\mathcal{C})$ solvers will only ground variables ranging over regular sorts (*regular variables*).

- Dividing predicates of the language into four types: *regular*, *constraint*, *defined* and *mixed*. Regular predicates denote relations among objects of regular sorts; constraint predicates denote primitive numerical relations among objects of constraint sorts; defined predicates are defined in terms of constraint, regular, and defined predicates; mixed predicates denote relations between objects which belong to regular sorts and those which belong to constraint sorts. Mixed predicates are not defined by the rules of the program and are similar to abducible relations of abductive logic programming.

  Consider for instance a regular sort $step = \{0..30\}$ used to denote steps of a trajectory of some acting agent, and a constraint sort $time = \{0..1000\}$ used to denote actual time (say in minutes). The relation $at(S, T)$ holds iff step S of the trajectory is executed at time T. It is a typical example of mixed relation. The corresponding declaration of this relation will be given by statement

$\#mixed\ \ at(step, time)$.

Without loss of generality, we will assume that in any mixed predicate m of Π's signature, constraint parameters follow regular parameters, i.e., every mixed atom formed by m can be written as $m(\bar{t}_r, \bar{t}_c)$ where $\bar{t}_r$ and $\bar{t}_c$ are the lists of regular and constraint terms respectively. According to our semantics a mixed predicate can be viewed as a function whose domain and range are collections of properly typed vectors of regular and constraint terms respectively. Hence $m(\bar{t}_r, \bar{t}_c)$ can be written as $m(\bar{t}_r) = \bar{t}_c$. If the range of m is boolean we write $m(\bar{t}_r)$ instead of $m(\bar{t}_r) = \mathit{true}$ and $\neg m(\bar{t}_r)$ instead of $m(\bar{t}_r) = \mathit{false}$.

Now let $acceptable\_time(T)$ be true iff time T belongs to the interval $[10, 20]$ or $[100, 120]$. It is natural to view this predicate as defined in terms of primitive constraint relation $\leq$. The corresponding declaration is as follows:

$\#defined\ \ acceptable\_time(time)$.

A predicate *occurs* has regular parameters and hence will be declared as

$\#regular\ \ occurs(action, step)$.

$\mathcal{AC}(\mathcal{C})$ does not require special declaration for the constraint predicates. They are to be specified in the parameter $\mathcal{C}$ of the language. A literal formed by a regular predicate will be called *regular literal.* Similarly for constraint, defined, and mixed literals.

Definition 3 [*Syntax of* $\mathcal{AC}(\mathcal{C})$]
A *standard* $\mathcal{AC}(\mathcal{C})$ *rule* over signature Σ is a statement of the form

$$h_1 or\ \ \ldots or\ \ h_k \leftarrow l_1, \ldots, l_m, not\ l_{m+1},\ \ \ldots, not\ l_n \tag{2}$$

such that

- if $k > 1$ then $h_1, \ldots, h_k$ are regular literals;
- if $k = 1$ then h_1 is a regular or defined literal[1]; and
- $l_1, \ldots, l_n$ are arbitrary literals of Σ.

[1] In what follows we use CLP tecniques to reason about defined literals. Since CLP does not allow disjunction in the heads of rules we prohibit such rules in our language.

A *coherence restoring rule* (cr-rule) of $\mathcal{AC}(\mathcal{C})$ is a statement of the form

$$r : l_1 \overset{+}{\leftarrow} l_1,\ \ldots, l_m,\ not\ \ l_{m+1}, \ldots, not\ l_n \tag{3}$$

where r is a term used to uniquely denote the name of the rule and l_i's are regular literals.

An $\mathcal{AC}(\mathcal{C})$ *program* Π consists of definitions of sorts of a signature Σ, declarations of variables - statements of the form $sort(V_1, \ldots, V_k) = sort\_name$, and a collection of standard and coherence restoring $\mathcal{AC}(\mathcal{C})$ rules over Σ.

Classification of literals of the signature Σ of an $\mathcal{AC}(\mathcal{C})$ program Π allows for partitioning Π into three parts:

- *regular part*, Π_r, consisting of rules built from regular literals,
- *defined part*, Π_d, consisting of rules whose heads are defined literals, and
- *middle part*, Π_m, consisting of all other rules of Π.

Elements of Π_r, Π_d and Π_m are called *regular rules*, *defined rules*, and *middle rules* respectively. Note that a standard (ground) ASP/CR-Prolog program Π can also be viewed as an $\mathcal{AC}(\mathcal{C})$ program all of whose predicates are defined as regular.

2.3 Semantics of $\mathcal{AC}(\mathcal{C})$

First we will need some terminology. Let R be a rule of an $\mathcal{AC}(\mathcal{C})$ program Π with signature Σ. A *ground instance* of R is obtained from R by

1. replacing variables of R by ground terms from the respective sorts; and
2. replacing all numerical terms by their values.

An ASP/CR-Prolog program $ground(\Pi)$ consisting of all ground instances of all rules in Π is called the *ground instantiation* of Π.

Definition 4 [*Partial interpretation*]
A consistent set S of ground literals over the signature Σ is called a *partial interpretation* of an $\mathcal{AC}(\mathcal{C})$ program Π if it satisfies the following conditions:

1. A constraint literal $l \in S$ iff l is true under the intended interpretation of its symbols;

2. For every mixed predicate $m(\bar{X}_r, \bar{Y}_c)$ and every ground instantiation $\bar{t}_r$ of $\bar{X}_r$, there is a unique ground instantiation $\bar{t}_c$ of $\bar{Y}_c$ such that $m(\bar{t}_r, \bar{t}_c) \in S$.

We first define the semantics for programs without cr-rules.

Definition 5 [*Answer sets of $\mathcal{AC}(\mathcal{C})$ programs without cr-rules*]
A partial interpretation S of the signature Σ of an $\mathcal{AC}(\mathcal{C})$ program Π is called an *answer set* of Π if there is a set M of ground mixed literals of Σ such that S is an answer set of the ASP program $ground(\Pi) \cup M$.

Now we give the semantics for programs with cr-rules. By $stand(\Pi)$ we denote the collection of standard rules of Π. By $\alpha(r)$, we denote a standard rule obtained from a cr-rule r by replacing $\overset{+}{\leftarrow}$ by $\leftarrow$. For a set $\mathcal{R}$ of cr-rules, $\alpha(\mathcal{R}) = \{\alpha(r) : r \in \mathcal{R}\}$. A minimal (with respect to set theoretic inclusion) collection $\mathcal{R}$ of cr-rules of Π such that $stand(\Pi) \cup \alpha(\mathcal{R})$ is coherent is called an *abductive support* of Π (see [5]).

Definition 6 [*Answer sets of arbitrary $\mathcal{AC}(\mathcal{C})$ programs*]
A set S is called an *answer set* of $\mathcal{AC}(\mathcal{C})$ program Π if it is an answer set of program $stand(\Pi) \cup \alpha(\mathcal{R})$ for some abductive support $\mathcal{R}$ of Π.

Let us illustrate the definition by the following example.

Example 1 [*$\mathcal{AC}(\mathcal{C})$ programs and their answer sets*]
Let P be an $\mathcal{AC}(\mathcal{C})$ program with sorts

time $= \{0..1000\}$.
step $= \{0..1\}$.
action $= \{a\}$.
fluent $= \{f\}$.

variable declarations

$sort(T) = time.$
$sort(S, S') = step.$

constraint relation $\leq$ defined on *time*, and declarations

$\#csort(time).$
$\#defined\ acceptable\_time(time).$
$\#mixed\ at(step, time).$
$\#regular\ occurs(action, step).$
$\#regular\ holds(fluent, step).$
$\#regular\ next(step, step).$

(To make our program executable with our implementations, we should use the *lparse* notation *time(0..1000)*, *step(0..1)*, *action(a)*, and *fluent(f)* for sorts, and *#domain time(T)* and *#domain step(S, S′)* for variable declarations.)

The program P also contains rules

$$\begin{array}{l}
acceptable\_time(T) \leftarrow 10 \leq T \leq 20. \\
acceptable\_time(T) \leftarrow 100 \leq T \leq 120. \\
\neg occurs(A, S) \leftarrow at(S, T), \\
\qquad\qquad\qquad\quad not\ acceptable\_time(T). \\
next(1, 0). \\
holds(f, S') \leftarrow occurs(a, S), \\
\qquad\qquad\quad\ \ next(S', S). \\
occurs(a, 0).
\end{array}$$

The first two rules comprise the defined part, P_d, of the program. Its middle part, P_m, consists of the third rule. The remaining rules form P's regular part, P_r.

Let $I = [10, 20] \cup [100, 120]$ and consider $t_0, t_1, t_2 \in time$ such that $t_0, t_1 \in I$ and $t_2 \notin I$. Let A_1 be a collection of atoms consisting of the specification of sorts, $step(0), step(1), action(a)$, etc, and atoms

$at(0, t_0), at(1, t_1),$
$next(1, 0), occurs(a, 0),$
$holds(f, 1),$
$acceptable\_time(t)$ for every $t \in I$.

Let $A_2 = (A_1 \setminus \{at(1, t_1)\}) \cup \{at(1, t_2), \neg occurs(a, 1)\}$. It is not difficult to check that A_1 and A_2 are answer sets of P.

Now let us consider a program P' obtained from P by replacing the rule

$$\begin{aligned} holds(f, S') \leftarrow\ & occurs(a, S), \\ & next(S', S). \end{aligned}$$

of P by

$$\begin{aligned} holds(f, S') \leftarrow\ & occurs(a, S), \\ & next(S', S), \\ & not\ ab(S, S'). \end{aligned}$$

and by adding rules

$$\neg holds(f, 1).$$

and

$$ab(S, S') \overset{+}{\leftarrow}.$$

It is not difficult to check that answer sets A'_1 and A'_2 of P' are obtained from answer sets A_1 and A_2 of P by replacing $holds(f, 1)$ by $\neg holds(f, 1)$ and $ab(0, 1)$.

3 Computing answer sets of $\mathcal{AC}(\mathcal{C})$ programs

We will need some terminology. To simplify the presentation we will, whenever necessary, identify an answer set A of a program Π with signature Σ with the set

$$A \cup \{not\ p : p \text{ is an atom of } \Sigma \text{ and } p \notin A\}.$$

Expression *not not l* will be identified with l. E-literals $p(\bar{t})$ and $not\ p(\bar{t})$ will be called *complementary*.

Definition 7 [*Query*]
A *query* is a set of defined and constraint e-literals. A ground set S of e-literals *satisfies* a query Q if there is a (sort respecting) substitution γ of variables of Σ by ground terms such that the result, $\gamma(Q)$, of this substitution is a subset of S. We will often refer to γ as a *solution* of Q w.r.t. S.

3.1 $\mathcal{AC}$*solver* – an algorithm for computing answer sets

In this section we present a (somewhat simplified) version of $\mathcal{AC}$*solver* – an algorithm for computing answer sets of a subclass of $\mathcal{AC}(\mathcal{C})$ programs.

Definition 8 [*Simple Programs*]
An $\mathcal{AC}(\mathcal{C})$ program Π is called *simple* if

1. Π contains no coherence restoring rules.
2. Π contains no disjunction.
3. Rules of Π with defined predicates in their heads contain neither mixed nor regular e-literals.
4. The defined part Π_d of Π has a unique answer set.

Restricting applicability of the algorithm to simple programs is made primarily to simplify the presentation. Expansion of the algorithm to programs with coherence restoring rules will be discussed at the end of this section. To remove the second condition one will need to use an ASP solver for disjunctive logic programs (or, whenever possible, to eliminate the disjunction). The third condition can be removed by modifying the corresponding constraint solver to allow proper treatment of global variables (values of the mixed predicates) and by calling constraint solver only when all the regular e-literals of the defined part, Π_d, of Π are assigned their values. (In this case we will also need an extra requirement prohibiting loops between regular predicates occurring in Π_r and Π_d). It is not clear to us if the fourth condition needs to be removed – constraint solvers normally assume acyclicity or other similar conditions which guarantee existence and uniqueness of the corresponding answer sets.

Definition 9 [*Canonical Program*]
We say that a simple $\mathcal{AC}(\mathcal{C})$ program Π is *canonical* if

1. Π contains no regular variables. (We refer to such programs as *r-ground.*)
2. Π contains no $\neg$.

3. Every mixed atom of Π has a form $m(\bar{t}_r, \bar{X})$ where $\bar{X}$ is a list of constraint variables.

4. If an atom occurs in the head of a middle rule it does not occur in the head of any other rule.

5. Negated mixed atoms are not allowed in the middle rules.

6. A middle rule of Π contains at most one occurrence of a defined atom and no occurrences of constraint atoms.

The restriction to canonical programs is much less severe. Indeed any simple $\mathcal{AC}(\mathcal{C})$ program Π can be, in a simple and natural way, reduced to its canonical form. Regular variables may be removed by a grounding process. The resulting program will be denoted by $gr(\Pi)$. Classical negation can be eliminated from $gr(\Pi)$ by viewing $\neg p$ as a new predicate symbol and adding constraints $\leftarrow p(\bar{t}), \neg p(\bar{t})$. Mixed atom $m(\bar{t}_r, t_c^1, \ldots, t_c^n)$ can be replaced by $m(\bar{t}_r, X_1, \ldots, X_n)$ and $X_1 = t_c^1, \ldots, X_n = t_c^n$. Middle rules $p(\bar{t}) \leftarrow B_1$ and $p(\bar{t}) \leftarrow B_2$ can be replaced by $p_1(\bar{t}) \leftarrow B_1$, $p_2(\bar{t}) \leftarrow B_2$, $p(\bar{t}) \leftarrow p_1(\bar{t})$ and $p(\bar{t}) \leftarrow p_2(\bar{t})$, where p_1 and p_2 are new regular predicate symbols. (Since only regular predicates can occur in the heads of middle rules the last two rules are regular.) Every occurrence of a negated mixed atom *not* $m(t_i, X_i)$ in the bodies of middle rules can be replaced by $m(t_i, Y_i), X_i \neq Y_i$.[2] Finally, a middle rule $p(\bar{t}) \leftarrow B_1, B_2$, where B_2 is the collection of the defined and constraint e-literals of the rule can be replaced by $d(\bar{X}) \leftarrow B_2$ and $p(\bar{t}) \leftarrow B_1, d(\bar{X})$. (Here $\bar{X}$ is the list of variables of B_2.)

$\mathcal{AC}$*solver* takes as an input a canonical $\mathcal{AC}(\mathcal{C})$ program Π, a set of ground regular e-literals B, and a query Q and returns an answer set A of Π which contains B and satisfies Q. If no such A exists the algorithm returns *false*.

[2]Consider an $\mathcal{AC}(\mathcal{C})$ program Π_1 and $\mathcal{AC}(\mathcal{C})$ program Π_2 obtained from Π_1 by this transformation. By Definition 5, S is an answer set of Π_1 iff S is an ASP answer set of $ground(\Pi_1) \cup M$ for some set M of ground mixed literals of Σ. By the Splitting Set Theorem, S is an answer set of $ground(\Pi_1) \cup M$ iff $S \setminus M$ is an answer set of $ground(\Pi_1)'$ which is the partial evaluation of $ground(\Pi_1)$ with respect to M. By our definition of partial interpretation there is exactly one x such that $m(t_i, x) \in M$. Now it is easy to check that $ground(\Pi_1)'$ is also the partial evaluation of $ground(\Pi_2) \cup M$ with respect to M, and thus Π_1 and Π_2 have exactly the same answer sets.

The computation starts with the derivation of the consequences of B with respect to program $\Pi_r \cup \Pi_m$. If the resulting set, B', is inconistent the algorithm returns *false*. Otherwise it collects all the middle rules whose regular literals are decided by B' and adds all the constraints that have to be made true to satisfy these rules to the query Q. For instance, if Π_m contains a rule $\leftarrow p(t), m(t,Y), d(Y)$ where $p(t)$ is a regular literal from B', $m(t,Y)$ is mixed and $d(Y)$ is defined then *not* $d(Y)$ is added to Q. After this is done, Q, together with Π_d is given as an input to a CLP solver. If a solution, say $\bar{x}$, is found, the corresponding literal $m(t,\bar{x})$ belongs to the answer set under construction while $d(\bar{x})$ does not. If no solution is found, the system backtracks. Often we limit ourselves to using CLP to simply check existence of a solution of Q with respect to Π_d and, if needed, do actual computation of the solution only once at the end of the algorithm.

The outline of the algorithm demonstrates the reason for dividing the original program Π into three parts. Essentially the first part Π_r, is used for reasoning with standard ASP algorithm. The middle part, Π_m serves to form a query to the third part, Π_d. In a sense, Π_m plays a role of the "bridge" between ASP and CLP parts of the algorithm. The query is answered by the CLP inference engine. The latter explains the prohibition of disjunction in the heads of Π_d. A substantial disjunctive information however can be handled by Π_r.

To make the idea above work we need to precisely define the computation of consequences of B with respect to $\Pi_r \cup \Pi_m$ and the formation of defined literals we add to query Q after this computation. This is done by two functions, Cn and *query* defined below[3].

Now we give a description of these functions.

We say that a set B of e-literals *falsifies* a set C of e-literals if C contains an e-literal complementary to some e-literal of B. The functions and the algorithm will be illustrated using program $gr(P)$ – the result of grounding the regular variables of program P from Example 1.

Program $gr(P)$

[3]An additional challange was presented by our desire to use intelligent grounding systems like *lparse* to partially ground the initial program. Such a grounding process is described and proven correct in [31].

1. *acceptable_time(T)* $\leftarrow 10 \leq T \leq 20$
2. *acceptable_time(T)* $\leftarrow 100 \leq T \leq 120$.
3. $\neg$*occurs(a,0)* $\leftarrow$ *at(0,T_0),*
   not acceptable_time(T_0).
4. $\neg$*occurs(a,1)* $\leftarrow$ *at(1,T_1),*
   not acceptable_time(T_1).
5. *occurs(a,0)*
6. *next(1, 0)*
7. *holds(f,0)* $\leftarrow$*occurs(a,0), next(0,0).*
8. *holds(f,1)* $\leftarrow$*occurs(a,0), next(1,0).*
9. *holds(f,0)* $\leftarrow$*occurs(a,1), next(0,1).*
10. *holds(f,1)* $\leftarrow$*occurs(a,1), next(1,1).*
11. $\leftarrow \neg$*occurs(a, 0), occurs(a, 0).*
12. $\leftarrow \neg$*occurs(a, 1), occurs(a, 1).*

(Note that the last two rules are the result of the elimination of classical negations).

Function *Cn*
Function $Cn(\Pi_r \cup \Pi_m, B)$ computes the consequences of a set of ground regular e-literals B under $\Pi_r \cup \Pi_m$. It is defined in terms of two auxiliary functions: *lower bound*, $lb(\Pi_r \cup \Pi_m, B)$, and *upper bound*, $ub(\Pi_r \cup \Pi_m, B)$. The former computes the minimal set X of e-literals containing B which is closed w.r.t. the following rules:

1. If $R \in \Pi_r$ and $body(R) \subseteq X$, then $head(R) \in X$.

2. If R is the only rule of $\Pi_r \cup \Pi_m$ whose body is not falsified by X and if $head(R) \in X$, then the regular e-literals of the body of R are in X.

3. If $(not\ l_0) \in X$, $(l_0 \leftarrow B_1, l, B_2) \in \Pi_r$, and $B_1, B_2 \subseteq X$, then $not\ l \in X$.

4. If there is no rule with head l_0, or the body of every rule of $\Pi_r \cup \Pi_m$ with head l_0 is falsified by X, then $not\ l_0 \in X$.

5. If X is inconsistent, X contains all e-literals.

Function $ub(\Pi_r \cup \Pi_m, B)$ returns the answer set X of a definite ground program $\alpha(\Pi_r \cup \Pi_m, B)$ obtained by

1. Removing all rules of $\Pi_r \cup \Pi_m$ whose bodies are falsified by B.
2. Removing any rule R of $\Pi_r \cup \Pi_m$ such that *not head*$(R) \in B$.
3. Removing all e-literals of the form *not* $p(\bar{t})$ from the rules of $\Pi_r \cup \Pi_m$.
4. Removing all defined and mixed e-literals from the rules of Π_m.

Now let

$$f_\Pi(B) = lb(\Pi_r \cup \Pi_m, B) \cup \{not\ p(\bar{t}) : p(\bar{t}) \notin ub(\Pi_r \cup \Pi_m, lb(\Pi_r \cup \Pi_m, B))\}.$$

The function $Cn(\Pi_r \cup \Pi_m, B)$ is defined as the least fixed point S of f_Π such that $B \subseteq S$. As usual, the existence of the fixpoint follows from the monotonicity of $f_\Pi(B)$. Moreover, $S = f_\Pi^{n-1}(B)$ for some number n.

One can check that $Cn(gr(P)_r \cup gr(P)_m, \emptyset)$ returns the set

$S_0 = \{$*next(1,0), holds(f,1), occurs(a,0), not next(0,0), not next(0,1), not next(1,1), not holds(f,0), not occurs(a,1), not* $\neg$*occurs(a,0)*$\}$.

Functions *Cn*, *lb* and *ub* can be viewed as slight modifications of functions *expand*, *Atleast* and *Atmost* of Smodels [38]. In fact for programs without defined and middle rules, our functions are exactly the same. In the general case our functions extend those of Smodels by taking into account the middle rules of Π.

Function *query*

Function *query* takes as an input a canonical program Π, a ground set of regular e-literals B and a query Q. It computes a collection of defined literals Q' such that Π has an answer set containing B and satisfying Q iff Π has an answer set containing B and satisfying $Q \cup Q'$. The function returns a new query, $Q \cup Q'$.

For simplicity of exposition, we limit ourselves to programs whose mixed predicates have one regular and one constraint parameter. We will need some terminology.

To every pair $\langle m, x\rangle$ where m is a mixed predicate and x is an element of the sort of its regular parameter, we assign a unique constraint variable V, called the *value variable* of $\langle m, x\rangle$.

An *r_ground* middle rule R is called *active* w.r.t. a set of ground regular e-literals B if B contains all the regular e-literals of the body of R. Program $pe(\Pi, B)$ is obtained from the middle part, Π_m, of Π by

1. Removing all rules which are not active w.r.t. B.
2. Removing every rule R such that $head(R) \notin B$ and $not\ head(R) \notin B$.
3. Removing regular e-literals from the bodies of all the remaining rules.
4. Renaming apart variables of the program to make sure that no variable occur in two rules of the program.
5. For every rule R containing a mixed literal $m(x, Y)$ replacing all occurrences of variable Y in R by the value variable V of $\langle m, x\rangle$.

For every rule $R \in pe(\Pi, B)$ we define a query $q(R)$ as follows:

- If $head(R) \in B$ and the body of R contains defined e-literal l then $q(R) = \{l\}$;
- If $head(R) \in B$ and the body of R contains no defined e-literal then $q(R) = \{true\}$;
- If $not\ head(R) \in B$ and the body of R contains defined e-literal l then $q(R) = \{\bar{l}\}$ where $\bar{l}$ is an e-literal complementary to l;
- If $not\ head(R) \in B$ and the body of R contains no defined e-literal then $q(R) = \{false\}$.

Symbols *true* and *false* above are constraint e-literals, say, $X = X$ and $X \neq X$ respectively.

Function $query(\Pi, B, Q)$ renames the variables of Q to make sure that no variable of Q occurs in the program $pe(\Pi, B)$ and returns

$$Q \cup \{q(R) : R \in pe(\Pi, B)\}.$$

Let us look at the program Π with $\Pi_m = \{\leftarrow p(t), m(t, Y), d(Y)\}$, $B = \{p(t)\}$, and $Q = \emptyset$. It is easy to see that, for these parameters, *query* returns $Q = \{not\ d(Y)\}$. If $\bar{x}$ is a solution of $not\ d(Y)$ with respect to Π_d then $\bar{x}$ will be used as the value of $m(t)$, i.e., $m(t, \bar{x})$ will belong to an answer set of Π

constructed by our program. If there is no such solution then the rule of Π_m cannot be satisfied by any answer set of Π containing B and satisfying the original query Q.

Let us illustrate the *query* algorithm by a more complete example in which $B = \{p1, not\, p2, p3, not\, p4\}$, $Q = \emptyset$, and the program Π with the middle part:

$p1 \leftarrow m1(t1, Y1), m(t2, Y2), d1(Y1, Y2).$
$p2 \leftarrow m3(t3, Y3), not\ d2(Y3).$
$p3 \leftarrow m4(t4, Y4), not\ d3(Y4).$
$\leftarrow m1(t1, Y5), d4(Y5).$
$p4 \leftarrow m1(t1, Y1).$

where p's are regular, m's are mixed and d's are defined predicates. The program $pe(\Pi, B)$ is

$R1: \quad p1 \leftarrow m1(t1, X1), m(t2, X2), d1(X1, X2).$
$R2: \quad p2 \leftarrow m3(t3, X3), not\ d2(X3).$
$R3: \quad p3 \leftarrow m4(t4, X4), not\ d3(X4).$
$R4: \quad \leftarrow m1(t1, X1), d4(X1).$
$R5: \quad p4 \leftarrow m1(t1, X1).$

where $X1, X2, X3, X4$ are the corresponding value variables.

Recall that since our program is canonical there are no other rules in Π with $p1$ in the head. Similarly for $p2$, $p3$, and $p4$. Since we are looking for an answer set of Π containing B and $p1 \in B$, we need to justify $p1$. This can be done only by finding a solution of constraint $d1(X1, X2)$. To justify $(not\, p2) \in B$ we need to find $X3$ such that $d2(X3)$. The justification of $p3 \in B$ is given by a solution of constraint $not\ d3(X4)$. To satisfy rule $R4$, $X1$ must be a solution of $not\ d4(X1)$. Since according to our definition of answer set, any answer set of the program should contain $m1(t1, x1)$ for some $x1$, the last rule is not satisfiable. According to our definition,

$q(R1) = \{d1(X1, X2)\}$,
$q(R2) = \{d2(X3)\}$,
$q(R3) = \{not\ d3(X4)\}$,
$q(R4) = \{not\ d4(X1)\}$,
$q(R5) = \{false\}$.

Not surprisingly, function *query* returns a constraint

$$d1(X1, X2) \wedge d2(X3) \wedge \mathit{not}\ d3(X4) \wedge \mathit{not}\ d4(X1) \wedge \mathit{false}$$

where the variables range over their respective sorts, and $\wedge$ is used in place of comma.

Now let us consider the program $gr(P)$ defined above and the set S_0 returned by $Cn(gr(P)_r \cup gr(P)_m, \emptyset)$, and compute $query(gr(P), S_0, \emptyset)$. The program has two middle rules (3) and (4) which are both active w.r.t. S_0. Since neither $\neg occurs(a, 1)$ nor $\mathit{not}\ \neg occurs(a, 1)$ is in S_0, function $pe(gr(\Pi)_m, S_0)$ will return

$$\begin{array}{lcl} \neg occurs(a, 0) & \leftarrow & at(0, T_0), \\ & & \mathit{not}\ acceptable\_time(T_0). \end{array}$$

Since $(\mathit{not}\ \neg occurs(a, 0)) \in S_0$, function *query* will return

$$acceptable\_time(T_0).$$

Function *c_solve*
We will need the following definition.

Definition 10 [*Constraint Program*]
A *constraint program* is a collection, Π, of defined rules formed by defined e-literals and primitive constraints such that Π has a unique answer set.

Function $c\_solve(\Pi, Q)$ takes as an input a constraint program Π and a query Q. The function returns a pair $\langle C, true\rangle$ where C is a consistent set of primitive constraints, such that for any solution γ of C, γ is a solution of Q w.r.t. the answer set of Π. If no such C exists the function returns *false*. (Note that when both Π and Q are empty, $c\_solve(\Pi, Q)$ returns $\langle \emptyset, true\rangle$).

For instance, $c\_solve(gr(\Pi)_d, acceptable\_time(T_0))$ may return $\langle C, true\rangle$ where $C = 10 \leq T_0 \leq 20$.

In what follows, we will assume the existence of such a function. There are, of course, many practical systems which implement such functions for various classes of queries and constraint programs [42, 21, 22, 37].

Function $\mathcal{AC}solver$
Now we are ready to present the main function (see Figure 1), $\mathcal{AC}solver$, which computes answer sets of canonical programs. If no such answer set exists $\mathcal{AC}solver$ returns *false*. Note that, since Π is a canonical program and thus a simple program, Π_d from line 3 of the algorithm is a constraint program by the fourth condition of the definition of simple programs. Hence it can be used as an input of our constraint solver, *c_solve*.

```
function $\mathcal{AC}solver$
  Input
      $\Pi$: $r$-ground canonical program and $\Pi_d$ is a constraint program;
      $B$: set of ground regular e-literals;
      $Q$: query;
  Output
      $false$ if no answer set of $\Pi$ contains $B$; $\langle A, C\rangle$, otherwise. Here, $A$ is
      a set of ground regular e-literals and $C$ is a consistent set of primitive
      constraints such that $A$ is the regular part of an answer set $W$ of $\Pi$
      containing $B$, and any solution $\gamma$ of $C$ is a solution of $Q$ w.r.t $W$.
begin
1.    $S := Cn(\Pi_r \cup \Pi_m, B)$;
2.    if $S$ is inconsistent return $false$;
3.    $O := c\_solve(\Pi_d, query(\Pi, S, Q))$;
4.    if $O = false$ then return $false$;
5.    if $O = \langle C, true\rangle$ and $p \in S$ or $not\ p \in S$ for any regular atom $p$ then
6.          return $\langle S, C\rangle$;
7.    pick a regular e-literal $l$ such that $l, (not\ l) \notin S$;
8.    $O := \mathcal{AC}solver(\Pi, S \cup \{l\}, Q)$ ;
9.    if $O = false$ return $\mathcal{AC}solver(\Pi, S \cup \{not\ l\}, Q)$;
10.   else return $O$;
end
```

Figure 1: The $\mathcal{AC}solver$ algorithm

Let us trace $\mathcal{AC}solver(gr(P), \emptyset, \emptyset)$. In the following, T_0 and T_1 are value variables of $\langle at, 0\rangle$ and $\langle at, 1\rangle$ respectively. We already computed the value S_0 of $Cn(gr(P)_r \cup gr(P)_m, \emptyset)$. S_0 is consistent, and $query(gr(P), S_0, \emptyset)$ returns $Q_0 = acceptable\_time(T_0)$. Function $c\_solve(gr(P)_d, Q_0)$ may return $\langle C_0, true\rangle$ with $C_0 = 10 \leq T_0 \wedge T_0 \leq 20$. For every t_0 satisfying C_0, atom $acceptable\_time(t_0)$ belongs to an answer set of $gr(P)$ compatible with S_0. The only regular atom undecided by S_0 is $\neg occurs(a,1)$. Suppose that $\mathcal{AC}solver$ selects this atom and calls $\mathcal{AC}solver(P, S_1, \emptyset)$ where $S_1 = S_0 \cup \{\neg occurs(a,1)\}$. $Cn(gr(P)_r \cup gr(P)_m, S_1)$ returns $S_2 = S_1$. Function $query(gr(P), S_2)$ returns $Q_1 = acceptable\_time(T_0) \wedge not\ acceptable\_time(T_1)$. The possible answer returned by $c\_solve(gr(P)_d, Q_1)$ may be, say, $\langle C_1, true\rangle$ with $C_1 = 10 \leq T_0 \leq 20 \wedge T_1 < 10$. Finally, $\mathcal{AC}solver(gr(P), \emptyset, \emptyset)$ returns $\langle S_2, C_1\rangle$. (Of course, the solver can be slightly modified to return some of the C_1's solutions.) If $\mathcal{AC}solver$ were to select $not\ \neg occurs(a,1)$ instead of $\neg occurs(a,1)$, then the returned value would be $\langle S_3, C_2\rangle$ with $S_3 = S_0 \cup \{not\ \neg occurs(a,1)\}$ and C_2, say, being $10 \leq T_0 \leq 20 \wedge 10 \leq T_1 \leq 20$.

It is easy to check that the algorithm always terminates. But it is not always correct. Consider for instance a program P_{ns}

$\#csort(s).$
$\#defined\ d(s), e(s).$
$s(0..2).$
$p \leftarrow e(Y).$
$d(1).\ \ d(2).$
$e(Y) \leftarrow d(Y), Y < 2.$

Every answer set of this program contains $d(1)$, $d(2)$, and p. Our algorithm however may also return sets not containing p. This happens when the algorithm picks literal *not* p, and *c_solve* returns, say, $Y = 0$.

To ensure correctness of the algorithm we will require a program to satisfy

Safety condition for constraint variables of Π_m: *Every constraint variable occurring in a middle rule of the program should have an occurrence in a mixed predicate from this rule.*

A rule is *safe* if it satisfies the safety condition; *non-safe* otherwise. A program is *safe* if it contains only safe rules.

Obviously, the middle rule $p \leftarrow e(Y)$ in the example above is non-safe. We can however construct a safe program P_s equivalent to P_{ns} with respect to the literals of P_{ns}. The program P_s contains a new defined predicate symbol d_0 and the rules

$d_0 \leftarrow e(Y).$
$p \leftarrow d_0.$
$d(1).\ \ d(2).$
$e(Y) \leftarrow d(Y), Y < 2.$

This transformation is rather general – it can be expanded to an arbitrary canonical program as follows.

Let $R \in \Pi$ be a non-safe middle rule. Assume that R is of the form $l \leftarrow B, l_d(\bar{X}, \bar{Y})$ with the defined e-literal $l_d(\bar{X}, \bar{Y})$ where $\bar{Y}$ is the list of constraint variables of l_d not occurring in the middle atoms of R. Let $d_0(\bar{X})$ be a new predicate symbol. By a *safe variant* of R we mean two rules

$d_0(\bar{X}) \leftarrow l_d(\bar{X}, \bar{Y}).$
$l \leftarrow B, d_0(\bar{X}).$

It is easy to check that a program *safe(*Π*)* obtained from Π by replacing its non-safe rules by their safe variants is equivalent to Π modulo the newly introduced predicates.

Now we are ready to formulate soundness conditions of our algorithm. We will use the following notation. If A is a set of ground literals over the signature of program Π, by $c(A, \Pi)$ we denote the set

$$A \cup \{not\ p : p \text{ is a literal of } \Sigma(\Pi) \text{ and } p \notin A\}.$$

Let Π be a safe canonical program, Σ a signature of Π, B a set of e-literals, and Q a query.

Theorem 1 [*Soundness*]

1. If $\mathcal{AC}solver(\Pi, B, Q)$ returns $\langle S, C \rangle$ then there exists an answer set A of Π satisfying conditions:
   - (i) $c(A, \Pi)$ contains B,
   - (ii) for any regular literal p of Σ, $p \in A$ iff $p \in S$,
   - (iii) A satisfies Q.
2. If $\mathcal{AC}solver(\Pi, B, Q)$ returns *false*, there is no answer set A of Π that contains B and satisfies Q.

Finally let us mention that our $\mathcal{AC}solver$ algorithm can be easily *expanded* to programs with cr-rules whose standard part is a canonical program satisfying our safety condition. We refer to such programs as *cr-canonical.* To see how this can be done it is sufficient to recall that standard CR-Prolog solver – an algorithm for computing answer sets of CR-Prolog [4] – employs an answer set solver to check if the standard part of the program is coherent. If it is coherent, the algorithm returns a corresponding answer set. Otherwise, it "guesses" a minimal collection $\mathcal{R}$ of coherence restoring rules and checks if the coherence is indeed restored by making a call to the corresponding answer set solver with $stand(\Pi) \cup \alpha(\mathcal{R})$ as an input. Note that if Π is a cr-canonical $\mathcal{AC}(\mathcal{C})$ program then $stand(\Pi)$ and $stand(\Pi) \cup \alpha(\mathcal{R})$ are canonical $\mathcal{AC}(\mathcal{C})$ programs, and the expansion of $\mathcal{AC}solver$ to cr-canonical programs can be obtained from the corresponding CR-Prolog solver by replacing calls to ASP solvers by calls to $\mathcal{AC}solver$.

3.2 Implementations

In this section we briefly discuss our prototype implementations of $\mathcal{AC}(\mathcal{C})$ solvers for two subclasses of our language.

Both prototypes use a *grounder*, $\mathcal{P}ground_d$, which grounds the regular variables of $\mathcal{AC}(\mathcal{C})$ program Π and outputs an r-ground program $gr(\Pi)$. The implementation of $\mathcal{P}ground_d$ uses intelligent grounder *lparse* [41]. Therefore it is only applicable to programs which satisfy *lparse*'s safety conditions. To allow the use of *lparse* for partial grounding, we implemented intermediate transformations that remove constraint variables, mixed and constrained atoms from Π before *lparse* grounding and then restore them back after *lparse* grounding. They ensure that the constraint variables are not grounded and the rules containing mixed atoms are not removed by *lparse*. (Recall that since mixed atoms do not occur in the heads of rules of the program, *lparse* believes them to be false and acts accordingly).

An $\mathcal{AC}(\mathcal{C})$ program Π can serve as an input to our first implementation, called $\mathcal{AD}engine$, if

- Π contains no defined predicates;
- Middle rules of Π are denials;
- Constraints of Π are of the form $X - Y > K$;

- The body of a middle rule of Π contains exactly one constraint literal.

We refer to such programs as $\mathcal{AC}_0$ *programs*. Since the middle rules of $\mathcal{AC}_0$ programs contain constraint predicates, our $\mathcal{AC}$*solver* algorithm is not directly applicable to them. It can, however, be easily modified to accommodate $\mathcal{AC}_0$ programs. To do that we need to modify function *query* with $Q = \emptyset$ by replacing "defined literal l" in the definition of $q(R)$ by "constraint atom $X - Y > K$." Since the head of R is never true in B, $q(R) = \{not\ X - Y > K\}$ which can be written as $\{X - Y \leq K\}$. Hence the *query* will return a collection of *difference constraints* – inequalities of the form $X - Y \leq K$. Such constraints can be efficiently solved by a number of algorithms available in the literature. The $\mathcal{AD}$*engine* uses an incremental difference constraint algorithm in [36]. The treatment of coherence restoring rules of $\mathcal{AC}_0$ programs is based on the modification of the CR-Prolog solver built on top of the ASP solver Surya [31]. According to our general strategy, calls to Surya in this CR-Prolog solver are replaced by calls to $\mathcal{AD}$*engine* with inputs not containing coherence restoring rules. The corresponding implementation can be downloaded from

`http://www.cs.ttu.edu/~mellarko/adsolver.html.`

The second implementation, called $\mathcal{AC}$*engine*, is currently under development. It is based on the ASP inference engine Surya (developed at TTU) and the CLP system CLP(R) [21] with constructive negation [40]. Completion of the $\mathcal{AC}$*engine* and the experimental evaluation of both systems is the subject of ongoing work.

4 Representing Knowledge in $\mathcal{AC}(\mathcal{C})$

Several examples presented in this section are meant to illustrate the use of $\mathcal{AC}(\mathcal{C})$ for knowledge representation. None of the examples can successfully run with traditional ASP solvers while all of them are easily solvable by the $\mathcal{AC}(\mathcal{C})$ solvers discussed above. We start with a simple planning and scheduling example.

Example 2 [*Planning and Scheduling*]
John, who is currently at work, needs to be in his doctor's office in one hour carrying the insurance card and money to pay for the visit. The card is at

home and money can be obtained from the nearby ATM. John knows the minimum time (in minutes) needed to travel between the relevant locations. Can he find a plan to make it on time? (assuming of course that there will be no delays and the actual time of travel will be the minimum time). To solve the problem we will divide it into two parts: planning and scheduling. The solution of the former will use the standard ASP based methods. We use variables P for *people*, L for *locations*, O for *objects* (cash and the insurance card), and S for steps of the planned trajectory. We will need an action $go\_to(P, L)$ and fluents $at\_loc(P, L)$, $at\_loc(O, L)$, and $has(P, O)$ with self-explanatory intuitive meaning. (To simplify the solution we assume that person P automatically gets the object O when both, P and O, share the same location). The transition diagram whose states are collections of fluents describing possible physical states of the domain and arcs are labeled by actions is defined by causal laws written as logic programming rules. For instance, direct effects of the actions are described by the rule

holds(at_loc(P,L),S1) $\leftarrow$ *next(S1, S0), occurs(go_to(P,L),S0).*

Indirect effects will be captured by the corresponding relationships between fluents:

holds(has(P,O),S) $\leftarrow$ *holds(at_loc(P,L),S), holds(at_loc(O,L),S).*
holds(at_loc(O,L),S) $\leftarrow$ *holds(at_loc(P,L),S), holds(has(P,O),S).*
$\neg$*holds(at_loc(X,L2),S)* $\leftarrow$ *holds(at_loc(X,L1),S), L1* $\neq$ *L2.*

The problem of representing the unchanged fluents is solved by the inertia axioms (variable F is used for fluents):

holds(F,S1) $\leftarrow$ *next(S1,S0), holds(F,S0), not* $\neg$*holds(F,S1).*
$\neg$*holds(F,S1)* $\leftarrow$ *next(S1,S0),* $\neg$*holds(F,S0),not holds(F,S1).*

John's goal and his options will be described by the rules:

occurs(go_to(john,L),S) or $\neg$*occurs(go_to(john,L),S).*

goal(S) $\leftarrow$ *holds(at_loc(john,doctor),S),*
holds(has(john,card),S),
holds(has(john,cash),S).
succeed $\leftarrow$ *goal(S).*

$\leftarrow$ *not succeed.*

Let us denote by D_r^n the above program with initial conditions and sort *step* defined as a collection of integers from 0 to n. Normally ASP planning is performed by computing answer sets of D_r^n for $n = 1, 2, \ldots$. The desired plans can be easily extracted from the answer sets of the first coherent program D_r^k. A possible plan returned by this planning method can be, say, [*go_to(john,home), go_to(john,atm),go_to(john,doctor)*].

Now we concentrate on the scheduling part of the problem. Actual time will range from 0 to 1440 (number of minutes in 24 hours). The schedule should assign time T to each step S of the plan. This will be achieved by introducing a mixed relation $at(S, T)$ and specifying the necessary constraints, e.g.,

$$\begin{array}{ll} \leftarrow & next(S1, S0), \\ & at(S0, T0), \\ & at(S1, T1), \\ & T1 < T0. \end{array}$$

$$\begin{array}{ll} \leftarrow & goal(S), \\ & at(0, T1), \\ & at(S, T2), \\ & T2 - T1 > 60. \end{array}$$

$$\begin{array}{ll} \leftarrow & next(S1, S0), \\ & occurs(go\_to(john, home), S0), \\ & holds(at\_loc(john, office), S0), \\ & at(S0, T0), \\ & at(S1, T1), \\ & T0 - T1 > -20. \end{array}$$

The first rule requires time to be a monotonic function of steps; the second guarantees that the trip does not take more than an hour; the third assumes that the trip from office to home takes at least twenty minutes. Other constraints are added in a similar fashion. Let us denote the resulting program by D^n.

It is easy to check that D^n is an $\mathcal{AC}_0$-program satisfying the safety condition, and hence the program can be run as an input to $\mathcal{AD}engine$. The solver will (almost instantaneously) return an answer set of D^n, containing a plan, say

[*occurs(go_to(john,home),0)*,
occurs(*go_to*(*john*, *atm*), *1*),
occurs(go_to(john,doctor),2)]

and a schedule, say, $at(0, 0)$, $at(1, 20)$, $at(2, 35)$, $at(3, 55)$ for executing its actions. It is guaranteed that if John performs the corresponding actions as scheduled he will get to see his doctor on time. If the initial conditions were modified to ensure that no plan of length n satisfies the desired goal the program would have no answer set and $\mathcal{AD}$*engine* would return *false*.

In the next example we show how our language can express and reason with disjunctive temporal constraints.

Example 3 [*Disjunctive Temporal Constraints*]
Suppose we would like to schedule an action "*a*" such that it occurs either between 3am and 5am or between 7am and 8am. To represent this restriction, we would require a constraint of the form, "*if action "a" occurs at step S and step S occurs at time T, then T cannot be outside intervals [3-5] or [7-8]*". Because of the disjunction we cannot represent this directly in $\mathcal{AC}_0$. Instead we introduce two regular atoms int_1 and int_2. The atom int_1 denotes that "action "*a*" occurs in interval [3-5]". Similarly for int_2. Assuming that time steps range from 0 to some n this disjunctive temporal constraint can be represented by the following rules.

% action a occurs in interval int_1 or int_2.

int_1 *or* int_2.

% If a occurs at step S and int_1 is true, then S should be assigned the time from $[3 - 5]$.

$$\leftarrow int_1, occurs(a, S), at(0, T_1), at(S, T_2), T_1 - T_2 > -3$$
$$\leftarrow int_1, occurs(a, S), at(0, T_1), at(S, T_2), T_2 - T_1 > 5$$

% If a occurs at step S and int_2 is true, then S should be assigned the time from $[7 - 8]$.

$$\leftarrow int_2, occurs(a, S), at(0, T_1), at(S, T_2), T_1 - T_2 > -7$$
$$\leftarrow int_2, occurs(a, S), at(0, T_1), at(S, T_2), T_2 - T_1 > 8$$

For simplicity we can also assume that step 0 occurred at time 0 and that action a occurred at step 1. The former can be expressed by

$\leftarrow at(0,T), T > 0.$

The latter by

$occurs(a, 1).$

To eliminate disjunction we replace the first (disjunctive) rule by

$int_1 \leftarrow not\ int_2.$
$int_2 \leftarrow not\ int_1.$

The resulting program can run as an input to $\mathcal{AD}engine$ which will returns an answer set containing $\{occurs(a, 1),\ int_2,\ at(0, 0),\ at(1, 7)\}$, where a occurs at 7 am.

In our next example we illustrate the use of our language for representing weak (defeasible) constraints.

Example 4 [*Planning with Weak Constraints*]
Let us now consider a variant of the story from Example 2 in which the requirement "*the trip does not take more than an hour*" is replaced "*the trip does not take more than an hour, but John prefers to make it in 50 minutes*". Such requirements are often referred to as *weak constraints.*

The new information can be encoded by the defeasible rule which says that "*under normal circumstances the trip will be made in 50 minutes (or less)*".

$$\begin{array}{ll} \leftarrow & goal(S), \\ & at(0, T1), \\ & at(S, T2), \\ & T2 - T1 > 50, \\ & not\ ab(S). \end{array}$$

The cr-rule

$ab(S) \overset{+}{\leftarrow} step(S).$

allows, when necessary, to consider exceptions to this rule. If John can get to the doctor's office in 50 minutes the program will find the corresponding plan and a proper schedule for its actions. If the initial time conditions are such that John cannot get to the doctor in 50 minutes the cr-rule will be used to defeat the weak constraint above and find a possible plan, say, requiring 55 minutes. The desired solutions can be easily found by $\mathcal{AD}$*engine*.

The next example illustrates the use of defined predicates of the language.

Example 5 [*Using Defined Predicates*]
Let us now consider an extension of the John's problem from Example 2 by assuming that "*John always travels in a taxi and that the taxi rate is $2.45 per minute. John knows the amount of money in his bank account as well as the amount he should pay to the doctor. Now he needs not only get to the doctor on time and ready but also make sure that he has enough money for the visit.*" To solve the problem one may try to encode this knowledge by the following rules.

bank_account(john, 200).
doctor_payment(130).
taxi_rate(2.45).

We assume that John has $200 in his bank account and that the doctor charges $130. The next rule defines the relation *enough_money(P)* which holds iff person P has enough money to accomplish the task.

```
enough_money(P) ← goal(S),
                  at(0,T1),
                  at(S,T2),
                  money_needed(P,T1,T2,Y1),
                  bank_account(P,Y2),
                  Y2 - Y1 ≥ 0.
```

The next two rules are self-explanatory.

```
money_needed(P,T1,T2,Y) ← doctor_payment(Y1),
                          taxi_payment(T1,T2,Y2),
                          Y = Y1 + Y2.

taxi_payment(T1,T2,Y) ← taxi_rate(Rate),
                        Y = Rate * (T2 - T1).
```

Since at the final state of the trajectory John should have enough money to pay the doctor and the taxi driver, we expand this program by a regular rule

$\leftarrow$ *not enough_money*$(john)$.

It is natural to declare *enough_money* as a regular predicate, and the rest, except the mixed predicate *at* and the primitive constraints, as defined predicates. Let us denote the resulting program by Π. Even though the program correctly represents our knowledge of the domain, due to the middle rule defining *enough_money*, the program is neither canonical nor safe and hence its answer sets cannot be computed by $\mathcal{AC}$*solver*. To remedy the problem we use the transformations of Section 3.1 to construct its canonical and safe counterpart, Π'. To this end we first introduce a new defined predicate $d_1(P, T1, T2, Y1, Y2)$ and replace the definition of *enough_money(P)* by

$$\begin{array}{ll} enough\_money(P) \leftarrow & goal(S), \\ & at(0,T1), \\ & at(S,T2), \\ & d_1(P,T1,T2,Y1,Y2). \end{array}$$

$$\begin{array}{ll} d_1(P,T1,T2,Y1,Y2) \leftarrow & money\_needed(P,T1,T2,Y1), \\ & bank\_account(P,Y2), \\ & Y2 - Y1 \geq 0. \end{array}$$

The resulting program is canonical but not safe because constraint variables $Y1, Y2$ of d_1 do not occur in the mixed predicates of the middle rule defining *enough_money*. To achieve safety, we introduce a new defined predicate $d_0(P, T1, T2)$ and replace this middle rule by the rules

$$\begin{array}{ll} enough\_money(P) \leftarrow & goal(S), \\ & at(0,T1), \\ & at(S,T2), \\ & d_0(P,T1,T2). \end{array}$$

$$d_0(P,T1,T2) \leftarrow d_1(P,T1,T2,Y1,Y2).$$

The resulting program Π' is canonical and safe. Now to solve our problem we simply need to call $\mathcal{AC}$*engine* with Π' as an input.

To test our inference engines in realistic applications we used an extension of USA-Advisor[6] – a decision support system for the reaction control system (RCS) of the space shuttle.

The RCS has primary responsibility for maneuvering the aircraft while it is in space. It consists of fuel and oxidizer tanks, valves and other plumbing needed to provide propellant to the maneuvering jets of the shuttle. It also includes electronic circuitry both to control the valves in the fuel lines and to prepare the jets to receive firing commands. Overall the system is rather complex, on that it includes 12 tanks, 44 jets, 66 valves, 33 switches, and around 160 computer commands (computer-generated signals). The RCS can be viewed, in a simplified form, as a directed graph whose nodes are tanks, jets and pipe junctions, and whose arcs are labeled by valves. For a jet to be ready to fire, oxidizer and fuel propellants need to flow through the nodes (tanks, junctions) and valves which are open and reach the jet. A node is pressurized when fuel or oxidizer reaches the node.

The system can be used for checking plans, planning and diagnosis. To test our solver, we have expanded the system to allow explicit representation of time to combine planning and scheduling. We will illustrate our extension by the following example.

Example 6 [*Planning and scheduling in USA-Advisor*]
Assume that after a node N gets pressurized it takes around 5 seconds for the oxidizer propellant to get stabilized at N and 10 seconds for fuel propellant to get stabilized. Furthermore, we cannot open a valve V which links N1 to N2 (link(N1,N2,V)), until N1 has been stabilized. Time steps of the program should be assigned actual time (in seconds) satisfying these constraints. We should be able to answer questions like: can a particular maneuver be performed in less than 30 secs?

To solve the problem, we expand the signature of USA-Adviser by constraint sorts time = [0..400] and mixed predicate $at(S, T)$ where S and T are variables for steps and actual time receptively. The relation holds if step S is performed at time T. In addition we need relations $otank(X)$ and $ftank(X)$ which hold if X is a oxidizer tank and fuel tank respectively. Fluent $got\_opened(V, S)$ is true when valve V was closed at step $S - 1$ and got opened at step S. Fluent $got\_pressurized(N, X, S)$ is true when node N is not pressurized at step $S - 1$ and is pressurized at step S by tank X. The new program contains all rules from original USA-advisor, and new rules describing the scheduling constraints. Here are typical examples of such rules.

The first rule, which is a part of the original USA-advisor, says that a tank node N_1 is pressurized by tank X at step S if it is connected by an open valve V to a node which is pressurized by tank X of sub-system R.

$$\begin{aligned} \mathit{holds}(\mathit{pressurized\_by}(N_1, X), S) \leftarrow\ & \mathit{step}(S), \mathit{tank\_of}(N_1, R), \\ & \mathit{holds}(\mathit{in\_state}(V, \mathit{open}), S),\ \mathit{link}(N_2, N_1, V), \\ & \mathit{tank\_of}(X, R),\ \mathit{holds}(\mathit{pressurized\_by}(N_2, X), S). \end{aligned}$$

The second rule defines a new relation *got_pressurized* in terms of *pressurized_by* and other relations of the old system.

$$\begin{aligned} \mathit{got\_pressurized}(N, X, S+1)\ \leftarrow\ & \mathit{link}(N_1, N, V),\ \mathit{tank\_of}(X, R), \\ & \mathit{not}\ \mathit{holds}(\mathit{pressurized\_by}(N, X), S), \\ & \mathit{holds}(\mathit{pressurized\_by}(N, X), S+1). \end{aligned}$$

Next four rules are typical examples of temporal constraints. The first rule says that if N_1 is pressurized by oxidizer tank, N_1 takes 5 seconds to stabilize.

$$\begin{aligned} \leftarrow\ & \mathit{link}(N_1, N_2, V), \mathit{got\_pressurized}(N_1, X, S_1), S_1 < S_2, \mathit{otank}(X), \\ & \mathit{got\_opened}(V, S_2), \mathit{at}(S_1, T_1), \mathit{at}(S_2, T_2), T_1 - T_2 > -5. \end{aligned}$$

The second guarantees that if N_1 is pressurized by fuel tank, N_1 takes 10 seconds to stabilize.

$$\begin{aligned} \leftarrow\ & \mathit{link}(N_1, N_2, V), \mathit{got\_pressurized}(N_1, X, S_1), S_1 < S_2, \mathit{ftank}(X), \\ & \mathit{got\_opened}(V, S_2), \mathit{at}(S_1, T_1), \mathit{at}(S_2, T_2), T_1 - T_2 > -10. \end{aligned}$$

The third specifies that some time should elapse between steps.

$$\leftarrow\ S_1 < S_2, \mathit{at}(S_1, T_1), \mathit{at}(S_2, T_2), T_1 - T_2 > -1.$$

Finally we require that the jets of a system should be ready to fire by 30 seconds.

$$\leftarrow\ \mathit{system}(R), \mathit{goal}(S, R), \mathit{at}(0, T_1), \mathit{at}(S, T_2), T_2 - T_1 > 30.$$

The program can serve as an input to $\mathcal{AD}$*engine*. We tested the solver on 450 auto-generated instances of the initial situations and maneuvers. The results show that $\mathcal{AD}$*engine* could compute answer sets for most of the instances tried in less than two minutes. (The acceptable performance given by our USA customers was 20 minutes). It is worth noting that for the standard translation of our program into a regular ASP program, the grounder *lparse1.1.1* (run on the same machine) can't ground the simplest program instance in a day. The latest version of grounder *gringo*[4] takes an hour to ground this instance, producing a file of size 16 Gbytes, but the answer set solver *clasp* can not produce any results in 30 hours.

5 Conclusion

In this paper we introduced a knowledge representation language $\mathcal{AC}(\mathcal{C})$ extending the syntax and semantics of ASP and CR-Prolog, gave some examples of its use for knowledge representation, and presented an algorithm, $\mathcal{AC}$*solver*, for computing answer sets of $\mathcal{AC}(\mathcal{C})$ programs. The algorithm does not require full grounding of a program and combines "classical" ASP solving methods with CLP techniques and CR-Prolog based abduction. The $\mathcal{AC}(\mathcal{C})$ based approach often allows to solve problems which are impossible to solve by more traditional ASP solving techniques. We believe that further investigation of the language and the development of more efficient and reliable solvers for its programs can help to substantially expand the domain of applicability of the answer set programming paradigm. The work is based on previous results by Baselice, Bonatti, and one of the authors [9]. In [16], an algorithm is developed to combine ASP computation with constraint solving for the purpose of reasoning with ASP aggregates. The corresponding language however does not allow classification of predicates and hence does not avoid grounding of variables (except the variables which are local w.r.t. the aggregates). An interesting line of work investigates ways of replacing ASP programs by the corresponding constraint programs (see for instance [14]). We hope that our approach will prove more attractive from the standpoint of knowledge representation and also more efficient but this is of course a matter for further research. There is also a substantial amount of work on the development of a generalization of ASP by rules which allows arbitrary "constraints atoms" [29, 26]. It remains to be seen if work on the development of

[4]Unreleased version obtained from the author in Aug 2008.

$\mathcal{AC}(\mathcal{C})$ solvers can profit from insights from this work.

6 Acknowledgments

This work was supported in part by NASA contract NASA-NEG05GP48G and ATEE/DTO contract ASU-06-C-0143. We thank Roland H. C. Yap for many discussions and help in our implementation of the prototype system $\mathcal{AC}$*engine*.

References

[1] K. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3,4):335–365, 1991.

[2] Yulia Babovich and Marco Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR-05*, Jan 2004.

[3] Marcello Balduccini. USA-Smart: Improving the Quality of Plans in Answer Set Planning. In *PADL'04*, Lecture Notes in Artificial Intelligence (LNCS), Jun 2004.

[4] Marcello Balduccini. CR-MODELS: An inference engine for CR-Prolog. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'07)*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 18–30. Springer, 2007.

[5] Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. In Patrick Doherty, John McCarthy, and Mary-Anne Williams, editors, *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, pages 9–18, Mar 2003.

[6] Marcello Balduccini, Michael Gelfond, and Monica Nogueira. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, 47:183–219, 2006.

[7] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press, Jan 2003.

[8] Chitta Baral, Karen Chancellor, Nam Tran, Nhan Tran, Anna Joy, and Michael Berens. A knowledge based approach for representing and reasoning about cell signalling networks. In *Proceedings of European Conference on Computational Biology, Supplement on Bioinformatics*, pages 15–22, 2004.

[9] Sabrina Baselice, Piero A. Bonatti, and Michael Gelfond. Towards an integration of answer set and constraint solving. In *Proceedings of ICLP-05*, pages 52–66, 2005.

[10] Daniel R. Brooks, Esra Erdem, James W. Minett, and Donald Ringe. Character-based cladistics and answer set programming. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages*, pages 37–51, 2005.

[11] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Adding Weak Constraints to Disjunctive Datalog. In *Proceedings of the 1997 Joint Conference on Declarative Programming APPIA-GULP-PRODE'97*, 1997.

[12] Weidong Chen, Terrance Swift, and David S. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–201, 1995.

[13] Keith Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[14] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. An experimental comparison of constraint logic programming and answer set programming. In *Proceedings of AAAI07*, pages 1622–1625, 2007.

[15] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A deductive system for nonmonotonic reasoning. In *International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR97*, LNAI 1265, pages 363–374. Springer Verlag, Berlin, 1997.

[16] Islam Elkabani, Enrico Pontelli, and Tran Cao Son. Smodelsa - a system for computing answer sets of logic programs with aggregates. In *LPNMR*, pages 427–431, 2005.

[17] M. Gebser, B. Kaufman, A. Neumann, and T. Schaub. Conflict-driven answer set enumeration. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 3662 of *lnai*, pages 136–148. Springer, 2007.

[18] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.

[19] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[20] Enrico Giunchiglia, Yulia Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377, 2006.

[21] J. Jaffar, S. Michaylov, P. J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[22] Joxan Jaffar and M. J. Maher. Constraint Logic Programming. *Journal of Logic Programming*, 19/20:503–581, 1994.

[23] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, George Gottlob, Stefania Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2006.

[24] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Proceedings of the eleventh international conference on Logic programming*, pages 23–37, Cambridge, MA, USA, 1994. MIT Press.

[25] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[26] Lengning Liu, Enrico Pontelli, Tran Cao Son, and Miroslaw Truszczynski. Logic programs with abstract constraint atoms: The role of computations. In *Proceedings of ICLP-07*, pages 286–301, 2007.

[27] Victor W. Marek and Miroslaw Truszczynski. *Nonmonotonic logics; context dependent reasoning*. Springer Verlag, Berlin, 1993.

[28] Victor W. Marek and Miroslaw Truszczynski. *Stable models and an alternative logic programming paradigm*, pages 375–398. The Logic Programming Paradigm: a 25-Year Perspective. Springer Verlag, Berlin, 1999.

[29] Victor W. Marek and Miroslaw Truszczynski. Logic programs with abstract constraint atoms. In *Proceedings of AAAI04*, pages 86–91, 2004.

[30] K. Marriott and Peter J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, Cambridge, MA, 1998.

[31] Veena S. Mellarkod. *Integrating ASP and CLP Systems: Computing Answer Sets from Partially Ground Programs*. PhD thesis, Texas Tech University, Dec 2007.

[32] Veena S. Mellarkod and Michael Gelfond. Enhancing asp systems for planning with temporal constraints. In *LPNMR 2007*, pages 309–314, May 2007.

[33] Ilkka Niemela. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[34] Ilkka Niemela and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 420–429, 1997.

[35] Ilkka Niemela, Patrik Simons, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, Jun 2002.

[36] G. Ramalingam, Junehwa Song, Leo Joskowicz, and Raymond E. Miller. Solving systems of difference constraints incrementally. *Algorithmica*, 23(3):261–275, 1999.

[37] SICStus Prolog. *SICStus Prolog User's Manual Version 4.* http://www.sics.se/isl/sicstuswww/site/index.html, 2007.

[38] Patrik Simons. Extending and implementing the stable model semantics. Research Report A58, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, Espoo, Finland, April 2000. Doctoral dissertation.

[39] Timo Soininen and Ilkka Niemella. Developing a declarative rule language for applications in product configuration. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages*, pages 305–319, 1998.

[40] Peter J. Stuckey. Constructive negation for constraint logic programming. In *LICS*, pages 328–339, 1991.

[41] Tommi Syrjanen. Implementation of logical grounding for logic programs with stable model semantics. Technical Report 18, Digital Systems Laboratory, Helsinki University of Technology, 1998.

[42] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

Appendix

In the following, we will use Π, Φ, and Ψ, possibly with indexes, to represent programs of $\mathcal{AC}(\mathcal{C})$, reduct programs, and "mixed" programs respectively. Rules will be represented by R with indexes. For atoms and e-literals, we use letters a and l respectively. Letters l_r, l_m, and l_d refer to regular e-literals, mixed literals and defined literals respectively. Sets of literals are denoted by S with indexes, and sets of atoms by A with indexes. For a rule R, $\mathit{head}(R)$ and $\mathit{body}(R)$ are the sets of e-literals in the head and body of R respectively. For a set of e-literals S, we define $\mathit{pos}(S) = \{a \ : \ a \in S\}$ and $\mathit{neg}(S) = \{a \ : \ \mathit{not}\ a \in S\}$; and notations $\mathit{mixed}(S)$, $\mathit{defined}(S)$ and $\mathit{regular}(S)$ refer to the set of mixed, defined, and regular e-literals of S respectively. The set S is *complete* with respect to a program if for any atom p of the program, either $p \in S$ or $not\, p \in S$. For a rule R, we also use $mixed(R)$, $defined(R)$, and $regular(R)$ to denote the set of mixed, defined, and regular e-literals in the body of R.

Given a program Π, $\mathcal{M}$ denotes the set of atoms of $\Sigma(\Pi)$ of the form $m(t, Y)$ where m is a mixed predicate, t is a ground regular term, and Y is the value variable of $\langle m, t\rangle$. Given a ground substitution θ, $\theta(\mathcal{M})$ is the result of replacing every value variable Y occurring in $\mathcal{M}$ by θ. If Π_d has an answer set, it has a unique answer set A_d because Π_d is a constraint program. We define

(1) $\Phi(\Pi, \theta) = \{head(R) \leftarrow regular(R) \ : \ R \in ground(\mathit{\Pi}_m),$
$\quad mixed(R) \subseteq \theta(\mathcal{M}), defined(R) \subseteq c(A_d, \mathit{\Pi}_d)\}$,

(2) $\Psi(\Pi, \theta) = ground(\Pi_r) \cup \Phi(\Pi, \theta)$, and

(3) $\mathit{\Pi}(\theta) = ground(\mathit{\Pi}) \cup \theta(\mathcal{M})$.

$A(\theta)$ and $A_\Psi(\theta)$ denote the answer set of the ASP program $\Pi(\theta)$ and $\Psi(\Pi, \theta)$ respectively.

Given a set of atoms A, $\overline{A}$ denotes all the ground atoms from the signature $\Sigma(\Pi)$ excluding those of A; $not\ A$ denotes $\{not\ a \ : \ a \in A\}$.

Lemma 1 Consider a canonical program Π, a ground substitution θ, and a set of e-literals S. Assume Π_d has the answer set A_d. $pos(S) \cup A_d \cup \theta(\mathcal{M})$ is an answer set of $\Pi(\theta)$ iff $pos(S)$ is an answer set of $\Psi(\Pi, \theta)$.

Proof.

Let A_1 be the set of ground atoms from signature $\Sigma(\Pi)$ formed by defined and mixed predicates. Since, by the definition of canonical programs, Π_d contains no regular e-literals, A_1 forms a splitting set of $\Pi(\theta)$. It splits the program into $\mathit{\Pi_1} = ground(\mathit{\Pi}_d) \cup \theta(\mathcal{M})$ and $\mathit{\Pi_2} = ground(\mathit{\Pi}_r) \cup ground(\mathit{\Pi}_m)$. From the construction, it immediately follows that

(4) $A_d \cup M$ is an answer set of Π_1.

Since $pos(S)$ and $M \cup A_d$ are disjoint, by splitting set theorem [24], (1) and (2), A is an answer set of $\Pi(\theta)$ iff $pos(S)$ is an answer set of $\Psi(\Pi, \theta)$. $\square$

Now we present the proof for the first part of Theorem 1.

Proof.

Let

(5) $< S, C >= \mathcal{AC}solver(\Pi, B, Q)$.

By line 5 and line 2 of Figure 1,

(6) S is complete and consistent.

To prove the theorem, we will construct an answer set, A, of Π from $< S, C >$.

The definition of $\mathcal{AC}solver$ and (5) imply that

(7) there is a query $Q' = \text{query}(\Pi, S, Q)$ such that

(8) $\langle C, \textit{true} \rangle = \textit{c\_solve}(\mathit{\Pi}_d, Q')$,

(9) C is consistent, and

(10) every solution of C is a solution of Q' w.r.t. the answer set of Π_d.

Let

(11) γ be a solution of C,

(12) $M = \gamma(\mathcal{M})$,

(13) $\Phi = \Phi(\Pi, \gamma)$,

(14) $\Psi = \Psi(\Pi, \gamma)$, and

(15) $\Pi(M) = \Pi(\gamma)$.

Let

(16) $A = pos(S) \cup M \cup A_d$.

By Definition 5 of answer set, to prove A is an answer set of Π, it is sufficient to show that A is an answer set of ASP program $\Pi(M)$.

To prove (i), we show $B \subseteq S$. By considering the $\mathcal{AC}solver$ algorithm it is easy to prove that there is a set of ground regular e-literals B' such that

(17) $B \subseteq B'$ and

(18) $S = Cn(\Pi_r \cup \Pi_m, B')$.

By the definition of *lb*, *ub*, and Cn,

(19) $B' \subseteq S$, which, together with (17), implies

(20) $B \subseteq S$.

Since S is consistent (6), $S \subseteq c(A, \Pi)$. Therefore, (20) implied condition (i): $B \subseteq c(A, \Pi)$.

Clearly $pos(S) = regular(A)$ because of (16), and thus condition (ii) of the theorem holds.

To prove (iii), note that the specification of function *query* ensures that $Q \subseteq Q'$ and thus, by (10) and (11),

(21) γ is a solution of Q w.r.t. the answer set of Π_d, i.e., A_d.

Since A_d is a subset of A (16), A satisfies Q. Hence, (iii) holds.

We next show that A is an answer set of ASP program $\Pi(M)$. By Lemma 1, A is an answer set of $\Pi(M)$ iff

(22) $pos(S)$ is an answer set of Ψ.

Let

(23) $S_3 = lb(\Pi_r \cup \Pi_m, S)$, and

(24) $S_4 = \{not\ a : a \notin ub(\Pi_r \cup \Pi_m, S_3)\}$.

By the definition of Cn,

(25) $S = S_3 \cup S_4$.

By the definition of *lb*,

(26) $S \subseteq S_3$. Since S is complete (6),

(27) S_3 is complete.

Since $S_3 \subseteq S$ (25), S_3 and S are complete (27,6), and S is consistent, we have

(28) $S = S_3$.

To prove (22), we first show (29) and then (30).

(29) The set $pos(S)$ satisfies the ASP program Ψ.

(30) The set $pos(S)$ is an answer set of the reduct $\Psi^{pos(S)}$.

Let

(31) R be a rule from Ψ such that $pos(S)$ satisfies the body of R.

To show that $head(R) \in S$, we consider two cases.

Case 1 R belongs to the regular part of $ground(\Pi)$. Then

(32) $body(R)$ is satisfied by $pos(S_3)$ (31, 28).

By (27) and (32) we have

(33) $body(R) \subseteq S_3$.

From (33), (23), and the clause (1) of the definition of *lb* we have that $head(R) \in S_3$. Together with (28), this implies

(34) $head(R) \in S$

Case 2 (35) $R \in \Phi$,

i.e., R is obtained by removing mixed and defined e-literals from the body of some rule $R' \in \mathit{ground}(\mathit{\Pi}_m)$ such that $\mathit{mixed}(R') \subseteq M$ and $\mathit{defined}(R') \subseteq c(A_d, \Pi_d)$.

Let us assume that

(36) $\mathit{head}(R) \notin \mathit{pos}(S)$.

Let

(37) R' be obtained by grounding a rule $R'' \in \Pi_m$.

Since $\mathit{head}(R'')$ is grounded, this, together with (36), implies that

(38) $\mathit{head}(R'') \notin \mathit{pos}(S)$.

Recall that to simplify the presentation we assumed that no rule R of Π_m contains more than one defined e-literal.

(39) $\mathit{defined}(R)$ is either empty or equal to $\{d(\bar{X})\}$ or $\{\mathit{not}\ d(\bar{X})\}$

for some defined predicate d and a list of variables $\bar{X}$.

Hence, $\mathit{defined}(R'')$ is equal to $\{d(\bar{X})\}$ or $\{\mathit{not}\ d(\bar{X})\}$, or is empty. Let us consider the first case,

(40) $\mathit{defined}(R'') = \{d(\bar{X})\}$

where $\bar{X} = [X_1, \ldots, X_k]$. Since program Π_m satisfies the safety condition, we have that variables $\bar{X}$ appear in the atoms of $\mathit{mixed}(R'')$. Recall that for simplicity of the presentation we assumed that mixed literals of the language have exactly one regular and one constraint parameter. This means that $\mathit{mixed}(R'')$ contains $m_1(t_1, X_1), \ldots, m_k(t_k, X_k)$.

Consider query Q' from (7). By the definition of function *query*, (31), (38), and (40) we have

(41) $not\ d(\bar{Y}) \in Q'$

where $\bar{Y} = [Y_1, \ldots, Y_k]$ with Y_i being the value variable of $\langle m_i, t_i \rangle$ for every $1 \leq i \leq k$.

Since γ is a solution of C (11), the definition of *c_solve* guarantees that $\gamma(Q') \subseteq c(A_d, \Pi_d)$. This, together with (41), implies

(42) $d(\gamma(\bar{Y})) \not\in A_d$

To obtain a contradiction we show that (42) cannot be true. By (37) we have that R' is obtained from R'' by a substitution which replaces occurrences of $\bar{X}$ by $\bar{x}$. Hence *defined*$(R') = d(\bar{x})$ where $\bar{x} = [x_1, \ldots, x_k]$. But, since $R \in \Phi$ (35), we have that $m_1(t_1, x_1), \ldots, m_k(t_k, x_k) \in M$. From the construction of M (12) we can conclude that

(43) $\bar{x} = \gamma(\bar{Y})$. Therefore,

(44) *defined*$(R') = d(\gamma(\bar{Y}))$.

This together with definition of Φ (13) and (35) implies that

(45) $d(\gamma(\bar{Y})) \in A_d$

which contradicts (42).

The case of

(46) *defined*$(R'') = \{not\ d(\bar{X})\}$

will be treated in a similar manner.

To complete the proof of (29), it suffices to notice that if *defined*(R'') were empty then, by the definition of function *query*, (38) and (31), query(Π, S, Q'') would return a set containing *false* which would contradict (8).

Now to complete the proof, we need to prove (30). Assume that $pos(S)$ is not the minimal set satisfying $\Psi^{pos(S)}$. Let

(47) S_0 be a complete and consistent set of e-literals such that $pos(S_0) \subset pos(S)$, and $pos(S_0)$ satisfies $\Psi^{pos(S)}$.

Let

(48) Φ_1 be $\alpha(\Pi_r \cup \Pi_m, S_3)$. We claim

(49) $pos(S_0)$ satisfies Φ_1

whose proof will be given later.

Since $ub(\Pi_r \cup \Pi_m, S_3)$ is the answer set of Φ_1 and $pos(S_0)$ satisfies Φ_1 (49), $ub(\Pi_r \cup \Pi_m, S_3) \subseteq pos(S_0)$, which in turn implies that $neg(S_0) \subseteq S_4$ (24). Since $S = S_3 \cup S_4$ (25), $S_4 \subseteq neg(S)$. Therefore, $neg(S_0) \subseteq neg(S)$. Since both S_0 and S are complete and consistent, $pos(S) \subseteq pos(S_0)$,which contradicts the assumption $pos(S_0) \subset pos(S)$. So, (30) holds.

To show (49), we prove

(50) For any rule $R \in \Phi_1$, $R \in \Psi^{pos(S)}$.

By the construction of Φ_1, there are two cases: R is obtained from $R' \in \Pi_r$ or $R' \in \Pi_m$.

Case 1 $R' \in \Pi_r$. By the construction of Φ_1(48), $body(R') \subseteq S_3$. Therefore, $R \in \Psi^{pos(S)}$ because $S = S_3$ (28).

Case 2 $R' \in \Pi_m$. By clause 1 and 2 of the definition of ub, $regular(R') \subseteq S_3$ and $head(R') \in S_3$. We next show that $head(R') \leftarrow regular(r') \in \Phi$. By (7), $defined(R') \in Q'$. Since c_solve returns $\langle C, true\rangle$ (8) and γ is a solution of C (11), $\gamma(defined(R')) \subseteq c(A_d, \Pi_d)$. By the definition of M (12), $\gamma(mixed(R')) \subseteq M$. Therefore, by (13) and (14), $head(R') \leftarrow regular(r') \in \Phi \subseteq \Psi$. Hence, $R \in \Psi^{pos(S)}$.

By (50), $\Phi_1 \subseteq \Psi^{pos(S)}$. Since $pos(S_0)$ satisfies $\Psi^{pos(S)}$ (47), $pos(S_0)$ satisfies Φ_1. $\square$

Let S be a set. A function $f : 2^S \rightarrow 2^S$ is *monotonic* if for any sets A and B, $A \subseteq B$ implies $f(A) \subseteq f(B)$.

Lemma 2 (Lemma A.1 [38])
Let S be a set, $f : 2^S \rightarrow 2^S$ a monotonic function, and $X \subseteq S$. If $f(X) \subseteq X$, then *lfp(f)* $\subseteq X$, where *lfp(f)* denotes the least fixed point of f.

A set of atoms A *agrees with* a set of e-literals S if $pos(S) \subseteq A$ and $neg(S) \cap A = \emptyset$. Alternatively, A agrees with S if $S \subseteq A \cup not\ \overline{A}$.

Corresponding to the five clauses in the definition of lb, we define the following functions on a collection of sets of e-literals.

Let

(51) $f_1(X) = \{head(R)\ :\ R \in \Pi_r, body(R) \subseteq X\}$,

(52) $f_2(X) = \{l \in body(R)\ :\ R \in \Pi_r \cup \Pi_m$, R is the unique rule with $head(R)$, $head(R) \in X$, and $body(R)$ is not falsified by $X\}$,

(53) $f_3(X) = \{not\ l\ :\ \exists(not\ l_0) \in X$, $(l_0 \leftarrow B_1, l, B_2) \in \Pi_r$, and $B_1, B_2 \subseteq X\}$, and

(54) $f_4(X) = \{not\ l_0\ :$ the body of every rule of $\Pi_r \cup \Pi_m$ with head l_0 is falsified by $X\}$.

Let U be the set of all ground regular e-literals from the signature $\Sigma(\Pi)$, and B a set of e-literals. We define

(55) $f(B, X) = B \cup X \cup f_1(X) \cup f_2(X) \cup f_3(X) \cup f_4(X)$.

(56) $f'(B, X) = f(B, X)$ if $f(B, X)$ is consistent, and U otherwise.

Lemma 3 The function $f'(\_, X)$ is monotonic with respect to its second parameter.

Proof.

Assuming

(57) $S \subseteq S'$,

we will show $f'(\_, S) \subseteq f'(\_, S')$. Consider the following two cases.

Case 1 (58) $f(\_, S')$ is inconsistent. By the definition of f', $f'(\_, S') = U$, implying that $f'(\_, S) \subseteq f'(\_, S')$.

Case 2 (59) $f(\_, S')$ is consistent. By the definition of f',

(60) $f'(\_, S') = f(\_, S')$.

One can verify that

(61) Functions f_1, f_3, f_4 are monotonic.

We next show that

(62) $f(\_, S) \subseteq f(\_, S')$.

Assuming that

(63) $f(\_, S) \not\subseteq f(\_, S')$,

we will later obtain a contradiction (to (59))

(64) $f(\_, S')$ is inconsistent.

By (59) and (62), $f(\_, S)$ is consistent. By the definition of f'

(65) $f'(\_, S) = f(\_, S)$.

Therefore, (62), together with (59) and (65), gives

$f'(\_, S) \subseteq f'(\_, S')$.

We show (64) below. By (63) and (61), there exists $a \in f_2(S) - f_2(S')$. By the definition of f_2,

(66) there is a unique rule $R \in \Pi_r \cup \Pi_m$ with $head(R)$,

(67) $body(R)$ is not falsified by S, and

(68) $head(R) \in S$.

(57) and (68) imply

(69) $head(R) \in S'$.

However, since $a \notin f_2(S')$, (66) and (69), $body(R)$ is falsified by S' by the definition of f_2. Furthermore, since R is the only rule with the head $head(R)$ (66),

(70) $not\ head(R) \in f_4(S')$

by the definition of f_4.

By the definition of f, we have $S' \subseteq f(\_, S')$ and $f_4(S') \subseteq f(\_, S')$. Hence, (69) and (70) imply that $f(\_, S')$ is inconsistent, i.e., (64) holds.

□

Lemma 4 Given a canonical program Π, θ and a set of e-literals B, if $A_\Psi(\theta)$ agrees with B, $A_\Psi(\theta)$ agrees with $f_i(B)$ for i from 1 to 4.

Proof.

One can verify that the claim is true for f_1, f_3, and f_4. To prove that $A_\Psi(\theta)$ agrees with $f_2(B)$, we show

(71) $pos(f_2(B)) \subseteq A_\Psi(\theta)$, and

(72) $neg(f_2(B)) \cap A_\Psi(\theta) = \emptyset$.

To prove (71) and (72), consider any literal $l \in f_2(B)$. By the definition of f_2,

(73) there is a unique rule $R \in \Pi_r \cup \Pi_m$ with $head(R)$,

(74) $body(R)$ is not falsified by B, and

(75) $head(R) \in B$, which implies

(76) $head(R) \in A_{\mathit{\Psi}}(\theta)$ because $A_{\Psi}(\theta)$ agrees with B.

Since R is the only rule with head $head(R)$ (73), (76), and $A_{\Psi}(\theta)$ is the answer set of $\Psi(\Pi, \theta)^{A_{\Psi}(\theta)}$, we have $head(R) \leftarrow pos(R) \in \mathit{\Psi}(\mathit{\Pi}, \theta)^{A_{\mathit{\Psi}}(\theta)}$. Hence, if l is an atom of $body(R)$, $l \in A_{\Psi}(\theta)$ and thus (71) holds; otherwise, $not\ l \notin A_{\Psi}(\theta)$ and thus (72) holds. □

Proposition 1 (Lower bound)
Given a canonical program Π, θ and a set of e-literals B, let $S = lb(\Pi_r \cup \Pi_m, B)$. If $A_{\Psi}(\theta)$ agrees with B, it agrees with S.

Proof.

By the definition of lb,,

(77) S is the minimal set of e-literals closed under the five clauses. Hence,

(78) S is the least fixed point of $f'(B, X)$ (56).

(79) By Lemma 3, $f'(B, X)$ is monotonic on its second parameter.

Let $S_{\Psi} = A_{\Psi}(\theta) \cup not\ \overline{A_{\Psi}(\theta)}$. We next show

(80) $f'(B, S_{\Psi}) \subseteq S_{\Psi}$.

By Lemma 4, $A_{\Psi}(\theta)$ agrees with $f_i(A_{\Psi}(\theta))$ ($i \in [1..4]$), implying that $f_i(A_{\Psi}(\theta)) \subseteq S_{\Psi}$ ($i \in [1..4]$). Since $A_{\Psi}(\theta)$ agrees with B, $B \subseteq S_{\Psi}$. By the definition of f, i.e., $f(B, S_{\Psi}) = B \cup S_{\Psi} \cup f_1(S_{\Psi}) \cup f_2(S_{\Psi}) \cup f_3(S_{\Psi}) \cup f_5(S_{\Psi})$,

(81) $f(B, S_{\Psi}) \subseteq S_{\Psi}$. Hence,

$f(B, S_{\Psi})$ is consistent because S_{Ψ} is so.

By the definition of f',

(82) $f'(S, \Psi) = f(B, S_{\Psi})$.

(82) and (81) imply that $f'(B, S_{\Psi}) \subseteq S_{\Psi}$.

Since S is the least fixed point of f' (78), $S \subseteq S_{\Psi}$. Therefore, $A_{\Psi}(\theta)$ agrees with S. □

Proposition 2 (Upper bound)
Given a canonical program Π, θ and a set of e-literals B, let $A_1 = ub(\Pi_r \cup \Pi_m, B)$. If $A_\Psi(\theta)$ agrees with B, $A_\Psi(\theta) \subseteq A_1$.

Proof.

Since $A_\Psi(\theta)$ agrees with B, we have

(83) $pos(B) \subseteq A_\Psi(\theta)$, and

(84) $neg(B) \cap A_\Psi(\theta) = \emptyset$.

Let

(85) $\Phi_1 = \Psi(\Pi, \theta)^{A_\Psi(\theta)}$, and

(86) $\Phi_2 = \alpha(\Pi_r \cup \Pi_m, B)$ and A_1 be the answer set of Φ_2.

Given a normal program P without default negation and a set of atoms X, we define functions

(87) $f(P, X) = X \cup \{head(R) \ : \ R \in P, body(R) \subseteq X\}$, and

(88) $g(X) = f(\Phi_1, A_\Psi(\theta) \cap X)$.

One can verify that

(89) function f is monotonic with respect to its second parameter, implying

(90) function g is monotonic.

We will show later that

(91) $g(A_1) \subseteq A_1$, and

(92) $g(X)$ and $f(\Phi_1, X)$ have the same least fixed point.

Since $A_\Psi(\theta)$ is the answer set of Φ_1 (85),

(93) $A_\Psi(\theta)$ is the least fixed point of $f(\Phi_1, X)$, which, together with (92), implies

(94) $A_\Psi(\theta)$ is the least fixed point of $g(X)$.

By Lemma 2, (91) and (94) result in

$A_\Psi(\theta) \subseteq A_1$.

The following is the proof for (91). We first show

(95) $f(\Phi_1, A_\Psi(\theta) \cap A_1) \subseteq f(\Phi_2, A_\Psi(\theta) \cap A_1)$.

Consider any atom

(96) $a \in f(\Phi_1, A_\Psi(\theta) \cap A_1)$.

If $a \in A_\Psi(\theta) \cap A_1$, $a \in f(\Phi_2, A_\Psi(\theta) \cap A_1)$ by the definition of (87). Otherwise, (87) implies that there exists $R \in \Phi_1$ such that $a = \mathit{head}(R)$ and

(97) $\mathit{body}(R) \subseteq A_\Psi(\theta) \cap A_1$.

(98) In forming Φ_1, let R be obtained from $R' \in \Psi(\Pi, \theta)$ that is obtained from $R'' \in \Pi_r \cup \Pi_m$.

Clearly, $\mathit{body}(R) = \mathit{pos}(\mathit{regular}(R''))$. Hence,

(97) implies

(99) $\mathit{pos}(\mathit{regular}(R'')) \subseteq A_\Psi(\theta) \cap A_1 \subseteq A_\Psi(\theta)$.

By (98),

(100) $\mathit{neg}(\mathit{regular}(R'')) \cap A_\Psi(\theta) = \emptyset$.

By (100) and (83),

(101) $\mathit{neg}(\mathit{regular}(R'')) \cap \mathit{pos}(B) = \emptyset$.

By (99) and (84),

(102) $\mathit{pos}(\mathit{regular}(R'')) \cap \mathit{neg}(B) = \emptyset$.

By (101) and (102),

(103) $\mathit{body}(R'')$ is not falsified by B.

Since $f(P, X)$ is monotonic with respect to X (89) and $A_\Psi(\theta)$ is the least fixed point of $f(\Phi_1, X)$ (93),

(104) $f(\Phi_1, A_\Psi(\theta) \cap A_1) \subseteq f(\Phi_1, A_\Psi(\theta)) = A_\Psi(\theta)$.

Since $a \in f(\Phi_1, A_\Psi(\theta) \cap A_1)$ (96), (104) implies

(105) $a \in A_\Psi(\theta)$. Therefore,

(106) $\mathit{not}\ a = \mathit{not}\ \mathit{head}(R'') \notin B$.

because $A_\Psi(\theta)$ agrees with B (84).

(103) and (106), together with clause 1 and 2 respectively in obtaining $\alpha(\Pi_r \cup \Pi_m, B)$, imply that R'' is not removed in forming Φ_2. By removing the default negations (clause 3) and mixed and defined literals (clause 4) from R'', we have the rule $\mathit{head}(R'') \leftarrow \mathit{pos}(\mathit{regular}(R''))$, i.e.,

(107) $R \in \Phi_2$.

By (97) and the definition of $f(P, X)$, we have $a \in f(\Phi_2, A_\Psi(\theta) \cap A_1)$, implying that (95) holds.

Since $f(P, X)$ is monotonic w.r.t. X and A_1 is the answer set of Φ_2 (86), by (95), we have

(108) $f(\Phi_1, A_\Psi(\theta) \cap A_1) \subseteq f(\Phi_2, A_\Psi(\theta) \cap A_1) \subseteq f(\Phi_2, A_1) = A_1$, implying

(109) $g(A_1) = f(\Phi_1, A_\Psi(\theta) \cap A_1) \subseteq A_1$. Hence, (91) holds.

Now, we prove (92) by showing first $A_3 \subseteq A_\Psi(\theta)$ and then $A_\Psi(\theta) \subseteq A_3$ where A_3 is the least fixed point of $g(X)$.

By the definition of g (88),

(110) $g(A_\Psi(\theta)) = f(\Phi_1, A_\Psi(\theta) \cap A_\Psi(\theta))$. Hence

(111) $g(A_\Psi(\theta)) = A_\Psi(\theta)$ because $A_\Psi(\theta)$ is the least fixed point of $f(\Phi_1, X)$.

Since A_3 is the least fixed point of $g(X)$,

(112) $A_3 \subseteq A_\Psi(\theta)$, and

(113) $A_3 = g(A_3) = f(\Phi_1, A_\Psi(\theta) \cap A_3)$.

By (112),

(114) $A_\Psi(\theta) \cap A_3 = A_3$.

(113) and (114) imply $A_3 = f(\Phi_1, A_3)$. Hence, A_3 is a fixed point of $f(\Phi_1, X)$. Therefore, $A_\Psi(\theta) \subseteq A_3$ because $A_\Psi(\theta)$ is the least fixed point of $f(\Phi_1, X)$ (93). □

Corollary 1 Given a canonical program Π, θ and a set of e-literals B, let $S = Cn(\Pi_r \cup \Pi_m, B)$. If $A_\Psi(\theta)$ agrees with B, it agrees with S.

Proof.

By definition of *Cn*, it is sufficient to prove that if $A_\Psi(\theta)$ agrees with B, it agrees with $f_\Pi(B)$.

Let

$S_1 = lb(\Pi_r \cup \Pi_m, B)$, and

$S_2 = \{not\ a : a \notin ub(\Pi_r \cup \Pi_m, S_1)\}$.

By the definition of f_Π,

(115) $f_\Pi(B) = S_1 \cup S_2$.

Since $A_\Psi(\theta)$ agrees with B, by Proposition 1, $A_\Psi(\theta)$ agrees with S_1, and by Proposition 2, $A_\Psi(\theta) \subseteq ub(\Pi_r \cup \Pi_m, S_1)$ and thus $A_\Psi(\theta)$ agrees with S_2. Therefore, $A_\Psi(\theta)$ agrees with $S_1 \cup S_2$, i.e., $f_\Pi(B)$. □

We are now in a position to prove the part 2 of Theorem 1.

Proof.

Prove by induction on the number n of undecided e-literals in B.

In the following, the line numbers refer to those in Figure 1.

Base case: $n = 0$.

By line 1 and 3 of Algorithm 1 of $\mathcal{AC}solver$, we have

(116) $S = Cn(\Pi_r \cup \Pi_m, B)$, and

(117) $O = c\_solve(\mathit{\Pi}_d, query(\mathit{\Pi}, S, Q))$.

When n is 0, there are only two cases for $\mathcal{AC}solver$ to return false:

(118) S is inconsistent (line 2), or

(119) $O = \mathit{false}$ (line 4).

Note that lines 7–10 are not reachable because there is no undecided e-literals in S.

We prove the base case by contradiction. Assume there is an answer set A such that it agrees with B and satisfies Q. Let θ be a solution of Q w.r.t. A such that $\theta(\mathcal{M}) \subseteq A$.

Consider the first case: S is inconsistent (118). By Corollary 1 and Lemma 1, $A_\Psi(\theta)$ agrees with S and thus S is consistent, a contradiction.

Next, consider the second case $O = \mathit{false}$ (119). Let $C = \{x_1 = \theta(x_1), x_2 = \theta(x_2), ...\}$ be a set of constraints. Clearly, the solution of C is a solution of Q. Therefore, *c_solve* will not return *false*, contradicting $O = \mathit{false}$.

Inductive hypothesis: assume the theorem is correct for B with $n < k (k \geq 0)$.

Prove that when $n = k$, the theorem holds.

Let l be the literal picked in line 7.

There are three cases for $\mathcal{AC}solver$ returns *false*:

(120) S is inconsistent (line 2),

(121) O is *false* (line 4), or

(122) $\mathcal{AC}solver(\Pi, S \cup \{l\})$ returns *false* at line 8, and $\mathcal{AC}solver(\Pi, S \cup \{not\ l\})$ returns *false* at line 9.

For the first two cases, the proposition can be proved in the similar fashion to those in the base case.

For the last case, we use proof by contradiction. Assume there is an answer set A such that it agrees with B and satisfies Q. Let θ be a solution of Q w.r.t. A such that $\theta(\mathcal{M}) \subseteq A$.

(123) $A_\Psi(\theta)$ agrees with B because $A(\theta)$ agrees with B.

By (116) and Corollary 1, (123) implies

(124) $A_\Psi(\theta)$ agrees with S.

Since either $l \in A_\Psi(\theta)$ or $l \notin A_\Psi(\theta)$, (124) implies that

(125) $A_\Psi(\theta)$ agrees with either $\{l\} \cup S$ or $\{not\ l\} \cup S$, hence,

(126) $A(\theta)$ agrees with $\{l\} \cup S$ or $\{not\ l\} \cup S$, and satisfies Q.

However, since both $\mathcal{AC}solver(\Pi, \{l\} \cup S, Q)$ and $\mathcal{AC}solver(\Pi, \{not\ l\} \cup S, Q)$ return *false*, and both $\{l\} \cup S$ and $\{not\ l\} \cup S$ have less than k undecided e-literals, by inductive hypothesis, we have

(127) there is no answer set of Π that agrees with $\{l\} \cup S$ and satisfies Q, and

(128) there is no answer set of Π that agrees with $\{not\ l\} \cup S$ and satisfies Q.

(127) and (128) contradict (126). $\square$