INTEGRATING ASP AND CLP SYSTEMS: COMPUTING ANSWER SETS FROM PARTIALLY GROUND PROGRAMS

by

VEENA S. MELLARKOD, M. S.

A PHD DISSERTATION

IN

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

DOCTOR OF PHILOSOPHY

Approved

Dr. Michael Gelfond
Committee Chairman

Dr. Tran Cao Son

Dr. Richard Watson

Dr. Yuan-Lin Zhang

Fred Hartmeister
Dean of the Graduate School

December, 2007

To my Guru Michael Gelfond.

I naively jumped into a deep ocean ...

Thank you for guiding me through.

ACKNOWLEDGMENTS

I have been privileged to receive all the support I needed to finish this work. First and foremost, I thank my parents and sister for their encouragement and for always being there for me. Thanks Mom and Dad! Thank you Vidhya for your love and immense confidence in me.

I am most fortunate to have known my guru Dr. Michael Gelfond and feel blessed to be his student. The good karma from my previous life must have resulted in the joyful years of my life as his student. The experience of learning logic and philosophy of life from him was incredible. Thank you for always being patient with me and teaching me as much as I could understand, Dr. Gelfond. You have been my pillar of intellectual strength from the time I blindly followed you to Lubbock from El paso. This work would not have been possible without your constant help, guidance, and support. I am sure your teachings will serve me well in all walks of my life.

Special thanks should be extended to Dr. Richard Watson, Dr. Tran Cao Son and Dr. Yuan-Lin Zhang for serving on my dissertation committee and for their detailed reviews and helpful suggestions. Dr. Zhang especially helped me come up with some of the examples proposed in this work. I should also thank Dr. Enrico Pontelli for reviewing my work and for his insightful comments.

Dr. David Sallach of Argonne National Laboratory has introduced me to agent-based modeling and other exciting areas of research that I had never known. Although this dissertation did not include those aspects, I enjoyed learning from him and the discussions which usually went on for several hours. I will never forget his constant support and guidance. I hope he enjoyed teaching me as much as I enjoyed learning from him. I should also thank Dr. Charles Macal and

Dr. Mike North, both at ANL, for including me in several of their discussions and work. Thanks Chick for also letting me in on the secrets of effective presentations.
Monica Nogueira and Marcello Balduccini deserve special mention for their constant guidance and support through my graduate study. They both have been incredibly helpful through the phases of my student life, classes and research. Marcy started helping me in Ramon's class while distinguishing classical negation from default negation when I was still a novice at logic programming and did not stop until I finished my dissertation work. He really has been a true friend whenever I needed. Monica was also ever supportive whenever I requested for her help. She had a big hand in shaping my Masters' thesis and now my doctoral dissertation. I especially enjoy her ever-positive attitude and hard work that made our work together exciting. I will cherish their friendships for life.
Gregory Gelfond must simply be thanked for being such a nice friend. Greg, I have enjoyed your friendship and am very happy that we will be friends forever. Thank you for reading my work and helping me whenever I come to you, including helping me format this work.
The members of Knowledge Representation Lab at Texas Tech University maintained such a happy and jovial environment in the lab, without which work would have gotten boring. Being in the lab was such fun with them around!
I wish to thank Lara Gelfond for her support and her wonderful cakes. Lara, thank you for being so thoughtful to bake my favorite cherry cake for my dissertation defense.
Seven years of my stay in Lubbock as a foreigner would not have been memorable but for the constant help and encouragement of my friends. Pradip Sahu, Javed Ali and Sudhir Prabhu have been such best friends that they have now become a part of my extended family. Sirisha and Rajmohan have been no different and

were constantly teaching tips in cooking and taking me out to lunch on Fridays.
I thank my brothers Vinayak, Vivek and Lakshmi Narayan, my parents-in-law Ranganathan and Kowsalya Ranganathan, my grand mothers Bhageerathy Ammal and Ramalakshmi Madabhushi for their incredible moral and emotional support. Vivek and Lakshmi, thank you both for entertaining me and taking care of me during the last phases of my dissertation work.
I would not even have embarked on a PhD program without moral and financial support from my uncles and aunts Sriram Melarcode, Rajam Sriram, Ganesh Melarkode, Uma Ganesh, Veerramani and Radha Veerramani. I am relieved that their thoughtful advice and suggestions have finally paid off into making me a better individual.
Finally, a BIG THANK YOU to the one person who above all has constantly showered me with his love, care and support. I am excited that we can now ride into the future together.

ABSTRACT

Answer set programming (ASP) has emerged as a declarative paradigm for knowledge representation and reasoning. In this approach, a logic program is used to represent the knowledge of the domain and various tasks are reduced to computing answer sets of this program. ASP languages A-Prolog and CR-Prolog have been proven as powerful tools for constructing complex reasoning systems. Constraint logic programming (CLP) emerged as an alternate paradigm through the fusion of logic programming and constraint solving. A CLP solver thus integrates resolution techniques from logic programming and constraint solving techniques from constraint satisfaction. While ASP is expressive for knowledge representation and reasoning, CLP solvers are efficient in reasoning with numerical constraints.

Every state-of-the-art ASP solver computes answer sets of programs from their ground equivalents. Though these systems solve large industrial problems, the ground programs become huge and unmanageable. This is especially true when programs contain variables that range over large numerical domains; huge memory requirements eventually force the solvers to fail.

The goal of this research is to address this issue by integrating different types of reasoning techniques to compute answer sets of programs. First, we investigate the integration of answer set reasoning, a form of abduction, and constraint solving techniques. We design a collection of languages, $\mathcal{V}(\mathcal{C})$, parameterized over a class $\mathcal{C}$ of constraints. An instance $\mathcal{AC}_0$ from this family is studied as a knowledge representation tool. An algorithm to compute answer sets of $\mathcal{AC}_0$ programs is developed. An $\mathcal{AC}_0$ solver is built that computes answer sets from partially ground programs. The use of this language and the efficiency of the solver are demonstrated.

We extend our investigation to develop methods to include resolution techniques. We design a collection of languages $\mathcal{AC}(\mathcal{C})$ parameterized over a class $\mathcal{C}$ of constraints. We develop an algorithm to compute answer sets of $\mathcal{AC}(\mathcal{C})$ programs from their partial ground instances by integrating the four reasoning techniques and prove correctness. A solver is built to compute answer sets for a class of $\mathcal{AC}(\mathcal{C})$ programs.

Our work is a significant step to declaratively solve problems that cannot be solved by pure ASP or CLP solvers. The solvers built are the first to tightly integrate different reasoning techniques to compute answer sets from partial ground programs.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Programming languages can be divided into two main categories, *algorithmic* and *declarative*. Programs in algorithmic languages describe sequences of actions for a computer to perform, while declarative programs can be viewed as collections of statements describing the objects of a domain and their properties. Such sets of statements are often called a *knowledge base*. The semantics of a declarative program Π is normally given by defining its models, i.e., possible states of the world compatible with Π. The work of computing these models, or consequences, is often done by an underlying inference engine. For example, Prolog is a logic programming language that has such an inference engine built into it. The programmer does not have to specify the steps of the computation and can therefore concentrate on the specifications of the problem. It is this separation of logic from control that characterizes declarative programming [48, 69, 64]. Declarative programs need to meet certain requirements. Some of these requirements are[48]:

- The syntax should be simple and there should be a clear definition of the meaning of the program.
- Knowledge bases constructed in this language should be elaboration tolerant. This means that a small change in our knowledge of a domain should result in a small change to our formal knowlegde base[71].
- Inference engines associated with declarative languages should be sufficiently general and efficient. It is often necessary to find a balance between the expressiveness of the language and the desired efficiency.

1.1 Declarative Paradigms

Two declarative programming paradigms of interest in this work are the *Answer Set Programming* (ASP) paradigm and the *Constraint Logic Programming* (CLP) paradigm. ASP paradigm was introduced by Michael Gelfond and Vladimir Lifschitz in 1988 [50] and CLP paradigm was introduced by Joxan Jaffer and Jean-Louis Lassez in 1987 [54]. We briefly describe these two paradigms before we state the goal of this research. Additional information on ASP and CLP with references to literature can be found in chapter 2.

1.1.1 ASP paradigm

Of the many languages in the ASP paradigm, the most studied and used is the declarative language *A-Prolog* [50]. The language has its roots in the research on the semantics of logic programming languages and non-monotonic reasoning [49, 51]. The syntax of *A-Prolog* is similar to Prolog, with some differences in connectives. For instance, unlike Prolog, *A-Prolog* allows disjunctions in the head and classical negation but does not allow the cut (!) operator. The following is a simple example of a program in *A-Prolog*.

Example 1.1.1. *Consider the program* Π *below:*

$$q(a).$$
$$q(b).$$
$$p(X) \leftarrow q(X).$$

Π *consists of three rules defining properties* $\{p,\ q\}$ *of objects* $\{a,\ b\}$. X *is a variable ranging over objects in the program.*

The semantics of *A-Prolog* is given by *answer set semantics*. There are several solvers/ inference engines that compute answer sets of programs written in

A-Prolog [19, 75, 68, 21, 62, 72]. These ASP solvers are quite efficient and have been used for solving complex applications [77, 36, 40, 41]. Computing answer sets of an *A-Prolog* program is a two step process. Given an *A-Prolog* program Π, ASP solvers first ground Π by replacing variables by constants to get a ground program, $ground(\Pi)$; then they use inference techniques to find answer sets of $ground(\Pi)$. The answer sets of $ground(\Pi)$ are the answer sets of the program Π.

Example 1.1.2. *Let* Π *be the program as in example 1.1.1. The ground program,* $ground(\Pi)$ *is:*

$$q(a).$$
$$q(b).$$
$$p(a) \leftarrow q(a).$$
$$p(b) \leftarrow q(b).$$

Program $ground(\Pi)$ *has a single answer set* $S = \{q(a),\ q(b),\ p(a),\ p(b)\}$. *The set* S *is the answer set of* Π.

There can be no answer sets or single answer set or several answer sets for a program. A detail description of the syntax and semantics of *A-Prolog* and ASP solvers can be found in chapter 2.

1.1.1.1 Language CR-Prolog

The knowledge representation language CR-Prolog [3] is an extension of *A-Prolog*. CR-Prolog allows natural encoding of "rare events". These events are normally ignored by a reasoner associated with the program and only used to

restore consistency of the reasoner's beliefs. For instance a program:

$$\neg p(X) \leftarrow not\ p(X).$$
$$q(a) \leftarrow \neg p(a).$$
$$[r(X)]\ :\ p(X) \stackrel{+}{\leftarrow}.$$

consists of two regular rules of ASP and the consistency restoring rule, $[r(X)]$, which says that in some rare cases, $p(X)$ may be true. The rule is ignored in the construction of the answer set $\{\neg p(a), q(a)\}$ of this program. If however the program is expanded by $\neg q(a)$ then the rule $r(a)$ will be used to avoid inconsistency. The answer set of the new program will be $\{p(a), \neg q(a)\}$.
CR-Prolog has been shown to be a useful tool for knowledge representation and reasoning [3, 6]. The language is expressive, and has a well understood methodology inherited from ASP, for representing defaults, causal properties of actions and fluents, various types of incompleteness, etc. In addition, it allows reasoning with complex exceptions to defaults and hence avoids some occasional inconsistencies of *A-Prolog*.
Simple CR-Prolog solvers [62, 4] built on top of the ASP solvers proved to be sufficiently efficient for building industrial size applications related to intelligent planning and diagnostics [6].

1.1.2 CLP Paradigm

Constraint logic programming (CLP) is a merger of two declarative paradigms: constraint solving [54], and logic programming [63, 25]. Constraint logic programming can be said to involve the incorporation of constraints and constraint "solving" methods in logic-based languages. This characterization suggests the possibility of many interesting languages, based on different

constraints and different logics. In general CLP(X) is used to denote a constraint logic programming language with constraint domain X.

The syntax of a typical language of CLP resembles prolog syntax. For instance, the program in example 1.1.2 can be viewed as a program in CLP. CLP languages differ on the type of constraints (relations) that are allowed and the domain in which these constraints operate. Constraints are a set of relations over a given domain, for instance the constraint domain for reals include standard arithmetic relations of the form: $X - Y \leq d$, $a * X + b * Y + c * Z = e$ etc., where X, Y, Z are variables ranging over reals and a, b, c, d, e are real constants.

Prolog can be said to be a CLP language where the constraints are equations over the algebra of terms (also called algebra of finite trees, or the Herbrand Domain) [56]. The CLP program in example below defines the relation $sumto(n, 1 + \cdots + n)$ for natural numbers.

Example 1.1.3. *[55]*

$$sumto(0, 0).$$
$$sumto(N, S) :- N \geq 1,\ N \leq S,\ sumto(N - 1, S - N).$$

The semantics of CLP languages is based on the operational interpretation of constraints rather than declarative interpretation like in logic programming languages. There are several state-of-the-art solvers for CLP languages with constraint domains over real, rational, finite trees and finite domains. Some of the current popular solvers are ECLiPSe (finite domain) [2], GNU Prolog (finite domain) [31], SICStus Prolog's CLP(Q,R) and CLP(FD) solvers (rational, real and finite domain) [18, 20]. There are a wide range of CLP applications from industry, business, manufacturing and science [90, 61, 53, 81, 22, 82, 88].

A CLP solver reads and compiles a user written program and then answers user's

queries (goals in the form of constraints). For instance, the program in example 1.1.3 above can be compiled in CLP(R), (a solver for CLP with real domain) and query $S <= 3, sumto(N, S)$ gives rise to three answers: $(N = 0, S = 0)$, $(N = 1, S = 1)$, and $(N = 2, S = 3)$ and terminates. A brief overview of constraint languages, solver and applications can be found in chapter 2.

1.1.2.1 Language and solver CLP(R)

In this work, we are interested in the language CLP(R) [58] which is an instance of constraint logic programming language because we have the source code of CLP(R) solver. The domain of computation R is the algebraic structure consisting of uninterpreted functors over real numbers. CLP(R) solver [52], reads a user input program in language CLP(R) and answers queries or goals. If a goal is successful i.e., has a solution, then the solver returns primitive $answer$ constraints whose solutions are solutions to the goal.

The CLP(R) solver's inference engine uses resolution techniques for reasoning with non-primitive constraints in the goal while accumulating primitive constraints. The primitive constraints are solved using constraint solving techniques. The underlying constraint solver is incremental and uses different constraint solving techniques based on the classes of constraints solved.

The following sections describe in more detail a motivation for this work and a proposed solution.

1.2 Problem Description

This section describes the problem of grounding *A-Prolog* programs containing numerical constraints with variables ranging over large domains and finding answer sets of such ground programs by ASP solvers. Before addressing the

Locations	Doctor	Home	Office	Atm
Doctor	0	20	30	40
Home	20	0	15	15
Office	30	15	0	20
Atm	40	15	20	0

Table 1.1: Travel time (in minutes) between several locations in Example 1.2.1

problem, let us look at a simple knowledge representation and planning example, involving large numerical domains.

Example 1.2.1. *Ram is at his office and has a dentist appointment in one hour. For the appointment, he needs his insurance card which is at home and cash to pay the doctor. He can get cash from the nearby atm. Table 1.1 shows the time in minutes needed to travel between locations: Doctor, Home, Office and Atm. For example, the time needed to travel between Ram's office to the Atm is 20 minutes. If the available actions are: moving from one location to another and picking items such as cash or insurance card then,*
(a) find a plan which takes Ram to the doctor on time, and
(b) find a plan which takes Ram to the doctor at least 15 minutes early.

The above planning problem can be expressed in *A-Prolog* language using direct effects, indirect effects and executability conditions for actions, along with inertia axiom, a planning module and initial situation (scenario). Declarative description of the problem requires representation of time. One natural way to do this is to use natural numbers from 0 to 1440 (which is the number of minutes in the day). Given a program Π to represent the above problem, classical ASP solvers [62, 72, 76], could not solve the problem. Even though *A-Prolog* is expressive enough to allow for an appropriate and concise representation of this problem, all current ASP solvers compute answer sets of a program from its ground

instantiation and therefore, are not capable of handling representations involving large ground instantiations as this one. Since the problem contains numerical constraints with time variables that range over large domains (here [0..1440]), the ground instantiation is quite huge. ASP solvers can not ground the program due to huge memory requirements and hence can not compute its answer sets.
When programs contain variables that range over large domains then classical methods of ASP solvers cannot be used to solve these problems. We need to find methodologies to compute answer sets without computing the whole ground instantiation. To further understand the significance of this problem, we present a simpler example that contains variables that range over large domains and compare it to the size of its ground instantiation.

Example 1.2.2. *Let* Π *be a program as given below:*

r_1 : $time(0..1440).$

r_2 : $\#domain\ time(T_1;T_2).$

$r_{3,4}$: $q(1).\quad q(2).$

r_5 : $p(X,Y) \leftarrow q(X),\ q(Y),\ X!=Y,\ not\ r(X,Y)$

r_6 : $r(X,Y) \leftarrow q(X),\ q(Y),\ X!=Y,\ not\ p(X,Y)$

r_7 : $\leftarrow r(X,Y),\ at(X,T_1),\ at(Y,T_2),\ T_1-T_2>3$

r_8 : $\leftarrow p(X,Y),\ at(X,T_1),\ at(Y,T_2),\ T_1-T_2>10$

r_9 : $1\ \{\ at(X,T)\ :\ time(T)\ \}\ 1 \leftarrow q(X)$

When we ground the above program using an intelligent grounder, lparse*, we get the number of rules in* $lparse(\Pi) = 8226921$.

The system lparse [98] is a state-of-the-art program used as a front-end to ASP inference engines to ground *A-Prolog* input programs before they can be processed by these engines. Since lparse uses intelligent grounding mechanisms, normally the size of $lparse(\Pi)$ is less than the size of its ground instantiation

$ground(\Pi)$, without losing answer sets of Π. We see in the example that the program with 9 rules has more than 8 million rules in $lparse(\Pi)$. Further notice that rules r_3 to r_6 and r_9 together comprise of only 7 rules in ground program $lparse(\Pi)$. The rules r_1 and r_2 specify the domain of time variables and are used only during grounding. An intelligent grounder would not output their ground instantiations. Therefore, the 8 million rules in the ground instantiation of Π are from rules r_7 and r_8 which contain numerical timing constraints.
It becomes clear that when there are variables that range over large domains the current ASP solvers are inefficient to compute answer sets and most times they fail. We need to investigate methods to compute answer sets of programs without computing the whole ground instantiation. This is one of the goals of this research.

1.3 Goal of this Research

The goal of this research is to investigate the integration of four types of reasoning capabilities:

- ▷ answer set reasoning from A-Prolog,
- ▷ abductive reasoning from CR-Prolog,
- ▷ resolution from CLP,
- ▷ constraint solving mechanisms from CLP.

We propose to achieve this in two steps:

1. Design a collection of languages $\mathcal{AC}(\mathcal{C})$, parameterized over a collection $\mathcal{C}$ of constraints. The set of constraints (relations) in $\mathcal{C}$ defines the difference between the languages in the collection $\mathcal{AC}(\mathcal{C})$. These languages differ from

A-Prolog and CR-Prolog by classifying predicates into different types. This information will be used by an inference engine of $\mathcal{AC}(\mathcal{C})$ to classify rules to different types and therefore apply different reasoning mechanisms to different types of rules.

2. Design an algorithm to compute answer sets of $\mathcal{AC}(\mathcal{C})$ programs. The algorithm would integrate four types of reasoning capabilities: answer sets, abduction, resolution and constraint solving mechanisms to compute answer sets. Further, the algorithm allows for computing answer sets of $\mathcal{AC}(\mathcal{C})$ programs from their partial ground instantiations.

1.3.1 Language $\mathcal{AC}(\mathcal{C})$

The first goal comprises of defining the syntax and semantics of $\mathcal{AC}(\mathcal{C})$. Since *A-Prolog* and CR-Prolog have been proven to be a useful tool for knowledge representation and reasoning [48, 10, 11, 5, 78], for this work, we desire

- ▷ to keep the syntax of $\mathcal{AC}(\mathcal{C})$ close to the syntax of *A-Prolog*, and
- ▷ to keep the semantics of $\mathcal{AC}(\mathcal{C})$ as a natural extension of semantics of *A-Prolog*.

To understand the intuitive reasoning and motivation for this work, we now present a brief introduction to this approach. We begin by giving a brief introduction to the language $\mathcal{AC}(\mathcal{C})$. The syntax and semantics of $\mathcal{AC}(\mathcal{C})$ are described in detail in chapter 3.

Signature of the language $\mathcal{AC}(\mathcal{C})$ is sorted. Predicates in the signature of the language are divided into four types: $regular$, $constraint$, $defined$ and $mixed$. Intuitively, regular predicates define relations with variables that range over small domains. Constraint predicates define primitive numerical relations with variables

ranging over large domains. Defined predicates which define non-primitive relations with variables ranging over large domains. The mixed predicates define relationships between *regular* and *defined*; and *regular* and *constraint* predicates. The atoms formed from these predicates are called regular, constraint, defined and mixed atoms.

The intuitive idea behind classifying predicates is to use different methods of inferencing: *answer set, abduction, resolution and constraint solving* for different types of rules formed from these atoms. Though, we give the formal syntax and semantics of $\mathcal{AC}(\mathcal{C})$ later in chapter 3, let us look at some parts of the example 1.2.1 and how we can represent it in the new language. The full example can be found in chapter 7.

We first represent the regular part of the example which consists of regular atoms and rules formed from these atoms. The representation of time that involves variables with large domains and constraint atoms is described later.

There is only one person *ram* in our example and locations: dentist place, ram's office, ram's home and the atm. We also have two items, insurance card and cash. The atom *step* represented below describes a higher level time frame involved in reasoning with time. A step in the world is a discrete time representation. These steps are used to perform higher intelligent reasonings. The real time is represented by atom *time* whose domain is [0..1440] and will be introduced later in the example along with rules for numerical reasoning. Finally, we will see that a mixed atom is used to eventually associate each discrete step to a real time in the world. To make it simple we only have one action: go_to. There are three fluents: a person P is at location L; an item I is at location L and a person P has item I. The following rules represent these informations.

```
person(ram).
```

```
item(icard).     item(cash).
loc(dentist).    loc(office).
loc(home).       loc(atm).
step(1..4).

% Actions
action(go_to(P,L)) :- person(P), loc(L).

% Fluents
fluent(at_loc(P,L)) :- person(P), loc(L).
fluent(at_loc(I,L)) :- item(I), loc(L).
fluent(has(P,I)) :- person(P), item(I).
```

The direct effect of a person going to a location is represented in the following rule. Note that the atom $next(S0, S1)$ is true when step $S1 = S0 + 1$. The atom $o(go\_to(P, L), S0)$ is read as: *action, person P goes to location L, occurs at step S0.* The atom $h(at\_loc(P, L), S1)$ is read as: *fluent, person P is at location L, holds at step S1.*

```
% If a person P goes to loc L at step S0 then he is at L at step S0+1.
h(at_loc(P,L),S1) :-    person(P), loc(L),
                        next(S0,S1),
                        o(go_to(P,L),S0).
```

The next two rules represent in-direct effects. Instead of introducing a new action *pick an item*, to make the representation simpler, we describe that a person has an item I if he is at the location where the item is. This allows us to represent the

first rule. The second in-direct effect describes the change in location of an item with changes in location of the person who is carrying the item.

```
% If Ram is at loc L and item I is at loc L then he has item I
h(has(P,I),S) :-    step(S),
                    h(at_loc(P,L),S),
                    h(at_loc(I,L),S).

% If a person has an item then the item is at same loc as the person
h(at_loc(I,L),S) :- step(S),
                    h(has(P,I),S),
                    h(at_loc(P,L),S).
```

Next we see how we can represent the constraint and mixed predicates in language $\mathcal{AC}(\mathcal{C})$. In order to represent a constraint predicate and sort, $\mathcal{AC}(\mathcal{C})$ uses the key word #csort. The first rule below is read as: *time is a constraint sort with values ranging from 1 to 1440*. The number 1440 is selected because it is the number of minutes in a day and we use minutes as the smallest measurement of time in this example. In order to represent mixed predicates, $\mathcal{AC}(\mathcal{C})$ uses the key word #mixed. The second rule is read as: *at is a mixed predicate of arity two with first parameter as a step and second parameter as time*. The default predicate type is regular. Therefore, if we do not mention the type of the predicate then it is regular. In this example, time is a constraint predicate, at is a mixed predicate and all others are regular. There are no defined predicates in this example. Recall that a mixed predicate defines relationships between regular and constraint parameters. In this example at is a mixed predicate relating a regular parameter step and a constraint parameter time.

```
#csort time(1..1440).
#mixed at(step,time).
```

Next we show how to represent the timing constraints like *the minimum time taken to travel between the dentist's place and ram's home is 20 minutes.* Using the mixed predicate at, we can assign a real time in minutes for each step. We can read at(S, T) as: *step* S *occurs at time* T. With the assumption that step 1 occurs prior to step 2 etc., we write the first rule which constrains models to assign real time for step increasingly. The other two rules represent the timing constraints. Though we show only two timing constraint rules, we need a rule for each of the timing constraint given in the example.

```
% Timing constraints
% assign times increasingly for steps.
:- next(S1, S2), at(S1,T1),  at(S2,T2), T1 - T2 > 0.

% minimum time for travel between dentist and home is 20 mins
:- h(at_loc(P, home), S1), o(go_to(P, dentist),S1),
    next(S1,S2), at(S1,T1), at(S2,T2), T1 - T2 > -20.

% minimum time for travel between home and atm is 15 minutes
:- h(at_loc(P, home), S1), o(go_to(P, atm),S1),
   next(S1,S2), at(S1,T1), at(S2,T2), T1 - T2 > -15.
```

Finally, we represent the planning module which selects one action for each step; a goal that Ram should be at the dentist with his insurance card and cash; a timing constraint for the goal that he should be at the dentist in sixty minutes; and

initial situation in which Ram is at his office and insurance card is at home and cash is at the atm.

```
%% Planning Module
1 { o(A,S) : action(A) } 1 :- step(S), not goal(S).

goal(S) :- h(at_loc(ram,dentist),S), h(has(ram,icard),S),
           h(has(ram,cash),S).

plan :- goal(S).
:- not plan.

% Problem (a): He should be at the dentist in 60 minutes
:- goal(S), at(0,T1), at(S,T2), T2 - T1 > 60.

% Initial Situation
h(at_loc(ram,office),0).    h(at_loc(icard,home),0).
h(at_loc(cash,atm),0).
```

Along with inertia for fluents and other timing constraints, this represents the formalization of example 1.2.1 in language $\mathcal{AC}(\mathcal{C})$. Notice that there are very few differences between the syntax of this $\mathcal{AC}(\mathcal{C})$ program and *A-Prolog* programs. In fact, we could easily modify this $\mathcal{AC}(\mathcal{C})$ program to a semantically equivalent *A-Prolog* program in a few steps as follows:

- ▷ First, we delete the following rules:

  ```
  #csort time(1..1440).
  #mixed at(step,time).
  ```

▷ and then add the following rules:

```
time(1..1440).
1 { at(S,T) : time(T) } 1 :- step(S).
```

The first *A-Prolog* rule added, states that time is a sort with values ranging from 1 to 1440. The next rule is a choice rule [75], which can be read as: *for each step* s, *select exactly one atom at(s,t), where* t *is of type time*. This rule is not needed for $\mathcal{AC}(\mathcal{C})$ programs as the mixed predicate at acts like a function which assigns a value T for every S. This shows how close the programs in $\mathcal{AC}(\mathcal{C})$ and *A-Prolog* are in syntax. These small differences however allow us to build efficient solvers for $\mathcal{AC}(\mathcal{C})$.

1.3.2 Algorithm *ACsolver*

After designing the language $\mathcal{AC}(\mathcal{C})$, we will address the second goal of this work, which is to design an algorithm *ACsolver* for computing answer sets of $\mathcal{AC}(\mathcal{C})$ programs and prove its correctness. The algorithm integrates different reasoning techniques: *answer set inferencing, abduction, resolution and constraint solving* to compute answer sets of $\mathcal{AC}(\mathcal{C})$ programs. This integration allows for selectively using a particular reasoning technique which is best suitable for the kind of reasoning needed. For instance, solving a set of numerical constraints using constraint solving mechanisms.

It was also clear in section 1.2, that when there are numerical constraints and variables that range over large domains, ASP solvers could not compute answer sets. The solvers were computing answer sets from ground instantiations. When the ground instantiations become huge and unmanageable, the solvers failed.

An interesting area for research is to look for algorithms that can compute answer sets without grounding the program. Our current work is a step towards this

ultimate goal. We design an algorithm to compute answer sets of $\mathcal{AC}(\mathcal{C})$ programs from their partial ground instantiations. For this, we develop a partial grounder which takes as input an $\mathcal{AC}(\mathcal{C})$ program Π and returns a partially ground program $\mathcal{P}(\Pi)$. The program $\mathcal{P}(\Pi)$ is used to compute answer sets of Π.

The partial grounder uses the classification of predicates in $\mathcal{AC}(\mathcal{C})$ program Π to partially ground Π. The chapter 4 describes in detail the partial grounding of $\mathcal{AC}(\mathcal{C})$ programs. The algorithm for partial grounder classifies each rule to a particular type and grounds the rules with the use of this classification. In slightly modified terms, we can think that the partial grounder grounds the regular atoms in a rule and leaves the constraint and defined atoms non-ground. The mixed atoms are ground partially that is, only the regular terms in a mixed atom are ground and the constraint terms are non-ground.

Clearly, a partially ground program is smaller than a fully ground instantiation. It is more interesting to see how small the size of partial ground program $\mathcal{P}(\Pi)$ is when compared to the fully ground instantiation $\mathrm{ground}(\Pi)$. For this let us use the simple example 1.2.2.

Example 1.3.1. *Let Π be the ASP program from example 1.2.2. Then an equivalent program Π' for Π in $\mathcal{AC}(\mathcal{C})$ is as follows:*

$\#csort\ \mathrm{time}(0..1440).$

$\#mixed\ \mathrm{at}(\mathrm{q},\ \mathrm{time}).$

$\mathrm{q}(1).\quad \mathrm{q}(2).$

$\mathrm{p}(X,Y)\ \leftarrow\ \mathrm{q}(X),\ \mathrm{q}(Y),\ X!=Y,\ not\ \mathrm{r}(X,Y)$

$\mathrm{r}(X,Y)\ \leftarrow\ \mathrm{q}(X),\ \mathrm{q}(Y),\ X!=Y,\ not\ \mathrm{p}(X,Y)$

$\leftarrow\ \mathrm{r}(X,Y),\ \mathrm{at}(X,T_1),\ \mathrm{at}(Y,T_2),\ T_1-T_2>3$

$\leftarrow\ \mathrm{p}(X,Y),\ \mathrm{at}(X,T_1),\ \mathrm{at}(Y,T_2),\ T_1-T_2>10$

Now let us look at the rules in $\mathcal{P}(\Pi')$:

$\#csort\ \mathrm{time}(0..1440).$

$q(1). \quad q(2).$

$p(1,2) \leftarrow not\ r(1,2)$

$p(2,1) \leftarrow not\ r(2,1)$

$r(1,2) \leftarrow not\ p(1,2)$

$r(2,1) \leftarrow not\ p(2,1)$

$\leftarrow\ r(1,2),\ at(1,T_1),\ at(2,T_2),\ T_1 - T_2 > 3$

$\leftarrow\ r(2,1),\ at(2,T_2),\ at(1,T_1),\ T_2 - T_1 > 3$

$\leftarrow\ r(1,1),\ at(1,T_1),\ at(1,T_1),\ T_1 - T_1 > 3$

$\leftarrow\ r(2,2),\ at(2,T_2),\ at(2,T_2),\ T_2 - T_2 > 3$

$\leftarrow\ p(1,2),\ at(1,T_1),\ at(2,T_2),\ T_1 - T_2 > 10$

$\leftarrow\ p(2,1),\ at(2,T_2),\ at(1,T_1),\ T_2 - T_1 > 10$

$\leftarrow\ p(1,1),\ at(1,T_1),\ at(1,T_1),\ T_1 - T_1 > 10$

$\leftarrow\ p(2,2),\ at(2,T_2),\ at(2,T_2),\ T_2 - T_2 > 10$

The number of rules in $\mathcal{P}(\Pi') = 14$.

The example above shows that the partial grounding of a program has 14 rules and is much smaller than the whole ground instantiation which has 8 million rules as seen in example 1.2.2.

The solver $\mathcal{AC}solver$ mainly consists of two parts.

- ▷ Partial Grounder $\mathcal{P}\mathrm{ground}$
- ▷ Inference Engine $\mathcal{AC}engine$

As described earlier, the input of $\mathcal{P}\mathrm{ground}$ is a $\mathcal{AC}(C)$ program and the output is a partially ground program $\mathcal{P}(\Pi)$. The correctness of the partial grounder $\mathcal{P}\mathrm{ground}$ is shown by proving the following proposition.

Proposition: *Answer sets of* Π *are same as answer sets of* $\mathcal{P}(\Pi)$.

Instead of using a naive grounder, we would like to use an intelligent grounder like $lparse$ [99], in our partial grounder to ground the regular terms. This makes the proof of correctness of $\mathcal{P}ground$ non-trivial.

The inference engine $\mathcal{AC}engine$ computes *simplified answer sets* of a $\mathcal{AC}(\mathcal{C})$ program Π from a partially ground program $\mathcal{P}(\Pi)$. A simplified answer set M of a program Π is a subset of an answer set M' of Π, more precisely, M contains all the regular atoms and mixed atoms of M'. The soundness of the inference engine is proved using the following proposition.

Proposition: *If* $\mathcal{AC}engine$ *returns* true *and a set* M*, then* M *is a simplified answer set of* Π.

The engine $\mathcal{AC}engine$ tightly couples the reasoning capabilities of a ASP solver (answer set and abduction reasoning techniques) and a constraint logic programming (CLP) solver (resolution and constraint solving techniques). At a higher level, we can describe that the inferences of $regular$ part are computed by ASP solver; the inferences of $consistency\ restoring$ part are computed by abductive reasoning techniques; the inferences of $defined$ and $constraint$ parts are computed by resolution and constraint solving mechanisms of CLP solver and finally the inferences of the mixed part are computed by communications between ASP and CLP solvers. The $\mathcal{AC}solver$ is implemented by integrating an existing CR-Prolog solver with ASP solver $Surya$ and CLP solver CLP(R).

1.3.3 Language $\mathcal{AC}(\mathcal{C})_{\mathrm{cr}}$

Language $\mathcal{AC}(\mathcal{C})_{\mathrm{cr}}$ is a natural extension of language $\mathcal{AC}(\mathcal{C})$ with consistency restoring capabilities of language CR-Prolog. The syntax of language $\mathcal{AC}(\mathcal{C})_{\mathrm{cr}}$ is a natural extension: allowing consistency restoring rules of CR-Prolog in $\mathcal{AC}(\mathcal{C})$.

The semantics of $\mathcal{AC}(\mathcal{C})_{\text{cr}}$ naturally extends the semantics of $\mathcal{AC}(\mathcal{C})$ using abductive reasoning of CR-Prolog.

Description of syntax and semantics of this language is out of scope of this work but we describe the syntax and semantics of a special case called $\mathcal{V}(\mathcal{C})$. We also study an instance $\mathcal{AC}_0$ of $\mathcal{V}(\mathcal{C})$ in this work. We use an existing implementation of a CR-Prolog solver in our integration to allow for abductive reasoning capabilities used in language $\mathcal{AC}_0$.

1.3.4 Language $\mathcal{V}(\mathcal{C})$

Next we introduce a collection of knowledge representation languages, $\mathcal{V}(\mathcal{C})$, parametrised over a class $\mathcal{C}$ of constraints. $\mathcal{V}(\mathcal{C})$ is a special case of $\mathcal{AC}(\mathcal{C})_{\text{cr}}$ with the following properties:

- ▷ there are no defined predicates in $\mathcal{V}(\mathcal{C})$, and
- ▷ if a rule r contains mixed predicates in the body then head of r is empty.

Note that the previous examples presented are examples of programs in $\mathcal{V}(\mathcal{C})$. $\mathcal{V}(\mathcal{C})$ is an extension to language CASP [12], with the introduction of consistency restoring capabilities from CR-Prolog. $\mathcal{V}(\mathcal{C})$ allows the separation of a program into two parts: a regular program of CR-Prolog and a collection of denials[1] whose bodies contain constraints from $\mathcal{C}$ with variables ranging over large domains. We study an instance $\mathcal{AC}_0$ from this family where $\mathcal{C}$ is a collection of constraints of the form $X - Y > K$, where X and Y are variables and K is any number. A solver for any language of $\mathcal{V}(\mathcal{C})$ can be built using answer set reasoning, abductive and constraint solving techniques. The syntax and semantics of $\mathcal{V}(\mathcal{C})$ is presented in chapter 6.

[1] By a denial we mean a logic programming rule with an empty head.

1.3.5 Language $\mathcal{AC}_0$

Our next goal is to study an instance $\mathcal{AC}_0$ of $\mathcal{V}(\mathcal{C})$ where $\mathcal{C}$ is a collection of constraints of the form $X - Y > k$, where X and Y are variables and k is a real number. Note that the examples presented so far are examples of $\mathcal{AC}_0$. In general, a program of $\mathcal{AC}_0$ consists of two modules:

- ▷ regular rules of ASP and consistency restoring rules of *CR-Prolog*;
- ▷ denials with constraints of type $X - Y > k$.

The variables of a $\mathcal{AC}_0$ program Π that occur in constraints are called constraint variables and the rest are regular variables. The semantics of $\mathcal{AC}_0$ described in chapter 6 extends ASP, CR-Prolog and CASP semantics in a natural way.
The interest in studying this language comes from the fact that, in addition to standard power of ASP, and CR-Prolog, $\mathcal{AC}_0$ allows natural encoding of constraints and consistency restoring rules, including

- ▷ *simple temporal constraints* like "John needs to be at the office in two hours",
- ▷ *disjunctive temporal constraints* like "action a can occur either between 8-9 am or between 4-5 pm",
- ▷ *qualitative soft temporal constraints* like "It is desirable for John to come to office 15 mins early"
- ▷ *disjunctive qualitative soft temporal constraints* like "It is desirable for action a to occur either between 2-4 pm or between 6-8 pm".

These temporal constraints are used widely in planning and scheduling applications [83, 44, 43, 84]. Further, the constraint literals in a $\mathcal{AC}_0$ program

occur only in the body of denials. This allows us to use simple constraint satisfaction algorithms instead of using a complex constraint logic programming algorithms (resolution techniques are not required). Also, note that the only constraint literals allowed are of the type $X - Y > k$. This allows us to use difference constraint satisfaction algorithms which are simpler to build than a general constraint solver. For this language, we can build a system tightly coupling an CR-Prolog inference engine and a simple difference constraint solver.

1.3.6 Solver $\mathcal{AD}solver$

An algorithm to compute answer sets of programs in $\mathcal{AC}_0$, integrates

1. Davis-Putnam type algorithm for computing answer sets of ASP programs [27],
2. the form of abduction from *CR-Prolog* [62], and
3. incremental constraint satisfaction algorithm for difference constraints [85].

We build a solver $\mathcal{AD}solver$ to compute answer sets of programs in $\mathcal{AC}_0$. $\mathcal{AD}solver$ consists of two parts:

(a). Partial grounder $\mathcal{P}\mathrm{ground}$.

- $\mathcal{P}\mathrm{ground}$ uses a modified version of lparse to ground regular variables of input program Π of $\mathcal{AC}_0$.
- The output program $\mathcal{P}(\Pi)$ is much smaller when compared to $\mathrm{lparse}(\Pi)$.
- **Proposition:** Answer sets of Π are same as answer sets of $\mathcal{P}(\Pi)$.

(b). Inference engine $\mathcal{AD}engine$.

- $\mathcal{AD}engine$ combines ASP, $CR\text{-}Prolog$ and difference constraint solving methods to compute answer sets of program $\mathcal{P}(\Pi)$ of $\mathcal{AC}_0$.

We build the inference engine $\mathcal{AD}engine$ that integrates a CR-Prolog engine with a difference constraint solver $\mathcal{D}solver$. The algorithm [85] was used to build $\mathcal{D}solver$. The CR-Prolog engine has a underlying ASP engine. The ASP engine was modified to tightly couple $\mathcal{D}solver$ and allows for the interaction of ASP reasoning with constraint solving techniques to compute answer sets of $\mathcal{AC}_0$. The implemented $\mathcal{D}solver$, computes a solution for given a set of difference constraints D of the form $X - Y \leq k$. $\mathcal{D}solver$ is an incremental constraint solver: given a set of constraints D, a solution S to D and a new constraint $X - Y \leq k$, $\mathcal{D}solver$ uses S to compute a solution for $D \cup \{X - Y \leq k\}$. Recall that the ASP reasoning techniques use Davis-Putnam strategy to compute answer sets. To tightly couple such a solver with a constraint solver, it is important (for efficiency sake) that the constraint solver is incremental and backtrackable. $\mathcal{D}solver$ built is incremental and backtrackable.

To illustrate the efficiency of $\mathcal{AD}solver$, consider the representation of example 1.2.1 shown before. Let us call the program Π. The partial ground program $\mathcal{P}(\Pi)$ has 610 rules and $\mathcal{AD}solver$ took 0.17 seconds to solve both problems in 1.2.1. Let us translate Π to an equivalent ASP program Π_a as shown before. The grounder $lparse$ could not ground the program Π_a. If constraint variables range from [0..100] instead of [0..1440], then $lparse$ could ground the program and its ground instantiation consists of 366060 rules. Using ASP solvers, it takes 183.18 and 161.20 seconds to solve first and second problems of example 1.2.1 respectively. Since $\mathcal{AD}solver$ uses a difference constraint solver, the domain size of constraint variables does not affect the efficiency of $\mathcal{AD}solver$. That is, $\mathcal{AD}solver$ solves both problems of 1.2.1 in 0.17 seconds independent of whether the constraint

variables range from [0..100] or [0..1440].

1.4 Summary

The contribution of this dissertation is tabulated as follows:

1. Develop a collection of languages $\mathcal{AC}(\mathcal{C})$ parameterized over $\mathcal{C}$.
   - ▷ syntax of $\mathcal{AC}(\mathcal{C})$ is an extension to syntax of *A-Prolog*.
   - ▷ semantics of $\mathcal{AC}(\mathcal{C})$ is a natural extension of semantics of *A-Prolog*.
2. Design an algorithm $\mathcal{AC}$*solver*, to compute answer sets of programs in $\mathcal{AC}(\mathcal{C})$ and prove its correctness.
   - ▷ partial grounder uses intelligent grounding mechanisms.
   - ▷ the algorithm tightly couples *ASP*, *CR-Prolog* and CLP's resolution and constraint solving techniques.
3. Implement $\mathcal{AC}$*solver* for a particular $\mathcal{C}$.
   - ▷ CLP(R)
4. Design a collection of languages $\mathcal{V}(\mathcal{C})$ parameterized over $\mathcal{C}$. These languages allow consistency restoring capabilities of *CR-Prolog* [3].
5. Study an instance of $\mathcal{V}(\mathcal{C})$ called $\mathcal{AC}_0$. The study concentrates on knowledge representation methodologies and reasoning capabilities of $\mathcal{AC}_0$.
6. Design an algorithm $\mathcal{AD}$*solver*, to compute answer sets of programs in $\mathcal{AC}_0$ and prove correctness.
   - ▷ partial grounder uses intelligent grounding mechanisms.

- ▷ the algorithm tightly couples *ASP*, *CR-Prolog* and difference constraint solving techniques.
- ▷ difference constraint solving is incremental.

7. Implement $\mathcal{AD}solver$.
8. Show efficiency of $\mathcal{AD}solver$ over classical *ASP* and *CR-Prolog* solvers for planning with temporal constraints.

This dissertation is organized in the following manner. Chapter 2 gives the necessary back ground information for ASP and CLP paradigms useful to understand the subject. Chapter 3 presents the syntax and semantics of the language $\mathcal{AC}(\mathcal{C})$. Chapter 4 presents the algorithm for partial grounder $\mathcal{P}\mathrm{ground}$. Chapter 5 describes the $\mathcal{AC}solver$ algorithm. Chapter 6 describes the language $\mathcal{AC}_0$ and knowledge representation methodologies. Chapter 7 gives examples of the use of the languages $\mathcal{AC}_0$ and $\mathcal{AC}(\mathcal{C})$ and efficiency results of $\mathcal{AD}solver$ and $\mathcal{AC}solver$. Chapter 8 presents the proofs of all the propositions in the work. Chapter 9 briefly describes other works related to this research. Finally, Chapter 10 give conclusions and future work.

CHAPTER 2

BACKGROUND

This chapter reviews two declarative programming paradigms:

- ▷ Answer Set Programming
- ▷ Constraint Logic Programming

2.1 Answer Set Programming Paradigm

A-Prolog is a knowledge representation language with roots in the research on the semantics of logic programming languages and non-monotonic reasoning [49, 51]. Language *A-Prolog* is based on the answer set semantics [50, 49].
A-Prolog has been shown to be a useful tool for knowledge representation and reasoning [3, 6]. The language is expressive, and has a well understood methodology for representing defaults, causal properties of actions and fluents, various types of incompleteness, etc. There are several derivatives and expansions of *A-Prolog*. The following section describes the *A-Prolog* language that is most generally used.

2.1.1 Language *A-Prolog*

The syntax of A-Prolog is determined by a typed signature Σ consisting of types, typed object constants, typed variables, typed function symbols and typed predicate symbols. We assume that the signature contains symbols for integers and for the standard functions and relations of arithmetic. Terms are built as in first-order languages. By simple arithmetic terms of Σ we mean its integer constants, that is $2 + 3$ and 3 are terms and 3 is a simple arithmetic term.

Atoms are expressions of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol with arity n and t_i's are terms of suitable types. Atoms formed by arithmetic relations are called arithmetic atoms. Atoms formed by non-arithmetic relations are called plain atoms. We allow arithmetic terms and atoms to be written in notations other than prefix notation, according to the way they are traditionally written in arithmetic (e.g. we write $3 = 1 + 2$ instead of $=(3, +(1, 2))$).
Literals are atoms and negated atoms, i.e. expressions of the form $\neg p(t_1, \ldots, t_n)$. Literals $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ are called complementary. By $\bar{l}$ we denote the literal complementary to l.

Definition 2.1.1. *A rule* r *of A-Prolog is a statement of the form:*

$$h_1 \ or \ \ h_2 \ or \ \ldots or \ \ h_k \leftarrow l_1, \ \ldots l_m, \ not \ l_{m+1}, \ \ldots, not \ l_n. \tag{2.1}$$

where $l_1, \ldots, l_m$ *are literals, and* h_i*'s and* $l_{m+1}, \ldots, l_n$ *are plain literals. We call* $h_1 \ or \ \ h_2 \ or \ \ldots or \ \ h_k$ *the head of the rule (head (r));*
$l_1, \ldots l_m, \ not \ l_{m+1}, \ \ldots, not \ l_n$ *is its body (*$body(r)$*), and* $pos(r)$, $neg(r)$ *denote, respectively,* $\{l_1, \ldots, l_m\}$ *and* $\{l_{m+1}, \ldots, l_n\}$.

The informal reading of the rule (in terms of the reasoning of a rational agent about its own beliefs) is "if you believe $l_1, \ldots, l_m$ and have no reason to believe $l_{m+1}, \ldots, l_n$, then believe one of $h_1, \ldots, h_k$." The connective "not" is called default negation.
A rule such that $k = 0$ is called denial, and is considered a shorthand of:

$$\bot \ \leftarrow \ not \ \bot, \ l_1, \ldots, l_m, \ not \ l_{m+1}, \ldots, not \ l_n.$$

Definition 2.1.2. *An A-Prolog program is a pair* $\langle \Sigma, \Pi \rangle$*, where* Σ *is a signature and* Π *is a set of rules.*

Whenever possible, we denote programs by their second element. The corresponding signature is denoted by $\Sigma(\Pi)$. The terms, atoms and literals of a program Π are denoted respectively by $terms(\Pi)$, $atoms(\Pi)$ and $literals(\Pi)$.
Rules containing variables (denoted by capital letters) are viewed as shorthands for the set of their ground instantiations, obtained by substituting the variables with all the terms of appropriate type from the signature of the program. The semantics of a program of *A-Prolog* is defined over its ground instantiation.
The semantics of *A-Prolog* is defined in two steps. The first step consists in giving the semantics of programs that do not contain default negation. We will begin by introducing some terminology.
An atom is in normal form if it is an arithmetic atom or if it is a plain atom and its arguments are either non-arithmetic terms or simple arithmetic terms. Notice that atoms that are not in normal form can be mapped into atoms in normal form by applying the standard rules of arithmetic. For example, $p(4+1)$ is mapped into $p(5)$. For this reason, in the following definition of the semantics of basic *A-Prolog*, we assume that all literals are in normal form unless otherwise stated.
A literal l is satisfied by a consistent set of plain literals S (denoted by $S \models l$) if:

- l is an arithmetic literal and is true according to the standard arithmetic inter- pretation;

- l is a plain literal and $l \in S$.

If l is not satisfied by S, we write $S \not\models l$. An expression $not\ l$, where l is a plain literal, is satisfied by S if $S \not\models l$. A set of literals is satisfied by S if each element of the set is satisfied by S.
We say that a consistent set of plain literals S *is closed under a program* Π *not*

containing default negation if, for every rule

$$h_1 \text{ or } \ldots \text{ or } h_k \leftarrow l_1, \ldots, l_m$$

of Π such that the body of the rule is satisfied by S, we have $\{h_1, \ldots, h_k\} \cap S \neq \emptyset$.

Definition 2.1.3. [**Answer Set of a program without default negation**] *A consistent set of plain literals, S, is an answer set of a program Π not containing default negation if S is minimally (set theoretic) closed under all the rules of Π.*

Programs without default negation and whose rules have at most one literal in the head are called definite. It can be shown that definite programs have at most one answer set [80, 100].

The second step of the definition of the semantics consists in reducing the computation of answer sets of *A-Prolog* programs to the computation of the answer sets of programs without default negation, as follows.

Definition 2.1.4. [**Reduct of a *A-Prolog* program**] *Let Π be an arbitrary A-Prolog program. For any set S of ground plain literals, let Π^S be the program obtained from Π by deleting:*

- *each rule, r, such that $\mathrm{neg}(r) \cap S \neq \emptyset$;*
- *all formulas of the form not l in the bodies of the remaining rules.*

Definition 2.1.5. [**Answer Set of a *A-Prolog* program**] *A set of plain literals, S, is an answer set of a A-Prolog program Π if it is an answer set of Π^S.*

2.1.2 ASP solvers

This section describes the solvers for computing answer sets of *A-Prolog* programs and describes briefly a standard algorithm for computing answer sets by *A-Prolog*

solvers.

The *smodels* algorithm [76] is a standard algorithm used for the computation of answer sets or stable models of a program in *A-Prolog*. The algorithm is based on $Davis - Putnam$ [27] strategy for computing answer sets of a program. Though *A-Prolog* programs allow to express disjunctions in the head, *smodels* algorithm does not allow disjunctions in the head. System DLV [19], allows *A-Prolog* programs with disjunctions in the head. The *Smodels* system is one of the state-of-the-art implementations of the *smodels* algorithm. The system has a two level architecture. The frontend called *lparse* [98], takes a program with variables and returns a ground program by replacing all variables by constants in the program.

Example 2.1.1. *The grounding of program* Π *shown in example 1.1.1 would result in the program:*

$$q(a).$$
$$q(b).$$
$$p(a) :- q(a).$$
$$p(b) :- q(b).$$

In reality, *lparse* does an intelligent grounding of the program, which is more than just replacing variables by constants. Therefore, the output of *lparse* is a ground program which is much smaller in size when compared to regular grounding. The second part of the *Smodels* system is the inference engine *smodels*. It takes the ground program output by *lparse* and computes the answer sets of the program. The *smodels* inference engine [92, 72] uses two methods of inference to compute the stable models or answer sets of a program. The first is called *lower closure* computation and second is called *upper closure* computation. Next, we briefly describe these two closures. The lower closure computation is based on a set of

inference rules. There are four inference rules in the smodels algorithm. Given a set S of ground literals and a program Π,

1. If the body of a rule r, in Π, is satisfied by S then add the head of r to S.
2. If an atom a is not in the head of any rule in Π, then $not\ a$ can be added to S.
3. If r is the only rule of Π with h in the head and $h \in S$ then the literals in the body of r can be added to S.
4. If h is in the head of rule r, in Π, $not\ h \in S$, and all literals in the body of r except l_i belong to S, then add $not\ l_i$ to S.

Consider the following example which demonstrates the use of inference rule #4.

Example 2.1.2. *Let* $S = \{a, b, not\ h\}$ *be a set of literals and rule* r *be of the form,*

$$h \leftarrow a, b, c.$$

Since literal not $h \in S$*, and literals* a *and* b *are true in* S*; by the inference rule #4 we can conclude not* c *and add it to* S*.*

The lower closure of a program Π with respect to a set of literals S is defined as the minimal set of literals that satisfy the four inference rules.
Let S be a set of literals and Π be a program. By $\alpha(\Pi, S)$ we denote a definite program obtained from Π by

1. Removing all rules in Π whose head or body are not satisfied by S.
2. Removing from the result of step one, all not-atoms from the bodies of rules.

The **upper closure** of a program Π with respect to S, denoted as $up(\Pi, S)$, is defined as the minimal set closed under the rules of $\alpha(\Pi, S)$. The set of atoms in $up(\Pi, S)$ comprises of all possible atoms that may be added to S during answer set computation. If an atom a does not belong to $up(\Pi, S)$ then it follows that for any answer set M that agrees with S, $a \notin M$. Therefore, we can add literal $not\ a$ to S.

Example 2.1.3. *Let* $S = \{\ \}$ *be an empty set of literals and program* Π *be as follows:*

$$a \leftarrow b.$$

$$b \leftarrow a.$$

$$c \leftarrow not\ a.$$

The upper closure of Π *with respect to* S *is* $U = \{c\}$. *Since atoms* a *and* b *do not belong to* U, *we can conclude not* a *and not* b *and add it to* S.

The lower closure and upper closure computations together form the $expand$ procedure in $smodels$ algorithm. The procedure $expand$ computes the set of consequences of Π with respect to S. Procedure $expand$ repeatedly computes $lower$ and $upper$ closures until no new literals can be added to S.
A simplified description of $smodels$ algorithm is as follows. The $smodels$ algorithm accepts as input a ground program Π of *A-Prolog* and a set of literals S. The output of $smodels$ is $true$ if there exists an answer set of Π agreeing with S; otherwise it returns $false$. If it returns $true$ then it also returns an answer set of Π agreeing with S. The algorithm has the following main steps:

1. Procedure $expand$ computes the set of consequences of Π given S and adds them to S.

2. If set S is not consistent then there exists no answer sets of Π agreeing with S and the algorithm returns $false$.

3. If S covers all atoms of Π then $atoms(S)$ is an answer set of Π. The algorithm returns $true$ and $atoms(S)$.

4. Otherwise, following Davis-Putnam strategy [27], the algorithm guesses a literal l, and calls itself recursively with inputs Π and $S \cup \{l\}$.

5. If it cannot find any answer sets of Π agreeing with $S \cup \{l\}$ then it calls itself recursively with inputs Π and $S \cup \{not\ l\}$.

This algorithm is the basis for many solvers [76, 72, 4] built for computing answer sets of program in *A-Prolog*. The inference methods have been studied well [76, 92] with respect to efficient implementations. Several techniques [92, 76, 72] have been adopted to improve the efficiency of the solvers like look-ahead, heuristics etc.

There are currently several inference engines for computing the answer sets of *A-Prolog* programs. Some of them are Smodels [92], Surya [72], DLV [19], Cmodels [35, 66], ASSAT [68] etc. The efficiency of these engines has led to some important applications including the use of Smodels, in the development of a decision support system for the space shuttle [77]. Other important applications are wire routing and satisfiability planning [40], encoding planning problems [32] , applications in product configuration [94], data integration application INFOMIX [65], and checking medical invoices for health insurance companies [59] etc. The next section describes some of the applications.

2.1.3 ASP Applications

One of the first complex dynamic ASP application built is a modeling and reasoning system *USA-Advisor* [77]. USA-Advisor can be viewed as a decision support system for the Reaction Control System (RCS) of the space shuttle. A

full description of the system can be found in Monica Nogueira's PhD thesis: *Building Knowledge Systems in A-Prolog* [77]. For extensions and investigations of the system refer [5, 78, 79, 7, 8, 6]. Now, we give a brief description of the system (from [6]) as we use it to test the solvers built in this work.

The RCS is the Shuttle's system that 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 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, in that it includes 12 tanks, 44 jets, 66 valves, 33 switches, and around 160 computer commands.

When an orbital maneuver is required, the astronauts must configure the RCS accordingly. This involves changing the positions of several switches, which are used to open or close valves or to energize the proper circuitry. Some of these switches may be faulty and valves may be leaky. This makes the pre-scripted sequence of actions inapplicable and further complicates the challenge of coming up with plans to achieve the desired results without causing any possibly dangerous side effects.

USA-Advisor can be viewed as a part of a decision support system for Shuttle flight controllers. It is an intelligent system capable of verifying and generating plans that prepare the RCS for a given maneuver. It can be used when the flight controllers have to come up with a plan for an emergency situation. It can also be used off-line to pre-determine the plans for possible fault conditions.

Other ASP applications include *wire routing* and *phylogeny re-construction*. Wire routing is the problem of determining the physical locations of all the wires interconnecting the circuit components on a chip. Some of the wires cannot

intersect with each other, they are competing for limited spaces, thus making routing a difficult combinatorial optimization problem. More details on wire routing application can be found at [40, 36, 42].

Phylogeny re-construction is a problem of constructing and labeling an evolutionary tree for a set of taxa (a taxanomic group of any rank such as species, family or class), which describes the evolution of the taxa in that set from their most common ancestors. More details of the application can be found at [42, 38, 39, 17, 16]. Other applications can be found at the *WASP: Working group on Answer Set Programming* website at [101, 45].

2.2 Constraint Logic Programming Paradigm

This section describes Constraint Logic Programming paradigm [58, 55, 70].

Constraint Logic Programming (CLP) began as a natural merger of two declarative paradigms: constraint solving and logic programming. This combination helps make CLP programs both expressive and flexible.

Constraint logic programming can be said to involve the incorporation of constraints and constraint "solving" methods in logic-based language. This characterization suggests the possibility of many interesting languages, based on different constraints and different logics.

Prolog can be said to be a CLP language where the constraints are equations over the algebra of terms (the Herbrand domain). The next sections describe the syntax, semantics of CLP languages, CLP solvers and applications.

2.2.1 Language of CLP

The syntax of a program in CLP is defined using a *signature* and *constraint domain* pair $\langle \Sigma, \mathcal{C} \rangle$. The signature Σ, defines the allowed set of typed function and

predicate symbols and associates an arity with each symbol. The constraint domain $\mathcal{C}$ defines the legitimate forms of constraints in the language.
The constraint domain specifies the syntax of the constraints. $\mathcal{C}$ gives the allowed constants, functions and constraint relations and their arity. The constraint domain also determines the values that variables can take.
For example, in the constraint domain of real numbers, which we call $\mathcal{R}$, variables take real number values. The set of function symbols for this constraint domain is $+, \times, -, /$, while the set of constants is all the floating point numbers. The constraint relation symbols are $=, <, \leq, >, \geq$, all of which take two arguments.
The constraint domain of language CLP(R) [58, 52] is the constraint domain of real numbers. Some CLP languages use the domain(s) as part of their names. For example CLP(R) for real domain, CLP(FT) for domain of finite trees, CLP(Q) for rational domain, CLP(FD) for finite domain and so on.
Given a constraint domain $\mathcal{C}$, the simplest form of constraint we can define is a primitive constraint. A primitive constraint consists of a constraint relation symbol from $\mathcal{C}$ together with appropriate number of arguments. These are constructed from the constants, functions of $\mathcal{C}$ and variables. The constraints $X > 3$ and $X + Y \times Y - 3 \times Z = 5$ are examples of primitive constraints of reals.
More complicated constraints can be built from primitive constraints by using the conjunctive connective $\wedge$ which stands for "and".

Definition 2.2.1. *A constraint is of the form* $c_1 \wedge \cdots \wedge c_n$ *where* $n \geq 0$ *and* $c_1, \ldots, c_n$ *are primitive constraints.*

The symbol $\wedge$ denotes and, so a constraints $c_1 \wedge \cdots \wedge c_n$ holds whenever all of the primitive constraints $c_1 \ldots c_n$ hold. There are two distinct constraints $true$ and $false$. The constraint $true$ always holds while $false$ never holds. The empty conjunction of constraints (when $n = 0$) is written as $true$.

Given a constraint, we can determine values of variables for which the constraint holds. By replacing variables by a value substitution θ, the constraint becomes variable-free and they can be evaluated and seen to be either true or false. If true then the substitution θ is a solution to the constraint.
An atom has the form $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and p is a n-ary predicate symbol from Σ.

Definition 2.2.2. *A CLP program is a collection of rules of the form:*

$$a :- b_1, \ldots, b_n. \tag{2.2}$$

where a *is an atom and* b_i*'s are either atoms or constraints.*

The atom a in rule 2.2 is called the head of the rule and $b_1, \ldots, b_n$ is called the body of the rule. A fact is a rule $a :- c$ where c is a constraint. A goal (or query) is a conjunction of constraints and atoms. The following is a CLP(R) program computing fibonacci numbers.

Example 2.2.1. *[58] The example defines a relation* $\mathrm{fib}(A, B)$ *which can be read as: "*A^{th} *fibonacci number is* B*".*

$$\mathrm{fib}(0, 1).$$
$$\mathrm{fib}(1, 1).$$
$$\mathrm{fib}(N, X_1 + X_2) :- N > 1,\ \mathrm{fib}(N-1, X_1),\ \mathrm{fib}(N-2, X_2).$$

A goal (query) which asks for a number A *such that* $\mathrm{fib}(A)$ *lies between 80 and 90 is*

$$?- 80 \leq B,\ B \leq 90,\ \mathrm{fib}(A, B)$$

The CLP languages use an operational semantics to give meaning to the programs. Next, we describe briefly the operational semantics of the language

CLP(R). For a detailed description of operational semantics of a generalized CLP language refer to [55, 56], for CLP(R) refer to [58].

The operational semantics of CLP(R) is presented using a transition system of *states*: $\langle A, C\rangle$, where A is a multi-set of atoms and constraints, and C is a multi-set of constraints. There is also a special state denoted by $fail$.

The set C is referred to as a constraint store. The constraints of C are divided into two categories: $solved$ and $delayed$. The solved constraints are simple linear constraints that can be checked for solvability. The delayed constraints are non-linear constraints and maybe difficult to check for solvability. The constraints in C are *solvable* if the set of solved constraints in C is consistent i.e. has a solution.

Given a program Π, a query $Q = l_1, \ldots, l_n$, and a constraint store C, there is a *derivation step* from a pair $\langle Q, C\rangle$ to another pair $\langle Q_1, C_1\rangle$ if:

($\rightarrow_c$) $Q_1 = l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n$, where l_i is a constraint and C_1 is a (possibly simplified) set of constraints equivalent to $C \cup \{l_i\}$. Furthermore, C_1 is solvable;

($\rightarrow_r$) or $Q_1 = l_1, \ldots, l_{i-1}, x_1 = t_1, \ldots, x_k = t_k, b_1, \ldots, b_m, l_{i+1}, \ldots, l_n$, where there is a rule $r \in \Pi$ such that head of r, h, can be unified with l_i, x_i are term parameters from l_i, t_i are term parameters from h, $body(r)$ is unified with the set of literals $\{b_1, \ldots, b_m\}$ and $C_1 = C$.

A *derivation sequence* is a possibly infinite sequence of transitions $\langle Q_0, C_0\rangle \rightarrow \langle Q_1, C_1\rangle \rightarrow \ldots$, starting with an initial query Q_0 and initial constraint store C_0, such that there is a derivation step to each pair $\langle Q_i, C_i\rangle, i > 0$, from the preceding pair $\langle Q_{i-1}, C_{i-1}\rangle$. A state which cannot be re-written further is called a *final state*.

A sequence is *successful* if it is finite and its last element is $\langle \emptyset, C_n \rangle$, where C_n is a set of solved constraints. A sequence is *conditionally successful* if it is finite and its last element is $\langle \emptyset, C_n \rangle$, where C_n consists of delayed and possibly some solved constraints. A sequence is *finitely failed* if it is finite, neither successful nor conditionally successful, and such that no derivation step is possible from its last element. There can be several derivation sequences for a query with respect to a program. Each of these sequences can be successful, conditionally successful, finitely failed or infinite.

The *computation tree* of a query Q for a program P in a CLP system is a tree with nodes labeled by states and edges labeled by $\rightarrow_r$ or $\rightarrow_c$ such that: root is labeled by $\langle Q, \emptyset \rangle$. If a node S has a outgoing edge labeled $\rightarrow_r$ then the node has a child for each rule in P and the state labeling each child is the state obtained from S by $\rightarrow_r$ transition for that rule.

Every branch of a computation tree is a derivation. Different selection strategies for selecting a literal in the derivation step, gives rise to different computation trees. Existing CLP languages use selection strategy based on the Prolog left-to-right computation rule. The problem of finding answers to a query can be seen as the problem of searching a computation tree. Most CLP languages employ a depth-first search with chronological backtracking.

2.2.2 CLP solvers

Some of the most used CLP solvers are ILOG [53], CHIP[33], CLP(R)[58], CLP(FD)[23], CLP(BNR) [14], Prolog IV, SICStus Prolog [18, 20], ECLiPSe [2], GNU Prolog [31]. Each of these solvers cover one domain or more. Most of them can handle linear systems of equations over real and rational domains. Many application domains like circuit verification, scheduling, resource allocation,

timetabling, control systems and combinatorial problems have been successfully tackled by these solvers [88].

2.2.3 CLP Applications

This section briefly describes some of the applications in which CLP has been successfully used [88].

Assignment problems are one of the first type of industrial applications where CLP was used. Examples in this domain are typically like allocation of gates to planes [22] in an airport or containers in harbor etc. The first industrial CLP application was developed for the HIT container harbor in Hong Kong [82], using language CHIP. Personnel assignments are a special case of assignment problems where humans are involved as a resource. The Gymnaste system [81] produces rosters for nurses in hospitals and is used in many hospitals in France. The OPTI-SERVICE system [24], generates weekly work plans for individual activities for 250 technicians and journalists of the French TV and radio station RFO.

The PLANETS system, developed by the University of Catalonia for a spanish electric company, is a tool for electrical power network reconfiguration which allows to schedule maintenance operations by isolating network segments without disrupting customer services. One of the most successful application domain for finite domain CLP are temporal scheduling problems [88]. The system ATLAS [91] is a constraint based scheduling application that schedules the production of herbicides at Monsanto Plant in Antwerp. The PLANE system [13] is used by Dassault Aviation to plan the production of jets.

The COBRA system [90] generates work plans for train drivers of North Western Trains in UK by scheduling nearly 25000 activities. ILOG Scheduler [53] can be used for pre-emptive scheduling problems where activities can be interrupted over

time. For a more detailed description of reasearch and applications in CLP refer [88, 93].

CHAPTER 3

LANGUAGE $\mathcal{AC}(\mathcal{C})$

This dissertation starts with the design of a collection, $\mathcal{AC}(\mathcal{C})$, of languages parametrised over a collection $\mathcal{C}$ of constraints. In this chapter, we define the syntax and semantics of $\mathcal{AC}(\mathcal{C})$.

3.1 Syntax

The syntax of $\mathcal{AC}(\mathcal{C})$ is determined by a sorted signature Σ, consisting of sorts $S_1, \ldots, S_n$, properly typed predicate symbols, variables and function symbols. By a sort we mean a non-empty countable collection of strings in some fixed alphabet. Strings of sort S_i will be referred as object constants of S_i.
Variables of $\mathcal{AC}(\mathcal{C})$ have a sort assigned to them. Each variable ranges over objects constants from their sort. A term of Σ is either a constant, a variable, or an expression $f(t_1, \ldots, t_n)$, where f is a function symbol of arity n, $t_1, \ldots, t_n$ are terms of proper sorts. The sort of value of a function $f : S_1, \ldots, S_n \rightarrow S$ is S.
Terms also have sorts and are assigned in the natural way. An *atom* is of the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms of proper sorts.
We assume that the standard numerical sorts like $\mathbb{N}$ and $\mathcal{R}$ and numerical functions like $+$, $-$ etc., and numerical relations like $>$, $<$ etc., are present in Σ. The numerical constants, functions and relations have their natural intended interpretations. For instance, relation $'>'$ is interpreted as numerical 'greater than' inequality relation; $'+'$ is interpreted as addition and $3 + 4$ is equal to 7. Terms of numerical sorts are called numerical terms.
Each sort is further distinguished as either a *regular sort* (denoted by s_r) or a

constraint sort (denoted by s_c). The object constants from regular sorts and constraint sorts are called *r-constants* and *c-constants* respectively. A variable is an *r-variable* (*c-variable*) if it ranges over constants from a regular (constraint) sort. A function symbol is either an *r-function* symbol or a *c-function* symbol depending on the sort of its value. We denote the set of r-constants, r-variables and r-function symbols in Σ by C_r, V_r and F_r respectively. Similarly, we denote the set of c-constants, c-variables and c-function symbols in Σ by C_c, V_c and F_c respectively. Terms from regular sorts and constraint sorts are called *r-terms* and *c-terms* respectively. Predicate symbols of Σ are divided into four disjoint sets called regular, constrained, mixed and defined, denoted by P_r, P_c, P_m and P_d respectively.

The collection of constraints, $\mathcal{C}$, is a set of formulas constructed in the natural way. The formulas in $\mathcal{C}$ are formed from a set of predicate symbols $\mathcal{P}$ and a set of function symbols $\mathcal{F}$. The c-predicate symbols P_c and c-function symbols F_c in Σ are subsets of $\mathcal{P}$ and $\mathcal{F}$ respectively.

An atom $p(t_1, \ldots, t_n)$ is

- an *r-atom* if $p \in P_r$ and $t_1, \ldots, t_n$ are r-terms;
- a *c-atom* if $p \in P_c$ and $t_1, \ldots, t_n$ are c-terms;
- a *m-atom* if $p \in P_m$ and $t_1, \ldots, t_n$ are r-terms or c-terms with at least one from each;
- a *d-atom* if $p \in P_d$ and where each parameter t_i is either an r-term or a c-term.

A *literal* is either an atom a or its negation $\neg a$. Literals formed by r-atoms, c-atoms, m-atoms and d-atoms are called *r-literals*, *c-literals*, *m-literals* and

d-literals respectively. An *extended literal* is either a literal l, or its negation *not* l. Literals of the form *not* l are called *negative literals*. The symbol *not* , denotes a logical connective known as *default negation*. The expression *not* l is read as *"there is no reason to believe in l"*. *Ground* expressions are expressions which do not contain variables. Ground terms are represented by lower case letters, and variables by upper case letters.

A rule r of $\mathcal{AC}(\mathcal{C})$ over Σ is a statement of the form:

$$h_1 \ or \ \ldots or \ \ h_k \leftarrow l_1, \ \ldots, \ l_m, \ not \ \ l_{m+1}, \ \ldots, \ not \ \ l_n \tag{3.1}$$

where $h_1, \ldots, h_k$ are r-literals or d-literals and $l_1, \ldots, l_n$ are arbitrary literals. Literals $h_1, \ldots, h_k$ constitute the head of the rule denoted by $head(r)$, and $l_1, \ldots, l_n$ constitute the body of the rule denoted by $body(r)$. Both the head and body of the rule may be empty. If the body is empty, i.e., $n = 0$, the rule is called a $fact$, and we write it as: $h_1 \ or \ \ldots or \ \ h_k$. If the head is empty then we call the rule a denial and assume $false$ as the head.

A rule is an *r-rule* (regular rule) if the literals in the head and body of the rule are all r-literals. A rule is a *d-rule* (defined rule) if all literals in the head are d-literals. The rest of the rules are called *m-rule*s (middle rules).

A *program* of $\mathcal{AC}(\mathcal{C})$ is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a collection of rules over Σ. Π can be divided into three disjoint sets of rules. The set consisting of r-rules of Π is called the regular part of Π denoted by Π_R. Similarly, the set consisting of m-rules of Π is called the middle part of Π denoted by Π_M and the set consisting of d-rules of Π is called the defined part of Π denoted by Π_D.

The following example shows a signature Σ and programs in the language $\mathcal{AC}(\mathcal{C})$. Note that, for readability, we use infix notations for numerical terms and atoms.

Example 3.1.1. Let Σ be a signature with constants $C_r = \{a, b\}$ and $C_c = \{0, \ldots, 100\}$; predicates $P_r = \{p(C_r, C_r),\ q(C_r)\}$, $P_m = \{\ at(C_r, C_c)\ \}$, $P_c = \{>,\ =,\ \leq,\ \geq\}$ with parameters from C_c and $P_d = \{\ equal(C_c, C_c)\ \}$; variables $V_r = \{X,\ Y\}$ range over C_r and $V_c = \{T_1,\ T_2\}$ range over C_c. Following are examples of programs in $\mathcal{AC}(\mathcal{C})$:

Π_1 :

$q(a).$

$q(b).$

$p(X, Y) \leftarrow q(X),\ q(Y),\ at(X, T_1),\ at(Y, T_2),\ T_1 > T_2.$

Π_2 :

$q(a).$

$q(b).$

$p(X, Y) \leftarrow q(X),\ q(Y),\ at(X, T_1),\ at(Y, T_2),\ equal(T_1, T_2).$

$equal(T_1, T_2) \leftarrow T_1 = T_2.$

3.2 Semantics

To give semantics of programs in $\mathcal{AC}(\mathcal{C})$, first we transform an arbitrary program Π into its ground instantiation *ground(Π)*. Then we define the semantics of ground(Π). The semantics of ground(Π), will be viewed as the semantics of program Π. A *ground instance* of a rule r is a rule obtained from r by:

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

A program $ground(\Pi)$ consisting of all ground instances of all rules in Π is called the *ground instantiation* of Π. A program $\langle \Sigma, \Pi \rangle$ is called *r-ground* if all rules of Π contain no r-variables. Given a program $\langle \Sigma, \Pi \rangle$, we can get r-ground($\Pi$) by

grounding only the r-terms using the procedure given above. Obviously $ground(\Pi)$ is an r-ground program.

Example 3.2.1. Let Π_1 be as in example 3.1.1. Then r-ground(Π_1) is:

$$q(a). \quad q(b).$$
$$p(a,a) \leftarrow q(a),\ q(a),\ at(a,T_1),\ at(a,T_2),\ T_1 > T_2.$$
$$p(a,b) \leftarrow q(a),\ q(b),\ at(a,T_1),\ at(b,T_2),\ T_1 > T_2.$$
$$p(b,a) \leftarrow q(b),\ q(a),\ at(b,T_1),\ at(a,T_2),\ T_1 > T_2.$$
$$p(b,b) \leftarrow q(b),\ q(b),\ at(b,T_1),\ at(b,T_2),\ T_1 > T_2.$$

To get the ground instantiation of Π_1, we need to ground variables T_1 and T_2 in r-ground(Π_1). For instance, ground instantiation of last rule above is:

$$p(b,b) \leftarrow q(b),\ q(b),\ at(b,0),\ at(b,0),\ 0 > 0.$$
$$\vdots$$
$$p(b,b) \leftarrow q(b),\ q(b),\ at(b,100),\ at(b,99),\ 100 > 99.$$
$$p(b,b) \leftarrow q(b),\ q(b),\ at(b,100),\ at(b,100),\ 100 > 100.$$

To give the semantics for $ground(\Pi)$, we need to introduce some terminology. If S is a set of ground literals, we say that S *satisfies* a ground literal l, $S \models l$, if $l \in S$ and S *satisfies not* l, $S \models not\ l$, if $l \notin S$. If S satisfies a literal l then l is said to be true in S.

A set of ground literals, S satisfies the head of a ground rule r of the form 3.1 if at least one of the literals in the head of r is satisfied by S. S satisfies the body of r if literals $l_1, \ldots, l_m$ belong to S and literals $l_{m+1}, \ldots, l_n$ do not belong to S. S satisfies rule r, if the head of r is satisfied by S whenever the body of r is satisfied by S. (Note that if the head of a rule is empty, then S satisfies the rule when at least one of the literals in the body is not satisfied by S). S satisfies a program Π, if it satisfies all of the rules of $ground(\Pi)$.

Without loss of generality, we will assume that in any m-literal, the c-term parameters follow r-term parameters. We often write $p(\bar{t}_r, \bar{t}_c)$, where $\bar{t}_r$ and $\bar{t}_c$ are the lists of r-terms and c-terms respectively. Let X be a set of ground m-atoms such that for every mixed predicate $p \in P_m$ and every list of ground r-terms $\bar{t}_r$, there is exactly one list of ground c-terms $\bar{t}_c$ such that $p(\bar{t}_r, \bar{t}_c) \in X$. We call a set of mixed atoms that satisfies this condition a *candidate-mixed* set.

Let M_c be the set of all ground c-atoms that are true in the intended interpretation of corresponding predicate symbols in P_c. For instance, if a c-atom represents $<$ (less than) over non-negative integers, M_c includes $\{0 < 1, 0 < 2, ..., 1 < 2, 1 < 3, ...\}$.

Given a set of literals S and a rule r of the form:

$$h_1 \ or \ \ldots or \ h_k \leftarrow l_1, \ \ldots, \ l_m, \ not \ l_{m+1}, \ \ldots, \ not \ l_n \tag{3.2}$$

The reduct of r with respect to S, r^S, is defined as follows:

$$r^S = \begin{cases} \emptyset & \text{if } l_{m+1}, \ldots, l_n \cap S \neq \emptyset, \\ h_1 \ or \ \ldots or \ h_k \leftarrow l_1, \ldots, l_m. & \text{otherwise.} \end{cases}$$

We define the reduct of a program Π with respect to a set of literals S, Π^S, as:

$$\Pi^S = \{r^S \mid r \in \Pi\}. \tag{3.3}$$

Definition 3.2.1. [**Deductive Closure**] *The deductive closure of* Π *is a minimal consistent set of literals that satisfies* Π.

Definition 3.2.2. [**Answer Set**] *Let* (Σ, Π) *be a program and* X *be a candidate-mixed set; a set* S *of ground literals over* Σ *is an answer set of* Π *if* S *is a deductive closure of* $(\mathrm{ground}(\Pi) \cup X \cup M_c)^S$.

Example 3.2.2. *Let* Π_1 *and* Π_2 *be programs as in example 3.1.1. An answer set for program* Π_1 *is* $S = A \cup X \cup M_c$*, where* $A = \{q(a),\ q(b)\}$*,* $X = \{at(a,3),\ at(b,3)\}$ *and* $M_c = \{1 > 0, 2 > 1, 3 > 2, 3 > 1 \ldots\}$*. An answer set for program* Π_2 *is* $S = A \cup X \cup D \cup M_c$*, where* $A = \{q(a), q(b), p(a,b), p(b,a)\}$*,* $X = \{at(a,2), at(b,2)\}$*,* $M_c = \{0 = 0, 1 = 1, 2 = 2, \ldots\}$*, and* $D = \{equal(0,0), equal(1,1), equal(2,2), \ldots\}$.

An alternative equivalent definition for answer sets of $\mathcal{AC}(C)$ programs which uses the definition of ASP answer sets can be given as:

Definition 3.2.3. [**Answer Set (using ASP)**] *Let* (Σ, Π) *be a program and* X *be a candidate-mixed set; a set* S *of ground literals over* Σ *is an answer set of* Π *if* S *is an asp answer set of* $ground(\Pi) \cup X \cup M_c$.

CHAPTER 4

PARTIAL GROUNDER $\mathcal{P}ground$

The solver for computing answer sets of programs in $\mathcal{AC}(\mathcal{C})$ is called $\mathcal{AC}solver$. It consists of three parts.

1. $\mathcal{P}ground$
2. $\mathcal{T}ranslator$
3. $\mathcal{AC}engine$

Given a $\mathcal{AC}(\mathcal{C})$ program Π, $\mathcal{AC}solver$ first calls $\mathcal{P}ground$ to ground r-terms of Π, the resulting r-ground program, $\mathcal{P}(\Pi)$, is transformed by $\mathcal{T}ranslator$ into an r-ground program $\mathcal{T}(\Pi)$. The $\mathcal{AC}engine$ combines answer set reasoning, a form of abduction, resolution and constraint solving techniques to compute answer sets of $\mathcal{T}(\Pi)$.

The algorithm works on a syntactically restricted class of programs of $\mathcal{AC}(\mathcal{C})$. Later, we prove that given a program Π of $\mathcal{AC}(\mathcal{C})$ that satisfies the syntactic restrictions, the answer sets of Π have one to one correspondence with answer sets of $\mathcal{P}(\Pi)$ and $\mathcal{T}(\Pi)$. In this chapter, the first section describes the syntactic restrictions and the next section describes $\mathcal{P}ground$.

4.1 Syntax Restrictions

In this section, we describe the syntactic restrictions that a program should satisfy. Before we enumerate the restrictions, we recall some definitions from [99]. The collection of rules of a program Π whose heads are formed by a predicate p, is called the definition of p in Π. A predicate p is called a *domain predicate* with respect to Π, if the definition of p in Π has no recursion through default negation.

Also, recall that a $\mathcal{AC}(\mathcal{C})$ program Π is divided into regular, middle and defined parts denoted by Π_R, Π_M and Π_D (see chapter 3).
We consider programs Π satisfying the following syntactic restrictions:

1. *There is only one literal in the head (non-disjunctive programs).* This restriction allows for a simpler description of the algorithm.

2. *Given a rule* $r \in \Pi_M$*, every c-variable of* r *should occur in m-literals of body(r).* This restriction ensures the correctness of the algorithm $\mathcal{AC}engine$.

3. *Each r-variable occurring in* $r \in \Pi_D$*, occurs in head(r).* The consequences of Π_D will be computed using constraint programming techniques. This restriction will ensure that a r-variable of a rule $r \in \Pi_D$ will be ground at the time of solving the head of r; thus allowing to perform lazy grounding.

4. *The only extended m-literals, d-literals and c-literals allowed in* Π *are atoms.* This restriction simplifies the description of the algorithm. It might be difficult to remove this restriction especially for negative mixed literals and needs more investigation.

5. *Mixed literals do not occur in rules of* Π_D. This restriction simplifies the description of the algorithm. We can allow the positive mixed atoms in the body of defined rules and the clp_solver function derivation sequence (see section 5.5.1.3) needs to be changed to include extra constraints. Allowing negative mixed literals needs more investigation.

6. $\Pi_R \cup \Pi_M$ *is r-domain restricted in the sense that every r-variable in a rule* $r \in \Pi_R \cup \Pi_M$*, must appear in an r-atom formed by a domain predicate in the body of* r. This restriction is similar to domain restriction

of the grounder lparse [99] and differs by restricting only r-variables. This restriction allows $\mathcal{P}$ground to use lparse to ground r-terms of $\Pi_R \cup \Pi_M$.

7. *There are no cyclic definitions between d-literals and r-literals.* This restriction simplifies the algorithm *ACsolver* and ensures the correctness of the algorithm *ACengine*.

Example 4.1.1. *Let*

$\Sigma = \{C_r = \{a, b\},\ C_c = [1, \cdots, 100],\ P_r = \{p(C_r, C_r),\ q(C_r),\ r(C_r)\},$
$P_m = \{at(C_r, C_c)\},\ P_c = \{>\},\ P_d = \{d(C_r, C_c, C_c)\}\}$ *be a signature and the programs* Π_1 *and* Π_2 *given below satisfy the syntax restrictions:*

Program Π_1*:*

$q(a).$ $\quad$ $q(b).$ $\quad$ $r(a).$

$p(X, Y) \leftarrow q(X),\ q(Y),\ at(X, T_1),\ at(Y, T_2),\ d(X, T_1, T_2).$

$d(X, T_1, T_2) \leftarrow not\ r(X),\ T_1 > T_2.$

Program Π_2*:*

$q(a) \leftarrow not\ q(b).$

$q(b) \leftarrow not\ q(a).$

$r(a).$ $\quad$ $r(b).$

$p(X, Y) \leftarrow r(X),\ r(Y), not\ q(Y),\ at(X, T_1),\ at(Y, T_2),\ T_1 > T_2.$

Example 4.1.2. *Let*

$\Sigma = \{C_r = \{a, b\},\ C_c = [1, \cdots, 100],\ P_r = \{p(C_r, C_r),\ q(C_r),\ r(C_r)\},$
$P_m = \{at(C_r, C_c)\},\ P_c = \{>\},\ P_d = \{d(C_r, C_c, C_c), d_2(C_r, C_c)\}\}$ *be a signature and the programs* Π_3 *and* Π_4 *given below do not satisfy the syntax restrictions:*
Program Π_3 *does not satisfy restrictions (1), (2), (3) and (6):*

$q(a)$ *or* $q(b).$ $\quad$ $r(a).$

$p(X,Y) \leftarrow q(X),\ at(X,T_1),\ d(T_1,T_2).$

$d(T_1,T_2) \leftarrow not\ r(X),\ T_1 > T_2.$

Program Π_4 *does not satisfy restrictions (4), (5), (6) and (7):*

$q(a) \leftarrow not\ q(b).$

$q(b) \leftarrow not\ q(a).$

$r(a).$ $\qquad$ $r(b).$

$p(X,Y) \leftarrow r(X),\ not\ q(Y),\ not\ at(X,T_1),\ at(Y,T_2),\ d(T_1,T_2).$

$d(T_1,T_2) \leftarrow r(X),\ d_2(X,T_1),\ not\ T_1 > T_2.$

$d_2(X,T) \leftarrow p(X,X),\ \neg at(X,T),\ T > 50.$

From now on, we assume that programs satisfy the given restrictions.

4.2 $\mathcal{P}$ground

In this section we describe the partial grounding procedure, $\mathcal{P}$ground. First, we introduce some terminology. Let us denote the set of c-variables occurring in a rule r by c-variables(r). Given a program Π and some signature Σ (not necessarily that of Π), we define a program $tc(\Sigma,\Pi)$, by induction on the cardinality of Π.

Definition 4.2.1. [$tc(\Sigma, \Pi)$] *Let* Π *be a program and* Σ *be an arbitrary signature,*

- $tc(\Sigma,\emptyset) = \emptyset$
- $tc(\Sigma,\Pi \cup \{r\}) = tc(\Sigma,\Pi)\ \cup\ r'$, *where* r' *is obtained as follows: for each* $X \in$ *c-variables(*r*) , replace every occurrence of* X *in* $body(r)$ *by new constant* x, *not belonging to signatures* Σ *and* $\Sigma_{tc(\Sigma,\Pi)\cup r}$. x *will be called a* tc_constant.

Example 4.2.1. *Let* Σ *and* Π_M *be as in example 4.1.1. The program* $tc(\Sigma,\Pi_M)$ *is as follows:*

$$p(X,Y) \leftarrow q(X),\ q(Y),\ at(X,t_1),\ at(Y,t_2),\ d(X,t_1,t_2).$$

The operator tc replaces all c-variables in Π by tc_constants. Let T_c be the set of pairs $\langle c,\ V\rangle$ where c is a tc_constant which replaced variable V in tc construction. Note that by definition of tc, for any tc_constant c, there is only one pair $\langle c,\ V\rangle$ in T_c. The following construction performs the reverse of tc by replacing tc_constants by variables.

Definition 4.2.2. [**reverse_tc(Π,T_c)**] *Let Π be a program with tc_constants and T_c be as defined above. The program* $reverse\_tc(\Pi, T_c)$ *is constructed as follows:*

- $reverse\_tc(\emptyset, T_c) = \emptyset$
- $reverse\_tc(\Pi \cup \{r\}, T_c) = reverse\_tc(\Pi, T_c) \cup r'$, *where r' is obtained from r by replacing each tc_constant c in r by variable V such that $\langle c,\ V\rangle \in T_c$.*

Example 4.2.2. *Let Π_1 be* $tc(\Sigma, \Pi_M)$ *as in example 4.2.1. From example 4.2.1,* $T_c = \{\langle t_1, T_1\rangle, \langle t_2, T_2\rangle\}$. *The program* $reverse\_tc(\Pi_1, T_c)$ *is as follows:*

r_4: $p(X,Y) \leftarrow q(X),\ q(Y),\ at(X,T_1),\ at(Y,T_2),\ d(X,T_1,T_2).$

To be able to use lparse for grounding regular terms, we will need to expand our program by adding definitions for mixed and defined atoms. In order to do that, we define an operator addr (add rules) defined as follows:

Definition 4.2.3. [**addr(Π)**] *Let Π be a program and P be the set of mixed and defined predicate symbols of Π. We define a program* $addr(\Pi) = addr(\Pi, P)$, *where* $addr(\Pi, X)$ *for $X \subseteq P$ is defined as follows:*

- $addr(\Pi, \emptyset) = \emptyset$

▷ $addr(\Pi, X \cup \{p\}) = addr(\Pi, X) \cup$

$$\{p(\bar{V}_r, \bar{V}_c) \leftarrow not\ np(\bar{V}_r, \bar{V}_c)\} \cup$$

$$\{np(\bar{V}_r, \bar{V}_c) \leftarrow not\ p(\bar{V}_r, \bar{V}_c)\}$$

where $p \in P \setminus X$, np *is a new predicate symbol not belonging to* $\Sigma_{\Pi \cup addr(\Pi,X)}$ *and* $\bar{V}_r, \bar{V}_c$ *are variables of appropriate sorts.*

Example 4.2.3. *Let* Σ *and* Π_1 *be as in example 4.1.1. The program* $addr(\Pi_1)$ *is as follows:*

$at(X, T_1) \leftarrow not\ n\_at(X, T_1).$
$n\_at(X, T_1) \leftarrow not\ at(X, T_1).$
$d(X, T_1, T_2) \leftarrow not\ n\_d(X, T_1, T_2).$
$n\_d(X, T_1, T_2) \leftarrow not\ d(X, T_1, T_2).$

Let Y be a set of m-atoms of Σ_Π such that for every mixed predicate $p \in P_m$ of Σ_Π and every list of properly typed ground r-terms $\bar{t}_r$, there is exactly one list of distinct c-variables $\bar{V}_c$ such that $p(\bar{t}_r, \bar{V}_c) \in Y$ and variables in $\bar{V}_c$ do not occur in any other atom from Y. We call the set of mixed atoms that satisfy this condition a r_ground_mixed set. Let f be the function which takes as input a mixed predicate p, a sequence of ground r-terms $\bar{t}_r$ and a r_ground_mixed set Y of Π and returns $[X_1, \ldots, X_m]$, where $p(\bar{t}_r, X_1, \ldots, X_m) \in Y$. The function f is well defined due to the definition of r_ground_mixed set.
Given an r-ground rule r and an r_ground_mixed set Y, we define a rule $rename(r, Y)$ as follows:

Definition 4.2.4. [**rename(r,Y)**] *Let* r *be a rule and* Y *be an r_ground_mixed set. The rule* $r' = rename(r, Y)$ *is obtained from* r *as follows:*

▷ *for each m-literal,* $m = p(\bar{t}_r, Y_1, \ldots Y_n)$ *in* $body(r)$*, replace occurrences of variables* $Y_1, \ldots, Y_n$ *in* m *by* $X_1, \ldots, X_n$ *respectively, where* $[X_1, \ldots, X_n] = f(p, \bar{t}_r, Y)$.

▷ *add constraints* $Y_1 = X_1, \ldots, Y_n = X_n$ *to the body of* r.

Example 4.2.4. Let r be a rule as shown below:

$r: \ p(a,b) \leftarrow at(a,T_1,T_1),\ at(b,T_1,T_2),\ d(a,b,T_1,T_2).$

such that $P_m = \{at\}$. Let $Y = \{at(a,V_1,V_2),\ at(b,V_3,V_4)\}$ be a r_ground_mixed set. Then $r' = rename(r,Y)$ is:

$r': \ p(a,b) \leftarrow at(a,V_1,V_2),\ at(b,V_3,V_4),\ d(a,b,T_1,T_2),$
$T_1 = V_1,\ T_1 = V_2,\ T_1 = V_3,\ T_2 = V_4.$

Definition 4.2.5. $[\beta(r,C,\Sigma)]$ *Let* C *be a set of c-literals,* r *be a rule and* Σ *be an arbitrary signature. We define* $\beta(r,C,\Sigma)$ *as a rule* r'*, where*

▷ $head(r') = head(r)$

▷ *If* r *consists only of predicates from* Σ *then* $body(r') = body(r) \cup c\_lits(C,r)$*, where* $c\_lits(C,r)$ *is the set of all c_lits from* C *whose tc_constants are exactly those in* r

▷ *If* r *contains some predicates not in* Σ *then* $body(r') = body(r)$

Given a program Π, $\beta(\Pi,C,\Sigma) = \{\ \beta(r,C,\Sigma) \mid r \in \Pi\ \}$

Example 4.2.5. *Let* $\Sigma = \{P_r = \{p(C_r,C_r),\ q(C_r)\},\ P_m = \{at\},\ P_c = \{>\},\ C_r = \{a,b\},\ C_c = [1,\cdots,100],\ tc\_cons = \{t_1,t_2\}\}$ *be a signature,* $C = \{t_1 > t_2\}$ *and* Π *be a program with the following rules:*

r_1*:* $\quad q(a).$

r_2*:* $\quad p(a,b) \leftarrow q(a),\ q(b),\ at(a,t_1),\ at(b,t_2).$

r_3: $\quad n\_at(b,t_1) \leftarrow not\ at(b,t_1).$

$\beta(\Pi, C, \Sigma)$ *is the following set of rules:*

$\beta(r_1, C, \Sigma)$: $\quad q(a).$

$\beta(r_2, C, \Sigma)$: $\quad p(a,b) \leftarrow q(a),\ q(b),\ at(a,t_1),\ at(b,t_2),\ t_1 > t_2.$

$\beta(r_3, C, \Sigma)$: $\quad n\_at(b,t_1) \leftarrow not\ at(b,t_1).$

Given a program Π_i, we denote the regular, middle and defined parts of Π_i by Π_{i_R}, Π_{i_M} and Π_{i_D} respectively. Now we describe the steps of $\mathcal{P}ground$ to obtain $\mathcal{P}(\Pi)$ from Π.

Definition 4.2.6. $[\mathcal{P}(\Pi)]$ Given a program Π of $\mathcal{AC}(\mathcal{C})$, we construct an r-ground program $\mathcal{P}(\Pi)$ as follows:

1. replace c-variables in Π_M by tc_constants, $\Pi_1 = tc(\Sigma_\Pi, \Pi_M) \cup \Pi_R \cup \Pi_D$
2. remove c-literals occurring in Π_{1_M} and store them in a set C,
   $\Pi_2 = \Pi_R \cup \Pi_D \cup (\Pi_{1_M} \setminus c\text{-}lits(\Pi_{1_M}))$
3. compute $addr(\Pi_2)$ and get $\Pi_3 = \Pi_2 \cup addr(\Pi_2)$
4. ground $\Pi_{3_R} \cup \Pi_{3_M}$ to get $\Pi_4 = lparse(\Pi_{3_R} \cup \Pi_{3_M} \cup addr(\Pi_2)) \cup \Pi_{3_D}$
5. remove ground instantiations of $addr(\Pi_2)$ to get
   $\Pi_5 = \Pi_4 \setminus ground(addr(\Pi_2))$
6. (a) put back c-literals removed in step (2), to get $\Pi_{6a} = \beta(\Pi_5, C, \Sigma_2)$
   (b) put back c-variables removed in step (1),
   $\Pi_6 = \Pi_{5_R} \cup \Pi_{5_D} \cup reverse\_tc(\Pi_{5_M}, T_c)$
7. compute rename(Π_{6_M},Y), to get $\Pi_7 = rename(\Pi_{6_M}, Y) \cup \Pi_{6_R} \cup \Pi_{6_D}$ where Y is a r_ground_mixed set of Π.

The r_ground_mixed set Y used in step (7) of construction of $\mathcal{P}(\Pi)$ is called $mcv\_set(\Pi)$ and read as *mixed candidate variable* set of Π. Note that all the c-variables in $\mathcal{P}(\Pi)$ occur in some mixed atom from $mcv\_set(\Pi)$. This set will be used by $\mathcal{AC}engine$ to construct a candidate_mixed set while computing answer sets of Π.

Also, note that the program obtained at step(3) after adding $addr(\Pi_2)$ contains mixed atoms in the head of rules and hence is not an $\mathcal{AC}(\mathcal{C})$ program but an arbitrary program of A-Prolog. The rules $addr(\Pi_2)$ are added to the program to make sure that $lparse$'s intelligent grounding does not remove rules with m-atoms and d-atoms in the body; this would happen without such rules because m-atoms and d-atoms do not occur in the heads of rules in $\Pi_{3_R} \cup \Pi_{3_M}$. The program $\mathcal{P}(\Pi)$ is an r-ground program of $\mathcal{AC}(\mathcal{C})$. The following proposition makes sure Π and $\mathcal{P}(\Pi)$ are equivalent, i.e., have the same answer sets.

Proposition 4.2.1. *Given a program Π of $\mathcal{AC}(\mathcal{C})$ that satisfies the syntax restrictions, S is an answer set of Π iff S is an answer set of $\mathcal{P}(\Pi)$.*

The steps of $\mathcal{P}(\Pi)$ are illustrated using the following example.

Example 4.2.6. Let Σ be formed by $C_r = \{1, 2\}$, $C_c = [0..100]$, $P_r = \{p, q, r, s, <, \neq\}$, $P_m = \{at\}$, $P_c = \{\leq, =\}$, $P_d = \{d\}$ and variables $V_r = \{X,\ Y\}$ and $V_c = \{T_1,\ T_2\}$. Π is as follows:

$r_{a_{[1,2]}}$: $q(1).\quad q(2).$

r_b : $p(X,Y) \leftarrow q(X),\ q(Y),\ r(X,Y),\ X < Y$

r_c : $r(X,Y) \leftarrow q(X),\ q(Y),\ not\ s(X,Y),\ X \neq Y$

r_d : $s(X,Y) \leftarrow q(X),\ q(Y),\ not\ r(X,Y),\ X \neq Y$

r_e : $p(X,Y) \leftarrow q(X),\ q(Y),\ at(X,T_1),\ at(Y,T_2),$
$T_1 \leq T_2,\ d(X,Y,T_1,T_2)$

r_f : $\quad d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X<Y,\ T_1 = T_2 + 10$

Program Π is divided into $\Pi_R = \{r_a, r_b, r_c, r_d\}$; $\Pi_M = \{r_e\}$; $\Pi_D = \{r_f\}$.

Now we show the changes to Π at each step of computation of $\mathcal{P}(\Pi)$. Performing step (1) to example 4.2.6, we get, $\Pi_1 = tc(\Sigma_\Pi, \Pi_M) \cup \Pi_R \cup \Pi_D$, where $tc(\Sigma_\Pi, \Pi_M) = \{r_{e_1}\}$ and

r_{e_1} : $\quad p(X,Y) \leftarrow q(X),\ q(Y),\ at(X,t_1),\ at(Y,t_2),\ t_1 \leq t_2, d(X,Y,t_1,t_2)$

Step (2) removes c-literals from Π_{1_M} and stores them in C, we get $\Pi_2 = \Pi_{1_R} \cup \Pi_{1_C} \cup \{r_{e_2}\}$, where r_{e_2} is obtained from r_{e_1} by removing c-literals in $body(r_{e_1})$.

r_{e_2} : $\quad p(X,Y) \leftarrow q(X),\ q(Y),\ at(X,t_1),\ at(Y,t_2),\ d(X,Y,t_1,t_2)$

and $C = \{t_1 \leq t_2\}$.

Step (3) adds rules from $addr(\Pi_2)$ to the program. $addr(\Pi_2)$ is as follows:

r_{m_1} : $\quad at(X,Z) \leftarrow \mathit{not}\ n\_at(X,Z).$

r_{m_2} : $\quad n\_at(X,Z) \leftarrow \mathit{not}\ at(X,Z).$

r_{m_3} : $\quad d(X,Y,Z_1,Z_2) \leftarrow \mathit{not}\ n\_d(X,Y,Z_1,Z_2).$

r_{m_4} : $\quad n\_d(X,Y,Z_1,Z_2) \leftarrow \mathit{not}\ d(X,Y,Z_1,Z_2).$

where, variables Z, Z_1, Z_2 range over tc_constants $\{t_1, t_2\}$ and variables X, Y range over $\{1,2\}$. *Instead of* $addr(\Pi_2)$*, a choice rule equivalent [75] with mixed and defined atoms will be added to improve efficiency during implementation.*

Step (4) is grounding regular terms in $\Pi_{3_R} \cup \Pi_{3_M} \cup addr(\Pi_2)$. Continuing the example, the resulting program $\Pi_4 = lparse(\Pi_{3_R} \cup \Pi_{3_M} \cup addr(\Pi_2)) \cup \Pi_{3_D}$ is as follows:

$rg_{a_{[1,2]}}$: $\quad q(1).\quad q(2).$

rg_{b_1} : $\quad p(1,2) \leftarrow q(1),\ q(2),\ r(1,2)$

rg_{c_1} : $\quad r(1,2) \leftarrow q(1),\ q(2),\ \mathit{not}\ s(1,2)$

rg_{c_2} : $\quad r(2,1) \leftarrow q(2),\ q(1),\ \mathit{not}\ s(2,1)$

rg_{d_1} : $s(1,2) \leftarrow q(1),\ q(2),\ not\ r(1,2)$

rg_{d_2} : $s(2,1) \leftarrow q(2),\ q(1),\ not\ r(2,1)$

rg_{e_1} : $p(1,1) \leftarrow q(1),\ q(1),\ at(1,t_1),\ at(1,t_2),\ d(1,1,t_1,t_2)$

rg_{e_2} : $p(1,2) \leftarrow q(1),\ q(2),\ at(1,t_1),\ at(2,t_2),\ d(1,2,t_1,t_2)$

rg_{e_3} : $p(2,1) \leftarrow q(2),\ q(1),\ at(2,t_1),\ at(1,t_2),\ d(2,1,t_1,t_2)$

rg_{e_4} : $p(2,2) \leftarrow q(2),\ q(2),\ at(2,t_1),\ at(2,t_2),\ d(2,2,t_1,t_2)$

rg_{m_1} : $at(1,t_1) \leftarrow not\ n\_at(1,t_1).$

rg_{m_2} : $at(2,t_1) \leftarrow not\ n\_at(2,t_1).$

...

rg_{m_8} : $n\_at(2,t_2) \leftarrow not\ at(2,t_2).$

rg_{m_9} : $d(a,a,t_1,t_1) \leftarrow not\ n\_d(a,a,t_1,t_1).$

$rg_{m_{10}}$: $d(a,b,t_1,t_1) \leftarrow not\ n\_d(a,a,t_1,t_1).$

...

rg_{m_n} : $n\_d(b,b,t_2,t_2) \leftarrow not\ n\_d(b,b,t_2,t_2).$

r_f : $d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X < Y,\ T_1 = T_2 + 10$

Step (5) removes rground instances $\{rg_{m_1}, \ldots, rg_{m_n}\}$ of $addr(\Pi_2)$, and we get Π_5 as: $rg_{a_{[1,2]}}$: $q(1).\quad q(2).$

rg_{b_1} : $p(1,2) \leftarrow q(1),\ q(2),\ r(1,2)$

rg_{c_1} : $r(1,2) \leftarrow q(1),\ q(2),\ not\ s(1,2)$

rg_{c_2} : $r(2,1) \leftarrow q(2),\ q(1),\ not\ s(2,1)$

rg_{d_1} : $s(1,2) \leftarrow q(1),\ q(2),\ not\ r(1,2)$

rg_{d_2} : $s(2,1) \leftarrow q(2),\ q(1),\ not\ r(2,1)$

rg_{e_1} : $p(1,1) \leftarrow q(1),\ q(1),\ at(1,t_1),\ at(1,t_2),\ d(1,1,t_1,t_2)$

rg_{e_2} : $p(1,2) \leftarrow q(1),\ q(2),\ at(1,t_1),\ at(2,t_2),\ d(1,2,t_1,t_2)$

rg_{e_3} : $p(2,1) \leftarrow q(2),\ q(1),\ at(2,t_1),\ at(1,t_2),\ d(2,1,t_1,t_2)$

rg_{e_4} : $p(2,2) \leftarrow q(2),\ q(2),\ at(2,t_1),\ at(2,t_2),\ d(2,2,t_1,t_2)$

r_f : $\quad d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X<Y,\ T_1 = T_2 + 10$

In step (6), c-literals removed in step (2) are replaced to the ground instances of the original rule. We also restore c-variables changed to constants in step (1). The resulting program Π_6 is:

$rg_{a_{[1,2]}}$: $\quad q(1).\quad q(2).$

rg_{b_1} : $\quad p(1,2) \leftarrow q(1),\ q(2),\ r(1,2)$

rg_{c_1} : $\quad r(1,2) \leftarrow q(1),\ q(2),\ not\ s(1,2)$

rg_{c_2} : $\quad r(2,1) \leftarrow q(2),\ q(1),\ not\ s(2,1)$

rg_{d_1} : $\quad s(1,2) \leftarrow q(1),\ q(2),\ not\ r(1,2)$

rg_{d_2} : $\quad s(2,1) \leftarrow q(2),\ q(1),\ not\ r(2,1)$

$rg_{e_{1'}}$: $\quad p(1,1) \leftarrow q(1),\ q(1),\ at(1,T_1),\ at(1,T_2), T_1 \leq T_2,\ d(1,1,T_1,T_2)$

$rg_{e_{2'}}$: $\quad p(1,2) \leftarrow q(1),\ q(2),\ at(1,T_1),\ at(2,T_2), T_1 \leq T_2,\ d(1,2,T_1,T_2)$

$rg_{e_{3'}}$: $\quad p(2,1) \leftarrow q(2),\ q(1),\ at(2,T_1),\ at(1,T_2), T_1 \leq T_2,\ d(2,1,T_1,T_2)$

$rg_{e_{4'}}$: $\quad p(2,2) \leftarrow q(2),\ q(2),\ at(2,T_1),\ at(2,T_2), T_1 \leq T_2,\ d(2,2,T_1,T_2)$

r_f : $\quad d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X<Y,\ T_1 = T_2 + 10$

Step (7) performs renaming of c-variables. Let $Y = \{at(1,V_1), at(2,V_2)\}$ be a r_ground_mixed set and function f be as defined before. The program $\mathcal{P}(\Pi)$ is as follows:

$rg_{a_{[1,2]}}$: $\quad q(1).\quad q(2).$

rg_{b_1} : $\quad p(1,2) \leftarrow q(1),\ q(2),\ r(1,2)$

rg_{c_1} : $\quad r(1,2) \leftarrow q(1),\ q(2),\ not\ s(1,2)$

rg_{c_2} : $\quad r(2,1) \leftarrow q(2),\ q(1),\ not\ s(2,1)$

rg_{d_1} : $\quad s(1,2) \leftarrow q(1),\ q(2),\ not\ r(1,2)$

rg_{d_2} : $\quad s(2,1) \leftarrow q(2),\ q(1),\ not\ r(2,1)$

rg_{e_a} : $\quad p(1,1) \leftarrow q(1),\ q(1),\ at(1,V_1),\ at(1,V_1), V_1 \leq V_1,\ d(1,1,V_1,V_1)$

rg_{e_b} : $\quad p(1,2) \leftarrow q(1),\ q(2),\ at(1,V_1),\ at(2,V_2), V_1 \leq V_2,\ d(1,2,V_1,V_2)$

$$rg_{e_c}: \quad p(2,1) \leftarrow q(2),\ q(1),\ at(2,V_2),\ at(1,V_1), V_2 \leq V_1,\ d(2,1,V_2,V_1)$$

$$rg_{e_d}: \quad p(2,2) \leftarrow q(2),\ q(2),\ at(2,V_2),\ at(2,V_2), V_2 \leq V_2,\ d(2,2,V_2,V_2)$$

$$r_f: \quad d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X < Y,\ T_1 = T_2 + 10$$

Recall that $\mathcal{P}ground$ uses $lparse$ for intelligent grounding of r-terms of a program Π. The system $lparse$ while grounding also transforms the program with *negative* literals ($\neg p(\bar{t})$) to an equivalent program (with respect to answer sets) without negative literals. Given a literal l and its negation $\neg l$, $lparse$ does the following transformation:

- ▷ replaces each occurrence of $\neg l$ in Π by a new literal l'
- ▷ adds the following rule to the program.

  $\leftarrow l,\ l'.$

Therefore, to simplify the description of the solver, we assume that programs with negated literals undergo the above transformation and $\mathcal{P}(\Pi)$ does not contain any negative literals.

CHAPTER 5
ALGORITHM

As described in chapter 4, $\mathcal{P}ground$ partially grounds an $\mathcal{AC}(C)$ program Π and returns an r-ground program $\mathcal{P}(\Pi)$. The $\mathcal{T}ranslator$ transforms $\mathcal{P}(\Pi)$ into a new r-ground program $\mathcal{T}(\Pi)$. We prove that the answer sets of $\mathcal{P}(\Pi)$ have one to one correspondence with answer sets of $\mathcal{T}(\Pi)$. The solver $\mathcal{AC}engine$ then computes answer sets of $\mathcal{T}(\Pi)$ using constraint programming techniques integrated with ASP techniques. The program $\mathcal{P}(\Pi)$ is translated to $\mathcal{T}(\Pi)$ in order to eliminate disjunctive queries to the CLP system (see section 5.5.2). In this chapter, the first section describes the $\mathcal{T}ranslator$ and the next section describes the inference engine $\mathcal{AC}engine$ to compute answer sets of $\mathcal{T}(\Pi)$.

5.1 $\mathcal{T}ranslator$

Let us introduce some terminology. We define the program $tr_1(\Pi)$ by induction on cardinality of Π.

Definition 5.1.1. $[tr_1(\Pi)]$ *Let* r *be a rule of the form:*

$$h \leftarrow l_1,\ \ldots,\ l_m,\ \mathit{not}\ \ l_{m+1},\ \ldots,\ \mathit{not}\ \ l_n \tag{5.1}$$

The program $tr_1(\Pi)$ *is defined as follows:*

- $tr_1(\emptyset) = \emptyset$
- $tr_1(\Pi \cup \{r\}) = tr_1(\Pi)\ \cup \{r_1, r_2\}$, *where rule* r_1 *is obtained from* r *by replacing it's head by a new r-atom* h' *not belonging to* $\Sigma_{tr_1(\Pi)\cup\{r\}}$ *as follows:*

$$r_1: \qquad h' \leftarrow l_1, \ldots, l_m, \mathit{not}\ \ l_{m+1}, \ldots, \mathit{not}\ \ l_n \tag{5.2}$$

and rule r_2 *is obtained by replacing* r*'s body by* h' *as follows:*

$$r_2: \qquad h \leftarrow h' \tag{5.3}$$

Example 5.1.1. *Let* $\Sigma = \{C_r = \{a, b\}, C_c = \{C_1 = [0 \ldots 86400],\ C_2 = [0 \ldots 1440]\},\ P_r = \{q(C_r), p(C_r, C_r)\},\ P_m = \{at(C_r, C_c\},\ P_c = \{\geq (C_c, C_c)\},\ P_d = \{d_1(C_r, C_r, C_c, C_c)\}\ \}$ *and Variables* V_1 *and* V_2 *range over sorts* C_1 *and* C_2 *respectively. Let* r *be as follows:*

$p(a, b) \leftarrow q(a),\ q(b),\ at(a, V_1),\ at(b, V_2),\ d_1(a, b, V_1, V_2), V_1 \geq V_2.$

Then $tr_1(r)$ *consists of the following rules:*

$p'(a, b) \leftarrow q(a),\ q(b),\ at(a, V_1),\ at(b, V_2),\ d_1(a, b, V_1, V_2), V_1 \geq V_2.$

$p(a, b) \leftarrow p'(a, b).$

Note that the new atoms introduced in definition tr_1 are regular atoms. Given a rule r, recall that we denote the set of extended r-literals, c-literals, d-literals and m-literals in $body(r)$ as $r\_lits(r)$, $c\_lits(r)$, $d\_lits(r)$ and $m\_lits(r)$ respectively. For any $\mathcal{AC}(C)$ program Π, we define the program $tr_2(\Pi)$ by induction on the cardinality of Π.

Definition 5.1.2. $[tr_2(\Pi)]$ *Let rule* r *be:*

$$h \leftarrow r\_lits(r), m\_lits(r), c\_lits(r), d\_lits(r). \tag{5.4}$$

We define the program $tr_2(\Pi)$ *as follows:*

- $tr_2(\emptyset) = \emptyset$
- $tr_2(\Pi \cup \{r\}) = tr_2(\Pi) \cup \{\ r_1, r_2\ \}$*, where* r_1 *and* r_2 *are obtained from* r *as follows:*

$$h \leftarrow r\_lits(r), m\_lits(r), d(\bar{t}_r, \bar{t}_c) \tag{5.5}$$

$$d(\bar{t}_r, \bar{t}_c) \leftarrow c\_lits(r), d\_lits(r) \tag{5.6}$$

where d *is a new d-predicate not belonging to* $\Sigma_{tr_2(\Pi)\cup\{r\}}$ *and* $\bar{t}_r$ *and* $\bar{t}_c$ *are the set of r-variables and c-variables in* $c\_lits(r) \cup d\_lits(r)$.

Example 5.1.2. *Let* $\Sigma = \{C_r = \{a, b\}, C_c = \{C_1 = [0 \dots 86400],\ C_2 = [0 \dots 1440]\},\ P_r = \{q(C_r), p(C_r, C_r)\},\ P_m = \{at(C_r, C_c\},\ P_c = \{\geq (C_c, C_c)\},\ P_d = \{d_1(C_r, C_r, C_c, C_c)\},\ \}$ *and Variables* V_1 *and* V_2 *range over sorts* C_1 *and* C_2 *respectively. Let* r *be as follows:*

$p(a, b) \leftarrow q(a),\ q(b),\ at(a, V_1),\ at(b, V_2),\ d_1(a, b, V_1, V_2), V_1 \geq V_2.$

Then $tr_2(r)$ *consists of the following rules:*

$p(a, b) \leftarrow q(a),\ q(b),\ at(a, V_1),\ at(b, V_2),\ d(a, b, V_1, V_2).$

$d(a, b, V_1, V_2) \leftarrow d_1(a, b, V_1, V_2), V_1 \geq V_2.$

The new predicates introduced in definition tr_2 are defined predicates. Therefore, rules of the form 5.6 are defined rules. *Note that the definition of* tr_2 *refers to r-variables even though* $\mathcal{P}(\Pi)$ *is r-ground. This is because applying* tr_2 *before grounding will be more efficient and gives less number of rules of the form 5.6.*

Definition 5.1.3. $[\mathcal{T}(\Pi)]$ *Let* Π *be an* $\mathcal{AC}(\mathcal{C})$ *program and* $\Pi_0 = \mathcal{P}(\Pi)$, *the program* $\mathcal{T}(\Pi)$ *is obtained by:*

1. *applying* tr_1 *to middle rules of* Π_0 *to get* $\Pi_1 = \Pi_{0_R} \cup \Pi_{0_D} \cup tr_1(\Pi_{0_M})$
2. *applying* tr_2 *to middle rules of* Π_1 *to get* $\mathcal{T}(\Pi) = \Pi_{1_R} \cup \Pi_{1_D} \cup tr_2(\Pi_{1_M})$.

Note that we apply transformation tr_1 only to middle rules of $\mathcal{P}(\Pi)$ and the new atoms introduced by tr_1 are r-atoms. Therefore, rules of the form 5.2 are middle rules and rules of the form 5.3 are regular rules. We apply transformation tr_2 to

middle rules of program obtained from transformation tr_1 and the new atoms introduced are d-atoms. Therefore the rules of the form 5.5 are middle rules and the rules of the form 5.6 are defined rules.

Lemma 5.1.1. *Given an $\mathcal{AC}(C)$ program Π, answer sets of $\mathcal{P}(\Pi)$ have one to one correspondence to answer sets of $\mathcal{T}(\Pi)$*

Example 5.1.3. *Let us continue example 4.2.6 from chapter 4. The program $\mathcal{P}(\Pi)$ as computed in example 4.2.6 is as follows:*

$rg_{a_{[1,2]}}$: $q(1).\quad q(2).$

rg_{b_1} : $p(1,2) \leftarrow q(1),\ q(2),\ r(1,2)$

rg_{c_1} : $r(1,2) \leftarrow q(1),\ q(2),\ not\ s(1,2)$

rg_{c_2} : $r(2,1) \leftarrow q(2),\ q(1),\ not\ s(2,1)$

rg_{d_1} : $s(1,2) \leftarrow q(1),\ q(2),\ not\ r(1,2)$

rg_{d_2} : $s(2,1) \leftarrow q(2),\ q(1),\ not\ r(2,1)$

rg_{e_a} : $p(1,1) \leftarrow q(1),\ q(1),\ at(1,V_1),\ at(1,V_1), V_1 \leq V_1,\ d(1,1,V_1,V_1)$

rg_{e_b} : $p(1,2) \leftarrow q(1),\ q(2),\ at(1,V_1),\ at(2,V_2), V_1 \leq V_2,\ d(1,2,V_1,V_2)$

rg_{e_c} : $p(2,1) \leftarrow q(2),\ q(1),\ at(2,V_2),\ at(1,V_1), V_2 \leq V_1,\ d(2,1,V_2,V_1)$

rg_{e_d} : $p(2,2) \leftarrow q(2),\ q(2),\ at(2,V_2),\ at(2,V_2), V_2 \leq V_2,\ d(2,2,V_2,V_2)$

r_f : $d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X < Y,\ T_1 = T_2 + 10$

First, we apply transformation tr_1 to middle rules of $\mathcal{P}(\Pi)$. We get the following rules:

$rg_{a_{[1,2]}}$: $q(1).\quad q(2).$

rg_{b_1} : $p(1,2) \leftarrow q(1),\ q(2),\ r(1,2)$

rg_{c_1} : $r(1,2) \leftarrow q(1),\ q(2),\ not\ s(1,2)$

rg_{c_2} : $r(2,1) \leftarrow q(2),\ q(1),\ not\ s(2,1)$

rg_{d_1} : $s(1,2) \leftarrow q(1),\ q(2),\ not\ r(1,2)$

rg_{d_2} : $s(2,1) \leftarrow q(2),\ q(1),\ not\ r(2,1)$

$rg_{e_{a_1}}$: $p'(1,1) \leftarrow q(1),\ q(1),\ at(1,V_1),\ at(1,V_1), V_1 < V_1,\ d(1,1,V_1,V_1)$

$rg_{e_{b_1}}$: $p'(1,2) \leftarrow q(1),\ q(2),\ at(1,V_1),\ at(2,V_2), V_1 < V_2,\ d(1,2,V_1,V_2)$

$rg_{e_{c_1}}$: $p'(2,1) \leftarrow q(2),\ q(1),\ at(2,V_2),\ at(1,V_1), V_2 < V_1,\ d(2,1,V_2,V_1)$

$rg_{e_{d_1}}$: $p'(2,2) \leftarrow q(2),\ q(2),\ at(2,V_2),\ at(2,V_2), V_2 < V_2,\ d(2,2,V_2,V_2)$

$rg_{e_{a_2}}$: $p(1,1) \leftarrow p'(1,1)$

$rg_{e_{b_2}}$: $p(1,2) \leftarrow p'(1,2)$

$rg_{e_{c_2}}$: $p(2,1) \leftarrow p'(2,1)$

$rg_{e_{d_2}}$: $p(2,2) \leftarrow p'(2,2)$

r_f : $d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X < Y,\ T_1 = T_2 + 10$

The middle rules of above program are $\{rg_{e_{a_1}}, rg_{e_{b_1}}, rg_{e_{c_1}}, rg_{e_{d_1}}\}$. *Now let us apply transformation* tr_2 *to middle rules of above program. We get the following rules:*

$rg_{a_{[1,2]}}$: $q(1).\quad q(2).$

rg_{b_1} : $p(1,2) \leftarrow q(1),\ q(2),\ r(1,2)$

rg_{c_1} : $r(1,2) \leftarrow q(1),\ q(2),\ not\ s(1,2)$

rg_{c_2} : $r(2,1) \leftarrow q(2),\ q(1),\ not\ s(2,1)$

rg_{d_1} : $s(1,2) \leftarrow q(1),\ q(2),\ not\ r(1,2)$

rg_{d_2} : $s(2,1) \leftarrow q(2),\ q(1),\ not\ r(2,1)$

$rg_{e_{a_1}}$: $p'(1,1) \leftarrow q(1),\ q(1),\ at(1,V_1),\ at(1,V_1),\ d_1(V_1)$

$rg_{e_{b_1}}$: $p'(1,2) \leftarrow q(1),\ q(2),\ at(1,V_1),\ at(2,V_2),\ d_2(V_1,V_2)$

$rg_{e_{c_1}}$: $p'(2,1) \leftarrow q(2),\ q(1),\ at(2,V_2),\ at(1,V_1),\ d_3(V_2,V_1)$

$rg_{e_{d_1}}$: $p'(2,2) \leftarrow q(2),\ q(2),\ at(2,V_2),\ at(2,V_2),\ d_4(V_2)$

$rg_{e_{a_3}}$: $d_1(V_1) \leftarrow V_1 < V_1,\ d(1,1,V_1,V_1)$

$rg_{e_{b_3}}$: $d_2(V_1,V_2) \leftarrow V_1 < V_2,\ d(1,2,V_1,V_2)$

$rg_{e_{c_3}}$: $d_3(V_2,V_1) \leftarrow V_2 < V_1,\ d(2,1,V_2,V_1)$

$$rg_{e_{d_3}}: \quad d_4(V_2) \leftarrow V_2 < V_2,\ d(2,2,V_2,V_2)$$

$$rg_{e_{a_2}}: \quad p(1,1) \leftarrow p'(1,1)$$

$$rg_{e_{b_2}}: \quad p(1,2) \leftarrow p'(1,2)$$

$$rg_{e_{c_2}}: \quad p(2,1) \leftarrow p'(2,1)$$

$$rg_{e_{d_2}}: \quad p(2,2) \leftarrow p'(2,2)$$

$$r_f: \quad d(X,Y,T_1,T_2) \leftarrow s(X,Y),\ X < Y,\ T_1 = T_2 + 10$$

Recall that $\mathcal{P}(\Pi)$ does not contain any negative ($\neg$) literals (see end of section 4.2). Therefore, $\mathcal{T}(\Pi)$ does not contain negative literals. Before we go to the next section, let us look at the structure of program $\mathcal{T}(\Pi)$.

$\mathcal{T}(\Pi)$ is an r-ground program and consists of three parts:

- ▷ regular part, Π_R, is ground and consists of rules of the form:

$$h \leftarrow a_1,\ \ldots,\ a_m,\ not\ \ b_{m+1},\ \ldots,\ not\ \ b_n$$

  where h, a's and b's are r-atoms;

- ▷ middle part, Π_M, is r-ground and consists of rules of the form:

$$h \leftarrow a_1,\ \ldots,\ a_m,\ not\ \ b_{m+1},\ \ldots,\ not\ \ b_n$$

  where h and b's are r-atoms, a's are r-atoms or m-atoms and contains exactly one d-atom;

- ▷ defined part, Π_D, is non-ground and consists of rules of the form:

$$h \leftarrow a_1,\ \ldots,\ a_m,\ not\ \ b_{m+1},\ \ldots,\ not\ \ b_n$$

  where h is a d-atom, b's are r-atoms, a's are r-atoms or d-atoms or c-atoms

The next section describes the solver $\mathcal{AC}\,engine$ in detail.

5.2 $\mathcal{AC}\,engine$

Before we describe the solver in detail, let us introduce some notation and terminology. Let Π be a r-ground program with signature Σ and B be a set of ground extended r-literals of Σ.

- ▷ We will identify an expression $not\ (not\ a))$ with a.
- ▷ $pos(B) = \{a \in Atoms(\Sigma) \mid a \in B\}$,
  $neg(B) = \{a \in Atoms(\Sigma) \mid not\ a \in B\}$,
  $Atoms(B) = pos(B) \cup neg(B)$.
- ▷ A set, M, of atoms *agrees* with B if $pos(B) \subseteq M$ and $neg(B) \cap M = \emptyset$.
- ▷ B *covers* a set of atoms M, $covers(B, M)$, if $M \subseteq Atoms(B)$
- ▷ B is *inconsistent* if $pos(B) \cap neg(B) \neq \emptyset$.

We consider the following rule throughout the chapter.

$$h \leftarrow l_1,\ \ldots,\ l_m,\ not\ \ l_{m+1},\ \ldots,\ not\ \ l_n \tag{5.7}$$

where l's and h are atoms. Let r be a rule of Π and B be a set of extended literals of Σ.

- ▷ Rule r is *falsified* by a set of extended literals B, if there exists a literal $l_i \in body(r)$ such that $not\ l_i \in B$.
- ▷ Rule r is *active* w.r.t. B, if $pos(r) \cap neg(B) = \emptyset$ and $neg(r) \cap pos(B) = \emptyset$.

function *ACengine*(Π: r-ground program, B: set of r-literals)

[a] $S := expand(\Pi_R \cup \Pi_M, B)$

[b] **if** $inconsistent(S)$ **return** $false$

[c] **if** $covers(S, rAtoms(\Pi))$ **then**

[d] $V :=$ c_solve(Π_D, S, query(Π,S), A)

[e] **if** V = true **return** true {a-set: $pos(S) \cup m\_atoms|_A$}

[f] **else return** false

[g] **else** V := c_solve(Π_D, S, pos(query(Π,S)), A)

[h] **if** V = false **return** false

[i] $pick(l, \bar{S})$

[j] **if** *ACengine*(Π,S ∪ {l}) **then**

[k] **return** true

[l] **else return** *ACengine*(Π,S ∪ {*not* l})

Figure 5.1: $\mathcal{AC}engine$: computation of answer sets of Π

Definition 5.2.1. [**simplified answer set**] *Let* $M = A \cup X \cup D \cup M_c$ *be an answer set of a program* Π*, where* A, X, D *are sets of regular, mixed and defined atoms respectively and* M_c *is the set of c-atoms representing the intended interpretation of c-predicates in* P_c*. The set* $A \cup X$ *is called a simplified answer set of* Π*.*

5.2.1 Main Computation Cycle

The function $\mathcal{AC}engine$ shown in figure 5.2.1, takes as inputs r-ground $\mathcal{AC}(\mathcal{C})$ program Π and a set of ground extended r-literals B. The function returns true if it finds an answer set M of Π, such that B agrees with M. Otherwise, it returns false. The solver uses functions **expand**, **pick** and **c_solve**. The accurate description of these functions will be given in the following sections.

The function $\mathcal{AC}engine$ 5.2.1 executes the following steps:

(a) Function *expand* computes the set of consequences of $\Pi_R \cup \Pi_M$ and B. This set of ground extended r-literals is stored in S. The set S has the following properties:

- $B \subseteq S$
- every answer set of Π that agrees with B agrees with S.

The computation of these consequences is defined by a set of closure rules described in section 5.3.

(b) If S is inconsistent then $inconsistent(S)$ returns true else it returns false. If S is inconsistent then there exists no answer set of Π that agrees with B and hence in step (b) $\mathcal{AC}engine$ returns $false$.

(c) If S covers the set of r-atoms in Π then $covers(S, rAtoms(\Pi))$ returns true, else it returns false. If it returns true then steps (d) and (e) are executed otherwise step (f) is executed.

(d) Function c_solve uses a constraint logic programming solver (clp-solver) to compute an answer to a query $Q = query(\Pi, S)$, using program Π_D. The query Q, which is a conjunction of r-ground d-literals, is constructed from the program Π_M and the set S. The description on how the query is built is given in section 5.5.2. If clp-solver fails to solve the query then c_solve returns false. If c_solve returns $false$ then

- there exists no answer set of Π agreeing with B.

If the clp-solver successfully solved query Q then c_solve can return either $true$ or $maybe$. The conditions when $maybe$ is returned instead of $true$ are

described in 5.5.2. At this step, when S covers the r-atoms of Π, c_solve returns $true$ if clp-solver successfully solves Q. If c_solve returns $true$ then it also outputs a set of answer constraints, A, consisting of constraints on the variables from Q. The set A holds the property that for any solution θ of A, we have $\Pi_D \cup M_c \cup S \models \textit{d-lits}(Q)|_\theta^{vars(Q)}$. Let X be a candidate_mixed set formed by substituting c-variables in $mcv\_set(\Pi)$ (see section 4.2) by values using substitution θ. At this step, after a successful return from c_solve, the set $pos(S) \cup X$ has the following property:

- ◇ There exists a set of defined literals D such that
  $M = pos(S) \cup X \cup D \cup M_c$ is an answer set of Π agreeing with B.

(e) If c_solve returns true then $\mathcal{AC}\,engine$ returns $true$ and $M_o = pos(S) \cup X$ (where $X = mcv\_set(\Pi)|_\theta^{vars(\Pi)}$) is the simplified answer set of Π agreeing with B.

(f) If c_solve returns $false$ then there exists no answer set agreeing with B; hence step (f) returns $false$.

(g) Function c_solve uses a clp-solver to compute an answer to query Q, using program Π_D. At this step, only positive part of the query (built as described in section 5.5.2) is sent as input to c_solve. When the function terminates, it returns $true$ or $false$ or $maybe$. If it returns false then there is no answer set agreeing with B.

(h) If c_solve returns false then there exists no answer set agreeing with B and $false$ is returned by $\mathcal{AC}\,engine$.

(i) The function **pick** chooses a r-literal l undecided in S.

(j) function *ACengine* recursively calls itself with program Π and $S \cup \{l\}$ as inputs and checks if it can find answer sets of Π agreeing with $S \cup \{l\}$.

(l) If it cannot find answer set agreeing with $S \cup \{l\}$ then the function calls itself recursively with program Π and $S \cup \{not\ l\}$ as inputs and checks if it can find answer sets of Π agreeing with $S \cup \{not\ l\}$.

Proposition 5.2.1. *Let Π be a program and B be a set of ground extended literals input to ACengine. If ACengine returns* true *and a set A then A is a simplified answer set of Π agreeing with B.*

5.3 The *expand* Cycle

Function *expand* computes the set S of ground extended r-literals consisting of all of consequences of r-ground program $\Pi_R \cup \Pi_M$ and a set of ground extended r-literals B. If S is consistent then it has the following properties:

- $B \subseteq S$
- every answer set that agrees with B also agrees with S

The first subsection introduces two auxillary functions atleast and atmost, used by the main loop of expand, then the next subsection describes expand.

5.3.1 Functions atleast, atmost

5.3.1.1 Function atleast

Let us introduce some terminology. Let Π be an r-ground program and B be a set of ground extended r-literals. We define $lc_0(\Pi, B)$ as a set of ground extended literals that is minimal (set theoretic) and satisfies the following conditions:

1. If $r \in \Pi_R$, and $body(r) \subseteq B$, then $head(r) \in lc_0(\Pi, B)$.

2. If an r-atom h is not in the head of any *active* rule in $\Pi_R \cup \Pi_M$ with respect to B then *not* $h \in lc_0(\Pi, B)$

3. If r is the only *active* rule of $\Pi_R \cup \Pi_M$ with respect to B such that $h = head(r)$ and $h \in B$ then $r\_lits(r) \subseteq lc_0(\Pi, B)$

4. If $r \in \Pi_R$, $h = head(r)$, *not* $h \in B$, and all literals in the body of r except l_i belong to B, then *not* $l_i \in lc_0(\Pi, B)$.

Proposition 5.3.1. *If the set* $lc_0(\Pi, B) \cup B$ *is consistent then* $lc_0(\Pi, B)$ *is unique.*

Definition 5.3.1. *Given a program* Π *and a set of ground extended r-literals* B*, the lower closure of* Π *with respect to* B*, denoted by* $lc(\Pi, B)$*, is the set of ground extended r-literals defined as follows:*

$$lc(\Pi, B) = \begin{cases} lc_0(\Pi, B) \cup B & \text{if } lc_0(\Pi, B) \cup B \text{ is consistent,} \\ r\text{-}lits(\Pi) & \text{otherwise.} \end{cases}$$

The definition of $lc(\Pi, B)$ is similar to definition in [75, 72] and differs in conditions (1) and (4) of the computation of $lc_0(\Pi, B)$, where the conditions are applied only to rules of Π_R and not to Π_M.

Example 5.3.1. *Let*

$\Sigma = \{C_r = \{a, b\}, C_c = \{C_1 = [0 \ldots 86400],\ C_2 = [0 \ldots 1440]\},\ P_r = \{q(C_r), r(C_r), p(C_r, C_r), s(C_r, C_r)\},\ P_m = \{at(C_r, C_c\},\ P_c = \{\geq (C_c, C_c)\},\ P_d = \{d_1(C_r, C_r, C_c, C_c), d_2(C_r, C_c, C_c)\},\ F_c = \{min(C_c, C_c), max(C_c, C_c), *(C_c, C_c)\},\}$

and variables V_1 *and* V_2 *range over sorts* C_1 *and* C_2 *respectively. Let* Π *be as follows:*

$q(a).\quad q(b).$

$p(a,b) \leftarrow q(a),\ q(b),\ not\ s(a,b),\ at(a,V_1),\ at(b,V_2),\ d_1(a,b,V_1,V_2).$

$s(a,b) \leftarrow q(a),\ q(b),\ at(a,V_1),\ at(b,V_2),\ d_2(a,V_1,V_2).$

One can check from definition that

$lc(\Pi,\{p(a,b)\}) = \{\ p(a,b),\ q(a),\ q(b),\ not\ r(a),\ not\ r(b),\ not\ s(a,b)\ \};$

and $lc(\Pi,\{not\ p(a,b)\}) = \{\ not\ p(a,b), q(a), q(b), not\ r(a), not\ r(b)\ \}$

Proposition 5.3.2. *Let* Π *be a program, and* B *be a set of extended r-literals.*

- $lc(\Pi, B)$ *is monotonic with respect to its second argument.*
- $lc(\Pi, B)$ *is unique.*
- *If* M *is an answer set of* Π*, then* M *agrees with* B *iff* M *agrees with* $lc(\Pi,B)$.

To make our terminology compatible with that of Smodels [75], the function computing the lower closure $lc(\Pi, B)$ of a r-ground program Π and a set of ground extended r-literals B will be called $atleast$. Note that the proposition 5.3.2 implies that if $lc(\Pi, B)$ is inconsistent then there is no answer set of Π that agrees with B; otherwise, every answer set of Π that agrees with B agrees with $lc(\Pi, B)$.

5.3.1.2 Function atmost

Function $expand$ uses a function called $atmost$ to compute the atoms that can possibly be true in an answer set of Π agreeing with B. Before we describe function $atmost$, let us introduce some terminology.

By $\alpha(\Pi, B)$, we denote a program obtained from Π by

1. Removing all rules in Π whose bodies are falsified by B.
2. Removing all rules r in Π where $head(r)$ is false in B.
3. Removing all not-atoms from bodies of rules in $\Pi_R \cup \Pi_M$

4. Removing all c-lits and d-lits from bodies of rules in Π_M

The definition of $\alpha(\Pi, B)$ has been modified from [72] by adding step (4) to work with middle rules.

Definition 5.3.2. *The* upper closure *of a program* Π *with respect to* B*, denoted as* $up(\Pi, B)$*, is defined as the deductive closure of* $\alpha(\Pi, B)$.

Example 5.3.2. *Let*

$\Sigma = \{C_r = \{a, b\}, C_c = \{C_1 = [0 \ldots 86400],\ C_2 = [0 \ldots 1440]\},\ P_r =$
$\{q(C_r), r(C_r), p(C_r, C_r), s(C_r, C_r)\},\ P_m = \{at(C_r, C_c\},\ P_c = \{\geq (C_c, C_c)\},$
$P_d = \{d_1(C_r, C_r, C_c, C_c), d_2(C_r, C_c, C_c)\},\ \}$ *and variables* V_1 *and* V_2 *range over sorts* C_1 *and* C_2 *respectively. Let* Π *be as follows:*

$q(a).\quad q(b).$

$p(a, b) \leftarrow q(a),\ q(b),\ not\ s(a, b),\ at(a, V_1),\ at(b, V_2),\ d_1(a, b, V_1, V_2).$

$s(a, b) \leftarrow q(a),\ q(b),\ at(a, V_1),\ at(b, V_2),\ d_2(a, V_1, V_2).$

One can check that $\alpha(\Pi, \emptyset)$ *is as follows:*

$q(a).\quad q(b).$

$p(a, b) \leftarrow q(a),\ q(b).$

$s(a, b) \leftarrow q(a),\ q(b).$

and $up(\Pi, \emptyset) = \{\ p(a, b), q(a), q(b), s(a, b)\ \}.$

The program $\alpha(\Pi, \{s(a, b)\})$ *is as follows:*

$q(a).\quad q(b).$

$s(a, b) \leftarrow q(a),\ q(b).$

and $up(\Pi, \{s(a, b)\}) = \{\ q(a), q(b), s(a, b)\ \}$

Proposition 5.3.3. *Let* Π *be a program and* B *be a set of extended r-literals.*

▷ $\mathrm{up}(\Pi, B)$ *is unique*

▷ *If* M *is an answer set of* Π *agreeing with* B *then r-atoms*$(M) \subseteq \mathrm{up}(\Pi, B)$

To make our terminology compatible with $Smodels$, we call the function that computes the upper closure, $\mathrm{up}(\Pi, B)$, of a program Π and a set of ground extended r-literals B ,$atmost$.

5.3.2 The *expand* Function

The expand function computes the set of consequences of a r-ground program Π with respect to a set of ground extended r-literals B. The inputs of the function are the program $\Pi_R \cup \Pi_M$ and set B. The function shown in figure 5.2 has the following main steps:

a. The variable S is initialized to B.

c. The variable S_0 is initialized to S

d. **atleast** computes $\mathrm{lc}(\Pi_R \cup \Pi_M, S)$ and adds it to S.

e. **atmost** returns $\mathrm{up}(\Pi_R \cup \Pi_M, S)$. The set
$\{ not\ l \mid l \in r\text{-}atoms(\Pi), l \notin \mathrm{atmost}(\Pi, S)\}$ is added to S.

(Since atoms which do not belong to $\mathrm{up}(\Pi_R \cup \Pi_M, S)$ cannot be consequences of Π and S, expand adds the negation (not) of atoms not in $\mathrm{up}(\Pi_R \cup \Pi_M, S)$ to S.)

f. The steps (c), (d) and (e) are executed until either $(S = S_0)$ or S becomes inconsistent.

g. S is returned by $expand$.

function expand (Π : r-ground program, B : set of r-literals)

```
        % var S, S_0 : set of r-literals
[a]     S := B
[b]     do
[c]         S_0 := S
[d]         S := S ∪ atleast(Π, S)
[e]         S := S ∪ { not l | l ∈ r-atoms(Π), l ∉ atmost(Π, S)}
[f]     while ( (S ≠ S_0) and consistent(S) )
[g]     return S
```

Figure 5.2: Function expand

Proposition 5.3.4. *Let* S *be the output of* $\mathrm{expand}(\Pi, B)$. *If* S *is consistent then an answer set* Y *of* Π *agrees with* B *iff* Y *agrees with* S. *Otherwise, there is no answer set of* Π *that agrees with* B.

Example 5.3.3. *Let* Π *be from example 4.2.6, and* $B = \{q(1), q(2), r(2,1), p'(1,2)\}$, *we get* $lc(\Pi_R \cup \Pi_M, B) = S = \{q(1), q(2), r(2,1), not\ s(2,1), p'(1,2), p(1,2)\}$; $\alpha(\Pi_R \cup \Pi_M, S)$ *and* $up(\Pi_R \cup \Pi_M, S)$ *are given below.*

$u_{a_{[1,2]}}$: $q(1).\quad q(2).$

u_{b_1} : $p(1,2) \leftarrow q(1),\ q(2),\ r(1,2)$

u_{c_1} : $r(1,2) \leftarrow q(1),\ q(2)$

u_{c_2} : $r(2,1) \leftarrow q(2),\ q(1)$

u_{d_1} : $s(1,2) \leftarrow q(1),\ q(2)$

$u_{e_{11}}$: $p(1,1) \leftarrow p'(1,1)$

$u_{e_{21}}$: $p(1,2) \leftarrow p'(1,2)$

$u_{e_{31}}$: $p(2,1) \leftarrow p'(2,1)$

$u_{e_{41}}$: $p(2,2) \leftarrow p'(2,2)$

$u_{e_{12}}$: $p'(1,1) \leftarrow q(1),\ q(1)$

$u_{e_{22}}$: $p'(1,2) \leftarrow q(1),\ q(2)$

$u_{e_{32}}$: $p'(2,1) \leftarrow q(2),\ q(1)$

$u_{e_{42}}$: $p'(2,2) \leftarrow q(2),\ q(2)$

$up(\Pi_R \cup \Pi_M, S) = \{q(1), q(2), p'(1,1), p'(1,2),\ p'(2,1), p'(2,2), s(1,2),\ r(1,2),$
$r(2,1),\ p(1,1),\ p(1,2),\ p(2,1), p(2,2)\}$.

Function expand returns $S = \{q(1), q(2), r(2,1), not\ s(2,1), p'(1,2), p(1,2)\}$.

5.4 The Query: $query(\Pi, S)$

This section describes the construction of a formula, $query(\Pi, S)$, used as an input to function c_solve. Let us introduce some terminology. Let V be a c-variable of Π that ranges over an interval $C = [lower \ldots upper]$ where lower and upper are numerical values. By $c\_sort(V)$, we mean the constraint $(lower \leq V) \bigwedge (V \leq upper)$. Given a set of variables Y, $c\_sort(Y) = \bigwedge_{V \in Y} c\_sort(V)$.

Example 5.4.1. *Let variables* V_1 *and* V_2 *range over the number of seconds* $[0 \ldots 86400]$ *and minutes* $[0 \ldots 1440]$ *of a day respectively. We get,*

$$c\_sort(\{V_1, V_2\}) = (0 \leq V_1) \wedge (V_1 \leq 86400) \wedge (0 \leq V_2) \wedge (V_2 \leq 1440)$$

Definition 5.4.1. **[pe(Π,S)]** *Given a r-ground rule* r *and a set of ground extended r-literals S, we define partial evaluation of* r *with respect to S,* $pe(r, S)$*, as a rule* r' *such that:*

> *if* $head(r)$ *is undecided with respect to S or r-lits(r) is falsified by S then* r' *is an empty rule. Otherwise,*

$$head(r') = head(r)$$

$$body(r') = \begin{cases} c\text{-}lits(r) \cup d\text{-}lits(r) & \text{if } c\text{-}lits(r) \cup d\text{-}lits(r) \neq \emptyset, \\ \{true\} & \text{otherwise.} \end{cases}$$

Given a program Π*, we define* $pe(\Pi, S) = \{pe(r, S) \mid r \in \Pi\}$.

Example 5.4.2. *Let*

$\Sigma = \{C_r = \{a, b\}, C_c = \{C_1 = [0 \ldots 86400],\ C_2 = [0 \ldots 1440]\},\ P_r =$
$\{q(C_r), r(C_r), p(C_r, C_r), s(C_r, C_r)\},\ P_m = \{at(C_r, C_c\},\ P_c = \{\geq (C_c, C_c)\},\ P_d =$
$\{d_1(C_r, C_r, C_c, C_c), d_2(C_r, C_c, C_c)\},\ F_c = \{min(C_c, C_c), max(C_c, C_c), *(C_c, C_c)\},\}$
and variables V_1 *and* V_2 *range over sorts* C_1 *and* C_2 *respectively. Let* Π *be as follows:*

$q(a).\quad q(b).$

$p(a, b) \leftarrow q(a),\ q(b),\ not\ s(a, b),\ at(a, V_1),\ at(b, V_2),\ d_1(a, b, V_1, V_2).$

$s(a, b) \leftarrow q(a),\ q(b),\ at(a, V_1),\ at(b, V_2),\ d_2(a, V_1, V_2).$

$d_1(a, b, V_1, V_2) \leftarrow r(a), r(b),\ min(V_1, V_2 * 60) \geq 2000.$

$d_2(a, V_1, V_2) \leftarrow not\ r(a),\ max(V_1, V_2 * 60) \geq 100.$

If $S_1 = \{q(a),\ q(b),\ p(a, b)\}$ *then* $pe(\Pi, S_1)$ *is as follows:*

$q(a) \leftarrow true.$

$q(b) \leftarrow true.$

$p(a, b) \leftarrow d_1(a, b, V_1, V_2).$

If $S_2 = \{q(a),\ q(b),\ p(a, b), not\ s(a, b)\}$ *then* $pe(\Pi, S_2)$ *is as follows:*

$q(a) \leftarrow true.$

$q(b) \leftarrow true.$

$p(a,b) \leftarrow d_1(a,b,V_1,V_2).$

$s(a,b) \leftarrow d_2(a,V_1,V_2).$

Given a rule r, we denote the formula obtained by conjunction of the set of literals in body of r by $\bigwedge body(r)$. For example, $\bigwedge body(r) = c \wedge d$ for the rule $a \leftarrow c, d$. We also assume that $\neg(\neg p) = p$. Now we describe $query(\Pi, S)$.

Definition 5.4.2. [**query(Π,S)**] *The formula* $query(\Pi, S)$*, is constructed via two auxiliary formulas* q_0 *and* q *defined as follows:*

1. *If* $h \in S$ *then* $q_0(\Pi, S, h) = \bigvee_{r \in pe(\Pi,S), h=head(r)} \bigwedge body(r)$.
2. *If not* $h \in S$ *then* $q_0(\Pi, S, h) = \bigwedge_{r \in pe(\Pi,S), h=head(r)} \neg \bigwedge body(r)$.
3. $q(\Pi, S) = \bigwedge_{h \in A} q_0(\Pi, S, h)$*, where*
   $A = \{h | h \in Atoms(S), h \in head(r), r \in \Pi_M\}$.

Finally, $query(\Pi, S)$ *is constructed as follows:*
$query(\Pi, S) = q(\Pi, S) \bigwedge c\_sort(Y)$*, where* Y *is the set of c-variables from* $mcv\_set(\Pi)$ *(see section 4.2).*

Example 5.4.3. *Let* Σ, Π, S_1, S_2, $pe(\Pi, S_1)$ *and* $pe(\Pi, S_2)$ *be as in example 5.4.2. We construct* $query(\Pi, S_1)$ *as follows:*

$q_0(\Pi, S_1, q(a)) = true \quad q_0(\Pi, S_1, q(b)) = true$

$q_0(\Pi, S_1, p(a,b)) = d_1(a,b,V_1,V_2)$

$q(\Pi, S_1) = d_1(a,b,V_1,V_2)$

$query(\Pi, S_1) = d_1(a,b,V_1,V_2) \wedge (0 \leq V_1) \wedge (V_1 \leq 86400) \wedge (0 \leq V_2) \wedge (V_2 \leq 1440)$

We construct $query(\Pi, S_2)$ *as follows:*

$q_0(\Pi, S_2, q(a)) = true \quad q_0(\Pi, S_1, q(b)) = true$

$q_0(\Pi, S_2, p(a,b)) = d_1(a,b,V_1,V_2)$

$q_0(\Pi, S_2, s(a,b)) = \neg d_2(a,V_1,V_2)$

$q(\Pi, S_2) = d_1(a, b, V_1, V_2)$

$query(\Pi, S_2) = d_1(a, b, V_1, V_2) \wedge \neg d_2(a, V_1, V_2) \wedge (0 \leq V_1) \wedge (V_1 \leq 86400) \wedge (0 \leq V_2) \wedge (V_2 \leq 1440)$

Note that if a rule $r \in pe(\Pi, S)$ has $false$ in the head then rule (2) from definiton 5.4.2 is used to build $q_0(\Pi, S, false)$. The application of transformation tr_1 of $\mathcal{T}ranslator$ ensures that for every rule $r \in \Pi_M$, if $h = head(r)$ then r is the only rule in Π with h in the head. Therefore, for any r-atom h in the head of rules in Π_M, there is no disjunction from step (1) of definition of $q_0(\Pi, S, h)$ above. Also the application of transformation tr_2 ensures that for every rule $r \in \Pi_M$, there is at most one d-literal in the body of r and no c-literals in body of r. Therefore, for any r-atom h in the head of rules in Π_M, there is no disjunction from step (2) of definition of $q_0(\Pi, S, h)$ above. *These two transformations guarantee that* $query(\Pi, S)$ *does not contain disjunction.* The following example shows how the transformations remove disjunction from the query.

Example 5.4.4. *Let* Σ *be signature as in example 5.4.2. Let* Π *be as follows:*

$q(a).$ $\quad q(b).$

$p(a, b) \leftarrow q(a),\ q(b),\ not\ s(a, b),\ at(a, V_1),\ at(b, V_2),\ d_1(a, b, V_1, V_2), d_3(V_1).$

$p(a, b) \leftarrow q(a),\ q(b),\ at(a, V_1),\ at(b, V_2),\ d_2(a, V_1, V_2), d_3(V_2).$

$d_1(a, b, V_1, V_2) \leftarrow min(V_1, V_2 * 60) \geq 2000.$

$d_2(a, V_1, V_2) \leftarrow not\ r(a),\ max(V_1, V_2 * 60) \geq 100.$

$d_3(V) \leftarrow V * 60 \geq 100.$

If $S_1 = \{q(a),\ q(b),\ p(a, b)\}$ *then* $pe(\Pi, S_1)$ *is as follows:*

$q(a) \leftarrow true.$

$q(b) \leftarrow true.$

$p(a, b) \leftarrow d_1(a, b, V_1, V_2),\ d_3(V_1).$

$p(a,b) \leftarrow d_2(a,V_1,V_2),\ d_3(V_2).$

We construct $\mathrm{query}(\Pi,S_1)$ *as follows:*

$q_0(\Pi,S_1,q(a)) = \mathrm{true} \quad q_0(\Pi,S_1,q(b)) = \mathrm{true}$

$q_0(\Pi,S_1,p(a,b)) = (d_1(a,b,V_1,V_2) \wedge d_3(V_1)) \vee (d_2(a,V_1,V_2) \wedge d_3(V_2))$

$q(\Pi,S_1) = (d_1(a,b,V_1,V_2) \wedge d_3(V_1)) \vee (d_2(a,V_1,V_2) \wedge d_3(V_2))$

$Q_1 = \mathrm{query}(\Pi,S_1) = ((d_1(a,b,V_1,V_2) \wedge d_3(V_1)) \vee (d_2(a,V_1,V_2) \wedge d_3(V_2))) \wedge (0 \leq V_1) \wedge (V_1 \leq 86400) \wedge (0 \leq V_2) \wedge (V_2 \leq 1440)$

The disjunction in the query Q_1 *occurs because there are two rules with* $p(a,b)$ *in the head. This is removed by transformation* tr_1*. Now consider* $S_2 = \{q(a), q(b), not\ p(a,b)\}$*, the query*

$Q_2 = \mathrm{query}(\Pi,S_2) = ((\neg d_1(a,b,V_1,V_2) \vee \neg d_3(V_1)) \wedge (\neg d_2(a,V_1,V_2) \vee \neg d_3(V_2))) \wedge (0 \leq V_1) \wedge (V_1 \leq 86400) \wedge (0 \leq V_2) \wedge (V_2 \leq 1440)$*. Note that there is disjunction in* Q_2 *because there is more than one literal in body of* $\mathrm{pe}(\Pi,S_2)$ *rules with head* $p(a,b)$*. This is removed by transformation* tr_2*.*

Hence, $\mathrm{query}(\Pi,S)$ is a conjunction of d-literals from Π and c-atoms of variable sorts.

As described in section 5.2, the formula $Q = \mathrm{query}(\Pi,S)$ is used as an input to the function c_solve in $\mathcal{AC}$*engine*. The function c_solve uses constraint logic programming techniques and attempts to solve Q. We can improve the efficiency of $\mathcal{AC}$*engine* by using only a part of the query Q as input to* c_solve. That is we delay solving the whole query and check whether only a part of the query is satisfiable. We now describe the part of the query Q that is sent as input to c_solve.

Let $Q = \mathrm{query}(\Pi,S)$ be the conjunction of literals $d_1,\ldots d_m, c_1,\ldots,c_n$, represented by the set of literals $\{d_1,\ldots d_m, c_1,\ldots,c_n\}$ where $d_1,\ldots d_m$ are

d-literals and $c_1, \ldots, c_n$ are c-literals. Let us remove from Q all d_i's that do not satisfy the following property called *r-support*: *there exists a rule* $r \in \Pi_M$ *with* $d_i \in body(r)$ *and* $r\_lits(r) \subseteq S$. The remaining set of literals of Q is used as input to c_solve. It is not difficult to show that the properties (see section 5.2.1) of the algorithm *ACengine* are still satisfied by using only a part of the query as described above.

Example 5.4.5. *Let* Π, S_1 *and* S_2 *be as in example 5.4.2. We get* $query(\Pi, S_1)$ *and* $query(\Pi, S_2)$ *as constructed in example 5.4.3. Since* $d_1(a, b, V_1, V_2)$ *does not satisfy the r-support property, the subset of* $query(\Pi, S_1)$ *that is input to c_solve is* $Q_1 = (0 \leq V_1) \wedge (V_1 \leq 86400) \wedge (0 \leq V_2) \wedge (V_2 \leq 1440)$. *The subset of* $query(\Pi, S_2)$ *from example 5.4.3 that is used as input to c_solve is* $Q_2 = query(\Pi, S_2)$.

5.5 The c_solve cycle

The c_solve function takes as inputs non-ground program Π_D, a set of ground extended r-literals S and a formula $Q = query(\Pi, S)$, where program Π_D is the set of defined rules of Π. Intuitively, S is the partial interpretation computed so far by function *expand* (see section 5.2). The query Q is a formula formed by d-literals and c-literals of Π (see section 5.4). The output of c_solve can be *true*, *false* or *maybe*. Before we describe the function, we describe an auxillary function called *clp-solver* that is used by c_solve.

5.5.1 The *clp-solver of ACengine*

The c_solve function uses a constraint logic programming solver (*clp-solver*) to find a consistent set of ground extended r-literals R and a set of constraints A such that, for any solution θ of A, $\Pi_D \cup R \cup M_c \models Q|_\theta^{vars(Q)}$, where M_c is the

intended interpretation of c-atoms in $\Pi_D \cup Q$.

The *clp-solver* function is a modified version of the standard constraint logic programming solver [58, 56]. The modification stems from the fact that rules of program Π_D might contain extended r-literals in the body. The definitions of r-literals belong to $\Pi_R \cup \Pi_M$ and therefore the original solver clp cannot derive the truth values of these literals. So we modify clp to work with r-literals. First we briefly describe clp [55], and then describe *clp-solver* of $\mathcal{AC}engine$.

5.5.1.1 Function clp

The inputs of the clp are a non-ground constraint logic program Π and a query Q which is a conjunction of literals. If clp terminates then the output is $true$, $false$ or $maybe$. If it returns $true$ then it also returns a set of constraints. The output constraints satisfy the following property: *there exists at least one solution for output constraints and any solution of the output constraints is a solution of input query.* If it returns $false$ then *there exists no solution for input query.* If it returns $maybe$ then it also returns a set of constraints. The output constraints satisfy the following property: *if there is a solution to the output constraints then it is a solution for the input query.* Note however that such a solution might not exist. More detailed description of clp requires some terminology from [57].

Let C be a set of formulas constructed from c-atoms (primitive constraints) referred to as a *constraint store*. The set of constraint formulas in C is divided into two categories: $solved$ and $delayed$. The delayed constraints are non-linear constraints and maybe difficult to check for solvability. The constraints in C are *solvable* if the set of solved constraints in C is consistent i.e. has a solution. Given a program Π, a query $Q = l_1, \ldots, l_n$, and a constraint store C, there is a

derivation step from a pair $\langle Q, C\rangle$ to another pair $\langle Q_1, C_1\rangle$ if:

- $Q_1 = l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n$, where l_i is a constraint formula and C_1 is a (possibly simplified) set of constraints equivalent to $C \cup \{l_i\}$. Furthermore, C_1 is solvable;
- or $Q_1 = l_1, \ldots, l_{i-1}, x_1 = t_1, \ldots, x_k = t_k, b_1, \ldots, b_m, l_{i+1}, \ldots, l_n$, where there is a rule $r \in \Pi$ such that head of r, h, can be unified with l_i, x_i are term parameters from l_i and t_i are term parameters from h, and $\mathrm{body}(r)$ is unified with the set of literals $\{b_1, \ldots, b_m\}$; and $C_1 = C$.

A *derivation sequence* is a possibly infinite sequence of pairs $\langle Q_0, C_0\rangle, \langle Q_1, C_1\rangle \ldots$, starting with an initial query Q_0 and initial constraint store C_0, such that there is a derivation step to each pair $\langle Q_i, C_i\rangle, i > 0$, from the preceding pair $\langle Q_{i-1}, C_{i-1}\rangle$.

Example 5.5.1. *Let* Π *be a program as follows:*

$$
\begin{aligned}
d(X,Y) &\leftarrow X^2 \geq Y,\ e(X).\\
e(X) &\leftarrow X * 4 \leq 20.\\
e(X) &\leftarrow X = 2.
\end{aligned}
$$

Let $Q = d(X,Y)$ *and let* $C = \emptyset$. *Then the following is a derivation sequence:*

$$
\begin{aligned}
\langle Q,\ C\rangle &= \langle\ \{d(X,Y)\},\ \emptyset\ \rangle\\
\langle Q_1, C_1\rangle &= \langle\ \{X^2 \geq Y, e(X)\},\ \emptyset\ \rangle\\
\langle Q_2, C_2\rangle &= \langle\ \{e(X)\},\ \{X^2 \geq Y\}\ \rangle\\
\langle Q_3, C_3\rangle &= \langle\ \{X * 4 \leq 20\},\ \{X^2 \geq Y\}\ \rangle\\
\langle Q_4, C_4\rangle &= \langle\ \emptyset,\ \{X^2 \geq Y,\ X \leq 5\}\ \rangle
\end{aligned}
$$

Note that when new constraints are added to the store, the old constraints maybe reduced to simpler equivalent constraints. For instance, in example 5.5.1, when we

add a new constraint $X * 4 \leq 20$ to C_3, we get the new store as C_4 which is equivalent to $\{X^2 >= Y,\ X * 4 \leq 20\}$. Similarly a non-linear constraint which is delayed might be reduced to become a linear constraint. When a delayed constraint becomes linear it becomes a solved constraint. The following example shows a derivation sequence where a non-linear constraint becomes linear.

Example 5.5.2. *Let* Π *be the program from example 5.5.1*
Let $Q = d(X, Y)$ *and let* $C = \emptyset$. *Then a derivation sequence (using second rule of* $e(X)$*) is as follows:*

$$\langle Q, C \rangle = \langle\ \{d(X, Y)\},\ \emptyset\ \rangle$$
$$\langle Q_1, C_1 \rangle = \langle\ \{X^2 \geq Y, e(X)\},\ \emptyset\ \rangle$$
$$\langle Q_2, C_2 \rangle = \langle\ \{e(X)\},\ \{X^2 \geq Y\}\ \rangle$$
$$\langle Q_3, C_3 \rangle = \langle\ \{X = 2\},\ \{X^2 \geq Y\}\ \rangle$$
$$\langle Q_4, C_4 \rangle = \langle\ \emptyset,\ \{Y \leq 4,\ X = 2\}\ \rangle$$

Notice that C_3 has non-linear constraints and C_4 is linear.
A sequence is successful if it is finite and its last element is $\langle \emptyset, C_n \rangle$, where C_n is a set of solved constraints. A sequence is *conditionally successful* if it is finite and its last element is $\langle \emptyset, C_n \rangle$, where C_n consists of delayed and possibly some solved constraints. A sequence is *finitely failed* if it is finite, neither successful nor conditionally successful, and such that no derivation step is possible from its last element. The derivation sequences in examples 5.5.1 and 5.5.2 are conditionally successful and successful respectively.
There can be several derivation sequences for a query with respect to a program. Each of these sequences can be successful, conditionally successful, finitely failed or infinite.
Here is a brief description of clp. Given a program Π and a query Q, clp

non-deterministically generates a derivation sequence D starting with $\langle Q, \emptyset \rangle$. If the sequence D finitely fails then clp backtracks to generate another derivation sequence. If the sequence D is successful (conditionally successful) then it returns true (*maybe*) and C, which is the set of c-atoms from the constraint store of the last element in D. The constraints in C are called *answer constraints*. If all derivation sequences are finitely failed then clp returns false.

Proposition 5.5.1. *Let* Π *be a constraint logic program and* Q *be a query such that neither* Π *nor* Q *contains* $\neg$ *and not . Let* V_Q *be the variables in* Q *and* M_c *be the set of intended interpretations of c-atoms in* Π*. Then*

- ▷ *If* clp *returns* true *or* maybe *then* $gr(\Pi) \cup M_c \models Q|_{\theta(A)}^{V_Q}$*, where* $\theta(A)$ *is any solution of answer constraints* A*.*
- ▷ *If* clp *returns* false *then for any substitution* θ *of* V_Q*,* $gr(\Pi) \cup M_c \not\models Q|_{\theta}^{V_Q}$.

The non-deterministic selection is implemented via depth first strategy. Hence, there are cases when there is a successful derivation sequence and clp is unable to find it. For instance, if D is infinite then clp keeps building D and does not terminate, but there might be other successful derivation sequences.

5.5.1.2 Constructive Negation

Note that a query $Q = query(\Pi, S)$ as defined in section 5.4 contains negative ($\neg$) literals. The definition of derivative step in function clp does not give a derivation for negative literals. So we need to modify the derivation step to allow negative literals in the query. This section recalls the concept of constructive negation from [96, 97, 34], that is used in *clp-solver* function.

Let Π be a program and Q be a query. Suppose Q and Π do not contain ($\neg$) negative literals. Let $D = \{D_1, \ldots, D_k\}$ be the set of all possible successful and conditionally successful derivation sequences of the pair $\langle Q, \emptyset \rangle$ as defined in previous section (5.5.1.1).

The query Q is logically equivalent to:

$$Q \equiv C_1 \vee \cdots \vee C_k$$

where C_i is the conjunction of formulas in the constraint store of the last element of sequence D_i.

Example 5.5.3. *Let Π be a program as follows:*

$$\begin{aligned} d(X,Y) &\leftarrow X \leq 3,\ Y > 4. \\ d(X,Y) &\leftarrow X > 5, X + Y < 20. \\ p(X,Y) &\leftarrow X^2 < 4, d(X,Y). \\ p(X,Y) &\leftarrow X^2 - Y^2 > 4. \end{aligned}$$

There are two derivation sequences for $\langle d(X,Y), \emptyset \rangle$ with answer constraints as: $C_1 = X \leq 3 \wedge Y > 4$ *and* $C_2 = X + Y < 20 \wedge X > 5$. *We get,*

$$d(X,Y) \equiv (X \leq 3 \wedge Y > 4) \vee (X + Y < 20 \wedge X > 5)$$

There are three derivation sequences for $\langle p(X,Y), \emptyset \rangle$ with answer constraints as:

$C_1 = X^2 < 4 \wedge X \leq 3 \wedge Y > 4$; $C_2 = X^2 < 4 \wedge X + Y < 20 \wedge X > 5$ *and*

$C_3 = Y^2 + 4 < X^2$. *We get,*

$$p(X,Y) \equiv (X^2 < 4 \wedge X \leq 3 \wedge Y > 4) \vee (X^2 < 4 \wedge X + Y < 20 \wedge X > 5) \vee (Y^2 + 4 < X^2)$$

Consider the negative goal, $\neg Q$. We can write it as

$$\neg Q \equiv \neg(C_1 \vee \cdots \vee C_k)$$

$$\neg Q \equiv \neg C_1 \wedge \cdots \wedge \neg C_k$$

where $Q \equiv C_1 \vee \cdots \vee C_k$.

Definition 5.5.1. [*c-neg*(Π, Q, C)] *Given a program* Π *and a query* Q*, let* $D = \{D_1, \ldots, D_k\}$ *be the set of all possible successful and conditionally successful derivation sequences of* $\langle Q, C\rangle$*. We define* *c-neg*$(\Pi, Q, C) = \neg(C_1 \vee \cdots \vee C_k)$*, where* C_i *is the answer constraint of derivation sequence* D_i*.*

Example 5.5.4. *Let* Π *be as in example 5.5.3. We get* $\neg d(X, Y) \equiv \neg((X \leq 3 \wedge Y > 4) \vee (X + Y < 20 \wedge X > 5))$*. Simplifying, we get,* $\neg d(X, Y) \equiv (\neg(X \leq 3) \vee \neg(Y > 4)) \wedge (\neg(X + Y < 20) \vee \neg(X > 5))$*. Further simplifying, we get,* $\neg d(X, Y) \equiv ((X > 3) \vee (Y \leq 4)) \wedge ((X + Y \geq 20) \vee (X \leq 5))$*. Likewise, we get* $\neg p(X, Y) \equiv \neg((X^2 < 4 \wedge X \leq 3 \wedge Y > 4) \vee (X^2 < 4 \wedge X + Y < 20 \wedge X > 5) \vee (Y^2 + 4 < X^2))$*. Simplifying we get,* $\neg p(X, Y) \equiv (X^2 \geq 4 \vee X > 3 \vee Y \leq 4) \wedge (X^2 \geq 4 \vee X + Y \geq 20 \vee X \leq 5) \wedge (Y^2 + 4 \geq X^2)$.

5.5.1.3 Function *clp-solver*

The function *clp-solver* is obtained from clp function by two modifications. The modifications are as follows:

- ▷ Apart from the constraint store C, *clp-solver* maintains a set of ground extended r-literals referred to as a *regular store* denoted by R. This

modification stems from the fact that the definitions of r-literals are not in the input of *clp-solver*.

- ▷ The derivation step is modified to deal with r-literals and negative d-literals ($\neg$) in the query.

We give the definition of a derivation step in two steps. First, we give the definition of a derivation step for programs and queries not containing $\neg$. Then we give the definition for when the query contains ($\neg$) negation.

Definition 5.5.2. [Derivation Step$^{\not\neg}$] *Let* Π *be a program and* $Q = l_1, \ldots, l_n$ *be a query both not containing* $\neg$. *Let* C *be a constraint store and* R *be a regular store. We say that there is a derivation step from a tuple* $\langle Q, C, R\rangle$ *to another tuple* $\langle Q_1, C_1, R_1\rangle$ *if:*

(1). $Q_1 = l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n$, *where* l_i *is a constraint formula;* $R_1 = R$ *and* C_1 *is a (possibly simplified) set of constraints equivalent to* $C \cup \{l_i\}$. *Furthermore,* C_1 *is solvable;*

(2). *or* $Q_1 = l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n$, *where* l_i *is a r-literal;* $C_1 = C$ *and* $R_1 = R \cup \{l_i\}$. *Furthermore,* R_1 *is consistent;*

(3). *or* $Q_1 = l_1, \ldots, l_{i-1}, x_1 = t_1, \ldots, x_k = t_k, b_1, \ldots, b_m, l_{i+1}, \ldots, l_n$, *where* $l_i = d(x_1, \ldots, x_k)$ *is a positive d-literal and there is a rule* $r \in \Pi$ *such that head of* r, $d(t_1, \ldots, t_k)$, *can be unified with* l_i *and* $\mathrm{body}(r)$ *can be unified with set* $\{b_1, \ldots, b_m\}$; $C_1 = C$; *and* $R_1 = R$.

A *derivation sequence* is a possibly infinite sequence of tuples $\langle Q_0, C_0, R_0\rangle$, $\langle Q_1, C_1, R_1\rangle \ldots$, starting with an initial tuple $\langle Q_0, C_0, R_0\rangle$, such that there is a derivation step to each element $\langle Q_i, C_i, R_i\rangle, i > 0$, from the preceding element $\langle Q_{i-1}, C_{i-1}, R_{i-1}\rangle$. A sequence is successful if it is finite and its last element is

$\langle \emptyset, C_n, R_n \rangle$, where C_n contains only solved constraints. A sequence is *conditionally successful* if it is finite and its last element is $\langle \emptyset, C_n, R_n \rangle$, where C_n contains solvable and delayed constraints. A sequence is *finitely failed* if it is finite, neither successful nor conditionally successful, and such that no derivation step is possible from its last element.

Example 5.5.5. *Let predicates* d, e *be defined predicates and* p *be a regular predicate. Let* Π *be a program as follows:*

$$\begin{aligned} d(X_1, Y_1) &\leftarrow X_1^3 \geq Y_1,\ e(X_1, Y_1). \\ e(X_2, Y_2) &\leftarrow X_2 = 2, X_2 + Y_2 \leq 20. \\ e(X_3, Y_3) &\leftarrow p(X_3), Y_3 * 4 \leq 20. \end{aligned}$$

Variables X_1, X_2, X_3 *are regular in the above program. Let* $Q = d(2, Z_1), d(4, Z_2)$*;* $C = \emptyset$ *and* $R = \emptyset$*. Then a derivation sequence is as follows: (To follow the steps easily, the constraint store is not simplified)*

$$\begin{aligned} \langle Q,\ C,\ R \rangle &= \langle\ \{d(2, Z_1), d(4, Z_2)\},\ \emptyset,\ \emptyset\ \rangle \\ \langle Q_1, C_1, R_1 \rangle &= \langle\ \{X_1 = 2, Z_1 = Y_1, X_1^3 \geq Y_1, e(X_1, Y_1), d(4, Z_2)\},\ \emptyset,\ \emptyset\ \rangle \\ \langle Q_2, C_2, R_2 \rangle &= \langle\ \{Z_1 = Y_1, X_1^3 \geq Y_1, e(X_1, Y_1), d(4, Z_2)\},\ \{X_1 = 2\},\ \emptyset\ \rangle \\ \langle Q_3, C_3, R_3 \rangle &= \langle\ \{X_1^3 \geq Y_1, e(X_1, Y_1), d(4, Z_2)\}, \{X_1 = 2, Z_1 = Y_1\},\ \emptyset\ \rangle \end{aligned}$$

$\langle Q_4, C_4, R_4\rangle = \langle\ \{e(X_1, Y_1), d(4, Z_2)\},\ \{X_1 = 2, Z_1 = Y_1, X_1^3 \geq Y_1\},\ \emptyset\ \rangle$

$\langle Q_5, C_5, R_5\rangle = \langle\ \{X_1 = X_2, Y_1 = Y_2, X_2 = 2, X_2 + Y_2 \leq 20, d(4, Z_2)\},$
$\{X_1 = 2, Z_1 = Y_1, X_1^3 \geq Y_1\},\ \emptyset\ \rangle$

$\langle Q_6, C_6, R_6\rangle = \langle\ \{Y_1 = Y_2, X_2 = 2, X_2 + Y_2 \leq 20, d(4, Z_2)\},$
$\{X_1 = 2, Z_1 = Y_1, X_1^3 \geq Y_1, X_1 = X_2\},\ \emptyset\ \rangle$

$\langle Q_7, C_7, R_7\rangle = \langle\ \{X_2 = 2, X_2 + Y_2 \leq 20, d(4, Z_2)\},$
$\{X_1 = 2, Z_1 = Y_1, X_1^3 \geq Y_1, X_1 = X_2, Y_1 = Y_2\},\ \emptyset\ \rangle$

$\langle Q_8, C_8, R_8\rangle = \langle\ \{X_2 + Y_2 \leq 20, d(4, Z_2)\},$
$\{X_1 = 2, Z_1 = Y_1, X_1^3 \geq Y_1, X_1 = X_2, Y_1 = Y_2, X_2 = 2\},\ \emptyset\ \rangle$

$\langle Q_9, C_9, R_9\rangle = \langle\ \{d(4, Z_2)\},$
$\{X_1 = 2, Z_1 = Y_1, X_1^3 \geq Y_1, X_1 = X_2, Y_1 = Y_2, X_2 = 2, X_2 + Y_2 \leq 20\},\ \emptyset\ \rangle$

$\langle Q_{10}, C_{10}, R_{10}\rangle = \langle\ \{X_4 = 4, Y_4 = Z_2, X_4^3 \geq Y_4, e(X_4, Y_4)\},$
$\{X_1 = 2, Z_1 = Y_1, X_1^3 \geq Y_1, X_1 = X_2, Y_1 = Y_2, X_2 = 2, X_2 + Y_2 \leq 20\},\ \emptyset\ \rangle$

$\langle Q_{11}, C_{11}, R_{11}\rangle = \langle\ \{Y_4 = Z_2, X_4^3 \geq Y_4, e(X_4, Y_4)\}, C_{10} \cup \{X_4 = 4\},\ \emptyset\ \rangle$

$\langle Q_{12}, C_{12}, R_{12}\rangle = \langle\ \{X_4^3 \geq Y_4, e(X_4, Y_4)\}, C_{10} \cup \{X_4 = 4, Y_4 = Z_2\},\ \emptyset\ \rangle$

$\langle Q_{13}, C_{13}, R_{13}\rangle = \langle\ \{e(X_4, Y_4)\}, C_{10} \cup \{X_4 = 4, Y_4 = Z_2, X_4^3 \geq Y_4\},\ \emptyset\ \rangle$

$\langle Q_{14}, C_{14}, R_{14}\rangle = \langle\ \{X_4 = X_5, Y_4 = Y_5, p(X_5), Y_5 * 4 \leq 20\}, C_{13},\ \emptyset\ \rangle$

$\langle Q_{15}, C_{15}, R_{15}\rangle = \langle\ \{Y_4 = Y_5, p(X_5), Y_5 * 4 \leq 20\}, C_{13} \cup \{X_4 = X_5\},\ \emptyset\ \rangle$

$\langle Q_{16}, C_{16}, R_{16}\rangle = \langle\ \{p(X_5), Y_5 * 4 \leq 20\}, C_{13} \cup \{X_4 = X_5, Y_4 = Y_5\},\ \emptyset\ \rangle$

$\langle Q_{17}, C_{17}, R_{17}\rangle = \langle\ \{Y_5 * 4 \leq 20\}, C_{13} \cup \{X_4 = X_5, Y_4 = Y_5\},\ \{p(4)\}\ \rangle$

$\langle Q_{18}, C_{18}, R_{18}\rangle = \langle\ \emptyset, C_{13} \cup \{X_4 = X_5, Y_4 = Y_5, Y_5 * 4 \leq 20\},\ \{p(4)\}\ \rangle$

The final regular store $R_{18} = \{p(4)\}$ *and the constraint store is* $C_{18} = \{X_1 = 2, Z_1 = Y_1, X_1^3 \geq Y_1, X_1 = X_2, Y_1 = Y_2, X_2 = 2, X_2 + Y_2 \leq 20, X_4 = 4, Y_4 = Z_2, X_4^3 \geq Y_4, X_4 = X_5, Y_4 = Y_5, Y_5 * 4 \leq 20\}$. *Simplifying the constraint store in terms of variables* Z_1 *and* Z_2*, we get,* $\{Z_1 \leq 8, Z_1 \leq 16, Z_2 \leq 64, Z_2 \leq 5\}$ *which is equivalent to* $\{Z_1 \leq 8, Z_2 \leq 5\}$.

Note that if the initial query Q is r-ground, then according to the syntactic restriction (3) in chapter 4, when a r-literal l is unified and added to Q using derivation step (3), it is ground. *Therefore, in derivation step (2) when a regular literal is added to regular store it will be a ground r-literal.*

Example 5.5.6. *Let predicates* d, e *be defined predicates and* p *be a regular predicate. Let* Π *be a program as follows:*

$$\begin{aligned} d(X,Y) &\leftarrow X^2 \geq Y,\ e(X). \\ e(X) &\leftarrow p(X,3), X > 4. \\ e(X) &\leftarrow not\ p(X,2). \end{aligned}$$

Let $Q = d(6,Y)$ *and* $C = \emptyset$ *and* $R = \emptyset$. *Then (a slightly simplified version of) a derivation sequence maybe written as follows:*

$$\begin{aligned} &\langle\ \{d(6,Y)\},\ \emptyset,\ \emptyset\ \rangle \\ &\langle\ \{36 \geq Y, e(6)\},\ \emptyset,\ \emptyset\ \rangle \\ &\langle\ \{e(6)\},\ \{36 \geq Y\},\ \emptyset\ \rangle \\ &\langle\ \{p(6,3), 6 > 4\},\ \{36 \geq Y\},\ \emptyset\ \rangle \\ &\langle\ \{6 > 4\},\ \{36 \geq Y\},\ \{p(6,3)\}\ \rangle \\ &\langle\ \emptyset,\ \{36 \geq Y\},\ \{p(6,3)\}\ \rangle \end{aligned}$$

Now we define the derivation step for queries containing negative literals. First, let us define $c\text{-}neg(\Pi, Q, C, R)$, which is a modification of $c\text{-}neg(\Pi, Q, C)$ from section 5.5.1.2.

Definition 5.5.3. $[c\text{-}neg(\Pi, \neg Q, C, R)]$ *Let* Π *be a program and* Q *be a query both not containing* $\neg$. *Let* C *be a constraint store and* R *be a regular store. Let* $D = \{D_1, \ldots, D_k\}$ *be the set of all possible successful and conditionally successful derivation sequences of* $\langle Q, C, R\rangle$ *using derivation step as defined in*

5.5.2. We define c-neg$(\Pi, \neg Q, C, R) = \neg(C_1 \vee \cdots \vee C_k)$*, where* C_i *is the conjunction of answer constraints of derivation sequence* D_i*. If all derivation sequences are finitely failed then if* R *is complete with respect to r-literals in* Π *then c-neg*$(\Pi, \neg Q, C, R) = \mathrm{false}$*, otherwise c-neg*$(\Pi, \neg Q, C, R)$ *is an empty formula.*

Example 5.5.7. *Let predicates* d, e *be defined predicates and* p *be a regular predicate. Let* Π *be a program as follows: (same as in example 5.5.5)*

$$\begin{aligned} d(X_1, Y_1) &\leftarrow X_1^3 \geq Y_1,\ e(X_1, Y_1). \\ e(X_2, Y_2) &\leftarrow X_2 = 2, X_2 + Y_2 \leq 20. \\ e(X_3, Y_3) &\leftarrow p(X_3), Y_3 * 4 \leq 20. \end{aligned}$$

Let $Q = d(2, Z)$ *and* $C = \emptyset$ *and* $R = \emptyset$*. There are two successful derivation sequences* D_1 *and* D_2 *for* $\langle Q, C, R\rangle$ *and no conditionally successful sequences. The first and last elements of* D_1 *and* D_2 *are as follows:*

$D_1 = \langle\ d(2, Z), \emptyset, \emptyset\ \rangle\ \ldots\ \langle\ \emptyset, \{Z \leq 8\}, \emptyset\ \rangle$

$D_2 = \langle\ d(2, Z), \emptyset, \emptyset\ \rangle\ \ldots\ \langle\ \emptyset, \{Z \leq 5\}, \{p(2)\}\ \rangle$

We get c-neg$(\Pi, \neg d(2, Z), \emptyset, \emptyset) = \neg(\ (Z \leq 8) \vee (Z \leq 5)\ )$.

Now we define the derivation step for queries which contain negative literals. We add an extra step to the previous definition.

Definition 5.5.4. [Derivation Step] *Let* Π *be a program not containing* $\neg$ *and* $Q = l_1, \ldots, l_n$ *be a query. Let* C *be a constraint store and* R *be a regular store. We say that there is a derivation step from a tuple* $\langle Q, C, R\rangle$ *to another tuple* $\langle Q_1, C_1, R_1\rangle$ *if:*

(1). $Q_1 = l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n$*, where* l_i *is a constraint formula;* $R_1 = R$ *and* C_1 *is a (possibly simplified) set of constraints equivalent to* $C \cup \{l_i\}$.

Furthermore, C_1 *is solvable;*

(2). *or* $Q_1 = l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n$, *where* l_i *is a r-literal;* $C_1 = C$ *and* $R_1 = R \cup \{l_i\}$. *Furthermore,* R_1 *is consistent;*

(3). *or* $Q_1 = l_1, \ldots, l_{i-1}, x_1 = t_1, \ldots, x_k = t_k, b_1, \ldots, b_m, l_{i+1}, \ldots, l_n$, *where* $l_i = d(x_1, \ldots, x_k)$ *is a positive d-literal and there is a rule* $r \in \Pi$ *such that head of* r, $d(t_1, \ldots, t_k)$, *can be unified with* l_i *and* $\mathrm{body}(r)$ *can be unified with set* $\{b_1, \ldots, b_m\}$; $C_1 = C$; *and* $R_1 = R$.

(4). *or* $Q_1 = l_1, \ldots, l_{i-1}, c\text{-}neg(\Pi, l_i, C, R), l_{i+1}, \ldots, l_n$, *where* $l_i = \neg d(x_1, \ldots, x_k)$ *is a negative d-literal and* $c\text{-}neg(\Pi, \neg d(x_1, \ldots, x_k), C, R)$ *is a formula from definition 5.5.3;* $C_1 = C$; *and* $R_1 = R$.

Example 5.5.8. *Let predicates* d, e *be defined predicates and* p *be a regular predicate. Let* Π *be a program as follows: (same as in example 5.5.5)*

$$\begin{aligned} d(X_1, Y_1) &\leftarrow X_1^3 \geq Y_1,\ e(X_1, Y_1). \\ e(X_2, Y_2) &\leftarrow X_2 = 2, X_2 + Y_2 \leq 20. \\ e(X_3, Y_3) &\leftarrow p(X_3), Y_3 * 4 \leq 20. \end{aligned}$$

Let $Q = \neg d(2, Z_1), d(4, Z_2)$, $C = \emptyset$ *and* $R = \emptyset$. *A derivation sequence is as follows:*

$$\begin{aligned} \langle Q,\ C,\ R \rangle &= \langle\ \{\neg d(2, Z_1), d(4, Z_2)\},\ \emptyset,\ \emptyset\ \rangle \\ \langle Q_1,\ C_1,\ R_1 \rangle &= \langle\ \{c\text{-}neg(\Pi, \neg d(2, Z_1), \emptyset, \emptyset), d(4, Z_2)\},\ \emptyset,\ \emptyset\ \rangle \end{aligned}$$

We use example 5.5.7 to get

$c\text{-}neg(\Pi, \neg d(2, Z_1), \emptyset, \emptyset) = \neg(\ (Z_1 \leq 8) \vee (Z_1 \leq 5)\ )$. *Hence,*

$\langle Q_1, C_1, R_1 \rangle = \langle\ \{\neg(\ (Z_1 \leq 8) \vee (Z_1 \leq 5)\ ), d(4, Z_2)\},\ \emptyset,\ \emptyset\ \rangle$

$\langle Q_2, C_2, R_2 \rangle = \langle\ \{d(4, Z_2)\},\ \{\neg(Z_1 \leq 8) \wedge \neg(Z_1 \leq 5)\},\ \emptyset\ \rangle$

$\langle Q_3, C_3, R_3 \rangle = \langle\ \{X_1 = 4, Y_1 = Z_2, X_1^3 \geq Y_1, e(X_1, Y_1)\}, \{\neg(Z_1 \leq 8) \wedge \neg(Z_1 \leq 5)\},\ \emptyset\ \rangle$

$\langle Q_4, C_4, R_4 \rangle = \langle\ \{Y_1 = Z_2, X_1^3 \geq Y_1, e(X_1, Y_1)\}, \{\neg(Z_1 \leq 8) \wedge \neg(Z_1 \leq 5), X_1 = 4\},\ \emptyset\ \rangle$

$\langle Q_5, C_5, R_5 \rangle = \langle\ \{X_1^3 \geq Y_1, e(X_1, Y_1)\}, \{\neg(Z_1 \leq 8) \wedge \neg(Z_1 \leq 5), X_1 = 4, Y_1 = Z_2\},\ \emptyset\ \rangle$

$\langle Q_6, C_6, R_6 \rangle = \langle\ \{e(X_1, Y_1)\}, \{\neg(Z_1 \leq 8) \wedge \neg(Z_1 \leq 5), X_1 = 4, Y_1 = Z_2, X_1^3 \geq Y_1\},\ \emptyset\ \rangle$

$\langle Q_7, C_7, R_7 \rangle = \langle\ \{X_3 = X_1, Y_3 = Y_1, p(X_3), Y_3 * 4 \leq 20\}, C_6,\ \emptyset\ \rangle$

$\langle Q_8, C_8, R_8 \rangle = \langle\ \{Y_3 = Y_1, p(X_3), Y_3 * 4 \leq 20\}, C_6 \cup \{X_3 = X_1\},\ \emptyset\ \rangle$

$\langle Q_9, C_9, R_9 \rangle = \langle\ \{p(X_3), Y_3 * 4 \leq 20\}, C_6 \cup \{X_3 = X_1, Y_3 = Y_1\},\ \emptyset\ \rangle$

$\langle Q_{10}, C_{10}, R_{10} \rangle = \langle\ \{Y_3 * 4 \leq 20\}, C_6 \cup \{X_3 = X_1, Y_3 = Y_1\},\ \{p(X_3)\}\ \rangle$

$\langle Q_{11}, C_{11}, R_{11} \rangle = \langle\ \emptyset, C_6 \cup \{X_3 = X_1, Y_3 = Y_1, Y_3 * 4 \leq 20\},\ \{p(X_3)\}\ \rangle$

Simplifying the constraint store, we get constraints on Z_1 *and* Z_2 *as:*

$\{Z_1 > 5, Z_2 \leq 5\}$.

The *clp-solver* takes 3 inputs:

- ▷ a program Π not containing $\neg$,
- ▷ an r-ground query Q which may possibly contain negative d-literals and
- ▷ a set of ground extended r-literals S.

Similar to clp, *clp-solver* returns $true$ ($maybe$), when it finds a successful (conditionally successful) derivation sequence. If all derivation sequences are finitely failed then it returns $false$. The main loop of *clp-solver* differs from the main loop of clp in the following ways:

- ▷ Unlike clp, *clp-solver* generates a derivation sequence starting with $\langle Q, \emptyset, S \rangle$,

- ▷ when a successful or conditionally successful derivation sequence D is found then *clp-solver* returns C and R, where C and R are respectively the set of c-atoms from the constraint store and r-atoms from the regular store of the last element in D

Similar to clp, there are cases when there is a successful derivation sequence and *clp-solver* is unable to find it.

Recall that the definition of an r-literal l belongs to program $\Pi_R \cup \Pi_M$, hence when *clp-solver* finds a regular literal l in the query, it cannot derive the truth value of l. Therefore, if l does not conflict the set of r-literals in regular store then it just adds l to the regular store. Intuitively, it assumes that l is successfully derived and continues building the sequence.

Note that in computing constructive negation of a d-literal$\neg l$, $c\text{-}neg(\Pi, \neg l, C, R)$, it suffices to use the definition of $DerivativeStep^{\not\perp}$ since the input program Π does not contain $\neg$ and l is positive.

It is also important to note a subtle difference in properties of $c\text{-}neg(\Pi, \neg l_i, C, R)$ to the case when Π does not contain r-literals and to the case when Π contains r-literals.

When Π does not contain r-literals, then *every solution of* $c\text{-}neg(\Pi, \neg l, C, R)$ *is a solution of* $\neg l$. That is,

$$\Pi \cup R \cup M_c \models \neg l \mid_\theta^{vars(l)}$$

where M_c is intended interpretation of c-atoms of Π and θ is any solution of $c\text{-}neg(\Pi, \neg l, C, R)$.

When Π contains r-literals then if R is complete, (i.e. for every r-atom a in Π either $a \in R$ or *not* $a \in R$), then we get:

$$\Pi \cup R \cup M_c \models \neg l \mid_\theta^{vars(l)}$$

where M_c is intended interpretation of c-atoms of Π and θ is any solution of $c\text{-}neg(\Pi, \neg l, C, R)$. For the case when R is not complete, the relationship of the formula $c\text{-}neg(\Pi, Q, C, R)$ with respect to $\neg l$ is complex. The definition of $c\text{-}neg(\Pi, Q, C, R)$ can be changed to get soundness and completeness property when R is not complete. This will be regarded in future work.
The intuitive reason for this difference is that while computing $c\text{-}neg(\Pi, \neg l, C, R)$, the derivation sequences of $\langle l, C, R\rangle$ change both constraint store and regular store, but the final result returned by *c-neg* captures only the conclusions in constraint store and ignores the regular store. This makes the definition of *c-neg* simpler while still retaining the correctness for the case when R is complete. Therefore, the query can contain negative literals only if R is complete. The following example gives more details.

Example 5.5.9. *Let predicates* d, e *be defined predicates and* p *be a regular predicate with domain of regular variable* $X = \{1, 2, 3, 4, 5\}$. *Let* Π *be a program as follows:*

$$\begin{aligned} d(X, Y) &\leftarrow not\ p(X), Y \geq 2. \\ d(X, Y) &\leftarrow p(X), Y \geq 4, X + Y \leq 10. \\ e(X, Y) &\leftarrow p(X), Y < 6. \\ e(X, Y) &\leftarrow p(X + 2), Y > 4. \end{aligned}$$

Let $Q = \neg e(3, Z)$. *Let* $C = \emptyset$ *and* $R = \emptyset$. *The following is the derivation sequence of* $\langle Q, C, R\rangle$*:*

$$\begin{aligned} \langle Q,\ C,\ R\rangle &= \langle\ \{\neg e(3, Z)\},\ \emptyset,\ \emptyset\ \rangle \\ \langle Q_1,\ C_1,\ R_1\rangle &= \langle\ \{c\text{-}neg(\Pi, \neg e(3, Z), \emptyset, \emptyset)\},\ \emptyset,\ \emptyset\ \rangle \end{aligned}$$

We compute $c\text{-}neg(\Pi, \neg e(3, Z), \emptyset, \emptyset)$ *as follows: There are two successful derivation sequences* D_1 *and* D_2 *for* $\langle e(3, Z), \emptyset, \emptyset\rangle$*: The first and last elements*

of D_1 *and* D_2 *are as follows:*

$D_1 = \langle\ e(3,Z),\emptyset,\emptyset\ \rangle\ \ldots\ \langle\ \emptyset,\{Z<6\},\{p(3)\}\ \rangle$

$D_2 = \langle\ e(3,Z),\emptyset,\emptyset\ \rangle\ \ldots\ \langle\ \emptyset,\{Z>4\},\{p(5)\}\ \rangle$

We get c-neg$(\Pi,\neg e(3,Z),\emptyset,\emptyset) = C_n = \neg(Z<6 \vee Z>4)$. *The formula* $C_n \equiv Z \geq 6 \wedge Z \leq 4$ *which does not have any solution. So* $\neg e(3,Z)$ *does not have a solution when* R *is not complete. Suppose* $R_1 = \{p(1), p(2), not\ p(3), p(4), not\ p(5)\}$. *We get c-neg*$(\Pi,\neg e(3,Z),C,R_1) = \emptyset$ *as* R *is complete and all derivations of* $\langle e(3,Z),C,R_1\rangle$ *are finitely failed. Therefore,* $\Pi \cup R_1 \models \neg e(3,Z), \forall real(Z)$.

Let $Q = \neg d(2,Z)$. *Let* $C = \emptyset$ *and* $R = \emptyset$. *The following is a derivation sequence of* $\langle Q,C,R\rangle$*:*

$$\langle Q,\ C,\ R\rangle = \langle\ \{\neg d(2,Z)\},\ \emptyset,\ \emptyset\ \rangle$$
$$\langle Q_1,\ C_1,\ R_1\rangle = \langle\ \{c\text{-}neg(\Pi,\neg d(2,Z),\emptyset,\emptyset)\},\ \emptyset,\ \emptyset\ \rangle$$

We compute c-neg$(\Pi,\neg d(2,Z),\emptyset,\emptyset)$ *as follows: There are two successful derivation sequences* D_1 *and* D_2 *for* $\langle d(2,Z),\emptyset,\emptyset\rangle$*: The first and last elements of* D_1 *and* D_2 *are as follows:*

$D_1 = \langle\ d(2,Z),\emptyset,\emptyset\ \rangle\ \ldots\ \langle\ \emptyset,\{Z\geq 2\},\{not\ p(2)\}\ \rangle$

$D_2 = \langle\ d(2,Z),\emptyset,\emptyset\ \rangle\ \ldots\ \langle\ \emptyset,\{Z\geq 4, Z\leq 8\},\{p(2)\}\ \rangle$

We get c-neg$(\Pi,\neg d(2,Z),\emptyset,\emptyset) = C_n = \neg(\ (Z\geq 2) \vee ((Z\geq 4)\wedge(Z\leq 8))\ )$. *The formula* C_n *is equivalent to* $(Z<2)\wedge((Z<4)\vee(Z>8))$. *Further we get* $C_n \equiv (Z<2\wedge Z<4)\vee(Z<2\wedge Z>8)$. *Note that c-neg*$(\Pi,\neg d(2,Z_1)$ *comprises of only the constraint formula though the regular store of* D_1 *and* D_2 *are non-empty. Consider* $Z = 10$, Z *does not satisfy the constraint formula* C_n, *but* $\Pi \cup \{p(2)\} \models \neg d(2,10)$.

Proposition 5.5.2. *Let* Π *be a program,* Q *be a query and* R *be a set of*

ground extended r-literals such that

- Q *is r-ground,*
- Π *satisfies syntax restriction (3) from section 4.1,*
- *if not* l *occurs in* Π *or* Q *then* l *is regular atom, and*
- $\neg$ *does not occur in* Π *and* Q.

Let V_Q *be the variables in* Q *and* M_c *be the set of intended interpretation of c-atoms in* Π. *Then,*

- *If clp-solver returns* $\mathtt{true}$ *or* $\mathtt{maybe}$ *then* $gr(\Pi) \cup R \cup M_c \models Q|^{V_Q}_{\theta(A)}$, *where* $\theta(A)$ *is any solution to answer constraints* A.
- *If clp-solver returns* $\mathtt{false}$ *then for any substitution* θ *of* V_Q, $gr(\Pi) \cup R \cup M_c \not\models Q|^{V_Q}_{\theta}$.

Proposition 5.5.3. *Let* Π *be a program,* Q *be a query and* R *be a set of ground extended r-literals such that*

- Q *is r-ground,*
- Π *satisfies syntax restriction (3) from section 4.1,*
- *if not* l *occurs in* Π *or* Q *then* l *is regular atom,*
- $\neg$ *does not occur in* Π,
- R *is complete with respect to r-literals in* Π.

Let V_Q *be the variables in* Q *and* M_c *be the set of intended interpretation of c-atoms in* Π. *Then,*

- ▷ *If clp-solver returns* true *or* maybe *then* $gr(\Pi) \cup R \cup M_c \models Q|^{V_Q}_{\theta(A)}$, *where* $\theta(A)$ *is any solution to answer constraints* A.
- ▷ *If clp-solver returns* false *then for any substitution* θ *of* V_Q, $gr(\Pi) \cup R \cup M_c \not\models Q|^{V_Q}_{\theta}$.

5.5.2 The *c_solve* Function

The c_solve function has three input parameters, a program Π_D, a set of ground extended r-literals S, and a query Q. It returns true, false or maybe. When it returns true or maybe, it also returns a set of constraints A. The algorithm is shown in Figure 5.3. First we introduce some terminology.

A *solution* of a set of constraints A, denoted by $a\_solution(A)$, is a binding of variables in A which satisfies constraints in A.

Let Π_D be the defined part of a program Π and S be a set of ground extended r-literals. Let $Q = query(\Pi, S)$, D be the set of d-literals in Q and V_Q be the set of variables in Q. Let A and R be the set of answer constraints and set of ground extended r-literals returned by *clp-solver(*Π,Q,A,R*)*. The following are the properties of the return value of c_solve:

- ▷ If c_solve(Π_D, S, Q, A) returns true then every answer set of Π agreeing with S contains $D|^{V_Q}_{\theta}$, where $\theta = a\_solution(A)$.
- ▷ If c_solve(Π_D, S, Q, A) returns maybe then there maybe answer sets of Π agreeing with $S \cup R$.
- ▷ If c_solve(Π_D, S, Q, A) returns false then there is no answer set of Π agreeing with S.

Recall from previous section that, when *clp-solver* returns true, it means that it found a successful or conditionally successful derivation D. Since during the

function c_solve (Π : Program, S : Set of r-lits, Q : Query, **var** A : Set of constraints)

```
        % var R : set of r-literals
        % var found : boolean
[a]     found := false
[c]     R := S
[d]     found := clp-solver(Π, Q, A, R)
[i]     if covers(S, r-atoms(Π)) then return found
[i]     else if Q is positive then
[i]             if R ⊆ S then return found
[i]             else if found = false then return false
[j]     end if
[j]     return maybe
```

Figure 5.3: Function c_solve

derivation step, the set of literals in R were assumed to be true, D is a successful derivation with the condition that R can be satisfied with respect to S. The derivation D is *r-successful* if the set of regular literals $R \cup S$ is consistent.

Let V_Q be the set of variables in Q. Let $V_M = vars(mcv\_set(\Pi))$. Recall that by construction of Q, $V_M \subseteq V_Q$. The answer constraints A returned by clp-solver consists of constraints on V_Q.

Let θ be a solution of A. We construct a candidate_mixed set X from $mcv\_set(\Pi)$ by substituting variables in $mcv\_set(\Pi)$ by values from binding θ written as $X = mcv\_set(\Pi)|_\theta^{V_M}$.

Proposition 5.5.4. *Let* Π *be a program* Π *and* S *be a set of ground extended r-literals. Let* $Q = query(\Pi, S)$, D *be the set of d-literals in* Q *and* $\bar{V}_Q$ *be the set of variables in* Q.

▷ *If c_solve*(Π_D, S, Q, A) *returns* true *then every answer set of* Π *agreeing*

with S *contains* $D|_\theta^{\bar{V}_Q}$, *where* $\theta = a\_solution(A)$.

- ▷ *If* $c\_solve(\Pi_D, S, Q, A)$ *returns* false *then there is no answer set of* Π *agreeing with* S.

Proposition 5.5.5. *Let* Π *be a program* Π *and* S *be a set of ground extended r-literals. Let* $Q = query(\Pi, S)$ *be a query,* A *be a set of answer constraints, and* θ *be any solution of* A. *Let* V *be the set of variables in* Q, $D = d\text{-}lits(Q)|_\theta^{\bar{V}}$ *be a set of d-literals and* $X = mcv\_set(\Pi)|_\theta^{\bar{V}}$ *be a candidate_mixed set. If* $c\_solve(\Pi_D, S, Q, A)$ *returns* true *then if* $S \cup D \cup X \cup M_c$ *is an* asp*-answer set of* $\Pi_R \cup \Pi_M \cup D \cup X \cup M_c$ *then there exists an answer set* M *of* Π *such that* $S \cup X$ *is the simplified part of* M.

Proposition 5.5.6. *Let a program* Π *and a set of extended r-literals* B *be inputs to* $\mathcal{AC}(C)$*_solver. Let*

(s_1) $S = expand(\Pi_R \cup \Pi_M, B)$ *is consistent and covers all r-atoms of* Π

(s_2) $Q = query(\Pi, S)$, $\bar{V} = vars(Q)$ *and* $c\_solve(\Pi_D, S, Q, A)$ *returns* true *at step (d) of* $\mathcal{AC}(C)$*_solver*

(s_3) $\theta = a\_solution(A)$, $D = d\text{-}lits(Q)|_\theta^{\bar{V}}$, $X = mcv\_set(\Pi)|_\theta^{\bar{V}}$ *and* M_c *is the intended interpretation of c-atoms(*Π*).*

Then $pos(S) \cup D \cup X \cup M_c$ *is an* asp*-answer set of* $\Pi_R \cup \Pi_M \cup D \cup X \cup M_c$ *agreeing with* B.

5.6 The pick Function

Function *pick* takes as input the set of r-atoms, $Y = r\text{-}atoms(\Pi) \setminus Atoms(S)$. It returns an extended r-literal formed from an atom of Y. The choice of the picked extended literal determines to a large degree the efficiency of our algorithm. The

selection is based on the heuristic function implemented in pick. Both Smodels [75] and Surya [72] have similar heuristic functions. The solver *ACengine* uses the same heuristic function as Surya.

CHAPTER 6

LANGUAGE $\mathcal{V}(\mathcal{C})$

Our work so far was in integrating four reasoning mechanisms: *answer set reasoning and abductive techniques* from ASP paradigm and *resolution and constraint solving techniques* from CLP paradigm. Towards this, we designed a collection of languages $\mathcal{AC}(\mathcal{C})$ and developed an algorithm for computing answer sets for a class of $\mathcal{AC}(\mathcal{C})$ programs. In the development of this algorithm, resolution techniques proved to be the most difficult to integrate with the others. Hence, before we integrated the four techniques, we investigated whether we can effectively integrate the other three techniques: *answer set reasoning, abductive techniques and constraint solving*. We were interested in developing languages which would allow us to integrate the three reasoning techniques to build a solver to compute answer sets. We wanted to design such languages and investigate their usefulness for knowledge representation.

Consider the subset of languages from $\mathcal{AC}(\mathcal{C})$ where signature of the languages do not contain any defined predicates. A program in such language does not contain any defined rules and the mixed part does not contain any defined atoms. This is an interesting collection of languages called CASP introduced in [12]. More information related to CASP can be found in chapter 9. We study an extension of CASP in this chapter.

This chapter introduces a collection of knowledge representation languages, $\mathcal{V}(\mathcal{C})$, parametrised over a class $\mathcal{C}$ of constraints. $\mathcal{V}(\mathcal{C})$ is an extension of both CR-Prolog [3] and CASP [12] allowing the separation of a program into two parts: a regular program of CR-Prolog and a collection of denials[1] whose bodies contain

[1] By a denial we mean a logic programming rule with an empty head.

constraints from $\mathcal{C}$ with variables ranging over large domains. We study an instance $\mathcal{AC}_0$ from this family where $\mathcal{C}$ is a collection of constraints of the form $X - Y > k$, where X and Y are variables and k is any real number.
We design and implement an algorithm computing the answer sets of programs of $\mathcal{AC}_0$ which does not ground constraint variables and tightly couples the classical ASP algorithm with an algorithm checking the consistency of difference constraints. This makes it possible to declaratively solve problems which could not be solved by pure ASP or by pure constraint solvers. The chapter is organized as follows: In section 6.1 we define the syntax and semantics of $\mathcal{V}(\mathcal{C})$ and $\mathcal{AC}_0$. Section 6.2 contains a description of the algorithm for computing answer sets of programs in $\mathcal{AC}_0$.

6.1 Syntax and Semantics of $\mathcal{V}(\mathcal{C})$

6.1.1 Syntax

The language $\mathcal{V}(\mathcal{C})$ contains a sorted signature Σ, with sorts partitioned into two classes: *regular*, s_r, and *constraint*, s_c. Intuitively, the former are comparatively small but the latter are too large for the ASP grounders. Functions defined on regular (constraint) classes are called r-functions (c-functions). Terms are built as in first-order languages. Predicate symbols are divided into three disjoint sets called *regular*, *constrained* and *mixed* and denoted by P_r, P_c and P_m respectively. Constraint predicate symbols are determined by $\mathcal{C}$. Parameters of regular and constraint predicates are of sorts s_r and s_c respectively. Mixed predicates have parameters from both classes with at least one from each. Atoms are defined as usual. A literal is an atom a or its negation $\neg a$. An extended literal is a literal l or $not\ l$, where not stands for *default negation*. Atoms formed from regular, constraint, and mixed predicates are called r-atoms, c-atoms and m-atoms

respectively. Similarly for literals. We assume that predicates of P_c have a predefined interpretation, represented by the set M_c of all true ground c-atoms. For instance, if $'>' \in P_c$, and ranges over integers, M_c consists of $\{...0 > -1, 1 > 0, 2 > 0, ..., 2 > 1, 3 > 1, ...\}$. The c-literals allowed in $\mathcal{V}(\mathcal{C})$ depend on the class $\mathcal{C}$. The $\mathcal{V}(\mathcal{C})$ rules over Σ are defined as follows.

Definition 6.1.1. [**rules**]

1. *A regular rule (r-rule)* ρ *is a statement of the form:*

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

   where $k >= 0$*;* h_i*'s and* l_i*'s are r-literals.*

2. *A constraint rule (c-rule) is a statement of the form:*

$$\leftarrow l_1, \ \cdots, l_m, \ not \ \ l_{m+1}, \cdots, not \ l_n$$

   where at least one l_i *is non-regular.*

3. *A consistency restoring rule (cr-rule) is a statement of the form:*

$$r: \ h_1 \ or \ \cdots or \ \ h_k \stackrel{+}{\leftarrow} l_1, \ \cdots, l_m, \ not \ \ l_{m+1}, \cdots, not \ l_n$$

   where $k > 0$*,* r *is a term which uniquely denotes the name of the rule and* h_i*'s and* l_i*'s are r-literals.*

$head(r) = h_0 \ or \ \cdots or \ \ h_k$; $body(r) = \{l_1, \ \cdots, l_m, \ not \ \ l_{m+1}, \cdots, not \ l_n\}$; and $pos(r)$, $neg(r)$ denote, respectively, $\{l_1, \cdots, l_m\}$ and $\{l_{m+1}, \cdots, l_n\}$.

A regular rule and constraint rule have the same intuitive reading as standard rules of ASP. The intuitive reading of a cr-rule is: *if one believes in* $l_1, \ldots l_m$ *and*

have no reason to believe $l_{m+1}, \ldots, l_n$*, then one may possibly believe one of* $h_1, \ldots, h_k$. The implicit assumption captured by the semantics of the language is that this possibility is used as little as possible, and only to restore consistency of the agent's beliefs.

Definition 6.1.2. [**program**] *A* $\mathcal{V}(\mathcal{C})$ *program is a pair (*Σ*,*Π*), where* Σ *is a sorted signature and* Π *is a set of* $\mathcal{V}(\mathcal{C})$ *rules over* Σ.

Example 6.1.1. *Let* $\Sigma = \langle\ C_r = \{a, b\},\ V_r = \{X, Y\},\ P_r = \{p, q\},\ C_c = \{-1000..1000\},\ V_c = \{T_1, T_2\},\ F_c = \{-\},\ P_c = \{>\},\ P_m = \{at\}\ \rangle$*, variables in* V_r *and* V_c *take values from* C_r *and* C_c *respectively. Let* Π *be:*

$q(a).\quad q(b).$

$p(X, Y)$ *or* $s(X, Y) \leftarrow q(X),\ q(Y).$

$\leftarrow p(X, Y),\ at(X, T_1),\ at(Y, T_2),\ T_1 - T_2 > 10.$

$\leftarrow s(X, Y), not\ r(X, Y),\ at(X, T_1),\ at(Y, T_2),\ T_1 - T_2 > 50.$

$c_r:\quad r(X, Y) \stackrel{+}{\leftarrow}$

The first three rules are r-rules and next two are c-rules and last rule is a cr-rule.

Example 6.1.2. *To represent conditions: "John goes to work either by car which takes 30 to 40 minutes, or by bus which takes at least 60 minutes", we start by defining the signature*

$\Sigma = \{C_r = \{start, end\},\ P_r = \{by\_car, by\_bus\},\ C_c = \{D_c = [0..1439], R_c = [-1439..1439]\},\ V_c = \{T_s, T_e\},\ F_c = \{-\}, P_c = \{>\},\ P_m = \{at\}\}$. *The sets* C_r *and* P_r *contain regular constants and predicates; elements of* C_c, V_c, F_c, *and* P_c *are constrained constants, variables, functions and predicate symbols.* P_m *is the set of mixed predicates. Values in* D_c *represent time in minutes. Consider one whole day from 12:00am to 11:59pm mapped to 0 to 1439 minutes.*

Regular atom "$\mathrm{by\_car}$*" says that "John travels by car"; mixed atom* $\mathrm{at}(\mathrm{start}, \mathrm{T})$ *says that "John starts from home at time* T*". Similarly for "*$\mathrm{by\_bus}$*" and "*$\mathrm{at}(\mathrm{end}, \mathrm{T})$*". Function "*$-$*" has the domain* $\mathrm{D_c}$ *and range* $\mathrm{R_c}$*;* $\mathrm{T_s}, \mathrm{T_e}$ *are variables for* $\mathrm{D_c}$*. The rules below represent the information from the story.*

% 'John travels either by car or bus' is represented by an r-rule

$\mathrm{r_a}:\ \mathrm{by\_car}$ *or* $\mathrm{by\_bus}$.

% Travelling by car takes between 30 to 40 minutes. This information is encoded by two c-rules

$\mathrm{r_b}:\ \leftarrow \mathrm{by\_car},\ \mathrm{at}(\mathrm{start}, \mathrm{T_s}),\ \mathrm{at}(\mathrm{end}, \mathrm{T_e}), \mathrm{T_e} - \mathrm{T_s} > 40.$

$\mathrm{r_c}:\ \leftarrow \mathrm{by\_car},\ \mathrm{at}(\mathrm{start}, \mathrm{T_s}),\ \mathrm{at}(\mathrm{end}, \mathrm{T_e}),\ \mathrm{T_s} - \mathrm{T_e} > -30.$

% Travelling by bus takes at least 60 minutes

$\mathrm{r_d}:\ \leftarrow \mathrm{by\_bus},\ \mathrm{at}(\mathrm{start}, \mathrm{T_s}),\ \mathrm{at}(\mathrm{end}, \mathrm{T_e}),\ \mathrm{T_s} - \mathrm{T_e} > -60.$

$\mathrm{T_e} - \mathrm{T_s} > 40$*,* $\mathrm{T_s} - \mathrm{T_e} > -30$*, and* $\mathrm{T_s} - \mathrm{T_e} > -60$ *are c-atoms.*

Example 6.1.3. *Let us expand the story from example 6.1.2 by new information: 'John prefers to come to work before 9am'. We add new constant* $'\mathrm{time0}'$ *to* $\mathrm{C_r}$ *of* Σ *which denotes the start time of the day, regular atom 'late' which is true when John is late and constrained variable* $\mathrm{T_t}$ *for* $\mathrm{D_c}$*. Time 9am in our representation is mapped to* 540^{th} *minute. We expand example 6.1.2 by the following rules:*

% Unless John is late, he comes to work before 9am

$\mathrm{r_e}:\ \leftarrow \mathrm{at}(\mathrm{time0}, \mathrm{T_t}),\ \mathrm{at}(\mathrm{end}, \mathrm{T_e}),\ \neg\mathrm{late},\ \mathrm{T_e} - \mathrm{T_t} > 540$

% Normally, John is not late

$\mathrm{r_f}:\ \neg\mathrm{late} \leftarrow$ *not* late

% On some rare occasions he might be late, which is encoded by a cr-rule

$\mathrm{r_g}:\ \mathrm{late} \xleftarrow{+}$

6.1.2 Semantics

We denote the sets of r-rules, cr-rules and c-rules in Π by Π^r, Π^{cr} and Π^c respectively. A rule r of (Π, Σ) will be called *r-ground* if regular terms in r are ground. A program is called *r-ground* if all its rules are r-ground. A rule r^g is called a *ground instance* of a rule r if it is obtained from r by: (1). replacing variables by ground terms of respective sorts; (2). replacing the remaining terms by their values. For example, $3+4$ will be replaced by 7. The program $\mathrm{ground}(\Pi)$ with all ground instances of all rules in Π is called the *ground instance* of Π. Obviously ground(Π) is an r-ground program.
We first define semantics for programs without cr-rules. We believe that this definition is slightly simpler than the equivalent definition from [12].

Definition 6.1.3. [**answer set 1**] *Given a program (Σ, Π), where Π contains no cr-rules, let X be a set of ground m-atoms such that for every predicate $p \in P_m$ and every ground r-term t_r, there is exactly one c-term t_c such that $p(\bar{t}_r, \bar{t}_c) \in X$. A set S of ground atoms over Σ is an answer set of Π if S is an asp answer set of $\mathrm{ground}(\Pi) \cup X \cup M_c$.*

Example 6.1.4. *Consider example 6.1.2 and let $X =$ $\{at(start, 430), at(end, 465)\}$. The set $S =$ $\{by\_car, at(start, 430), at(end, 465)\} \cup M_c$ is an asp answer set of $\mathrm{ground}(\Pi) \cup X \cup M_c$ and therefore is an answer set of Π. According to S, John starts to travel by car at 7:10am and reaches work at 7:45am. Of course there are other answer sets where John travels by car and his start and end times differ but satisfy given constraints. There are also answer sets where John travels by bus.*

Now we give the semantics for programs with cr-rules. By $\alpha(r)$, we denote a regular rule obtained from a cr-rule r by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$; α is expanded in a

standard way to a set R of cr-rules. Recall that according to [6], a minimal (with respect to set theoretic inclusion) collection R of cr-rules of Π such that $\Pi^r \cup \Pi^c \cup \alpha(R)$ is consistent (i.e. has an answer set) is called an *abductive support* of Π. Definition 6.1.4 is a simplification of the original definition from [3], which includes special preference rules.

Definition 6.1.4. [**answer set 2**] *A set S is called an answer set of Π if it is an asp answer set of program $\Pi^r \cup \Pi^c \cup \alpha(R)$ for some abductive support R of Π.*

Example 6.1.5. *Consider example 6.1.3 and let* $X = \{at(start, 430), at(end, 465)\}$. *The set* $S = \{by\_car, \neg late,\ at(time0, 0),\ at(start, 430),\ at(end, 465)\} \cup M_c$ *is an answer set of* $ground(\Pi) \cup X \cup M_c$ *and therefore is an answer set of* Π. *According to S, John starts by car at 7:10am and reaches work at 7:45am and is not late. The cr-rule was not applied and* $\alpha(\emptyset) = \emptyset$.

Example 6.1.6. *Let us consider example 6.1.3, and add new information that 'John's car is not available and he could not start from home until 8:15am'.*

% 'John's cannot use his car' is encoded by an r-rule with empty head.

r_h : $\leftarrow by\_car.$

% 'John cannot start before 8:15am' is encoded as a c-rule:

r_i : $\leftarrow at(time0, T_t),\ at(start, T_s),\ T_t - T_s > -495.$

Let $X = \{at(time0, 0), at(start, 495), at(end, 560)\}$. $S = \{\ by\_bus,\ late,\ at(time0, 0),\ at(start, 495),\ at(end, 560)\ \} \cup M_c$ *is an answer set of the program, where John arrives late by bus at 9:20am. The cr-rule* r_g *was used and* $\alpha(\{r_g\}) = \{late \leftarrow\}$.

6.2 $\mathcal{AD}solver$

In this section we describe the algorithm which takes a program (Σ, Π) of $\mathcal{AC}_0$ as input and returns a simplified answer set $A \cup X$ (regular and mixed atoms) such that $M = A \cup X \cup M_c$ is an answer set of Π. The algorithm works for a class of $\mathcal{AC}_0$ programs satisfying the following syntax restrictions:

- ▷ *There are no disjunctions in the head of rules.* This restriction simplifies the $\mathcal{AD}solver$ algorithm.
- ▷ *Every c-rule of the program contains exactly one c-literal in the body.* This restriction simplifies the $\mathcal{AD}solver$ algorithm.

From now, we consider only programs that satisfy the above syntax restrictions. $\mathcal{AD}solver$ consists of a partial grounder $\mathcal{P}\mathrm{ground}_\mathrm{d}$ and an inference engine $\mathcal{AD}engine$. Similar to $\mathcal{AC}solver$, given a $\mathcal{AC}_0$ program Π, $\mathcal{AD}solver$ first calls $\mathcal{P}\mathrm{ground}_\mathrm{d}$ to ground r-terms of Π, to get an r-ground program, $\mathcal{P}_\mathrm{d}(\Pi)$. The $\mathcal{AD}engine$ combines constraint solving techniques with answer set reasoning and abduction techniques to compute answer sets of $\mathcal{P}_\mathrm{d}(\Pi)$. The next sections describe them in detail.

6.2.1 $\mathcal{P}\mathrm{ground}_\mathrm{d}$

Given a $\mathcal{AC}_0$ program Π, $\mathcal{P}\mathrm{ground}_\mathrm{d}$ grounds the r-variables in Π and outputs a r-ground program $\mathcal{P}_\mathrm{d}(\Pi)$. The steps of $\mathcal{P}\mathrm{ground}_\mathrm{d}$ follow the steps of $\mathcal{P}\mathrm{ground}$ (see section 4.2) but need not consider defined predicates as $\mathcal{AC}_0$ does not allow defined predicates.

The implementation of $\mathcal{P}\mathrm{ground}_\mathrm{d}$ uses intelligent grounder $lparse$ [98]. Example 6.2.1 presents a $\mathcal{AC}_0$ program and the corresponding r-ground program output by $\mathcal{P}\mathrm{ground}_\mathrm{d}$.

Example 6.2.1. *Let* a1 *and* a2 *be two actions. For representing the condition "a1 should occur 30 minutes before a2", we begin by defining a signature.* $\Sigma = \{C_r = \{\{a1, a2\}, \{1, 2\}\}, V_r = \{S\}, P_r = \{o\}, P_m = \{at\}, C_c = \{D_c = \{0..1440\}, R_c = \{-1440..1440\}\}, V_c = \{T_1, T_2\}, F_c = \{-\}, P_c = \{>\}\}$ *and* Π *be the following rules:*

```
step(1..2).
% one action occurs at each step
o(a1, S) :- step(S), not o(a2, S).
o(a2, S) :- step(S), not o(a1, S).
% an action can occur at most once
:- step(S1), step(S2), o(a1, S1), o(a1, S2), S1 != S2.
:- step(S1), step(S2), o(a2, S1), o(a2, S2), S1 != S2.
% define 'time' as a csort, and 'at' as a mixed predicate
#csort time(0..1440).
#mixed at(step,time).
% time should be increasingly assigned to steps
:- step(S1), step(S2), at(S1, T1), at(S2, T2), S1 < S2, T1 - T2 > 0.
% a1 should occur 30 minutes before a2
:- step(S1), step(S2), o(a1, S1), o(a2, S2),
   at(S1, T1), at(S2, T2), T1 - T2 > -30.
```

We get $\mathcal{P}_d(\Pi)$ *as follows:*

```
step(1).       step(2).
o(a1, 1) :-  not o(a2, 1).
o(a1, 2) :-  not o(a2, 2).
o(a2, 1) :-  not o(a1, 1).
o(a2, 2) :-  not o(a1, 2).
```

```
:-  o(a1, 1), o(a1, 2).
:-  o(a2, 1), o(a2, 2).
#csort time(0..1440).
:- at(1, V1), at(2, V2), V1 - V2 > 0.
:- o(a1, 1), o(a2, 2), at(1, V1), at(2, V2), V1 - V2 > -30.
:- o(a1, 2), o(a2, 1), at(2, V2), at(1, V1), V2 - V1 > -30.
```

6.2.2 $\mathcal{AD}engine$

The $\mathcal{AD}engine$ integrates a standard CR-Prolog solver and a difference constraint solver. CR-Prolog solver consists of a meta layer and computes answer sets by using an underlying ASP inference engine. For $\mathcal{AD}engine$, we use Surya [72] as the underlying inference engine.

Suppose there are no c-rules in a program Π, then Π is a CR-Prolog program. A typical CR-Prolog solvers available now, compute answer sets of Π as follows:

1. a meta-layer selects a minimal set R of cr-rules of Π called a candidate abductive support of Π;
2. an ASP inference engine is used to check program $\Pi^r \cup \alpha(R)$ for consistency and compute an answer set.
3. if an answer set is found at step (2) then R is an abductive support with respect to Π and the answer set computed is an answer set of Π and is returned[2]; otherwise the solver loops back to step(1) to find another minimal set R not tried so far.

A full description of the CR-Prolog solver is out of the scope of this work and can be found at [4]. To compute answer sets of $\mathcal{AC}_0$ programs, we modify the solver to

[2]This algorithm is a simplification of the actual algorithm [], which requires additional checking due to special preference rules allowed in the language.

accept c-rules; and then change step(2) of the algorithm. Given a $\mathcal{AC}_0$ program Π, we modify the underlying inference engine Surya to compute answer sets of $\Pi^r \cup \Pi^c \cup \alpha(R)$. Note that the program $\Pi^r \cup \Pi^c \cup \alpha(R)$ does not contain cr-rules but only r-rules and c-rules.

$\mathcal{AD}engine$ integrates a form of abductive reasoning using the meta-layer with answer set reasoning and constraint solving of the underlying inference engine. Surya has been modified to tightly couple with a difference constraint solver (for constraint solving).

In this section, we just present the modifications of the ASP engine Surya. First, we will describe the difference constraint solver $\mathcal{D}solver$ that was built to be integrated with ASP engine.

6.2.2.1 $\mathcal{D}solver$

Before, we describe the solver, we need to introduce some terminology. A difference constraint is of the form $X - Y \leq k$, where X, Y are variables and k is a real number. Given a set of difference constraints, D, *a consistent solution* of D is defined as the assignment of values to the variables in the constraints such that the constraints are satisfied.

Example 6.2.2. *Here is an example of a set of difference constraints and one of its solution. Let* $D = \{X - Y \leq 3,\ Y - Z \leq -4,\ X - Z \leq -3\}$ *be a set of difference constraints. A consistent solution for* D*,*
$S_D = \{X = -3,\ Y = -4,\ Z = 0\}$.

A difference constraint solver takes as input a set of difference constraints (viewed as conjunction of difference constraints) and outputs a consistent solution. The set of difference constraints input to the solver is sometimes called a *constraint store* to the solver.

$\mathcal{D}solver$ is a difference constraint solver. Unlike a naive difference constraint solver, $\mathcal{D}solver$ is an incremental solver and follows the algorithm from [85]. The inputs to $\mathcal{D}solver$ are a constraint store D, a solution S to the constraint store and a difference constraint c. The output of $\mathcal{D}solver$ is $true$ if $D \cup c$ has a consistent solution and $false$ otherwise. If the solver returns $true$ then it also returns a solution S' of $D \cup c$.

An important fact to note with an incremental solver is that given a constraint store D, a solution S to D and a difference constraint c, the solver uses S to compute a solution for $D \cup c$. This is different from a naive difference constraint solver which computes a solution for $D \cup c$ from scratch. The complexity of finding a solution using an incremental solver is $O(e + v\log v)$ where e and v are the number of constraints and variables respectively. The complexity of a naive difference constraint solver is $O(ev)$.

6.2.2.2 $Surya_{\mathrm{d}}$

In this section, we describe the integration of an ASP inference engine Surya and difference constraint solver $\mathcal{D}solver$. We call this new solver $Surya_{\mathrm{d}}$. The solver $Surya_{\mathrm{d}}$ tightly couples an ASP engine and a difference constraint solver. Algorithm of $Surya_{\mathrm{d}}$ follows a standard asp algorithm [75, 72] and is shown in Figure 6.1. The inputs of $Surya_{\mathrm{d}}$ are a r-ground program Π, a set of r-literals B and a set of difference constraints D. The output of $Surya_{\mathrm{d}}$ is $true$ if there exists an answer set of Π agreeing with B; otherwise it returns $false$. If $Surya_{\mathrm{d}}$ returns $true$ then it also returns a simplified answer set of Π agreeing with B. The set of constraints D is used as a constraint store to store the active constraints of Π with respect to B and will be described in next section. The integration of constraint solving in $Surya_{\mathrm{d}}$ is built inside $expand\_dc$ function. Next, we will describe

function $Surya_{\mathrm{d}}$ (Π: r-ground program, B: set of r-literals, D: set of constraints)

[a] $S := expand\_dc(\Pi, B, D)$

[b] **if** $inconsistent(S)$ or $inconsistent(D)$ **return** $false$

[c] **if** $covers(S, rAtoms(\Pi))$ **then**

[d] **return** true {answer-set is $pos(S) \cup m\_atoms(\Pi)|^{vars(D)}_{solution(D)}$}

[e] $pick(l, \bar{S})$

[f] **if** $Surya_{\mathrm{d}}(\Pi, S \cup \{l\}, D)$ **then**

[g] **return** true

[h] **else return** $Surya_{\mathrm{d}}(\Pi, S \cup \{not\ l\}, D)$

Figure 6.1: $Surya_{\mathrm{d}}$: computation of answer sets of Π

function $expand\_dc$.

6.2.2.3 Function $expand\_dc$

Let us introduce some terminology. Recall that a r-ground program Π input to $expand\_dc$ consists of r-rules and c-rules. Let the set of r-rules be denoted by Π_R and set of c-rules be denoted by Π_M. Further, recall that due to syntax restrictions, there is only one c-literal in every c-rule of Π.

Definition 6.2.1. [**active_dc(Π,B)**] *Let Π be a r-ground $\mathcal{AC}_0$ program and B be a set of extended literals. The set of constraints*
$D = \{\neg c \mid r \in \Pi,\ c = c\text{-}lit(r),\ r\text{-}lits(r) \subseteq B\}$ *is called the set of active constraints of Π with respect to B and denoted as* $active\_dc(\Pi, B)$.

Note that the constraints in $active\_dc(\Pi, B)$ are all difference constraints because $\mathcal{AC}_0$ programs contain only constraints of type $X - Y > k$ and its negation $X - Y \leq k$ is a difference constraint.

Example 6.2.3. *Consider the r-ground program $\Pi_g = \mathcal{P}_{\mathrm{d}}(\Pi)$ from example*

6.2.1. By definition, $active\_dc(\Pi_g, \emptyset) = \{V1 - V2 \leq 0\}$
$active\_dc(\Pi_g, \{o(a1,2), o(a2,1)\}) = \{V1 - V2 \leq 0, V2 - V1 \leq -30\}$.

Function *expand_dc* takes as input a r-ground program Π, a set of extended r-literals B and a set of difference constraints D. The output of the function is the set S of consequences of Π and B. Further, it expands D to consist of the set of constraints of Π active with respect S. For clarity, we denote the input and output values of the set of constraints D to *expand_dc* as D_i and D_o respectively. If S and D_o are consistent then it has the following properties:

- $B \subseteq S$
- $D_i \subseteq D_o$
- every answer set that agrees with B also agrees with S

Function $expand\_dc$ differs from function $expand$ (see section 5.3) in the following ways:

- Apart from using ASP reasoning techniques, *expand_dc* uses constraint solving techniques to compute consequences of Π, B and D.
- Apart from computing the set of consequences of Π with respect to B, the function also computes the set of active constraints of Π with respect to S, adds them to D and ensures D_o is consistent.

Function *expand_dc* uses two auxiliary functions $atleast\_dc$ and $atmost$. We describe these functions next.

Function $atleast\_dc$**:**
Let us introduce some terminology. Let Π be an r-ground program and B be a set of ground extended r-literals and D be a set of difference constraints. We define

$lc_0(\Pi, B, D)$ as a set of ground extended literals that is minimal (set theoretic) and satisfies the following conditions:

1. If $r \in \Pi_R$, and $body(r) \subseteq B$, then $head(r) \in lc_0(\Pi, B, D)$.
2. If $r \in \Pi_M$, $c = c\text{-}lits(r)$, and $r\text{-}lits(r) \subseteq B$, $D \cup \{\neg c\}$ is not consistent, then $head(r) \in lc_0(\Pi, B, D)$.
3. If an r-atom h is not in the head of any *active* rule in $\Pi_R \cup \Pi_M$ with respect to B then *not* $h \in lc_0(\Pi, B, D)$
4. If r is the only *active* rule of $\Pi_R \cup \Pi_M$ with respect to B such that $h = head(r)$ and $h \in B$ then $r\_lits(r) \subseteq lc_0(\Pi, B, D)$
5. If $r \in \Pi_R$, $h = head(r)$, *not* $h \in B$, and all literals in the body of r except l_i belong to B, then *not* $l_i \in lc_0(\Pi, B, D)$.
6. If $r \in \Pi_M$, $h = head(r)$, $c = c\text{-}lits(r)$, *not* $h \in B$, all literals in the body of r except l_i belong to B and $D \cup \{\neg c\}$ is not consistent then *not* $l_i \in lc_0(\Pi, B, D)$.

Example 6.2.4. *Consider r-ground program* $\Pi_g = \mathcal{P}_d(\Pi)$ *from example 6.2.1. We get,* $lc_0(\Pi_g, \emptyset, \emptyset) = \{step(1),\ step(2)\}$ *and* $lc_0(\Pi_g, \{o(a1,2)\}, \{V1 - V2 \leq 0\}) = \{step(1),\ step(2),\ o(a1,2),\ not\ o(a1,1),\ o(a2,1),\ not\ o(a2,2),\ false\}$. *The atom* false *is added by condition (2) in definition of* lc_0, *as* $D \cup \{V2 - V1 \leq -30\}$ *is not consistent (does not have a solution).*

Proposition 6.2.1. *If the set* $lc_0(\Pi, B, D) \cup B$ *is consistent then* $lc_0(\Pi, B, D)$ *is unique.*

Definition 6.2.2. *Given a program* Π *and a set of ground extended r-literals* B*, the lower closure of* Π *with respect to* B*, denoted by* $lc(\Pi, B, D)$*, is the set of ground extended r-literals defined as follows:*

$$lc(\Pi, B, D) = \begin{cases} lc_0(\Pi, B, D) \cup B & \text{if } lc_0(\Pi, B, D) \cup B \text{ is consistent and} \\ & \quad active\_dc(\Pi, lc_0(\Pi, B, D) \cup B) \cup D \text{ is consistent,} \\ r\text{-}lits(\Pi) & \text{otherwise.} \end{cases}$$

Example 6.2.5. *Let us use example 6.2.4. We get*
$lc(\Pi_g, \emptyset, \emptyset) = \{step(1),\ step(2)\}$*, since* $lc_0(\Pi_g, \emptyset, \emptyset) \cup \emptyset$ *is consistent and* $active\_dc(\Pi, lc_0(\Pi_g, \emptyset, \emptyset)) \cup \emptyset = \{V1 - V2 \leq 0\}$ *is consistent.*
$lc(\Pi_g, \{o(a1, 2)\}, \{V1 - V2 \leq 0\}) = r\text{-}lits(\Pi_g)$ *as*
$lc_0(\Pi_g, \{o(a1, 2)\}, \{V1 - V2 \leq 0\}) \cup B$ *is not consistent (we always assume atom* $'true'$ *belongs to* B *and is complementary to* $false$*). Note that* $active\_dc(\Pi, lc_0(\Pi, B, D)) \cup D$ *is also not consistent.*

The inputs of function $atleast\_dc$ are a r-ground program Π, a set of extended literals B and a set D of difference constraints. The function computes the lower closure $lc(\Pi, B, D)$ of Π with respect to B and D. Further, it updates D to contain the constraints of Π active with respect to $lc(\Pi, B, D)$. The following are the steps in function $atleast\_dc$:

1. computes $S := lc(\Pi, B, D)$
2. if S is consistent then updates $D := D \cup active\_dc(\Pi, S)$
3. returns S

Example 6.2.6. *Let us use the r-ground program* $\Pi_g = \mathcal{P}_d(\Pi)$ *from example 6.2.1.*

1. *The call* $\mathtt{atleast\_dc}(\Pi_g, \emptyset, \emptyset)$ *returns* $S = \{\mathrm{step}(1), \mathrm{step}(2)\}$ *and updates* D *to* $\{V1 - V2 \leq 0\}$.

2. *The call* $\mathtt{atleast\_dc}(\Pi_g, \{o(a1,2)\}, \{V1 - V2 \leq 0\})$ *returns an inconsistent set* $S = r\text{-}lits(\Pi_g)$ *and leaves* $D = \{V1 - V2 \leq 0\}$.

3. *The call* $\mathtt{atleast\_dc}(\Pi_g, \{not\ o(a1,2)\}, \{V1 - V2 \leq 0\})$ *returns* $S = \{\mathrm{step}(1),$ $\mathrm{step}(2)$, *not* $o(a1,2)$, $o(a1,1)$, *not* $o(a2,1)$, $o(a2,2)\}$ *and updates* $D = \{V1 - V2 \leq 0, V1 - V2 \leq -30\}$. *Both* S *and* D *are consistent.*

Note that, to compute $\mathrm{lc}(\Pi, B, D)$, function $\mathtt{atleast\_dc}$ uses function $\mathcal{D}solver$ to check consistency in steps (2) and (6) of computation of $\mathrm{lc}_0(\Pi, B, D)$. For actual implementation, $\mathcal{D}solver$ maintains its own constraint store D consisting of the active constraints of Π with respect to B and a solution S_D of D. Updating D occurs at step (2) of computation of $\mathrm{lc}_0(\Pi, B, D)$ and solution of updated $D := D \cup \{\neg c\}$ is computed by $\mathcal{D}solver(D, S_D, \neg c)$ using the existing solution S_D of D

Proposition 6.2.2. *Let* Π *be a program,* B *be a set of extended r-literals and* D *be a set of constraints,*

- $\mathrm{lc}(\Pi, B, D)$ *is monotonic with respect to its second argument.*
- $\mathrm{lc}(\Pi, B, D)$ *is unique.*
- *If* M *is an answer set of* Π*, then* M *agrees with* B *iff* M *agrees with* $\mathrm{lc}(\Pi, B, D)$.

Note that the proposition 6.2.2 implies that if $\mathrm{lc}(\Pi, B, D)$ is inconsistent then there is no answer set of Π that agrees with B; otherwise, every answer set of Π that agrees with B agrees with $\mathrm{lc}(\Pi, B, D)$.

Note that in the definition of $lc_0(\Pi, B, D)$, the conditions (2) and (6) are added to integrate the constraint solving techniques and uses $\mathcal{D}solver$. This definition assumes that the number of constraint atoms in the body of middle rules is exactly one. If we allow arbitrary number of constraint atoms in the body of middle rules then the conditions (2) and (6) should be replaced by the following conditions:

(2) If $r \in \Pi_M$ and *r-lits*$(r) \subseteq B$, $D \cup \neg$*c-lits*(r) is not consistent, then $head(r) \in lc_0(\Pi, B, D)$.

(6) If $r \in \Pi_M$, $h = head(r)$, *not* $h \in B$, all literals in the body of r except l_i belong to B and $D \cup \neg$*c-lits*(r) is not consistent then *not* $l_i \in lc_0(\Pi, B, D)$.

By *c-lits*(r), we mean the conjunction of all c-literals in r. The formula $\neg$*c-lits*(r) is the disjunction of negated c-literals of r. The restriction of allowing only one c-literal in the body of middle rules eliminates this disjunction, so we can use a simple difference constraint solver. The $\mathcal{D}solver$ is a difference constraint solver and does not allow disjunctions.

Function $atmost$:

Function atmost is same as described earlier (see section 5.3.1.2). Note that in a $\mathcal{AC}_0$ program Π, the set of rules in Π_M are denials (with empty head) where the head is assumed to be $false$. When computing $\alpha(\Pi, B)$ for computation of upper closure (see section 5.3), the rules in Π_M are removed by step (2) of $\alpha(\Pi, B)$. Therefore, the upper closure of Π with respect to B is same as upper closure of Π_R with respect to B. Therefore, it is interesting to note that the upper closure definition used in [75, 72] suffices here.

Function *expand_dc*:

Function *expand_dc* shown in Figure 6.2 follows the algorithm of function $expand$ from section 5.3 but calls function $atleast\_dc$ instead of function

function expand_dc (Π : r-ground program, B : set of r-literals, **var** D : set of constraints)

```
        % var S, S_0 : set of r-literals
[a]     S := B
[b]     do
[c]             S_0 := S
[d]             S := atleast_dc(Π, S, D)
[e]             S := S ∪ { not l | l ∈ r-atoms(Π), l ∉ atmost(Π, S)}
[f]     while ( (S ≠ S_0) and consistent(S) )
[g]     return S
```

Figure 6.2: Function *expand_dc*

atleast. The inputs of the function are an r-ground program Π, a set of extended r-literals B and a set of difference constraints D. The output of the function is the set of consequences S of Π, B and D.

Let D_i and D_o be respectively the input and output values of D to function *expand_dc*. Recall that the set D is updated by function atleast_dc to contain active constraints of Π with respect to lower closure S computed at step (d) in Figure 6.2. The do loop of *expand_dc* ensures the following lemma.

Lemma 6.2.1. *Let* Π *be a r-ground* $\mathcal{AC}_0$ *program and* B *be a set of extended r-literals, and* D *be a set of difference constraints input to expand_dc. Let* S *be the consistent set of consequences returned by expand_dc. Let* D_i *and* D_o *be the input and output values of* D *respectively. Then* D_o *is consistent and contains the set of active constraints of* Π *with respect to the set of consequences* S*, that is,* active_dc$(\Pi, S) \subseteq D_o$.

Further, the set of constraints in D is not changed in any other step of algorithm $Surya_d$ in Figure 6.1 other than in *expand_dc*. This can be used to easily prove the lemma below.

Lemma 6.2.2. *Let Π be a r-ground program, B_i's be set of extended r-literals and D_i's be set of difference constraints. Let $Surya_d(\Pi, B_i, D_i)$ be the i^{th} recursive call of $Surya_d$ for computing answer sets of Π. Then,*

- ▷ *for $i > 1$, $\text{active\_dc}(\Pi, B_i \setminus \{l, not\ l\}) \subseteq D_i$, where l is the chosen literal selected by pick function at $i-1^{th}$ call.*

Example 6.2.7. *This example shows the computation of an answer set using algorithm $Surya_d$. Consider the r-ground program $\Pi_g = \mathcal{P}_d(\Pi)$ from example 6.2.1. Since this program does not have any cr-rules, Π_g is directly used to compute answer sets. $\mathcal{AD}$engine calls $Surya_d(\Pi_g, \emptyset, \emptyset)$.*

1. *During initialization, a new variable V representing csort time is introduced. The constraints $0 \leq V_i - V \leq 1440$ for each time variable V_i in the program is added to the constraint store. The program Π_g contains two time variables $V1$ and $V2$. Therefore the constraint store is initialized to*
   $D_1 = \{V1 - V \leq 1440,\ V - V1 \leq 0,\ V2 - V \leq 1440,\ V - V2 \leq 0\}$.

2. *$Surya_d$ calls $expand\_dc(\Pi_g, \emptyset, D_1)$ and returns $S_2 = \{\text{step}(1), \text{step}(2)\}$ and $D_2 = D_1 \cup \{V1 - V2 \leq 0\}$ (see step (1) in example 6.2.6).*

3. *Since S_2 is consistent and not complete, $Surya_d$ picks a literal l from $\bar{S}_2$. Suppose it picks atom $o(a1, 2)$ (action $a1$ occurs at step 2). $Surya_d(\Pi_g, S_3, D_2)$ is called recursively, where $S_3 = S_2 \cup \{o(a1, 2)\}$.*

4. *$Surya_d$ calls $expand\_dc(\Pi_g, S_3, D_2)$. $expand\_dc$ returns an inconsistent set $S_4 = r\text{-}lits(\Pi_g)$ and leaves D_2 as is. (see step(2) in example 6.2.6).*

5. *$Surya_d$ backtracks and calls $Surya_d(\Pi_g, S_4, D_2)$, where $S_4 = S_2 \cup \{not\ o(a1, 2)\}$ (negation of previously picked literal $o(a1, 2)$ is*

tried).

6. *Surya*$_d$ *calls expand_dc*(Π_g, S_4, D_2). *expand_dc returns a consistent set* $S_5 = \{step(1),\ step(2),\ not\ o(a1,2),\ o(a1,1),\ not\ o(a2,1),\ o(a2,2)\}$ *and updates* $D_5 = \{V1 - V2 \leq 0, V1 - V2 \leq -30\}$. *(see step(3) in example 6.2.6).*

7. S_5 *is complete and covers all atoms of* Π_g *and* D_5 *is consistent. Therefore,* $pos(S) \cup m\text{-}atoms(\Pi_g)|^{vars(\Pi_g)}_{solution(D_5)}$ *is a simplified answer set and is returned. A solution for the set of constraints* D_5 *is* $solution(D_5) = \{V1 = 0, V2 = 31\}$. *The m-atoms in* Π_g *are* $\{at(1,V1),\ at(2,V2)\}$. *The substitution of variables in m-atoms of* Π_g *using* $solution(D_5)$ *yields* $\{at(1,0),\ at(2,31)\}$. *Therefore, the simplified answer set* $\{step(1),\ step(2),\ o(a1,1),\ o(a2,2), at(1,0), at(2,31)\}$ *is returned. In this answer set, action* $a1$ *occurs at step 1 which is executed at 0th minute and* $a2$ *occurs at step 2 which is executed at 31st minute.*

Note that while checking for consistency, $\mathcal{D}solver$ computes and maintains a solution for the current D, this is used as $solution(D)$ to compute $m\text{-}atoms(\Pi_g)|^{vars(\Pi_g)}_{solution(D)}$.

CHAPTER 7

KNOWLEDGE REPRESENTATION

In this chapter we look at representing some example problems[1] in languages $\mathcal{AC}_0$ and $\mathcal{AC}(\mathcal{R})$ where $\mathcal{R}$ is a real domain. Then we present some execution times for computing answer sets of the example problems using $\mathcal{AD}solver$ and $\mathcal{AC}solver$. For easy reference, we present a complete list of languages, their features and solvers in Tables 7.1 and 7.2.

Languages	Features
$\mathcal{V}(\mathcal{C})$	allows regular, mixed and constraint atoms m-atoms and c-atoms occur only in denials allows cr-rules
$\mathcal{AC}(\mathcal{C})$	allows regular, mixed, constraint and defined atoms allows non-empty head rules with m-atoms and c-atoms in body allows complex user-defined relations using defined predicates
$\mathcal{V}(\mathcal{C})_{\mathrm{cr}}$	extension of $\mathcal{V}(\mathcal{C})$ to allow preferences from CR-Prolog syntax and semantics is a natural extension of CR-prolog
$\mathcal{AC}(\mathcal{C})_{\mathrm{cr}}$	extension of $\mathcal{AC}(\mathcal{C})$ to allow cr-rules and preferences from CR-Prolog syntax and semantics is a natural extension of CR-prolog

Table 7.1: Languages and their Features

Languages	Instance	Solvers
$\mathcal{AC}_0$	Instance of $\mathcal{V}(\mathcal{C})$ with $\mathcal{C} = \{X - Y > k\}$	$\mathcal{AD}solver$
$\mathcal{AC}(\mathcal{R})$	Instance of $\mathcal{AC}(\mathcal{C})$ with $\mathcal{C} = \mathcal{R}$ (real)	$\mathcal{AC}solver$
$\mathcal{AC}_{0\mathrm{cr}}$	Instance of $\mathcal{V}(\mathcal{C})_{\mathrm{cr}}$ with $\mathcal{C} = \{X - Y > k\}$	$\mathcal{AD}solver$
$\mathcal{AC}(\mathcal{R})_{\mathrm{cr}}$	Instance of $\mathcal{AC}(\mathcal{C})_{\mathrm{cr}}$ with $\mathcal{C} = \mathcal{R}$ (real)	$\mathcal{AC}solver$

Table 7.2: Languages and Solvers

[1] To make it simpler, the syntax of mixed declarations and rules in the examples have been slightly modified from the real syntax accepted by the solvers, refer [74, 73] for the syntax accepted by the solvers.

7.1 Representing Knowledge in $\mathcal{AC}_0$ and $\mathcal{AC}_{0cr}$

This section discusses questions like, "How to select mixed predicates?", "What types of knowledge can be represented using $\mathcal{AC}_0$?", "Can we successfully compute answer sets for complex planning and scheduling problems using the new solver?". Normally, a variable with a large domain is viewed as a constraint variable. Those with small domains are regular variables. We select mixed predicates as those which contain both these variables. A limitation to select mixed predicates is to note that these predicates can be used only in body of denials. With respect to $\mathcal{AC}_0$, only constraints of the form $X - Y > k$ are allowed. Therefore knowledge represented by constraint variables in mixed predicates are limited to these constraints. Interestingly, these constraints are used widely in constraint programming [83, 44, 43, 84].

$\mathcal{AC}_0$ is good for representing planning and scheduling problems. Given a task of executing n actions and time restrictions on their executions, a scheduling problem consists of finding times $T_1, \ldots, T_n$ such that action a_i occurs at time T_i and satisfies all the time restrictions. The timing restrictions can be temporal distance constraints between any two actions. Such constraints can be represented in $\mathcal{AC}_0$ as follows. Let $a_1, \ldots, a_n$ be n actions and $S_1, \ldots, S_n$ be variables in domain $[1..m]$. The r-atom $occurs(a_i, S_i)$ is read as, "action a_i occurs at step S_i". The step S_i denotes a time point T_i and is represented by an m-atom $at(S_i, T_i)$. Atom $at(S, T)$ is read as 'step S happens at time T'. The domain of a step S is comparatively smaller than the domain of a time variable T. Hence S is a r-variable and T is a c-variable used to write timing constraints.

Our approach allows a rather straightforward representation of actions with durations. When actions have durations, the scheduling problem finds the start and end time points of actions such that all timing restrictions are satisfied. One

method of representing constraints on action durations in $\mathcal{AC}_0$ is as follows. Let $a_1, \ldots, a_n$ be actions and $S_1, \ldots, S_n$ be variables from the domain $[1..m]$. The r-atom $occurs(a_i, S_i)$ is read as, "action a_i occurs at step S_i". The step S_i denotes a time interval $[T_{si}, T_{ei}]$. The time interval of step S_i is represented by two m-atoms $at(S_i, start, T_{si})$ and $at(S_i, end, T_{ei})$. Atom $at(S, start, T)$ is read as 'step S starts at T'. We can write temporal constraints using the c-variables T_{si} and T_{ei}.

Example 7.1.1. *[84]* [**Breakfast problem**] *We have a scheduling problem, "Prepare coffee and toast. Have them ready within 2 minutes of each other. Brew coffee for 3-5 minutes; toast bread for 2-4 minutes." We start by defining signature,* $\Sigma = \{C_r = \{$ start, end, brew, toast, $S_c = [1..2]$ $\}, P_r = \{step, occurs\}, C_c = \{D_c = [0..1439], R_c = [-1439..1439]\}, V_c = \{T_1, T_2\}, F_c = \{-\}, P_c = \{>\}, P_m = \{at\}\}$. *Constants "*brew, toast*" represents actions 'brewing coffee' and 'toasting bread'. To solve this, we first represent constraints and then we have a small planning module to represent action* a_i *occurs at some time step* S_i. *The constraints are as follows.*

% Brew coffee for 3 to 5 minutes is represented using two c-rules

$\leftarrow$ $occurs(brew, S), at(S, start, T_1), at(S, end, T_2), T_2 - T_1 > 5.$

$\leftarrow$ $occurs(brew, S), at(S, start, T_1), at(S, end, T_2), T_1 - T_2 > -3.$

% Toast bread for 2 to 4 minutes is represented by two c-rules

$\leftarrow$ $occurs(toast, S), at(S, start, T_1), at(S, end, T_2), T_2 - T_1 > 4.$

$\leftarrow$ $occurs(toast, S), at(S, start, T_1), at(S, end, T_2), T_1 - T_2 > -2.$

% Coffee and bread should be ready between 2 minutes of each other

$\leftarrow$ $occurs(brew, S_1), occurs(toast, S_2), at(S_1, end, T_1),$
$at(S_2, end, T_2), T_2 - T_1 > 2.$

$\leftarrow$ $occurs(brew, S_1), occurs(toast, S_2), at(S_1, end, T_1),$

$at(S_2, end, T_2), T_1 - T_2 > -2.$

% Start time of step 1 is before step 2

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

% A simple planning module to represent occurrence of actions:

$step(1..2).$

$occurs(brew, S)\ or\ occurs(toast, S)\ \leftarrow\ step(S).$

$\leftarrow\ action(A),\ occurs(A, S_1),\ occurs(A, S_2),\ S_1 \neq S_2.$

The first c-rule is read as: 'If brewing coffee occurs at step S, then duration between start and end of S cannot be more than 5 minutes'. The second c-rule says that 'start and end times for S cannot be less than 3 minutes. The c-atom is written as $T_1 - T_2 > -3$ instead of $T_2 - T_1 < 3$ as the implementation allows only constraints of the form $X - Y > K$. The disjunctions in the head of the rules of the above program can be eliminated using non-disjunctive rules. A solution to the above breakfast scheduling problem can be found by computing answer sets of the program using $\mathcal{AD}solver$. A solution could be to start brewing coffee at 0th minute and end at 3rd minute; start toasting bread at 2nd minute and end at 4th minute. This solution can be extracted from an answer set $\{occurs(brew, 1), occurs(toast, 2), at(1, start, 0), at(1, end, 3), at(2, start, 2), at(2, end, 4)\} \cup M_c$. Suppose we would like to schedule an action a such that it occurs either between 3am and 5am or between 7am and 8am. (Now, we assume that actions do not have duration and are instantaneous). 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]am or [7-8]am.* We cannot represent this directly in $\mathcal{AC}_0$ because of the disjunction. Instead we introduce two r-atoms int_1 and int_2 to represent intervals [3-5] and [7-8] respectively. The r-atom int_1 denotes that action a occurs in interval [3-5]. We write a disjunction

on int_i to choose the interval and then use int_i to write the constraints. The following example shows the representation of the constraint.

Example 7.1.2. *"Action a should be performed in between intervals [3-5] am or [7-8] am". Let r-atom* int_1 *(*int_2*) be true when action* a *occurs in interval [3-5]am ([7-8]am). To keep it simple, let us suppose that action* a *occurs at some step say 1. We need to assign time for this step. Atom* $at(0,T)$ *denotes time of step 0 and represents start time for our problem 12 am.*

$occurs(a,1)$.

% 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 between [3-5]', is encoded using two c-rules*

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

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

% 'If a *occurs at step* S *and* int_2 *is true, then* S *should be between [7-8]', is encoded using two c-rules*

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

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

An answer set for this program would be $\{occurs(a,1)$, int_2, $at(0,0)$, $at(1,7)\} \cup M_c$, where a occurs at 7 am. The temporal constraints shown above are examples of simple temporal constraints and disjunctive temporal constraints [28]. The following example is from [28]. We show that we can represent the problem and answer some of the questions asked in the example. Though, syntax of $\mathcal{AC}_0$ does not allow choice rules and cardinality constraints [75], $\mathcal{AD}solver$ built on top of $lparse$ and $Surya$ allows these type of rules in its input language. For concise representation, we use choice rules and cardinality constraints in the following

example.

Example 7.1.3. *[28]* [**Carpool**] *John goes to work either by car (30-40 mins), or by bus (at least 60 mins). Fred goes to work either by car (20-30 mins), or in a car pool (40-50 mins). Today John left home between 7:10 and 7:20, and Fred arrived between 8:00 and 8:10. We also know that John arrived at work about 10-20 mins after Fred left home. We wish to answer queries such as: "Is the information in the story consistent?","Is it possible that John took the bus, and Fred used the carpool?", "What are the possible times at which Fred left home?".*

```
%% John goes to work either by car or by bus. (We use a choice rule)
1{ j_by_car, j_by_bus }1.
%% Fred goes to work either in car or by car pool
1{ f_by_car, f_by_cpool }1.

%% define 'time' as csort and 'at' as a mixed predicate
#csort time(0..1440).
timepoint(start_time; start_john; end_john; start_fred; end_fred).
#mixed at(timepoint, time).

%% It takes John 30 to 40 minutes by car
:- j_by_car, at(start_john, T1), at(end_john, T2), T2 - T1 > 40.
:- j_by_car, at(start_john, T1), at(end_john, T2), T1 - T2 > -30.

%% "It takes John atleast 60 minutes by bus" is represented by a c-rule
:- j_by_bus, at(start_john, T1), at(end_john, T2), T1 - T2 > -60.
```

```
%% "It takes Fred 20 to 30 minutes by car" is represented by two c-rules
:- f_by_car, at(start_fred, T1), at(end_fred, T2), T2 - T1 > 30.
:- f_by_car, at(start_fred, T1), at(end_fred, T2), T1 - T2 > -20.

%% It takes Fred 40 to 50 minutes by car pool
:- f_by_cpool, at(start_fred, T1), at(end_fred, T2), T2 - T1 > 50.
:- f_by_cpool, at(start_fred, T1), at(end_fred, T2), T1 - T2 > -40.

%% We view the start time as 7am, that is 0 minutes = 7am
%% Today John left home between 7:10 and 7:20
:- at(start_john, T), at(start_time, T0), T0 - T > -10.
:- at(start_john, T), at(start_time, T0), T - T0 > 20.

%% Fred arrived at work between 8:00 and 8:10
:- at(end_fred, T), at(start_time, T0), T0 - T > -60.
:- at(end_fred, T), at(start_time, T0), T - T0 > 70.

%% John arrived at work about 10-20 mins after fred left home
:- at(end_john, T1), at(start_fred, T2), T1 - T2 > 20.
:- at(end_john, T1), at(start_fred, T2), T2 - T1 > -10.
```

Now let us look at answering each of the questions in the problem.

```
Question (1) Is the information in story consistent ?
To answer this question, we find answer sets of the program.
Answer Set: j_by_car f_by_cpool at(start_time,0) at(start_john, 10)
            at(end_john, 40) at(start_fred, 20) at(end_fred, 60)
```

The above answer set corresponds to John using the car and Fred using the

car pool. John starts at 7:10 am and reaches at 7:40 am. Fred starts at 7:20 am and reaches at 8:00 am. The information is consistent since the program has an answer set. The time taken by ADsolver was 0.065 seconds of which 0.018 seconds was used for grounding and loading.

Question(2) Is it possible that John took the bus and Fred used carpool?
To answer this question, we add the following knowledge to our program and compute answer sets.

```
j_by_bus.
f_by_cpool.
```

There are no answer sets for this new program.

Therefore, according to the story, it is not possible for John to take a bus and Fred to use a carpool and have the other timing information true. The time taken by ADsolver was 0.029 seconds of which grounding and loading took 0.018 seconds.

Question (3) What are the possible times that Fred left home?
To answer this question, we need to find the interval of time when Fred can leave home and still have all the information in the story true.

This answer cannot be found using ADsolver, as the underlying constraint solver Dsolver *is a simple constraint solver and computes solutions for a set of difference constraints and does not compute intervals (as needed in this example). In [28], the authors use a more complex Floyd-Warshall algorithm [26] to compute answer to this question. Further, such type of questions can be answered by looking at all the answer sets and this is not possible with ADsolver or ACsolver. ACsolver can answer interval questions like "If Fred and John both used cars today, then what are the possible times that Fred left*

home". Notice that such interval questions are queried upon single answer sets and need not look at all answer sets put together.

Using cr-rules in $\mathcal{AC}_0$ we can represent important information like, "an event e may happen but it is very rare". Such information is very useful in default reasoning. Combining such information together with c-rules allows us to represent qualitative soft constraints [86, 87, 103, 60, 47] like, "an event e may happen but it is very rare; if event e happens then ignore constraint c". The following example is an extension of 7.1.3 and shows the representation of qualitative soft temporal constraints.

Example 7.1.4. *This example shows the use of cr-rules to express qualitative soft temporal constraints. For this we use example 7.1.3. We remove information that "John arrived at work about 10-20 mins after Fred left home" and extend the story as follows: It is desirable for Fred to arrive atleast 20 mins before John.*

```
%% It is desirable for Fred to arrive atleast 20 mins before John.
:- at(end_fred, T1), at(end_john, T2), not is_late, T1 - T2 > -20.
%% CR-rule r1: We may possibly believe that Fred is late
r1: is_late +-.
```

For the newly added information, we get two models where Fred arrives before John in each of them.

```
Answer set (1): j_by_bus f_by_car  at(start_john,20) at(end_john,100)
                at(start_fred,30) at(end_fred,60) at(start_time,0)
Answer set (2): j_by_bus f_by_carpool at(start_john,20) at(end_john,80)
                at(start_fred,20) at(end_fred,60) at(start_time,0)
```

To compute the two models, ADsolver took 0.064 seconds of which 0.019 seconds were used for grounding and loading. Now we would like to expand our story, "We come to know that Fred's car is broken and therefore, he cannot use it". We add the following rule to the program.

```
:- f_by_car.
```

For the new program, we get one model where John travels by bus and Fred uses the carpool and still reaches before John.

```
Answer set: j_by_bus f_by_carpool  at(start_john,20) at(end_john,80)
            at(start_fred,20) at(end_fred,60) at(start_time,0)
```

ADsolver took 0.053 seconds to compute the model. Suppose we know that John used his car today. Will Fred arrive atleast 20 mins before John as desired? For this, we add the following rule to the program.

```
j_by_car.
```

There is no model where Fred arrives 20 minutes before John and the cr-rule was fired to give the following answer set.

```
Answer set: j_by_car f_by_cpool is_late at(start_john,20) at(end_john,60)
            at(start_fred,20) at(end_fred,60) at(start_time,0)
```

Fred is late and cannot arrive 20 minutes before John as desired. ADsolver took 0.052 seconds to compute the model.

Simple temporal constraints, disjunctive temporal constraints and qualitative soft constraints can be expressed in $\mathcal{AC}_0$. The implemented solver is faster than a standard ASP solver when domains of constraint variables are large. The language of CR-Prolog also allows preferences on the cr-rules [3]. Given two cr-rules r_1 and

r_2, the statement $prefer(r_1, r_2)$ allows preference to cr-rule r_1 when compared to cr-rule r_2. CR-Prolog allows static and dynamic preferences. The language $\mathcal{AC}_0$ does not allow preferences but $\mathcal{AC}_0$ syntax can be easily extended to allow CR-Prolog style preferences and the semantics would be a natural extension of CR-Prolog. Though language $\mathcal{AC}_0$ does not allow preferences, the solver $\mathcal{AD}solver$ which is built using the meta layer of CR-Prolog solver, allows preferences. So, we can express soft qualitative temporal constraints with preferences which is used in constraint programming applications [86, 87].

Example 7.1.5. *This example shows the representation of qualitative soft temporal constraints with preferences. Let us use example 7.1.4. We remove information that "John arrived at work about 10-20 mins after Fred left home and Fred arrived between 8:00 and 8:10" and extend the story as follows: It is desirable for Fred to arrive atleast 20 mins before John. If possible, Fred desires to start from home after 7:30am. We also know that Fred's car is broken and John used his car today.*

```
%% If possible, Fred desires to leave after 7:30am
:- at(start_time, T1), at(start_fred,T2), not start_early, T1 - T2 > -30.
%% CR-rule r2: sometimes, Fred may need to start early.
r2: start_early +-.
```

The above rules along with other rules from examples 7.1.4 and 7.1.3 represent the information in the story. We get two answer sets where cr-rules were used in both.

```
Answer set (1):
j_by_car f_by_cpool is_late at(start_john,20) at(end_john,60)
          at(start_fred,30) at(end_fred,70) at(start_time,0)
```

```
Answer set (2):
j_by_car f_by_cpool start_early at(start_john,20) at(end_john,60)
                at(start_fred,0) at(end_fred,40) at(start_time,0)
```

Now we add new preference information that "Fred prefers coming before John than starting late from home". we represent the preference as follows:

```
% Prefer starting early to reaching late
prefer(r2,r1).
```

Now, we get only one model:

```
Answer set:
j_by_car f_by_cpool start_early at(start_john,20) at(end_john,60)
                at(start_fred,0) at(end_fred,40) at(start_time,0)
```

The other model is not preferred when compared to this one and therefore is not returned. $\mathcal{AD}$solver computed the answer set in 0.13 seconds.

The above example clearly shows the use of preferences from CR-Prolog along with c-rules gives a natural representation of qualitative soft constraints with preferences. Similarly, we can use cr-rules, cr-preferences and c-rules together to represent disjunctive soft temporal constraints and disjunctive soft temporal constraints with preferences which are also useful for scheduling problems. Another investigation we are concerned with is whether $\mathcal{AC}_0$ can be used for complex planning and scheduling problems. Also, whether we can use the implemented solver to compute answer sets in realistic time for these problems. To test this, we have used the system USA-Advisor[7] , 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.

USA-Advisor can be used for checking correctness of plans and for planning and diagnosis. The system does not contain any timing restrictions or constraints. To test our solver, we expand the system to allow explicit representation of time and reason with time constraints. We use it to solve planning and scheduling tasks. We will illustrate our extension by the following example.

Example 7.1.6. [**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. Further, we cannot open a valve V which links N1 to N2, (link(N1,N2,V)), until N1 has been stabilized. We would like to assign real times to the time steps given in the program such that this constraint is satisfied. Also, can we answer questions like: can the whole maneuver take less than 30 secs?*

$\Sigma = \Sigma_{\text{old}} \cup \{P_r = \{\text{otank}, \text{ftank}, \text{got\_opened}, \text{got\_pressurized}\}$,

$P_m = \{at\}, C_c = \{D_c = [0..400], R_c = [-400..400]\}, F_c = \{-\}, P_c = \{>\}\}$. *Atoms* otank(X) *and* ftank(X) *denote that* 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. *Atom* at(S, T) *is read as 'step* S *is performed at time* T*', where* S *is a regular variable with domain 0 to plan length;* T *is a constraint variable with domain [0..400] seconds. The new program contains all rules from original advisor, and new rules describing the scheduling constraints. The first rule is from USA-Advisor, followed by some new rules. The second rule shows the connection between original program and new one.*

% Tank node N_1 *is pressurized by tank X if it is connected by an open valve to a node which is pressurized by tank X of sub-system R*

$$h(\text{pressurized\_by}(N_1, X), S) \leftarrow \text{step}(S), \text{tank\_of}(N_1, R), h(\text{in\_state}(V, \text{open}), S), \text{link}(N_2, N_1, V), \text{tank\_of}(X, R), h(\text{pressurized\_by}(N_2, X), S).$$

% node gets pressurized when it was not pressurized at S and pressurized at S+1.

$$\text{got\_pressurized}(N, X, S+1) \leftarrow \text{link}(N_1, N, V), \text{tank\_of}(X, R), \textit{not}\ h(\text{pressurized\_by}(N, X), S), h(\text{pressurized\_by}(N, X), S+1).$$

% A valve V *linking* N_1 *to* N_2 *cannot be opened unless* N_1 *is stabilized.*

% If N_1 *is pressurized by oxidizer tank,* N_1 *takes 5 seconds to stabilize.*

$$\leftarrow \text{link}(N_1, N_2, V), \text{got\_pressurized}(N_1, X, S_1), S_1 < S_2, \text{otank}(X), \text{got\_opened}(V, S_2), \text{at}(S_1, T_1), \text{at}(S_2, T_2), T_1 - T_2 > -5$$

% If N_1 *is pressurized by fuel tank,* N_1 *takes 10 seconds to stabilize.*

$\leftarrow$ $link(N_1, N_2, V), got\_pressurized(N_1, X, S_1), S_1 < S_2, ftank(X),$
$got\_opened(V, S_2), at(S_1, T_1), at(S_2, T_2), T_1 - T_2 > -10$

% time should be increasingly assigned to steps

$\leftarrow$ $S_1 < S_2, at(S_1, T_1), at(S_2, T_2), T_1 - T_2 > -1$

% The jets of a system should be ready to fire by 30 seconds

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

The USA-Advisor extension was used for testing efficiency of $\mathcal{AD}solver$. About 900 test cases available at [9] were used. The USA-Advisor files (old) used were *rcs1, heuristics, plan, problem-base* available at [9]. Along with these files, we used a file containing the scheduling constraints (example 7.1.6) and an instance file available at [9]. The instance file specifies the faults in the system (for example, a leaky valve or a switch stuck at a position) and a goal (for example to ready jets to accomplish a right manuever) for planning. $\mathcal{AD}solver$ was used to compute plans and an answer set would give the sequence of actions to be performed in order to achieve the goal in a required number of steps. The answer sets returned properly scheduled plans that satisfies the above contraints. Experiments were run on a Mac OS X powerPC G4 with 1 GHz processor and 512 MB SDRAM. Timing results of $\mathcal{AD}solver$ are shown in Figures 7.1, 7.2 and 7.3. Timing results in Figure 7.1 correspond to 50 instances (instance001..instance050) from folders 'instance-auto /ins' (left) and 'instance-auto /ins-4' (right). Timing results in Figure 7.2 correspond to 50 instances (instance001..instance050) from folders 'instance-auto /ins-8' (left) and 'instance-monica/ins-3-0' (right). Timing results in Figure 7.3 correspond to 50 instances (instance001..instance050) from folders 'instance-monica/ins-5-0' (left) and 'instance-monica/ins-8-0' (right). Each instance was used in three different runs to compute plans of length $n = 3$, $n = 4$ and $n = 5$. Therefore, each image shows timing results of 50 instances each at

$n = 3$, $n = 4$ and $n = 5$ plan lengths (total 150 time points). The time results shown correspond to the number of seconds taken by $\mathcal{AD}solver$ to compute an answer set giving the plan or to return $false$ saying there exists no plan for the given problem for specified number of steps.

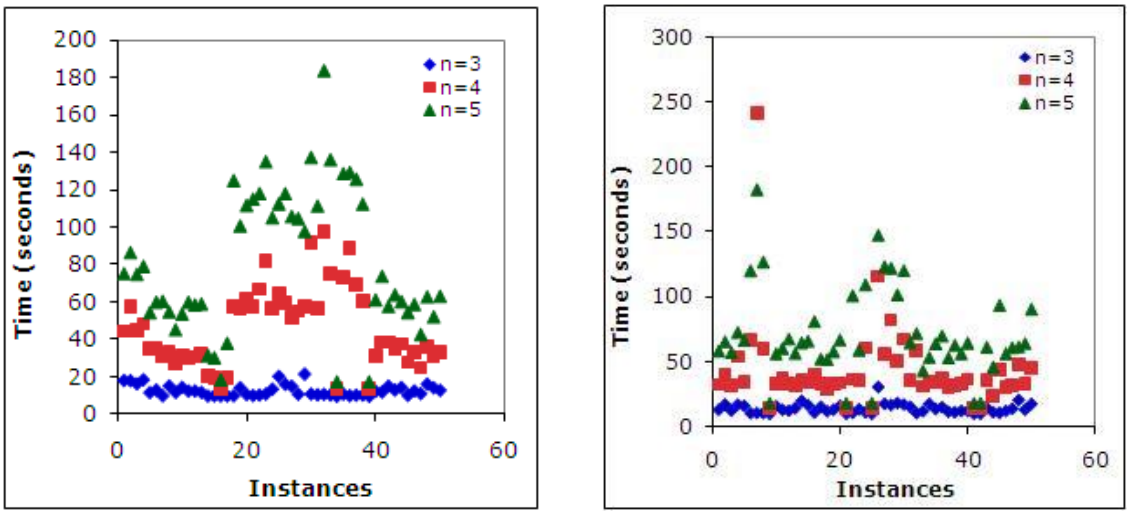

Figure 7.1: $\mathcal{AD}solver$ Time Results (1) on Planning and Scheduling in USA-Advisor

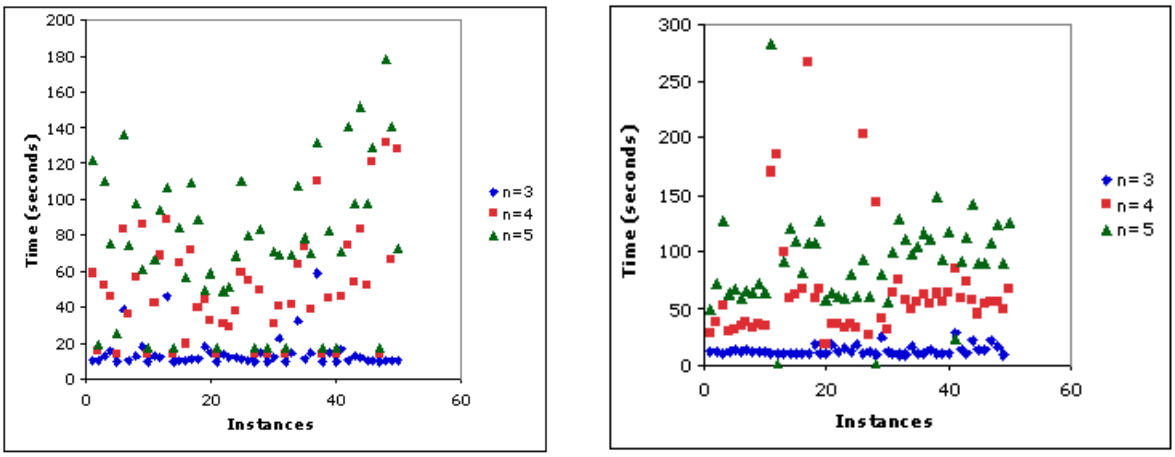

Figure 7.2: $\mathcal{AD}solver$ Time Results (2) on Planning and Scheduling in USA-Advisor

Timing results clearly show that for most of the instances, $\mathcal{AD}solver$ took less than 150 seconds. There were some instances for which $\mathcal{AD}solver$ took longer

Folder	Instance	Plan Length	has Found	Time (seconds)
ins-4	008	4	no	3359.59
ins-8	009	3	no	717.768
ins-8	009	5	yes	2111.38
ins-8	050	5	yes	2672.36
ins-3-0	011	4	yes	1370.32
ins-3-0	012	4	no	1784.36
ins-3-0	012	5	yes	17509.6
ins-3-0	013	5	yes	911.337
ins-3-0	016	5	yes	832.474
ins-3-0	020	5	yes	9358.20
ins-3-0	041	5	yes	2923.75
ins-3-0	044	5	yes	840.977
ins-5-0	011	4	yes	1540.06
ins-5-0	016	4	no	738.710
ins-5-0	013	5	yes	3067.71
ins-5-0	016	5	yes	4259.71
ins-5-0	017	5	yes	5482.55
ins-8-0	034	3	no	1115.15
ins-8-0	022	4	no	2030.4
ins-8-0	006	5	yes	7409.42
ins-8-0	015	5	yes	1052.09
ins-8-0	022	5	yes	1207.68

Table 7.3: $\mathcal{AD}solver$ Time Results (4) on Planning and Scheduling in USA-Advisor

times and their timing results are shown in Table 7.3. These instances were removed from the images so that the other timing results are seen better. For all the instances finding plans of length three, the number of partially ground rules in $\mathcal{P}(\Pi)$ was approximately around 96000 rules. Similarly for plan length $n = 4$, $\mathcal{P}(\Pi)$ was approximately 126000 rules and for $n = 5$, $\mathcal{P}(\Pi)$ was approximately 150000 rules.

Table 7.3 shows the timing of instances that took longer than 500 seconds. These instances are not shown in the plots. The first and second columns of the table

give the folder and instance number of the instance. The third column gives the length of the plan n for which the instance was run. The fourth column shows yes if $\mathcal{AD}solver$ returned an answer set and shows no if $\mathcal{AD}solver$ returned $false$ meaning there is no answer set.

Any $\mathcal{AC}_0$ program can be translated to a *CR-Prolog* program. To translate a $\mathcal{AC}_0$ program to $CR-Prolog$ program, we need to follow these steps:

1. Remove rules of type $\#csort$ $\mathrm{cpred}(0..n)$
   and replace by rules $\mathrm{cpred}(0..n)$.

2. Remove rules of type $\#mixed$ $\mathrm{mpred}(\bar{t}_r, \bar{t}_c)$ and add a choice rule of the form:

   $$1\ \{\ \mathrm{mpred}(\bar{t}_r, \bar{t}_c)\ :\ \mathrm{cpred}(\bar{t}_c)\ \}\ 1\ :-\ \mathrm{rpred}(\bar{t}_r)$$

   where cpred and rpred are appropriate predicate symbols for the c-terms and r-terms. Note that we can write regular rules equivalent of the above choice rule.

The USA Advisor planning and scheduling example was translated to a *CR-Prolog* program. The example was tested using Lparse as the grounder and Surya and Smodels as the inference engines. The domain of the time variables was set as $time(0..400)$ seconds. The grounder lparse could not ground the program and returned a malloc error denoting out of memory. When the domains were reduced to $time(0..31)$, lparse could ground the program giving a ground instantiation of around 8 million rules. Surya and Smodels could not compute answer sets in the alloted two hour time limit. The domain of the c-variables in a program does not influence the efficiency of $\mathcal{AD}solver$. The timing results would be the same if we change the domain of time to $time(0..31)$ or $time(0..1440)$. The results show that $\mathcal{AD}solver$ performs well in cases where classical ASP solvers fail.

7.2 Representing Knowledge in $\mathcal{AC}(\mathcal{R})$ and $\mathcal{AC}(\mathcal{R})_{\rm cr}$

Language $\mathcal{AC}(\mathcal{R})$ is language $\mathcal{AC}(\mathcal{C})$ with real constraint domain ($\mathcal{C} = \mathrm{R}$). In this section we will look at methodology of representing knowledge in $\mathcal{AC}(\mathcal{R})$ and its expressiveness.

Note that the syntax restrictions (see section 4.1) on $\mathcal{AC}$*solver* programs need to be satisfied to use the $\mathcal{AC}$*solver* built. Further, the current version of $\mathcal{AC}$*solver does not allow regular literals in the body of defined rules.*

Language $\mathcal{AC}(\mathcal{R})$ can express all problems in language $\mathcal{AC}_0$ with the exception of cr-rules. Let $\mathcal{AC}(\mathcal{R})_{\rm cr}$ be an extension of $\mathcal{AC}(\mathcal{R})$ whose syntax allows cr-rules and preferences and the semantics is a natural extension for cr-rules like in $\mathcal{AC}_0$. The implemented $\mathcal{AC}$*solver* is for language $\mathcal{AC}(\mathcal{R})_{\rm cr}$ and allows cr-rules and preferences from language CR-Prolog, just like solver $\mathcal{AD}$*solver* does. Therefore, all example problems in section 7.1 can be represented and answer sets can be computed using $\mathcal{AC}$*solver*.

Some interesting things to note about $\mathcal{AC}$*solver* is that it allows more than one constraint in the body of the rules and allows non-empty head for middle rules. This allows a direct representation of interval problems.

Example 7.2.1. *In example 7.1.1, the information: 'Brew coffee for 3-5 minutes' can be represented using a mixed rule and a regular rule as follows:*

```
proper_coffee :- brew_coffee, at(start_brew,T1), at(end_brew,T2),
                 3 <= T2 - T1, T2 - T1 <= 5.
:- not proper_coffee.
```

Note that proper_coffee *is a regular atom and is in the head of the first rule which contains mixed and constraint atoms in the body. Such rules are not allowed in language $\mathcal{V}(\mathcal{C})$.*

Recall that *ACsolver* computes the simplified answer set of a program and uses a CLP(R) solver to integrate resolution and constraint solving. Therefore, unlike *ADsolver* which returns a value for the constraint variables in the program, *ACsolver* returns answer constraints of the constraint variables in the program. For instance, the example below shows the answer constraints returned by *ACsolver*

Example 7.2.2. *Consider the program in example 7.1.3. ACsolver can be used to compute answer sets of the program. An answer set returned by ACsolver is:*

```
Answer set (1): f_by_car j_by_bus at(start_fred,V0) at(end_john,V1)
                at(start_time,V2) at(end_fred,V3) at(start_john,V4)
                Constraints: V1=V0+20  V2=V0-50  V3=V0+20  V4=V0-40
                V0<=1420  50<=V0
```

The answer set shows that if Fred and John both travelled by car then for the story to be consistent, Fred takes 20 minutes to reach (V3 = V0+20) the office and John started 40 minutes before Fred left home (V4= V0-40).

These constraints can be used to find a solution to the constraint variables. In *ADsolver* the constraints allowed are always on two variables (representing distance between two variables). Therefore we cannot set a variable directly to a particular value. In $\mathcal{AC}(\mathcal{R})$, we can set a variable to a particular value.

Example 7.2.3. *In example 7.1.3, to represent that John left home between 7:10 and 7:20 am, we introduced a new time variable representing the start of time T0, and wrote the following rules:*

```
%% We view the start time as 7am, that is 0 minutes = 7am
```

```
%% Today John left home between 7:10 and 7:20
:- at(start_john, T), at(start_time, T0), T0 - T > -10.
:- at(start_john, T), at(start_time, T0), T - T0 > 20.
```

We do not set the variable T0 to 0. Given a solution S *to a set of difference constraints* D *and a constant* d*, the assignment obtained by adding* d *to every variable in* S *is also a solution to* D *[26]. Using this lemma, $\mathcal{AD}$solver sets the value of one variable and computes the solution. In $\mathcal{AC}(\mathcal{R})$, we can directly set the value of T0 as follows:*

```
% start time is 7am (= 0 minutes)
:- at(start_time,T0), T0 != 0.
```

$\mathcal{AC}$solver computes an answer set of the story along with the new information:

```
Answer set (1): f_by_car j_by_bus at(start_time,V0) at(start_fred,V1)
                at(end_john,V2) at(end_fred,V3) at(start_john,V4)
                Constraints: V0=0  V1=50  V2=70  V3=70  V4=10
```

The constraints are simplified with the new information. When John and Fred both travel by car then for the story to be consistent, Fred starts at 7:50am and reaches at 8:10am; John starts at 7:10am and reaches office at 8:10am.

In chapter 1, we saw an example of Ram visiting a dentist. Next we will see the full representation of the problem in language $\mathcal{AC}_0$ and then we will extend the example to show expressiveness of language $\mathcal{AC}(\mathcal{R})$.

Example 7.2.4. [**Visit a dentist**] *Ram is at his office and has a dentist appointment in one hour. For the appointment, he needs his insurance card*

Locations	Doctor	Home	Office	Atm
Doctor	0	20	30	40
Home	20	0	15	15
Office	30	15	0	20
Atm	40	15	20	0

Table 7.4: Travel time (in minutes) between several locations in Example 7.2.4

which is at home and cash to pay the doctor. He can get cash from the nearby atm. Table 1.1 shows the time in minutes needed to travel between locations: Doctor, Home, Office and Atm. For example, the time needed to travel between Ram's office to the Atm is 20 minutes. If the available actions are: moving from one location to another and picking items such as cash or insurance card then,
(a) find a plan which takes Ram to the doctor on time, and
(b) find a plan which takes Ram to the doctor at least 15 minutes early.

```
% Objects
person(ram).     item(icard).    item(cash).
loc(dentist).    loc(office).     loc(home).     loc(atm).
step(0..4).
% domain variables
#domain person(P).               #domain item(I).
#domain step(S).                 #domain loc(L;L1;L2).
% Actions
% To make representation simple, we use only one action
action(go_to(P,L)) :- person(P), loc(L).
% Fluents
fluent(at_loc(P,L)) :- person(P), loc(L).
```

```
fluent(at_loc(I,L)) :- item(I),   loc(L).
fluent(has(P,I))    :- person(P), item(I).
% Effects of action go_to
% If a person goes to loc L then he is at L in next moment of time.
h(at_loc(P,L),S1) :-    next(S0,S1), o(go_to(P,L),S0).
% If Ram is at loc L and item I is at loc L then he has item I
h(has(P,I),S) :-    h(at_loc(P,L),S), h(at_loc(I,L),S).
% If a person has an item then it is at same loc as the person
h(at_loc(I,L),S) :- h(has(P,I),S), h(at_loc(P,L),S).
% A person cannot go to a loc he is already in
:-  h(at_loc(P,L),S), o(go_to(P,L),S).

% Inertia
h(F,S1) :-  fluent(F), next(S0,S1), h(F,S0), not -h(F,S1).
-h(at_loc(P,L),S) :- h(at_loc(P,L1),S), neq(L,L1).
-h(at_loc(I,L),S) :- h(at_loc(I,L1),S), neq(L,L1).
% next
next(S,S+1) :- step(S+1).

%% Timing Constraints
% csort time
#csort time(0..1440).
% mixed relation: step S is 'at' time T
#mixed at(step,time).
% Assign times increasingly.
:- next(S1, S2), at(S1,T1),  at(S2,T2), gt(T1 - T2, 0).
```

```
% Time taken to travel between dentist and home is atleast 20 minutes
:- h(at_loc(P, home), S1), o(go_to(P, dentist),S1), next(S1,S2),
   at(S1,T1), at(S2,T2), gt(T1 - T2, -20).
% Time taken to travel between home and dentist is atleast 20 minutes
:- h(at_loc(P, dentist), S1), o(go_to(P, home),S1), next(S1,S2),
   at(S1,T1), at(S2,T2), gt(T1 - T2, -20).
% Time taken to travel between dentist and office is atleast 30 minutes
:- h(at_loc(P, office), S1), o(go_to(P, dentist),S1), next(S1,S2),
   at_time(S1,T1), at_time(S2,T2), gt(T1 - T2, -30).
% Time taken to travel between office and dentist is atleast 30 minutes
:- h(at_loc(P, dentist), S1), o(go_to(P, office),S1), next(S1,S2),
   at(S1,T1), at(S2,T2), gt(T1 - T2, -30).
% Time taken to travel between dentist and atm is atleast 40 minutes
:- h(at_loc(P, atm), S1), o(go_to(P, dentist),S1), next(S1,S2),
   at(S1,T1), at(S2,T2), gt(T1 - T2, -40).
% Time taken to travel between atm and dentist is atleast 40 minutes
:- h(at_loc(P, dentist), S1), o(go_to(P, atm),S1), next(S1,S2),
   at(S1,T1), at(S2,T2), gt(T1 - T2, -40).
% Time taken to travel between home and office is atleast 15 minutes
:- h(at_loc(P, home), S1), o(go_to(P, office),S1), next(S1,S2),
   at(S1,T1), at(S2,T2), gt(T1 - T2, -15).
% Time taken to travel between office and home is atleast 15 minutes
:- h(at_loc(P, office), S1), o(go_to(P, home),S1), next(S1,S2),
   at(S1,T1), at(S2,T2), gt(T1 - T2, -15).
% Time taken to travel between home and atm is atleast 15 minutes
:- h(at_loc(P, home), S1), o(go_to(P, atm),S1), next(S1,S2),
```

```
   at(S1,T1), at(S2,T2), gt(T1 - T2, -15).
% Time taken to travel between atm and home is atleast 15 minutes
:- h(at_loc(P, atm), S1), o(go_to(P, home),S1), next(S1,S2),
   at(S1,T1), at(S2,T2), gt(T1 - T2, -15).
% Time taken to travel between office and atm is atleast 20 minutes
:- h(at_loc(P, office), S1), o(go_to(P, atm),S1), next(S1,S2),
   at(S1,T1), at(S2,T2), gt(T1 - T2, -20).
% Time taken to travel between atm and office is atleast 20 minutes
:- h(at_loc(P, atm), S1), o(go_to(P, office),S1), next(S1,S2),
   at(S1,T1), at(S2,T2), gt(T1 - T2, -20).


%% Planning Module
1 { o(go_to(Px,Lx),S) : person(Px) : loc(Lx) } 1 :- step(S), not goal(S).
goal(S) :- h(at_loc(ram,dentist),S),
           h(has(ram,icard),S),
           h(has(ram,cash),S).
plan :- goal(S).
:- not plan.
%% Initial Situation
h(at_loc(ram,office),0).
h(at_loc(icard,home),0).
h(at_loc(cash,atm),0).
```

To answer the question (a), we add the following rule to the program:

```
% Problem (a). He should be at the dentist in 60 minutes
:- goal(S), at(0,T1), at(S,T2), gt(T2 - T1, 60).
% We can set step 0's time to be at 0 minute
```

```
:- at(0, T), T != 0.
```

An answer set computed by $\mathcal{AC}$solver is as follows:

```
Answer set:
o(go_to(ram,atm),0) o(go_to(ram,home),1) o(go_to(ram,dentist),2)
at(0,V0) at(1,V1) at(2,V2) at(3,V3) at(4,V4)
Constraints: V0=0  0<=V4  0<=V3  V3<=1440  0<=V2  V2<=1440
     V1<=1440  V4<=60  V3<=V4  V2+20<=V3  V1+15<=V2  20<=V1
```

Note that the answer set contains constraints over the variables. Now if we add the information that Ram needs to be at the dentist at least in 55 minutes, then we get an answer set that contains assignment of values to variables because there exists only one solution:

```
% He should be at the dentist in 55 minutes
:- goal(S), at(0,T1), at(S,T2), gt(T2 - T1, 55).
Answer set:
o(go_to(ram,atm),0) o(go_to(ram,home),1) o(go_to(ram,dentist),2)
at(0,V0) at(1,V1) at(2,V2) at(3,V3) at(4,V4)
Constraints: V0=0  V1=20  V2=35  V3=55  V4=55
```

To answer the question (b), we remove previous question and add the following rule.

```
% Problem (b). He should be at the dentist in 45 minutes
:- goal(S), at(0,T1), at(S,T2), gt(T2 - T1, 45).
```

There are no answer sets for this problem, hence there is no plan where Ram can be at the dentist 15 minutes before his appointment.

Language $\mathcal{AC}(\mathcal{R})$ allows literals in the head of mixed rules. This allows for writing more expressive rules than language $\mathcal{AC}_0$ and take advantage of different reasoning mechanisms to compute answer sets. The type of constraints used in the programs can be complex linear and non-linear equations. Functions abs, min, max, sin, cos, tan, exp can also be used along with constraint variables in mixed and defined rules. Further, we can use real numbers in mixed and defined rules. The following example illustrates some of these features.

Example 7.2.5. *We extend visiting dentist example 7.2.4 by asking following questions.*

- ▷ *(c). If Ram takes a cab for all his trips and the cab rate is $2.45 per minute. If the doctor's fees is $130 then how much money should Ram withdraw from the atm to pay all his expenses.*
- ▷ *(d). If Ram's expenses are more than the amount in his bank account then he needs to borrow. If Ram has $160 in his account then does he need to borrow money to cover his expenses?*

To express problem (c), we add the following rules to the program.

```
% (c). If the cab rate is $2.45 per minute and the doctor's fees is $130,
%      how much should Ram withdraw from the bank to pay his expenses.
doctor_expense(130).
#csort money(0..2000).
bank(atm).
% relation need_amount(bank,money)
#mixed need_amount(bank,money).
enough_money :- reached_goal(S), at(0,T1), at(S,T2), doctor_expense(X),
```

```
                total_expense(X,T1,T2,Y), need_amount(atm,Y).
:- not enough_money.
{: %start of defined part
total_expense(X,T1,T2,Y) <- Y = X + 2.45 * (T2 - T1).
:} % end of defined part
reached_goal(S) :- goal(S), not ngoal(S).
ngoal(S) :- step(S1), S1<S, goal(S1).
```

An answer set computed by $\mathcal{AC}$solver is as follows:

```
Answer set (1):
o(go_to(ram,atm),0) o(go_to(ram,home),1) o(go_to(ram,dentist),2)
enough_money at(0,V0) at(1,V1) at(2,V2) at(3,V3) at(4,V4)
need_amount(atm,V5)
Constraints: V0=0  V1=20  V2=35  V3=55  V4=55  V5=264.75
```

To express problem (d), we add the following rules to the program

```
% (d) If Ram has $160 in his account then does he need to
%     borrow money to cover his expenses?
amount(atm,160).
borrow :- amount(atm,X), need_amt(atm,Y), Y > X.
```

An answer set computed by $\mathcal{AC}$solver shows that he needs to borrow money to cover his expenses.

```
Answer set:
o(go_to(ram,atm),0) o(go_to(ram,home),1) o(go_to(ram,dentist),2)
enough_money borrow at(0,V0) at(1,V1) at(2,V2) at(3,V3) at(4,V4)
need_amount(atm,V5)
Constraints: V0=0  V1=20  V2=35  V3=55  V4=55  V5=264.75
```

Consider a problem which uses defaults, real numbers and functions on real numbers. ASP solvers can deal with defaults but not real numbers and functions on real numbers. Likewise, CLP solvers can deal with real numbers and functions on reals but not default statements. The following example shows the representation of these features combined and the computation of answer sets using *ACsolver*.

Example 7.2.6. We are given a database of students and their family relations including profession and death records of their parents. For example,

```
student(jane).  student(greg).   student(adam).

father(john, jane).   father(mike, greg).   father(jack, adam).
mother(lily, jane).   mother(lara, greg).   mother(sara, adam).

police(jack).  prof(mike).    police(john).
dead(jack).    dead(john).    dead(sara).
```

The database assumes that police officers who are dead, died in the line of duty. Exceptions are recorded by explicit statements,

```
-died_on_duty(john).
```

The second part of the database contains records with two fields: students name followed by a list of his or her grades in subjects. For example,

```
grades(greg, [42.5, 88.2, 56, 99, 70]).
grades(jane, [70.5, 75.5, 76.25, 67.3, 66]).
grades(adam, [95, 25, 89.9, 47, 92]).
```

Our goal is to use this database to define the following scholarship policy:
Normally scholarships are awarded to students who are orphans and children of police officers. There are exceptions to giving this scholarship. First, there should be money to award scholarship. Second, if a police officer did not die in the line of duty then his or her child may or may not get the scholarship. If there is no student satisfying this condition and there is money then the scholarship goes to the student who gets the highest average score in his or her subjects. If there is a tie, then the student who gets the lower standard deviation is preferred.
The following regular rules define the relations orphan and parent:

```
orphan(P) :- parents_dead(P).
parents_dead(P) :- father(F, P), mother(M, P), dead(F), dead(M).
-orphan(P) :- not orphan(P).


parent(X, P) :- father(X, P).
parent(X, P) :- mother(X, P).
```

Next we represent the default for the scholarship and the exceptions.

```
% scholarship is awarded to a student who is orphan and child
% of a police officer
award(S) :- student(S), orphan(S),
            parent(P,S), police(P),
            not ab(d(P)), not -award(S).


% a strong exception to awarding scholarship is lack of money.
-scholarship :- -money.
```

```
-award(S) :- -scholarship.

% a child of a police officer who did not die in line of duty
% may or may not receive scholarship
ab(d(P)) :- police(P), dead(P), -died_on_service(P).

% Since a police officer always intervenes even while off duty, we
% assume that the officer was on service unless stated otherwise.
died_on_service(P) :- police(P), dead(P), not -died_on_service(P).

% Otherwise, the best student gets the scholarship
award(S) :- student(S), best_student(S),
            not -best_award(S), not -scholarship.
-best_award(S1) :-  best_student(S1), S1 != S2, award(S2).
```

Notice that all rules defined so far are regular rules. The relation *best_student* is defined next.

```
% csort score takes values between 0 and 100
#csort score(0..100).
% mixed relation: average score of student
#mixed avg_score(student, score).
% mixed relation: standard deviation of student
#mixed std_dev(student, score).

% best student
best_student(S) :- not  -best_student(S).
-best_student(S1) :- S1 != S2, better_student(S2,S1).
```

```
% Student S1 is better than S2 if his average score is higher
better_student(S1, S2) :-  S1 != S2, avg_score(S1,A1),
                           avg_score(S2,A2), A1 > A2.

% In case the students have same average, a better student
% is determined by a lower standard deviation
better_student(S1, S2) :-  avg_score(S1,A1), avg_score(S2,A2),
                           std_dev(S1, D1), std_dev(S2, D2),
                           S1 != S2, A1 = A2, D1 < D2.

% The next two rules ensure that correct average is calculated
% for student S. Note that average is a defined predicate.
correct_avg(S) :- avg_score(S,A), average(S,A).
:- student(S), not correct_avg(S).

% The next two rules ensure that correct standard deviation
% is calculated for student S. Note sdeviation is a defined predicate.
correct_std(S) :- std_dev(S,D), sdeviation(S, D).
:- student(S), not correct_std(S).
```

Note that all rules which contain mixed atoms formed by mixed predicates avg_score and std_dev are middle rules. The rest are regular rules. Now, we define $average$ and $sdeviation$ by defined rules.

```
{: % start of defined part
% average of student X is A
average(X, A) <- grades(X, L), sum(L, S), count(L, N), A = (S / N).
```

```
% standard deviation of student S is D
sdeviation(S, D) <- grades(S, L), average(S, M),
                    variance(L, M, V), D = pow(V, 0.5).
% variance calculation
variance(L, M, V) <- count(L, N), sumsqd(L, M, Ds), V = Ds / (N-1).

% definition of sum of a list of numbers
sum([], 0).
sum([H |T], H + S) <- sum(T, S).

% count of number of items in a list
count([], 0).
count([_ | T], 1 + N) <- count(T, N).

% sum of square of the difference of items in a list to their mean
sumsqd([], _, 0).
sumsqd([H | T], M, pow(H-M,2) + S) <-  sumsqd(T, M, S).
:} % end of defined part
```

Now we can use *AC solver* to compute answer sets of the above program given the database information shown before. *AC solver* computed the following answer set.

```
Answer set (1): orphan(adam) award(adam) best_student(jane)
                avg_score(jane,V0) avg_score(greg,V1) avg_score(adam,V2)
                std_dev(jane,V3) std_dev(greg,V4) std_dev(adam,V5)
                Constraints: V0=71.11  V1=71.11  V2=69.78  V3=4.65543
                V4=23.0575  V5=31.8542
```

```
Duration: 0.110
Ground+Load Time: 0.031
Number of Rules: 141
Number of Atoms: 141
```

If we come to know that Jack died while he was not in the line of duty. The we add the following statement to the database and compute answer sets.

```
-died_on_service(jack).

Answer set (1): orphan(adam) award(jane) best_student(jane)
                avg_score(jane,V0) avg_score(greg,V1) avg_score(adam,V2)
                std_dev(jane,V3) std_dev(greg,V4) std_dev(adam,V5)
                Constraints: V0=71.11  V1=71.11  V2=69.78  V3=4.65543
                V4=23.0575  V5=31.8542
Duration: 0.111
Ground+Load Time: 0.031
Number of Rules: 142
Number of Atoms: 141
```

Note that Greg received the same highest score as Jane but Jane's standard deviation was lower, so she was awarded the scholarship.

This chapter describes methodologies for representing knowledge in languages $\mathcal{AC}_0$ and $\mathcal{AC}(\mathcal{R})$. Further, it also presents experimental results of an USA advisor planning and scheduling problem. The results show that $\mathcal{AC}_0$ can be used for complex planning and scheduling problems and $\mathcal{AD}solver$ can be used to compute answer sets in realistic time for these problems. We also presented

examples where *$\mathcal{AC}$ solver* could be used to compute answer sets where classical ASP and pure CLP solvers cannot be used to solve the problem.

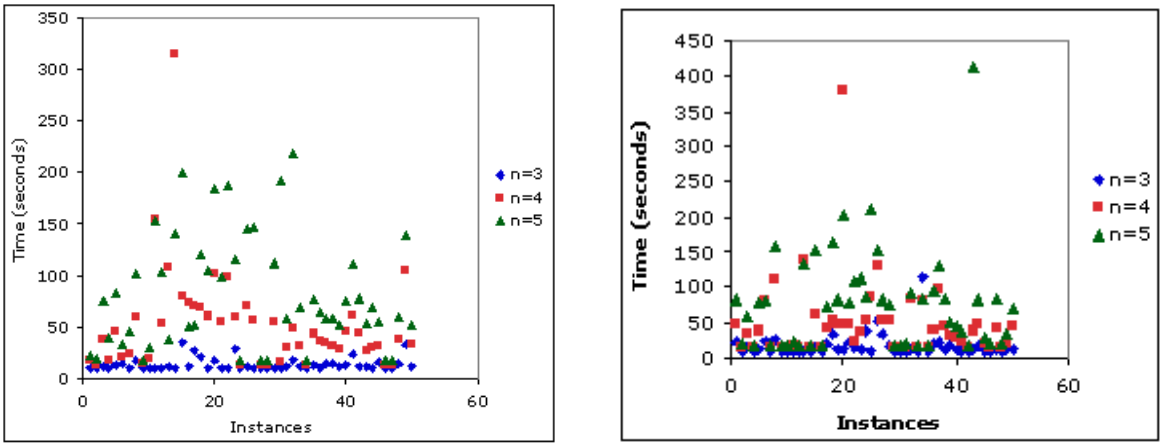

Figure 7.3: *ADsolver* Time Results (3) on Planning and Scheduling in USA-Advisor

CHAPTER 8

PROOFS

In this chapter we give proofs of several propositions we have seen so far. Next section proves that the partial grounding procedure we describe in chapter 4 is correct. Section 8.2 proves the propositions related to $expand$ function. Section 8.3 proves the propositions related to c_$solve$ function. Finally section 8.4 proves the correctness of $\mathcal{AC}(C)$_solver algorithm.

8.1 Partial Grounding

In this section we prove the correctness of $\mathcal{P}ground$ described in chapter 4. First we introduce some terminology. Let $m = p(\bar{t}_r, \bar{t}_c)$ be an r-ground m-atom and X be a candidate_mixed set. By construction of X, there exists exactly one m-atom m_i of the form $p(\bar{t}_r, \bar{t}'_c)$ (such that the regular terms in m and m_i are same). Let us denote the m-atom m_i as $image(m, X)$, read as *image of* m *in* X.

Example 8.1.1. *Let* $at(C_r, C_c)$ *be a mixed predicate where regular sort* C_r *ranges over* $\{a, b, c\}$ *and constraint sort* C_c *ranges over* $[1 \ldots 1000]$.
Set $X = \{at(a, 5),\ at(b, 3),\ at(c, 4)\ \}$ *is a candidate mixed set.*
Image of r-ground atom $p(a, T_1)$ *in* X *is* $image(p(a, T_1), X) = p(a, 5)$.
and $image(p(b, T_2), X) = p(b, 3)$.

Let c be a tc_constant and $m = p(\bar{t}_r, c_1, \ldots, c_n)$ be an r-ground m-atom, where $c_1, \cdots, c_n$ are tc_$constants$ with possibly some c_i's equal to c. Let X be a candidate_mixed set and $m_\nu = p(\bar{t}_r, \nu_1, \cdots, \nu_n)$ be the image of m in X. We define the *candidate values of* c *with respect to* m *and* X as $\delta(c, m, X) = \{\nu_i \mid c_i$ in m, such that $c_i = c\}$. Given a rule $r \in \Pi$, we define the *candidate values of* c *with respect to* r *and* X as $\delta(c, r, X) = \bigcup_{m \in m\_atoms(r)} \delta(c, m, X)$.

Let r be a rule and $c_1, \ldots, c_n$ be tc_constants occurring in r. A *value vector* of r with respect to X is defined as a vector $\langle v_1, \cdots, v_n \rangle$ where $v_i \in \delta(c_i, r, X)$.

Example 8.1.2. *Let* $at(C_r, C_c, C_c)$ *be a mixed predicate where* $C_r = \{a, b\}$ *and* $C_c = [1 \ldots 1000]$. *Let* $X = \{at(a, 3, 50), at(b, 5, 40)\}$ *be a candidate mixed set. Let* r *be a rule as shown below:*

$$p(a, b) \leftarrow at(a, c_1, c_1),\ at(b, c_1, c_2),\ d(a, b, c_1, c_2).$$

where c_1 *and* c_2 *are tc_constants.*

$\delta(c_1,\ at(a, c_1, c_1),\ X) = \{3, 50\}$. $\quad$ $\delta(c_2,\ at(a, c_1, c_1),\ X) = \{\ \}$.

$\delta(c_1,\ r,\ X) = \{3, 50, 5\}$. $\quad$ $\delta(c_2,\ r,\ X) = \{40\}$.

value vectors of r *with respect to* X *are* $\langle 3, 40\rangle$, $\langle 50, 40\rangle$, $\langle 5, 40\rangle$.

Definition 8.1.1. $[\gamma(\Pi, X)]$ *Let* Π *be a program and* X *be a candidate_mixed set. Let* $c_1, \cdots, c_n$ *be the tc_constants occurring in a rule* $r \in \Pi$ *and* $\bar{v} = \langle v_1, \cdots, v_n \rangle$ *be a value vector of* r *with respect to* X. $\gamma_0(r, X, \bar{v})$ *is defined as the rule obtained as follows: for every* $i = 1$ *to* n, *replace all occurrences of* c_i *in* r *by* v_i. *Further,* $\gamma(r, X) = \{\gamma_0(r, X, \bar{v}) \mid \bar{v}$ *is a value vector of* r *w.r.t.* $X\ \}$ *and* $\gamma(\Pi, X) = \{\gamma(r, X) \mid r \in \Pi\}$.

Example 8.1.3. *Let* $\Sigma = \{P_r = \{p(C_r, C_r), q(C_r, C_r)\},\ P_m = \{at(C_r, C_c)\},\ P_c = \{>\ \},\ C_r = \{a, b\},\ C_c = [1, \cdots, 100],\ tc\_cons = \{t_1, t_2\}\}$ *be the signature of program* Π *containing a rule*

r: $\quad p(a, b) \leftarrow q(a),\ q(b),\ at(a, t_1),\ at(b, t_2),\ t_1 > t_2.$

Let $X = \{\ at(a, 3),\ at(b, 6)\ \}$, *then* $\gamma(r, X)$ *is:*

$$p(a, b) \leftarrow q(a),\ q(b),\ at(a, 3),\ at(b, 6),\ 3 > 6.$$

The set $\gamma(r, X)$ can contain more than one rule. For instance, the following example shows $\gamma(r, X)$ contains three rules.

Example 8.1.4. *Let rule* r *and candidate set* X *be as in example 8.1.2. In example 8.1.2, we computed three value vectors of* r *with respect to* X*. The value vectors are used to compute* $\gamma(r, X)$*, which is as follows:*

$$p(a, b) \leftarrow at(a, 3, 3),\ at(b, 3, 40),\ d(a, b, 3, 40).$$
$$p(a, b) \leftarrow at(a, 50, 50),\ at(b, 50, 40),\ d(a, b, 50, 40).$$
$$p(a, b) \leftarrow at(a, 5, 5),\ at(b, 5, 40),\ d(a, b, 5, 40).$$

We recall definition of $\beta(r, C, \Sigma)$ from chapter 4.

Definition 8.1.2. $[\beta(r, C, \Sigma)]$ *Let* C *be a set of c-literals,* r *be a rule and* Σ *be an arbitrary signature. We define* $\beta(r, C, \Sigma)$ *as a rule* r'*, where*

- $head(r') = head(r)$
- *If* r *consists only of predicates from* Σ *then* $body(r') = body(r) \cup c\_lits(C, r)$*, where* $c\_lits(C, r)$ *is the set of all c_lits from* C *whose tc_constants are exactly those in* r
- *If* r *contains some predicates not in* Σ *then* $body(r') = body(r)$

Given a program Π, $\beta(\Pi, C, \Sigma) = \{\ \beta(r, C, \Sigma) \mid r \in \Pi\ \}$

Example 8.1.5. *Let* Σ *be signature from example 8.1.3 and* $C = \{t_1 > t_2\}$ *and* Π' *be a program with the following rules:*

r_1*:* $q(a).$

r_2*:* $p(a, b) \leftarrow q(a),\ q(b),\ at(a, t_1),\ at(b, t_2).$

r_3*:* $n\_at(b, t_1) \leftarrow not\ at(b, t_1).$

$\beta(\Pi', C, \Sigma)$ *is the following set of rules:*

$\beta(r_1, C, \Sigma)$*:* $q(a).$

$\beta(r_2, C, \Sigma)$*:* $p(a, b) \leftarrow q(a),\ q(b),\ at(a, t_1),\ at(b, t_2),\ t_1 > t_2.$

$\beta(r_3, C, \Sigma)$*:* $n\_at(b, t_1) \leftarrow not\ at(b, t_1).$

8.1.0.4 Splitting Sets

Splitting set theorem is used several times for the proofs. We briefly recall some definitions from [67]. Readers may skip this section and go directly to the proofs. A *splitting set* of a program P is any set U of literals such that, for every rule $r \in P$, if $head(r) \cap U \neq \emptyset$ then $lit(r) \subset U$. The set of rules $r \in P$ such that $lit(r) \subset U$ is called the bottom of P relative to U denoted by $b_U(P)$. The set $P \setminus b_U(P)$ is the top of P relative to U. Consider two sets of literals U, X and a program P. For each rule $r \in P$ such that $pos(r) \cap U$ is a part of X and $neg(r) \cap X$ is disjoint from X, take the rule r' defined by

$$head(r') = head(r), \quad pos(r') = pos(r) \setminus U, \quad neg(r') = neg(r) \setminus U.$$

The program consisting of all the rules r' obtained from Π will be denoted by $eval_U(\Pi, X)$. Let U be a splitting set of a program P. A solution to P (with respect to U) is a pair $\langle X, Y \rangle$ of set of literals such that

- X is an answer set of $b_U(P)$,
- Y is an answer set of $eval_U(P \setminus b_U(P), X)$,
- $X \cup Y$ is consistent.

Splitting Set Theorem: Let U be a splitting set for a program P. A set A of literals is a consistent answer set for P if and only if $A = X \cup Y$ for some solution $\langle X, Y \rangle$ to P with respect to U.

8.1.1 $\mathcal{P}(\Pi)$

Note that, in the proofs, we denote the ground instantiation of a program Π, $ground(\Pi)$, as $gr(\Pi)$. Throughout this section, we use X for a candidate mixed set and M_c for the intended interpretation of c-atoms of a program.

Proposition: *Given a program* Π *of* $\mathcal{AC}(\mathcal{C})$ *satisfying the syntax restrictions,* S *is an answer set of* Π *iff* S *is an answer set of* $\mathcal{P}(\Pi)$.

Proof: First we give proof when there are no defined literals in Π. Therefore, $\Pi_D = \emptyset$. We look at each step (N) in the procedure $\mathcal{P}(\Pi)$ and present some properties of the program resulting from step N. These properties are proved as lemmas later in the section. Let Π be an $\mathcal{AC}(\mathcal{C})$ program without defined literals.

(a). By definition of $\mathcal{AC}(\mathcal{C})$ answer set, S is an answer set of Π, iff S is an αsp answer set of $gr(\Pi) \cup X \cup M_c$, where M_c is the intended interpretation of c-atoms of Π and X is some candidate mixed set.

(b). By (a) and definition of gr, S is an answer set of Π iff S is an αsp answer set of $\Pi \cup X \cup M_c$, where X is some candidate mixed set. By definition of answer set, $X \subseteq S$.

Next, we use the steps in $\mathcal{P}(\Pi)$ to prove that S is an asp answer set of $\Pi \cup X \cup M_c$ iff S is an αsp answer set of $\mathcal{P}(\Pi) \cup X \cup M_c$. The steps in $\mathcal{P}(\Pi)$ and their properties are as follows:

1. Step (1) of $\mathcal{P}(\Pi)$, replaces all c-variables in Π_M by tc_constants. Since $\Pi_D = \emptyset$, we get $\Pi_1 = tc(\Sigma_\Pi, \Pi_M) \cup \Pi_R$. The following property holds after step (1): S *is an answer set of* $\Pi \cup X \cup M_c$ *iff* S *is an answer set of* $\gamma(gr(\Pi_1), X) \cup X \cup M_c$.

2. Step (2) removes all c-literals from Π_1. Let us denote the resulting program as Π_2. Let the c-literals be stored in C. The program Π_2 contains only r-literals and m-literals. Let Σ_2 be the signature of Π_2. The following property holds after step (2): S *is an answer set of* $\gamma(gr(\Pi_1), X) \cup X \cup M_c$ *iff* S *is an answer set of* $\gamma(\beta(gr(\Pi_2), C, \Sigma_2), X) \cup X \cup M_c$.

3. Step (3) adds rules to Π_2 with m-atoms in the head. The resulting program $\Pi_3 = \Pi_2 \cup addr(\Pi_2)$ is not an $\mathcal{AC}(\mathcal{C})$ program but an ASP program. Since new predicates (np) are added by $addr(\Pi_2)$, the signature Σ_3 of Π_3 is different from Σ_2. Let
$\tilde{X} = \{\ np(\bar{t}_r, \bar{t}_c) \mid p(\bar{t}_r, \bar{t}_c) \notin X \text{ and } p(\bar{t}_r, \bar{t}_c) \in m\_atoms(\Pi)\}$, where $\bar{t}_r$ and $\bar{t}_c$ are ground. The following property holds after step (3): *S is an answer set of $\gamma(\beta(gr(\Pi_2), C, \Sigma_2), X) \cup X \cup M_c$ iff $S \cup \tilde{X}$ is an answer set of $\gamma(\beta(gr(\Pi_3), C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$.*

4. Step (4) grounds the regular and middle part of program Π_3. The resulting program $\Pi_4 = lparse(\Pi_{3R} \cup \Pi_{3M} \cup addr(\Pi_2))$. The following property holds after step (4): *S is an answer set of $\gamma(\beta(gr(\Pi_3), C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$ iff S is an answer set of $\gamma(\beta(\Pi_4, C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$.*

5. Step (5) removes the ground instantiations of $addr(\Pi_2)$. The resulting program $\Pi_5 = \Pi_4 \setminus ground(addr(\Pi_2))$. The following property holds after step (5): *$S \cup \tilde{X}$ is an answer set of $\gamma(\beta(\Pi_4, C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$ iff S is an answer set of $\gamma(\beta(\Pi_5, C, \Sigma_2), X) \cup X \cup M_c$.*

6. Step (6) consists of two sub-steps:

   (a) Step (6a) adds the c-literals stored in C at step (2) to Π_5 to obtain $\Pi_{6a} = \beta(\Pi_5, C, \Sigma_2)$. The following property holds after step (6a): *S is an answer set of $\gamma(\beta(\Pi_5, C, \Sigma_2), X) \cup X \cup M_c$ iff S is an answer set of $\gamma(\Pi_{6a}, X) \cup X \cup M_c$.*

   (b) Step (6b) replaces the tc_constants by c-variables to get Π_6. The following property holds after step (6b): *S is an answer set of $\gamma(\Pi_{6a}, X) \cup X \cup M_c$ iff S is an answer set of $\Pi_6 \cup X \cup M_c$.*

7. Step (7) renames c-variables to get $\Pi_7 = rename(\Pi_{6_M}, Y) \cup \Pi_{6_R}$, where Y is a r_ground_mixed set. The following property holds after step (7): *S is an answer set of* $\Pi_6 \cup X \cup M_c$ *iff S is an answer set of* $\Pi_7 \cup X \cup M_c$.

(c). Since $\Pi_7 = \mathcal{P}(\Pi)$, by (1) to (7), we get that S is an answer set of $\Pi \cup X \cup M_c$ iff S is an answer set of $\mathcal{P}(\Pi) \cup X \cup M_c$.

(d). By definition of answer sets of $\mathcal{AC}(\mathcal{C})$ programs, we get that S is an answer set of Π iff S is an answer set of $\mathcal{P}(\Pi)$.

■

8.1.2 Step (1) of $\mathcal{P}(\Pi)$

We now prove the properties used at each step in the previous proof. At step (1) of $\mathcal{P}(\Pi)$, program Π_1 is obtained by replacing c-variables in Π_M by tc_constants. Observe that $\Pi_1 = \Pi_R \cup tc(\Sigma_\Pi, \Pi_M)$. The regular part and middle part of Π_1 are denoted by Π_{1R} and Π_{1M} respectively. Before proving lemma 8.1.2, we prove an auxiliary lemma 8.1.1. First we introduce some terminology.
Let $r_g \in gr(r)$ be a rule obtained from r by substituting $V_1 = x_1, \cdots, V_n = x_n$ of variables in r. Let us denote $\zeta_0(r, r_g, V_i)$ to be equal to value x_i and $\zeta(r, r_g, \langle V_1, \cdots, V_n \rangle) = \langle x_1, \cdots, x_n \rangle$ such that $\zeta_0(r, r_g, V_i) = x_i$.

Lemma 8.1.1. *Let* Π *be an* $\mathcal{AC}(\mathcal{C})$ *program,* X *be a candidate_mixed set, and* $U = m\_atoms(\Pi)$. *We have* $eval_U(gr(\Pi_M), X) = eval_U(\gamma(gr(tc(\Sigma_\Pi, \Pi_M)), X), X)$.

Proof:

1. $\Longrightarrow$ Let $r_e = l \leftarrow B, E \in eval_U(gr(\Pi_M), X)$, such that $B = r\_lits(r_e)$ and $E = c\_lits(r_e)$. Note that by definition of r_e, m-literals do not belong to the body of r_e.

We show that $r_e \in eval_U(\gamma(gr(tc(\Sigma_\Pi, \Pi_M)), X), X)$.

2. By (1) and definition of $eval$, $\exists\ r_a \in gr(\Pi_M)$ such that $r_e = eval(r_a, X)$. By definition of $eval$, $m\text{-}lits(r_a) \subseteq X$.

3. Now we show that $r_a \in \gamma(gr(tc(\Sigma_\Pi, \Pi_M)), X)$.

4. For this, first we construct a rule $r \in gr(tc(\Sigma_\Pi, \Pi_M))$ and then show that $r_a \in \gamma(r, X)$.

   (a) By (2) and definition of gr, $\exists r_u \in \Pi_m$ such that $r_a \in gr(r_u)$; Let r_a be obtained from r_u by some substitution θ substituting variables $V_1, \cdots, V_n$ by constants $x_1, \cdots, x_n$ respectively. Let $V_1, \cdots, V_k$ be c-variables and $V_{k+1}, \cdots, V_n$ be r-variables.

   (b) By definition of tc, there exists a rule r_t such that $r_t = tc(\Sigma_\Pi, r_u)$; Let r_t be obtained from r_u by substituting c-variables $V_1, \cdots, V_k$ of r_u by some tc_constants $c_1, \cdots, c_k$ respectively. Now r_t contains only r-variables $V_{k+1}, \cdots, V_n$.

   (c) By definition of gr, $\exists\ r \in gr(r_t)$, such that r is obtained from r_t by substitution θ substituting variables $V_{k+1}, \ldots, V_n$ by constants $x_{k+1}, \ldots, x_n$.

5. Now, we show that $r_a \in \gamma(r, X)$.

   (a) By construction of r and r_a, we get $r_a \in \gamma(r, X)$ iff the tc_constants $c_1, \ldots, c_k$ of r are substituted by constants $x_1, \ldots x_k$.

   (b) Hence by definition of γ, it is enough to show that $\bar{v} = \langle x_1, \cdots, x_k \rangle$ is a value vector for r with respect to X.

   (c) $\bar{v}$ is a value vector of r with respect to X iff $x_i \in \delta(c_i, r, X)$ for $i = 1$ to k

(d) Consider an arbitrary tc_constant c_i from r. We show that $x_i \in \delta(c_i, r, X)$.

(e) By syntax restriction (2) on Π(see section 4.1), c_i occurs in some m-atom $m \in r$.

(f) Let m_ν be the image of m in X. By (2) and construction of X, $m_\nu \in m\text{-}lits(r_a)$.

(g) By construction of r_a, we have $\zeta_0(r_u, r_a, V_i) = x_i$. Hence $\delta(c_i, m, X) = x_i$

(h) Therefore, we get $x_i \in \delta(c_i, r, X)$ for all $i = 1$ to k

(i) Hence $\bar{\nu}$ is a value vector of r with respect to X

(j) We get $\gamma_0(r, X, \bar{\nu}) = r_a$ and hence $r_a \in \gamma(r, X)$.

6. Since $r \in gr(tc(\Sigma_\Pi, \Pi_M), X)$, we get $r_a \in \gamma(gr(tc(\Sigma_\Pi, \Pi_M)), X)$. By definition of $eval$ and (2), $r_e \in eval_U(\gamma(gr(tc(\Pi_M)), X), X)$

7. The proof from the other side is straightforward as $\gamma(gr(tc(\Pi_M)), X) \subseteq gr(\Pi_M)$.

■

Lemma 8.1.2. *S is an answer set of* $\Pi \cup X \cup M_c$ *iff S is an answer set of* $\gamma(gr(\Pi_1), X) \cup X \cup M_c$.

Proof:

1. S is an answer set of $\Pi \cup X \cup M_c$ iff S is an answer set of $gr(\Pi) \cup X \cup M_c$ [By definition of answer set]

2. (1) is true iff S is an answer set of $gr(\Pi_R) \cup gr(\Pi_M) \cup X \cup M_c$ [Since $\Pi = \Pi_R \cup \Pi_M$ and $gr(\Pi) = gr(\Pi_R) \cup gr(\Pi_M)$

3. (2) is true iff S is an answer set of $eval_U(gr(\Pi_R) \cup gr(\Pi_M) \cup M_c, X) \cup X$
   [as $U = m\text{-}atoms(\Pi)$ splits $gr(\Pi) \cup X \cup M_c$ and $top_U = gr(\Pi) \cup M_c$]

4. (3) is true iff S is an answer set of $gr(\Pi_R) \cup eval_U(gr(\Pi_M), X) \cup X \cup M_c$
   [Since m-atoms(Π) do not appear in rules $\Pi_R \cup M_c$, we get
   $eval_U(gr(\Pi_R) \cup M_c, X) = gr(\Pi_R) \cup M_c$]

5. (4) is true iff S is an answer set of
   $gr(\Pi_R) \cup eval_U(\gamma(gr(\Pi_{1M}), X), X) \cup X \cup M_c$
   [By lemma (8.1.1), $eval_U(gr(\Pi_M), X) = eval_U(\gamma(gr(tc(\Sigma_\Pi, \Pi_M)), X), X)$ and
   by definition $\Pi_{1M} = tc(\Sigma_\Pi, \Pi_M)$]

6. (5) is true iff S is an answer set of
   $eval_U(gr(\Pi_R) \cup \gamma(gr(\Pi_{1M}) \cup M_c, X), X) \cup X$
   [By definition of $eval$ and fact that $gr(\Pi_R)$ does not contain $m\text{-}atoms(\Pi)$]

7. (6) is true iff S is an answer set of $eval_U(\gamma(gr(\Pi_R \cup \Pi_{1M} \cup M_c), X), X) \cup X$
   [By definition of γ and gr]

8. (7) is true iff S is an answer set of $\gamma(gr(\Pi_R \cup \Pi_{1M}), X) \cup X \cup M_c$
   [by splitting set theorem]

9. (8) is true iff S is an answer set of $\gamma(gr(\Pi_1), X) \cup X \cup M_c$
   [Since $\Pi_1 = \Pi_R \cup \Pi_{1M}$]

■

8.1.3 Step (2) of $\mathcal{P}(\Pi)$

The program Π_2 is obtained by removing c-literals from Π_1. The c-literals are stored in C and the signature of Π_2 is Σ_2.

Lemma 8.1.3. *S is an answer set of $\gamma(gr(\Pi_1), X) \cup X \cup M_c$ iff S is an answer set of $\gamma(\beta(gr(\Pi_2), C, \Sigma_2), X) \cup X \cup M_c$.*

Proof:

1. To prove the lemma, we show that $gr(\Pi_1) = \beta(gr(\Pi_2), C, \Sigma_2)$
2. $\Longrightarrow$ Let $r \in gr(\Pi_1)$. We show that $r \in \beta(gr(\Pi_2), C, \Sigma_2)$.
3. By (2), $\exists\ r_1 \in \Pi_1$ with variables $V_1, \cdots V_n$, such that $r \in gr(r_1)$ and r is obtained from r_1 by some substitution θ substituting $V_1 = x_1, \cdots, V_n = x_n$.
4. By (3) and definition of Π_2, $\exists\ r_2 \in \Pi_2$, such that r_2 is obtained by removing c_lits from r_1. Hence, $head(r_2) = head(r_1)$ and $body(r_2) = body(r_1) \setminus c\_lits(r_1)$.
5. By (4) and definition of gr, $\exists\ r_g \in gr(r_2)$, such that r_g is obtained from r_2 by same substitution θ (from(3)) substituting $V_1 = x_1, \cdots, V_n = x_n$.
6. By (3) and (5), $head(r_g) = head(r)$ and $body(r_g) = body(r) \setminus c\_lits(r)$.
7. Let r contain tc_constants $c_1, \cdots, c_n$. By construction of tc, $c_1, \cdots, c_n$ occur in C only in c_literals from $c\_lits(r_1)$.
8. By construction of r_g (see (4) and (5)), $c_1, \cdots, c_n$ are the tc_constants in r_g.
9. By definition of β, $\beta(r_g, C, \Sigma_2) = r'$, such that $head(r') = head(r_g)$ and $body(r') = body(r_g) \cup c\_lits(C, r_g)$.
10. By (7) and (8), $c\_lits(C, r_g) = c\_lits(r_1) = c\_lits(r)$
11. By (9) and (10), $r' = r$ and therefore $r \in \beta(gr(\Pi_2), C, \Sigma_2)$
12. $\Longleftarrow$ proof is similar

13. Hence, $gr(\Pi_1) = \beta(gr(\Pi_2), C, \Sigma_2)$

14. Therefore, $\gamma(gr(\Pi_1), X) \cup X \cup M_c = \gamma(\beta(gr(\Pi_2), C, \Sigma_2), X) \cup X \cup M_c$

15. Therefore lemma 8.1.3 is true.

■

8.1.4 Step (3) of $\mathcal{P}(\Pi)$

At step (3) of $\mathcal{P}(\Pi)$, we compute the ASP program $\Pi_3 = \Pi_2 \cup addr(\Pi_2)$. The signature Σ_3 of Π_3 is different from Σ_2 and contains extra predicates np (added by $addr(\Pi_2)$). Let $\tilde{X} = \{\ np(\bar{t}_r, \bar{t}_c) \mid p(\bar{t}_r, \bar{t}_c) \notin X \text{ and } p(\bar{t}_r, \bar{t}_c) \in \textit{m-atoms}(\Pi)\}$, where $\bar{t}_r$ and $\bar{t}_c$ are ground.

Lemma 8.1.4. *S is an answer set of $\gamma(\beta(gr(\Pi_2), C, \Sigma_2), X) \cup X \cup M_c$ iff $S \cup \tilde{X}$ is an answer set of $\gamma(\beta(gr(\Pi_3), C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$*

Proof:

1. $S \cup \tilde{X}$ is an answer set of $\gamma(\beta(gr(\Pi_3), C, \Sigma_2), X) \ \cup\ X \ \cup\ \tilde{X} \cup M_c$ iff $S \cup \tilde{X}$ is an answer set of
   $\gamma(\beta(gr(\Pi_2), C, \Sigma_2) \ \cup\ \beta(gr(addr(\Pi_2)), C, \Sigma_2), X) \ \cup\ X \ \cup\ \tilde{X} \cup M_c$.
   [as $\Pi_3 = \Pi_2 \cup addr(\Pi_2)$]

2. (1) is true iff $S \cup \tilde{X}$ is an answer set of
   $\gamma(\beta(gr(\Pi_2), C, \Sigma_2) \ \cup\ gr(addr(\Pi_2)), X) \ \cup\ X \ \cup\ \tilde{X} \cup M_c$;
   [as predicate np occurs in every rule of $gr(addr(\Pi_2))$ and np is not in Σ_2, we get $\beta(gr(addr(\Pi_2)), C, \Sigma_2) = gr(addr(\Pi_2))$]

3. (2) is true iff $S \cup \tilde{X}$ is an answer set of
   $\gamma(\beta(gr(\Pi_2), C, \Sigma_2), X) \ \cup\ \gamma(gr(addr(\Pi_2)), X) \ \cup\ X \ \cup\ \tilde{X} \cup M_c$;
   [by definition of γ]

4. (3) is true iff $S \cup \tilde{X}$ is an answer set of
   $eval_U(\gamma(\beta(gr(\Pi_2), C, \Sigma_2), X) \cup M_c,\ X \cup \tilde{X})\ \cup\ (X \cup \tilde{X})$;
   [as $U = m\_atoms(\Pi) \cup \{np(\bar{t}_r, \bar{t}_c) \mid p(\bar{t}_r, \bar{t}_c) \in m\_atoms(\Pi)\}$ splits Π_3 with
   $bot_U = \gamma(gr(addr(\Pi_2)), X)\ \cup X \cup \tilde{X}$ which has unique answer set $X \cup \tilde{X}$]

5. (4) is true iff $S \cup \tilde{X}$ is an answer set of $\gamma(\beta(gr(\Pi_2), C, \Sigma_2), X) \cup M_c \cup\ (X \cup \tilde{X})$;
   [by splitting set theorem]

6. (5) is true iff S is an answer set of $\gamma(\beta(gr(\Pi_2), C, \Sigma_2), X)\ \cup\ X \cup M_c$;
   [as $U = \tilde{X}$ splits with $bot_U = \tilde{X}$ and $top_U = \gamma(\beta(gr(\Pi_2), C, \Sigma_2), X)\ \cup\ X \cup M_c$
   ]

■

8.1.5 Step (4) of $\mathcal{P}(\Pi)$

After step (4), the program $\Pi_4 = lparse(\Pi_{3R} \cup \Pi_{3M} \cup addr(\Pi_2)) \cup \Pi_{3C}$. Since $\Pi_{3C} = \emptyset$, the program $\Pi_4 = lparse(\Pi_3)$. Note that Π_3 and Π_4 are ASP programs. We need some terminology. Let us call the set of all predicates in $\Sigma_3 \setminus \Sigma_2$, n_mixed predicates. Let us denote the atoms formed by n_mixed predicates in program Π_3 by $n\_matoms(\Pi_3)$.

Recall from [99], the notion of dependency graph for an ASP program. The dependency graph $G_\Pi = (V_\Pi, E_\Pi)$ of program Π is a directed graph constructed as follows:

- ▷ $V_\Pi = \{p \mid p$ is a predicate symbol in $\Pi\}$.
- ▷ $(p, q) \in E_\Pi$ iff there exists a rule $r \in \Pi$ where p is a predicate symbol in head and q is a predicate symbol in body.

A predicate p depends on a predicate q iff there exists a path from p to q in the dependency graph. A predicate p is a *domain predicate* iff it holds that every

path starting from nodes $p \in V_\Pi$ is negative cycle free. Atoms formed by domain predicates are referred to as domain atoms. Let us denote the set of all ground domain atoms in a program Π by $Dom(\Pi)$. Given a ground rule r, let $dom(r) = body(r) \cap Dom(\Pi)$.

Given a ASP program Π, $U = Dom(\Pi)$ splits the program into domain part Π^d and non-domain part Π^n. Π^d consists of all rules r such that $head(r) \in Dom(\Pi)$. The rest is non-domain part. Since Π^d is stratified [1], it has a unique answer set A [50]. We base our proof on the following proposition about $lparse$.

Proposition 8.1.1. *Let* $\Pi_g = gr(\Pi)$ *and* $\Pi_l = lparse(\Pi)$. *We can write* $\Pi_l = A \cup R$, *where* A *is a set of atoms and* R *is a set of rules such that,*

(l_1) A *is the unique answer set of the domain part* Π^d.

(l_2) *If* $r_l \in R$ *then* $\exists\ r_g \in \Pi_g$ *such that* r_l *is obtained from* r_g *by removing literals in* $body(r_g)$ *that belong to* A.

(l_3) S *is an answer set of* Π_g *iff* S *is an answer set of* Π_l

The proof of lemma 8.1.7 depends on the following statements (which can be easily proven):

(i_1) $mixed$ and n_mixed predicates in Π_3 and Π_4 are non-domain predicates.

(i_2) The set of predicates that depend on $mixed$ predicates are non-domain predicates.

(i_3) $U = Dom(\Pi)$ is a splitting set of Π_i with $bot_U(\Pi_i) \subseteq \Pi_{iR}$, for $i = 3, 4$.

We prove two auxiliary lemmas 8.1.5, 8.1.6 before lemma 8.1.7.

Lemma 8.1.5. *Let program* $T_g = \gamma(\beta(gr(\Pi_3), C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$ *and program*

$T_l = \gamma(\beta(lparse(\Pi_3), C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$.

1. $U = Dom(\Pi_3)$ *is a splitting set of both* T_l *and* T_g.

2. $bot_U(T_l)$ *and* $bot_U(T_g)$ *have same unique answer set* A.

Proof:

1. Let $\Pi_g = gr(\Pi_3)$, $\Pi_l = lparse(\Pi_3)$ and $U = Dom(\Pi_3)$.

2. We show that U splits T_g. Proof that U splits T_l is similar.

3. U is a splitting set of Π_g.
   [By (i_3) and definition of gr]

4. (3) is true iff $\forall r \in \Pi_g$, if $head(r) \in U$ then $body(r) \in U$.
   [By definition of splitting set]

5. (4) is true iff $\forall r \in \Pi_{gR}$, if $head(r) \in U$ then $body(r) \in U$.
   [By (i_3) and definition of gr]

6. (5) is true iff $\forall r \in T_{gR}$, if $head(r) \in U$ then $body(r) \in U$.
   [By definition of γ and β, $T_g =$
   $\gamma(\beta(\Pi_g, C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c = \Pi_{gR} \cup \gamma(\beta(\Pi_g \setminus \Pi_{gR}, C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$]

7. (6) is true iff $\forall r \in T_g$, if $head(r) \in U$ then $body(r) \in U$.
   [By (i_1), (i_2) and construction of T_g, if $r \in T_g \setminus T_{gR}$, $head(r) \notin U$]

8. (7) is true iff U splits T_g.

9. By definition of domain atoms, $bot_U(\Pi_{gR})$ is stratified. Therefore, $bot_U(\Pi_{gR})$ has a unique answer set A.

10. It is easy to see that $bot_U(T_g) = bot_U(\Pi_{gR})$. Therefore $bot_U(T_g)$ has a unique answer set A.

■

Lemma 8.1.6. *Let program* $T_g = \gamma(\beta(gr(\Pi_3), C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$ *and program*
$T_l = \gamma(\beta(lparse(\Pi_3), C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$. *We can write* $T_l = A \cup R$ *where* A *is a set of r-atoms such that*

1. A *is the unique answer set of* $bot_U(T_l)$, *where* $U = Dom(T_l)$.
2. *If* $r_l \in R$ *then* $\exists\ r_g \in T_g$, *such that* r_l *is obtained from* r_g *by removing literals in* A *from* $body(r_l)$.

Proof:

1. From lemma 8.1.5, we can see that A is a unique answer set of $bot_U(T_l)$ as $Dom(T_l) = Dom(\Pi_3)$.
2. Now we prove the second part of the lemma.
3. Let $r_l \in R$.
4. We construct $r_g \in T_g$ such that r_l is obtained from r_g by removing literals from $body(r_l)$ that belong to A.
5. To do that, we first construct $r_0 \in lparse(\Pi_3)$ such that $r_l \in \gamma(\beta(r_0, C, \Sigma_2), X)$.
6. Next we construct $r_u \in gr(\Pi_3)$ such that $r_g \in \gamma(\beta(r_u, C, \Sigma_2), X)$.
7. We construct r_0 as follows: by (3), $r_l \in T_l$ and therefore $\exists\ r_0 \in lparse(\Pi_3)$ such that $r_l \in \gamma(\beta(r_0, C, \Sigma_2), X)$.
8. Let $c_1, \cdots, c_n$ be the tc_constants in r_0. This means that r_l was obtained from r_0 by

(a) adding $c\_lits(C, r_0)$ to body of r_0,

(b) replacing $c_1, \cdots, c_n$ by some value vector say $\langle v_1, \cdots, v_n \rangle$.

9. We construct r_u as follows: since $r_0 \in lparse(\Pi_3)$, by proposition (8.1.1), we know that $\exists\ r_u \in gr(\Pi_3)$ such that r_0 is obtained from r_u by removing literals in A from $body(r_u)$.

10. We construct r_g as follows:

(a) adding to $c\_lits(C, r_0)$ to body of r_u;

(b) replacing $c_1, \cdots, c_n$ by value vector $\langle v_1, \cdots, v_n \rangle$.

11. Now we show that $r_g \in \gamma(\beta(r_u, C, \Sigma_2), X)$.

(a) Let us refer rules obtained at steps (8a) and (10a) as r_8 and r_{10} respectively.

(b) First we show that $r_{10} = \beta(r_u, C, \Sigma_2)$.

(c) Notice that by construction of r_u, the tc_constants in r_u are exactly the same as tc_constants in r_0. Therefore, $c_1, \cdots, c_n$ are the only tc_constants in r_u.

(d) Therefore, $c\_lits(C, r_0) = c\_lits(C, r_u)$ and hence $r_{10} = \beta(r_u, C, \Sigma_2)$.

(e) Now to show that $r_g \in \gamma(\beta(r_u, C, \Sigma_2), X)$, it suffices to show that $\langle v_1, \cdots, v_n \rangle$ is a value vector of r_{10} with respect to X.

(f) In step (8b), we see that $\langle v_1, \cdots, v_n \rangle$ is a value vector of r_8 with respect X. By construction, we know that $m\_atoms(r_0) = m\_atoms(r_u) = m\_atoms(r_{10})$ and also that $m\_atoms(r_0) = m\_atoms(r_8)$. We get, $m\_atoms(r_{10}) = m\_atoms(r_8)$ and by definition of γ, $\langle v_1, \cdots, v_n \rangle$ is a value vector of r_{10}.

12. Now we show that r_l can be obtained from r_g by removing literals in A from body of r_g.

    (a) By construction, $head(r_l) = head(r_0) = head(r_u) = head(r_g)$

    (b) By construction, $r\_atoms(r_l) = r\_atoms(r_0)$;
    $r\_atoms(r_u) = r\_atoms(r_g)$ and $r\_atoms(r_0) = r\_atoms(r_u) \setminus A$.
    Hence, $r\_atoms(r_l) = r\_atoms(r_g) \setminus A$.

    (c) By construction, $m\_atoms(r_8) = m\_atoms(r_{10})$, and
    $c\_atoms(r_8) = c\_atoms(r_{10})$. Rules r_l and r_g are obtained from r_8 and r_{10} respectively by replacing the tc_constants by the same value vector.
    Hence, $m\_atoms(r_l) = m\_atoms(r_g)$ and $c\_atoms(r_l) = c\_atoms(r_g)$.

    (d) Therefore, $head(r_l) = head(r_g)$ and $body(r_l) = body(r_g) \setminus A$.

    (e) r_l can be obtained from r_g by removing literals in A from $body(r_g)$.

■

Lemma 8.1.7. *S is an answer set of $\gamma(\beta(gr(\Pi_3), C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$ iff S is an answer set of $\gamma(\beta(\Pi_4, C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$*

Proof:

1. Let $\Pi_g = gr(\Pi_3)$ and $\Pi_l = lparse(\Pi_3)$. Let
   $T_g = \gamma(\beta(\Pi_g, C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$ and $T_l = \gamma(\beta(\Pi_l, C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$.

2. By lemma 8.1.5, U is a splitting set of T_g and T_l; and $bot_U(T_l)$ and $bot_U(T_g)$ have same unique answer set A.

3. Let $T_g^t = top_U(T_g) = \Pi_g \setminus bot_U(T_g)$ and $T_l^t = top_U(T_l) = \Pi_l \setminus bot_U(T_l)$.

4. By splitting set theorem, Lemma 8.1.7 is true iff (5) is true

5. $S \setminus A$ is an answer set of $eval_U(T_g^t, A)$ iff $S \setminus A$ is an answer set of $eval_U(T_l^t, A)$.

6. We show (5) is true by showing that $eval_U(T_g^t, A) = eval_U(T_l^t, A)$.

7. $\Longrightarrow$

8. Let $r_e \in eval_U(T_g^t, A)$.

9. We construct $r' \in T_l^t$ such that $r_e = eval_U(r', A)$.

10. To do that we first construct $r_0 \in \Pi_3$ and $r_g \in \gamma(\beta(gr(r_0), C, \Sigma_2), X)$ such that $r_e = eval_U(r_g, A)$.

11. Next, we construct $r_l \in lparse(r_0)$ and $r' \in \gamma(\beta(r_l, C, \Sigma_2), X)$ such that (9) is satisfied.

12. We construct r_0 and r_g as follows:

    (a) By (8), $\exists r_g$ such that $r_e = eval(r_g, A)$ and $r_g \in T_g^t$.

    (b) Since $T_g^t \subseteq T_g$, $r_g \in T_g$ and r_g is obtained from some rule $r_0 \in \Pi_3$, such that $r_g \in \gamma(\beta(gr(r_0), C, \Sigma_2), X)$

13. Let $V_1, \cdots, V_n$ be the r-variables and $c_1, \cdots, c_k$ be the tc_constants that occur in r_0. This means that r_g is obtained from r_0 by

    (a) replacing $V_1, \cdots, V_n$ by some constants say $x_1, \cdots, x_n$

    (b) adding $c\_lits(C, r_0)$ to body of rule obtained from (a);

    (c) replacing $c_1, \cdots, c_k$ by constants from some value vector say $\langle v_1, \cdots, v_k \rangle$;

14. Let us construct r_l as follows:

(a) Let r_u be the rule obtained in step (13a).

(b) Let r_l be obtained from r_u by removing literals in A from $body(r_u)$.

15. Now we show that $r_l \in lparse(r_0)$

(a) By definition of $lparse$, $r_l \in lparse(r_0)$ if $U \cap r\_pos(r_u) \subseteq A$ and $A \cap r\_neg(r_u) = \emptyset$.

(b) Since $r_e = eval_U(r_g, A)$; by definition of eval, we know that $U \cap r\_pos(r_g) \subseteq A$ and $A \cap r\_neg(r_g) = \emptyset$.

(c) By construction of r_u, $r\_atoms(r_u) = r\_atoms(r_g)$. Therefore, we get $U \cap r\_pos(r_u) \subseteq A$ and $A \cap r\_neg(r_u) = \emptyset$.

(d) Therefore, $r_l \in lparse(r_0)$.

16. Now we construct the rule r' required by (9) as follows:

(a) add $c\_lits(C, r_0)$ to body of r_l;

(b) replace tc_constants $c_1, \cdots, c_k$ by constants from value vector $\langle v_1, \cdots, v_k \rangle$;

17. Now we show that $r' \in \gamma(\beta(r_l, C, \Sigma_2), X)$.

(a) Let us refer to the rule obtained in (16a) by r_a.

(b) First, we show that $r_a = \beta(r_l, C, \Sigma_2)$.

(c) Since $r_l \in lparse(r_0)$, the tc_constants that occur in r_l and r_0 are same. Therefore, $c_1, \cdots, c_k$ are the only tc_constants in r_l.

(d) Therefore, by definition of c_lits, $c\_lits(C, r_0) = c\_lits(C, r_l)$.

(e) Since $\beta(r_l, C, \Sigma_2)$ adds $c\_lits(C, r_l)$ to r_l; by construction of r_a, we get $r_a = \beta(r_l, C, \Sigma_2)$.

(f) To show that $r' \in \gamma(\beta(r_l, C, \Sigma_2), X)$, it suffices to show that $\langle v_1, \cdots, v_k \rangle$ is a value vector of r_a with respect to X.

(g) Let us refer to the rule obtained in (13b) by r_b. Notice that by construction, r_a and r_b have the same m_atoms. From (13c), we see that $\langle v_1, \cdots, v_k \rangle$ is a value vector of r_b with respect to X. Hence, by definition of γ, $\langle v_1, \cdots, v_k \rangle$ is a value vector of r_a with respect to X.

(h) Therefore, $r' \in \gamma(\beta(r_l, C, \Sigma_2), X)$.

18. Now we show that $r_e = eval_U(r', A)$.

(a) By construction, $head(r') = head(r_l) = head(r_u) = head(r_g)$.

(b) By construction, $r\_atoms(r') = r\_atoms(r_l) = r\_atoms(r_u) \setminus A$ and $r\_atoms(r_u) = r\_atoms(r_g)$. Hence, $r\_atoms(r') = r\_atoms(r_g) \setminus A$.

(c) Rules r' and r_g are obtained by using same value vector $\langle v_1, \cdots, v_k \rangle$ on r_a and r_b respectively. By construction, $m\_atoms(r_a) = m\_atoms(r_b)$ and $c\_atoms(r_a) = c\_atoms(r_b)$; therefore $m\_atoms(r') = m\_atoms(r_g)$ and $c\_atoms(r') = c\_atoms(r_g)$.

(d) Therefore, $head(r') = head(r_g)$ and $body(r') = body(r_g) \setminus A$.

(e) Since by (10), $r_e = eval_U(r_g, A)$; by definition of $eval$, we get $r_e = eval_U(r', A)$.

19. $\Longleftarrow$

20. Let $r_e \in eval_U(T_l^t, A)$.

21. We construct $r_g \in T_g^t$ such that $r_e = eval_U(r_g, A)$.

22. By (20), we know that $\exists\ r_l \in T_l^t$, such that $r_e = eval(r_l, A)$. Since $T_l^t \subseteq T_l$, $r_l \in T_l$.

23. By lemma 8.1.6, we know that $\exists\ r_g \in T_g$ such that r_l can be obtained from r_g by removing literals in A from $body(r_g)$.

24. Since $r_l \in T_l^t$, we have $r_g \in T_g^t$.

25. Now we have to show that $r_e = eval_U(r_g, A)$

26. We know that $r_e = eval_U(r_l, A)$; $head(r_l) = head(r_g)$ and $body(r_l) = body(r_g) \setminus A$. From definition of $eval$, $r_e = eval_U(r_g, A)$.

■

8.1.6 Step (5) of $\mathcal{P}(\Pi)$

In step (5), we remove the ground instances of rules added in step 3. We get $\Pi_5 = \Pi_4 \setminus lparse(addr(\Pi_2))$

Lemma 8.1.8. *$S \cup \tilde{X}$ is an answer set of $\gamma(\beta(\Pi_4, C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$ iff S is an answer set of $\gamma(\beta(\Pi_5, C, \Sigma_2), X) \cup X \cup M_c$*

Proof:

1. $S \cup \tilde{X}$ is an answer set of $\gamma(\beta(\Pi_4, C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$ iff $S \cup \tilde{X}$ is an answer set of $\gamma(\beta(\Pi_5 \cup lparse(addr(\Pi_2)), C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$
   [By definition $\Pi_5 = \Pi_4 \setminus lparse(addr(\Pi_2))$]

2. (1) is true iff $S \cup \tilde{X}$ is an answer set of $\gamma(\beta(\Pi_5, C, \Sigma_2), X) \cup \gamma(\beta(lparse(addr(\Pi_2)), C, \Sigma_2), X) \cup X \cup \tilde{X} \cup M_c$
   [By definition of γ]

3. (2) is true iff $S \cup \tilde{X}$ is an answer set of $\gamma(\beta(\Pi_5, C, \Sigma_2), X) \cup \gamma(lparse(addr(\Pi_2)), X) \cup X \cup \tilde{X} \cup M_c$
   [Since by definition of β, $\beta(lparse(addr(\Pi_2)), C, \Sigma_2) = lparse(addr(\Pi_2))$]

4. (3) is true iff $S \cup \tilde{X}$ is an answer set of
   $eval_U(\gamma(\beta(\Pi_5, C, \Sigma_2), X) \cup M_c, \quad X \cup \tilde{X}) \quad \cup \; X \cup \tilde{X}$
   [as $U = m\_atoms(\Pi) \cup \{np(\bar{t}_r, \bar{t}_c) \mid p(\bar{t}_r, \bar{t}_c) \in m\_atoms(\Pi)\}$ splits the program with $bot_U = X \cup \tilde{X} \cup \gamma(lparse(addr(\Pi_2)), X)$]

5. (4) is true iff $S \cup \tilde{X}$ is an answer set of $\gamma(\beta(\Pi_5, C, \Sigma_2), X) \; \cup \; X \cup \tilde{X} \cup M_c$
   [splitting set theorem]

6. (5) is true iff S is an answer set of $\gamma(\beta(\Pi_5, C, \Sigma_2), X) \; \cup \; X \cup M_c$
   [as $U = \tilde{X}$ splits with $bot_U = \tilde{X}$ and $top_U = \gamma(\beta(\Pi_5, C, \Sigma_2), X) \; \cup \; X \cup M_c$]

■

8.1.7 Step (6) of $\mathcal{P}(\Pi)$

In step (6a), we add the stored c-literals to Π_5; $\Pi_{6a} = \beta(\Pi_5, C, \Sigma_2)$.

Lemma 8.1.9. *S is an answer set of $\gamma(\beta(\Pi_5, C, \Sigma_2), X) \cup X \cup M_c$ iff S is an answer set of $\gamma(\Pi_{6a}, X) \cup X \cup M_c$*

Proof:

1. S is an answer set of $\gamma(\beta(\Pi_5, C, \Sigma_2), X) \cup X \cup M_c$ iff S is an answer set of $\gamma(\Pi_{6a}, X) \cup X \cup M_c$ [by construction of Π_{6a}]

■

In step (6b), we replace tc-constants in Π_{6a} by c-variables to get Π_6. Before we prove lemma 8.1.11, we prove an auxiliary lemma 8.1.10.

Lemma 8.1.10. *Let Π_{6a} and Π_6 be programs from steps (6a) and (6b) of $\mathcal{P}(\Pi)$, X be a candidate_mixed set and $U = m\text{-}atoms(\Pi)$. We have $eval_U(gr(\Pi_{6M}), X) = eval_U(\gamma(\Pi_{6aM}, X), X)$.*

Proof:

1. $\Longleftarrow$ By definition of gr and γ, we have $\gamma(\Pi_{6aM}, X) \subseteq gr(\Pi_{6M})$. Hence, if $r \in eval_U(\gamma(\Pi_{6aM}, X), X)$ then $r \in eval_U(gr(\Pi_{6M}), X)$.
2. $\Longrightarrow$ Let $r \in eval_U(gr(\Pi_{6M}), X)$. We show that $r \in eval_U(\gamma(\Pi_{6aM}, X), X)$.
3. By (2), $\exists\ r_1 \in gr(\Pi_{6M})$, such that $r = eval_U(r_1, X)$ and $m\text{-}lits(r_1) \subseteq X$.
4. By (3), $\exists\ r_u \in \Pi_{6M}$, such that r_1 is obtained from r_u by substituting c-variables $V_1, \cdots, V_n$ in r_1 by $x_1, \cdots, x_n$ respectively.
5. By (4) and definition of Π_{6aM}, $\exists\ r_0 \in \Pi_{6aM}$, such that r_u is obtained from r_0 by replacing tc_constants $c_1, \cdots, c_n$ by $V_1, \cdots, V_n$ respectively.
6. We show that $r_1 \in \gamma(r_0, X)$
7. By (4), if $\bar{\nu} = \langle x_1, \cdots, x_n \rangle$ is a value vector of r_0 with respect to X then $\gamma_0(r_0, X, \bar{\nu}) = r_1$.
8. $\bar{\nu}$ is a value vector of r_0 with respect to X iff $x_i \in \delta(c_i, r_0, X)$ for all $i = 1$ to n
9. Consider an arbitrary tc_constant c_i in r_0, let c_i occur in a m_atom m in r_0. By construction of r_1, $\exists\ m_\nu \in r_1$, such that r-terms in m and m_ν are same. By (3) and by construction of X, $image(m, X) = m_\nu$.
10. By (5), tc_constant c_i in r_0 was substituted by V_i. By (3), $\zeta_0(r_u, r, V_i) = x_i$
11. Since $m_\nu \in X$ and $\zeta_0(r_u, r, V_i) = x_i$, we have $x_i \in \delta(c_i, m, X)$
12. Therefore, we get $x_i \in \delta(c_i, r_0, X)$ for all $i = 1$ to n
13. By (8), $\bar{\nu}$ is a value vector of r_0 with respect to X
14. By (7), $\gamma_0(r_0, X, \bar{\nu}) = r_1$ and hence $r_1 \in \gamma(r_0, X)$.

15. Therefore $r_1 \in \gamma(\Pi_{6aM}, X)$. By (3), $r \in eval_U(\gamma(\Pi_{6aM}, X), X)$.

■

Lemma 8.1.11. *S is an answer set of $\gamma(\Pi_{6a}, X) \cup X \cup M_c$ iff S is an answer set of $\Pi_6 \cup X \cup M_c$*

Proof:

1. S is an answer set of $\Pi_6 \cup X \cup M_c$ iff S is an answer set of $gr(\Pi_6) \cup X \cup M_c$
2. (1) is true iff $S \setminus X$ is an answer set of $eval_U(gr(\Pi_6) \cup M_c, X)$
   [$U = m\text{-}atoms(\Pi)$ splits $gr(\Pi_6) \cup X \cup M_c$ with $top_U = gr(\Pi_6) \cup M_c$]
3. (2) is true iff $S \setminus X$ is an answer set of $eval_U(\Pi_{6R} \cup gr(\Pi_{6M}) \cup M_c, X)$
   [Since $gr(\Pi_6) = \Pi_{6R} \cup gr(\Pi_{6M})$]
4. (3) is true iff $S \setminus X$ is an answer set of $eval_U(gr(\Pi_{6M}), X) \cup \Pi_{6R} \cup M_c$
   [Since $eval_U(\Pi_{6R} \cup M_c, X) = gr(\Pi_{6R}) \cup M_c$]
5. (4) is true iff $S \setminus X$ is an answer set of $eval_U(\gamma(\Pi_{6aM}, X), X) \cup \Pi_{6R} \cup M_c$
   [By lemma 8.1.10]
6. (5) is true iff $S \setminus X$ is an answer set of $eval_U(\gamma(\Pi_{6aM} \cup \Pi_{6R}, X) \cup M_c, X)$
   [Since $\gamma(\Pi_{6R}, X) = \Pi_{6R}$]
7. (6) is true iff $S \setminus X$ is an answer set of $eval_U(\gamma(\Pi_{6a}, X) \cup M_c, X)$
   [Since $\Pi_{6a} = \Pi_{6R} \cup \Pi_{6aM}$]
8. (7) is true iff S is an answer set of $\gamma(\Pi_{6a}, X) \cup M_c \cup X$
   [splitting set theorem]

■

8.1.8 Step (7) of $\mathcal{P}(\Pi)$

In step (7), we rename c-variables to get $\Pi_7 = rename(\Pi_{6_M}, Y) \cup \Pi_{6_R}$, where Y is a r_ground_mixed set.

Lemma 8.1.12. *S is an answer set of $\Pi_6 \cup X$ iff S is an answer set of $\Pi_7 \cup X$*

Proof:

The proof easily follows from the fact that variables are just renamed in this step.

■

The proof for $\mathcal{P}(\Pi)$ when defined literals are present is very similar.

8.2 Function $expand$

In this section, we prove properties of functions $atleast$, $atmost$ and $expand$.

Lemma 8.2.1. $lc(\Pi, B)$ *is monotonic in its second argument.*

Proof:

1. Let $X \subseteq Y$, we show that $lc(\Pi, X) \subseteq lc(\Pi, Y)$.

   Notice that when $Y \cup lc_0(\Pi, Y)$ is in-consistent then by definition of lc, $lc(\Pi, Y) = r\text{-}lits(\Pi)$. Therefore, the lemma is trivially true.

2. Now we look at the case when $Y \cup lc_0(\Pi, Y)$ is consistent. By definition of lc, $lc(\Pi, Y) = Y \cup lc_0(\Pi, Y)$.

3. Let r-literal $l \in lc(\Pi, X)$, we show that $l \in lc(\Pi, Y)$.

   We know that l was added by one of the four inference rules of lc_0 or $l \in X$. If $l \in X$ then $l \in Y$, we will look at the other cases now.

4. Case(i): Let l be added by first inference rule.
   (a) By inference rule (1), $\exists\ r \in \Pi_R$ such that $l = head(r)$ and $body(r) \subseteq X$.
   (b) Since $X \subseteq Y$, we get $body(r) \subseteq Y$.
   (c) By inference rule (1), $head(r) \in lc_0(\Pi, Y)$. Therefore, $l \in lc(\Pi, Y)$.
5. Case(ii): Let l be added by second inference rule.
   (a) By inference rule (2), $l = not\ h$ and atom h is not in the head of any active rule in $\Pi_R \cup \Pi_M$ with respect to X.
   (b) By definition of active, $\forall r \in \Pi_R \cup \Pi_M$ such that $h = head(r)$, we have $pos(r) \cap neg(X) \neq \emptyset$ or $neg(r) \cap pos(X) \neq \emptyset$.
   (c) Since $X \subseteq Y$, we get $pos(r) \cap neg(Y) \neq \emptyset$ or $neg(r) \cap pos(Y) \neq \emptyset$.
   (d) Therefore, $\forall r \in \Pi_R \cup \Pi_M$ such that $h = head(r)$, r is not active with respect to Y.
   (e) By inference rule (2), $not\ h \in lc_0(\Pi, Y)$ and $l \in lc(\Pi, Y)$.
6. Case(iii): Let l be added by third inference rule.
   (a) By inference rule (3), $\exists\ h \in X$ such that $r \in \Pi_R \cup \Pi_M$ is the only active rule with $h = head(r)$. So *r-lits*(r) are added to $lc_0(\Pi, X)$ and $l \in$ *r-lits*(r).
   (b) Since $h \in X$, we have $h \in Y$.
   (c) Since r is the only active rule with respect to X such that $h = head(r)$, for any other rule $r' \in \Pi_R \cup \Pi_M$ such that $h = head(r')$, we have $pos(r') \cap neg(X) \neq \emptyset$ or $neg(r') \cap pos(X) \neq \emptyset$.
   (d) Since $X \subseteq Y$, by definition of active, for any rule $r' \in \Pi_R \cup \Pi_M$ other than r such that $h = head(r')$, we get r' is not active with respect to Y.

(e) Rule r can be active or not active with respect to Y.

i. If r is active then r is the only rule with h in the head that is active with respect to Y. By (6a) and inference rule (3), $r\_lits(r) \subseteq lc_0(\Pi, Y)$ and $l \in lc(\Pi, Y)$.

ii. If r is not active then there is no rule with h in the head that is active with respect to Y. By inference rule (2), $not\ h \in lc_0(\Pi, Y)$. Since by (2) $Y \cup lc_0(\Pi, Y)$ is consistent. By (6b) this case is not possible, hence r is active with respect to Y.

7. Case(iv): Let l be added by fourth inference rule.

(a) By inference rule (4), $l = not\ l'$ and $\exists\ r \in \Pi_R$ and $h = head(r)$ and $not\ h \in X$ and all literals in $body(r)$ except l' belong to X. Then $l \in lc_0(\Pi, X)$.

(b) Since $X \subseteq Y$, we know that $not\ h \in Y$ and all literals in $body(r) \setminus \{l'\}$ belong to Y.

(c) Case (i): If $l' \in Y$, then $body(r) \subseteq Y$, therefore, $h \in lc_0(\Pi, Y)$. Since $not\ h \in Y$, $Y \cup lc_0(\Pi, Y)$ is in-consistent. This is not possible, so $l' \notin Y$. Therefore by inference rule (4), $not\ l' \in lc_0(\Pi, Y)$. Therefore, $l \in lc(\Pi, Y)$.

8. By (4), (5), (6) and (7), we get lc is monotonic in its second argument.

Lemma: *Let* Π *be a program and* B *be a ground set of extended r-literals. If* $lc_0(\Pi, B)$ *is consistent then it is unique.*

Proof:

1. Let $lc_0(\Pi, B)$ be consistent.

We show that it is unique by proof of contradiction.

2. Suppose $lc_0(\Pi, B)$ is not unique and there are two sets X, Y such that $X = lc_0(\Pi, B)$ and $Y = lc_0(\Pi, B)$.

3. Let $Z = X \cap Y$. We show that $Z = lc_0(\Pi, B)$, by showing that Z is closed under each of the four inference rules.

4. Let r be a rule in Π_R and $body(r) \subseteq B$. We show that $head(r) \in Z$.

   (a) Since $X = lc_0(\Pi, B)$ and $Y = lc_0(\Pi, B)$, by inference rule (1), $head(r) \in X$ and $head(r) \in Y$.

   (b) Therefore by construction of Z, $head(r) \in Z$. Hence Z is closed under inference rule (1).

5. Z is closed under other inference rules and can be proven similarly.

6. Z is closed under all the inference rules, and $Z = X \cap Y$. This is not possible as X and Y are minimal sets closed under the inference rules. We reach a contradiction.

7. $lc_0(\Pi, B)$ is unique.

The other propositions and lemmas of *expand* have similar proofs to the corresponding propositions and lemmas in [75].

8.3 Function c_solve

we prove two auxiliary lemmas before we prove the proposition 5.5.5.

Lemma 8.3.1. *Let* P *and* Q *be two ASP programs and* A *and* B *be disjoint sets of atoms such that:*

(a) $\mathrm{Lit}(P) \subseteq A \cup B$ *and* $\mathrm{Lit}(Q) \subseteq A \cup B$.

(b) $\mathrm{head}(P) \subseteq A$

(c) $\mathrm{head}(Q) \subseteq B$

(d) if not l *occurs in* P *or* Q *then* $l \in A$

(e) There are no loops in $P \cup Q$ *between predicates from* A *and* B.

For any $S \subseteq A$ *and* $D \subseteq B$*, if* $S \cup D$ *is an answer set of* $P \cup D$ *and* $S \cup D$ *is an answer set of* $Q \cup S$ *then* $S \cup D$ *is an answer set of* $P \cup Q$.

Proof:

1. Let $S \cup D$ be an answer set of $P \cup D$
2. Let $S \cup D$ be an answer set of $Q \cup S$
3. First we show that $S \cup D$ is closed under the rules of $(P \cup Q)^{(S \cup D)}$.
   (a) If *not* l occurs in P then by properties (a), (d) and the fact that $D \subseteq B$, we have $l \in S \cup D$ iff $l \in S$. Therefore, we have $(P \cup D)^{S \cup D} = P^S \cup D$, $(Q \cup S)^{S \cup D} = Q^S \cup S$ and $(P \cup Q)^{S \cup D} = (P \cup Q)^S = P^S \cup Q^S$.
   (b) By (1), $S \cup D$ is closed under $(P \cup D)^{S \cup D}$, and therefore under $P^S \cup D$.
   (c) By (2), $S \cup D$ is closed under $(Q \cup S)^{S \cup D}$, and therefore under $Q^S \cup S$.
   (d) By (3a), (3b) and (3c), $S \cup D$ is closed under $(P \cup Q)^{S \cup D}$.
4. Now we show that $S \cup D$ is minimally closed under rules of $(P \cup Q)^S$.
5. Let $S_0 \cup D_0 \subseteq S \cup D$ be minimally closed under rules of $(P \cup Q)^S$. We show that $S_0 \cup D_0 = S \cup D$.

6. First, we show that for each $l \in S \cup D$, there exists a rule $r \in (P \cup Q)^S$, such that $head(r) = l$ and $S \cup D \models body(r)$.

   ▷ By (1), if $l \in S$, then there exists a rule in P^S such that $head(r) = l$ and $S \cup D \models body(r)$.

   ▷ By (2), if $l \in D$, then there exists a rule in Q^S such that $head(r) = l$ and $S \cup D \models body(r)$.

7. Case(i): Suppose $S_0 = S$ and $D_0 \subset D$.

   ▷ Since $(P \cup Q)^S = P^S \cup Q^S$, by (5), $S_0 \cup D_0$ is closed under $Q^S \cup S = (Q \cup S)^{(S \cup D)}$.

   ▷ Since $S_0 \cup D_0 \subset S \cup D$, by (2), this is not possible.

8. Case(ii): Suppose $D_0 = D$ and $S_0 \subset S$.

   ▷ Since $(P \cup Q)^S = P^S \cup Q^S$, by (5), $S_0 \cup D_0$ is closed under $P^S \cup D = (P \cup D)^{(S \cup D)}$.

   ▷ Since $S_0 \cup D_0 \subset S \cup D$, by (1), this is not possible.

9. Case(iii): Suppose $S_0 \subset S$ and $D_0 \subset D$. We show that this case is not possible.

10. First we show that if $S_0 \subset S$ and $D_0 \subset D$ then one of the following statement holds:

    (l_1) There exists a cycle (loop) in P^S such that the literals in the loop do not depend on literals from $D \setminus D_0$.

    (l_2) There exists a cycle in Q^S such that the literals in the loop do not depend on literals from $S \setminus S_0$.

11. Next we construct a set $X \subset S \cup D$ of literals and show that X is either closed under $(P \cup D)^S$ or closed under $(Q \cup S)^S$. By (1) and (2) this is not possible.

    Now we show that (10) holds.

12. Let $L = (S \cup D) \setminus (S_0 \cup D_0)$. Let R be the set of rules $r \in (P \cup Q)^S$ such that head of r is in L and $body(r) \subseteq S \cup D$. By (6), we know that, for every $l \in L$, there exists a rule $r \in R$ such that $head(r) = l$.

13. Therefore, for every $r \in R$, since $head(r) \notin S_0$, $body(r) \not\subseteq S_0 \cup D_0$. By construction of R, $body(r) \subseteq S \cup D$.

14. By construction of L, and (13), $body(r) \cap L \neq \emptyset$. That is for every rule r in R, there is a literal from L in body(r).

    Now we show that there is a cycle in rules of R. For this we construct a digraph as follows.

15. Let us construct a digraph $G = \langle V, E \rangle$ as follows: The set of nodes $V = L$. There is an edge from node a to node b in E if there is a rule r in R such that $a = head(r)$ and $b \in body(r)$.

16. By (13), for every node $l \in V$, there is at least one edge starting from l and ending at some other node in V. The out-degree of each node in V is greater or equal to one. Therefore, there is at least one cycle in G.

17. By construction of R and (16), there is at least one cycle in R. Now we show that one of the cycles from R satisfy statement (10).

18. Let $L = L_s \cup L_d$ such that $L_s = L \cap S$ and $L_d = L \cap D$. According to property (e), no cycle in G can contain both a literal from L_s and a literal from L_d.

Let us color the nodes in G corresponding to literals L_s red and let us color the nodes in G corresponding to literals in L_d green. All nodes of G are either red or green. There is no cycle in G containing both a red node and a green node.

19. Let $r_0, \ldots, r_k$ be the red cycles of G (containing only red nodes of literals in L_s) and $g_0, \ldots, g_m$ be green cycles of G (containing only green nodes of literals in L_d). Let $r_{k+1}, \ldots, r_x$ be the red nodes not in any cycle (literals in L_s that are not part of any cycle). Let $g_{m+1}, \ldots, g_y$ be the green nodes not in any cycle (literals in L_d that are not in any cycle).

20. A literal a depends on a literal b if there is a directed path from a to b in G. A literal a *depends* on a cycle i if there is a node $b \in i$ such that there is a directed path from a to b in G.

21. It is easy to show that each literal in $r_{k+1}, \ldots, r_x$ and $g_{m+1}, \ldots, g_y$ depends on some cycle in $r_0, \ldots, r_k$, $g_0, \ldots, g_m$.

22. A cycle i *depends* on cycle j if there is a node $a \in i$ and a node $b \in j$ such that there is a directed path from a to b in G. A cycle i *depends* on a literal b, if there is a node $a \in i$, such that there is a directed path from a to b in G.

23. Let $G_c = \langle V_c, E_c \rangle$ be a directed graph such that $V_c = \{r_0, \ldots, r_x, g_0, \ldots, g_y\}$ be the nodes of G_c. The nodes $r_0, \ldots, r_k, g_0, \ldots, g_m$ corresponds to cycles $r_0, \ldots, r_k, g_0, \ldots, g_m$ respectively. The nodes $r_{k+1}, \ldots r_x, g_{m+1}, \ldots, g_y$ correspond to literals $r_{k+1}, \ldots, r_x, g_{m+1}, \ldots, g_y$ respectively. Let (a, b) be a directed edge in E_c if a depends on b in G.

24. Now, we show that either there exists a red cycle r_i such that r_i does not

depend on any green node g_j in G_c or there exists a green cycle g_i such that g_i does not depend on any red node r_j in G_c.

- ▷ Consider a subgraph G_d of G_c, where $V_d = V_c$ and $E_d \subseteq E_c$ such that for $0 \leq i \leq x$ and $0 \leq j \leq y$, if $(r_i, g_j) \in E_c$ then $(r_i, g_j) \in E_d$ and if $(g_j, r_i) \in E_c$ then $(g_j, r_i) \in E_d$.
- ▷ By (18), a cycle in G cannot contain a red node and a green node. Since graph G_d shows dependency of red nodes on green nodes and vice versa, there cannot be a cycle in graph G_d, it should be acyclic. Therefore, the out-degree of all nodes in G_d cannot be greater than or equal to one.
- ▷ Hence, there exists at least one node in G_d that has out-degree equal to zero. Let $n_1, \ldots, n_n$ be nodes in G_d with out-degree equal to zero. we show that at least one of them is a cycle node.

  We prove by contradiction.
- ▷ Suppose $n_1, \ldots, n_n$ are all literal nodes. Consider an arbitrary node n_i.
- ▷ Case(a): Let n_i be a red node ($n_i \in L_s$). By (21), we know that n_i depends on some cycle in $r_0, \ldots, r_k, g_0, \ldots, g_m$. Since n_i has out-degree equal to zero in graph G_d, we can conclude that n_i depends on some red cycle from $r_0, \ldots, r_k$. Suppose n_i depends on r_j. If red cycle r_j depends on some green node g_l from $g_0, \ldots, g_y$ then n_i depends on g_l, which means out-degree of n_i will not be zero in G_d. Therefore we can conclude that cycle r_j does not depend on any node from $g_0, \ldots g_y$. The out-degree of r_j in G_d is equal to zero and therefore $n_1, \ldots, n_n$ are all not literal nodes.
- ▷ Case(b): Let n_i be a green node ($n_i \in L_d$). By similar argument, there is a cycle node g_j in G_d with out-degree equal to zero and $n_1, \ldots, n_n$

are all not literal nodes.

- ▷ So there exists a red cycle node r_i that does not depend on any green node g_j in G_c or there exists a green cycle node g_i that does not depend on any red node r_j in G_c.

25. Consider a cycle corresponding to the node with out-degree equal to zero in G_d. It can be either some red r_i or green g_i. If it is r_i then literals from r_i do not depend on literals from $L_d = D \setminus D_0$. If it is g_i then literals from g_i do not depend on literals from $L_s = S \setminus S_0$. Therefore, statement (10) holds true.

    Now we show statement (11) is true.

26. Case(a): Suppose it is red cycle r_i. Let $H = L_0 \cup L_1$ be the set of literals where L_0 is the set of literals from cycle r_i and L_1 is the set of literals $b \in L$ such that there exists $a \in L_0$ and a depends on b in G.

27. Now, we show that $H \subseteq L_s$.

    - ▷ From (25), we know that red cycle r_i does not depend on any green node in $g_0, \ldots, g_y$. So literals corresponding to nodes $g_0, \ldots, g_y$ do not belong to H.
    - ▷ By construction of G_c, the literals corresponding to $g_0, \ldots, g_y$ are all literals from L_d. Therefore, $H \cap L_d = \emptyset$ and $H \subseteq L_s$.

28. By construction of H, R and (27), for every rule r in R such that $head(r) \in H$, there exists literal from H in the body of r.

29. Let $S_1 = S \setminus H$. We show that $S_1 \cup D$ is closed under $(P \cup D)^{S \cup D}$.

30. By properties (a), (d) and fact that $D \subseteq B$, we get $S_1 \cup D$ is closed under $(P \cup D)^{(S \cup D)}$, if it is closed under $P^S \cup D$

31. $S_1 \cup D$ is trivially closed under rules of D. Now we show that $S_1 \cup D$ is closed under P^S.

32. Case(1): Let $r \in P^S$ such that no literals in r belong to H.

    ▷ If $S_1 \cup D \models body(r)$ then $S \cup D \models body(r)$. Since $head(r) \in S \setminus H$ and $S_1 = S \setminus H$. we get $head(r) \in S_1 \cup D$.

33. Case(2): Let $r \in P^S$ such that $head(r) \in H$.

    ▷ By construction of R, if $r \notin R$, then $S \cup D \not\models body(r)$. Hence, $body(r) \not\subseteq S \cup D$. Therefore, $body(r) \not\subseteq S_1 \cup D$ and $S_1 \cup D \not\models body(r)$.

    ▷ If $r \in R$, then by (28), $\exists l \in body(r)$, such that $l \in H$. Therefore, $S_1 \cup D \not\models body(r)$.

34. Case(3): Let $r \in P^S$ such that $\exists l \in body(r)$ and $l \in H$. We know $S_1 \cup D \not\models body(r)$.

35. From Case(1), Case(2) and Case(3), we get $S_1 \cup D$ is closed under $P^S \cup D$. Since $S_1 \cup D \subset S \cup D$, by (1), this is not possible.

36. Therefore, if statement (l_1) is true then we arrive at a contradiction.

37. Case(b): Suppose the cycle with property in (25) is green g_i. Let $H = L_0 \cup L_1$ be the set of literals where L_0 is the set of literals from cycle g_i and L_1 is the set of literals $b \in L$ such that there exists $a \in L_0$ and a depends on b in G.

38. Now, we show that $H \subseteq L_d$.

    ▷ From (25), we know that green cycle g_i does not depend on any red node in $r_0, \ldots, r_x$. So literals corresponding to red nodes $r_0, \ldots, r_x$ do not belong to H.

- ▷ By construction of G_c, the literals corresponding to $r_0, \ldots, r_x$ are all literals from L_c. Therefore, $H \cap L_c = \emptyset$ and $H \subseteq L_d$.

39. By construction of H, R and (38), for every rule r in R such that $head(r) \in H$, there exists literal from H in the body of r.

40. Let $D_1 = D \setminus H$. We show that $S \cup D_1$ is closed under $(Q \cup S)^{S \cup D}$.

41. $S \cup D_1$ is closed under $(Q \cup S)^{(S \cup D)}$, if it is closed under $Q^S \cup S$.

42. $S \cup D_1$ is trivially closed under rules of S. Now we show that $S \cup D_1$ is closed under Q^S.

43. Case(1): Let $r \in Q^S$ such that no literals in r belong to H.

    - ▷ If $S \cup D_1 \models body(r)$ then $S \cup D \models body(r)$. Since $head(r) \in D \setminus H$ and $D_1 = D \setminus H$, we get $head(r) \in S \cup D_1$.

44. Case(2): Let $r \in Q^S$ such that $head(r) \in H$.

    - ▷ If $r \notin R$, then by construction of R, $S \cup D \not\models body(r)$. Hence, $body(r) \not\subseteq S \cup D$. Therefore, $body(r) \not\subseteq S \cup D_1$ and $S \cup D_1 \not\models body(r)$.
    - ▷ If $r \in R$, then by (39), $\exists l \in body(r)$, such that $l \in H$. Therefore, $S \cup D_1 \not\models body(r)$.

45. Case(3): Let $r \in Q^S$ such that $\exists l \in body(r)$ and $l \in H$. We know $S \cup D_1 \not\models body(r)$.

46. From Case(1), Case(2) and Case(3), we get $S \cup D_1$ is closed under $Q^S \cup S$. Since $S \cup D_1 \subset S \cup D$, by (2), this is not possible.

47. Therefore, if statement (l_2) is true then we arrive at a contradiction.

48. By (36) and (47), we see that l_1 and l_2 cannot hold. If $S_0 \subset S$ and $D_0 \subset D$, then either l_1 or l_2 holds. Therefore, it is not possible that $S_0 \subset S$ and $D_0 \subset D$.

49. By (7), (8) and (48), we get $S_0 = S$ and $D_0 = D$ and $S \cup D$ is a minimal set closed under $P^S \cup Q^S$.

50. Therefore, $S \cup D$ is an answer set of $P \cup Q$.

■

Lemma 8.3.2. *Let* P *be an ASP program and* S *be an answer set of* P. *Let* T *be a set of atoms such that*

(p_1) $head(P) \cap T = \emptyset$ *and*

(p_2) *if not* l *occurs in* P *then* $l \notin T$.

For every rule r *such that* $body(r) \cap T \neq \emptyset$, *if*

(a) $\exists l \in (pos(r) \setminus T)$ *such that* $l \notin S$ *or*

(b) $\exists l \in (neg(r) \setminus T)$ *such that* $l \in S$ *or*

(c) $head(r) \in S$

then $S \cup T$ *is an answer set of* $P \cup T$.

Proof:

1. Let S be an answer set of P.

2. First we show that $S \cup T$ is closed under $(P \cup T)^{(S \cup T)}$.

   (a) By (1), S is closed under P^S.

(b) Since T is a set of atoms, by property (p_2), we have $(P \cup T)^{(S \cup T)} = P^S \cup T$. Therefore, it suffices to show that $S \cup T$ is closed under $P^S \cup T$.

(c) Let $r \in P^S$ such that $body(r) \subseteq S \cup T$. We show that $head(r) \in S \cup T$.

(d) Case(i): Suppose $body(r) \cap T = \emptyset$. By (2c), $body(r) \subseteq S$. By (2a), we get $head(r) \in S$. Therefore, $head(r) \in (S \cup T)$.

(e) Case(ii): Suppose $body(r) \cap T \neq \emptyset$. By definition of reduct, $\exists r_0 \in P$ such that $r = r_0^S$ and condition (b) is not satisfied by r_0. By (2c), condition (a) is not satisfied. Therefore, condition (c) is satisfied by r_0 and $head(r_0) \in S$. Since $head(r_0) = head(r)$, $head(r) \in S$.

(f) By cases (i) and (ii), and fact that $S \cup T$ is trivially closed under rules of T, we get $S \cup T$ is closed under $P^S \cup T$.

3. Now we show that $S \cup T$ is minimally closed under $P^S \cup T$.

4. Let $V \subseteq S \cup T$ be minimally closed under $P^S \cup T$.

5. Let $V_0 = V \cap S$. By (4), $T \subseteq V$. Therefore $V = V_0 \cup T$.

6. Now we show that V_0 is closed under P^S.

(a) By (4), $V = V_0 \cup T$ is closed under $P^S \cup T$.

(b) Case(i): Let $r \in P^S$ such that $body(r) \cap T = \emptyset$. If $body(r) \subseteq V_0$ then $body(r) \subseteq V$. By (6a), $head(r) \in V$. By property (p1), $head(r) \cap T = \emptyset$, since $V = V_0 \cup T$, we get $head(r) \in V_0$.

(c) Case(ii): Let $r \in P^S$ such that $body(r) \cap T \neq \emptyset$. Then by construction of V_0, we get $body(r) \not\subseteq V_0$.

(d) V_0 is closed under P^S.

7. By (1), $V_0 = S$ and by definition of V, $V = S \cup T$.

8. By (4), $S \cup T$ is answer set of $P \cup T$.

■

Now we prove proposition 5.5.5.

Proposition 5.5.5: *Let* Π *be a program and* S *be a set of r-literals. Let* $Q = \text{query}(\Pi, S)$ *be a query,* A *be a set of answer constraints, and* θ *be any solution of* A*. Let* V *be the set of variables in* Q*,* $D = \textit{d-lits}(Q)|_\theta^{\bar{V}}$ *be a set of d-literals and* $X = \text{mcv\_set}(\Pi)|_\theta^{\bar{V}}$ *be a candidate_mixed set. If* $\textit{c\_solve}(\Pi_D, S, Q, A)$ *returns* true *then if* $S \cup D \cup X \cup M_c$ *is an* asp*-answer set of* $\Pi_R \cup \Pi_M \cup D \cup X \cup M_c$ *then there exists an answer set* M *of* Π *such that* $S \cup X$ *is the simplified part of* M*.*

Proof:

1. Let $Y = S \cup D \cup X \cup M_c$ be an asp-answer set of $\Pi_R \cup \Pi_M \cup D \cup X \cup M_c$.

2. The program $\Pi_D \cup S \cup X \cup M_c$ is stratified and therefore has a single answer set. Let $S \cup D_m \cup X \cup M_c$ be an answer set of $\Pi_D \cup S \cup X \cup M_c$, where D_m is a set of d-literals.

3. We show that the set $M = S \cup D_m \cup X \cup M_c$ is $\mathcal{AC}(\mathcal{C})$ answer set of Π.

4. Using lemma 8.3.2, first we show that M is an asp-answer set of $\Pi_R \cup \Pi_M \cup D_m \cup X \cup M_c$.

5. Consider the set $T = D_m \setminus D$ and the program $P = \text{ground}(\Pi_R \cup \Pi_M) \cup D \cup X \cup M_c$. By (1), Y is the answer set of P.

6. First we show that P and T satisfy the properties in lemma 8.3.2.

(a) By definition of P and T, we get $head(P) \cap T = \emptyset$. Therefore, property (p_1) from lemma 8.3.2 is satisfied.

(b) By syntactic restriction in $\mathcal{AC}(C)$ programs, if $not\ l$ occurs in rules of P then l is a r-literal and therefore $l \notin T$. Property (p_2) is satisfied.

Now we show that for every rule r in P such that $body(r) \cap T \neq \emptyset$, either condition (a) or (b) or (c) is satisfied.

(c) Let R_g be the rules in P such that $body(r) \cap T \neq \emptyset$. By definition of P and T, $R_g \subseteq ground(\Pi_M)$.

(d) Let $R \subseteq \Pi_M$ such that $R_g \subseteq ground(R)$.

(e) Consider a rule $r_g \in R_g$, by definition of $ground$, $\exists r \in R$ such that $r_g \in ground(r)$. Note that r is r-ground.

(f) Case(i): If $Y \not\models r\_lits(r)$ then either $r\_pos(r) \not\subseteq Y$ or $r\_neg(r) \cap Y \neq \emptyset$. Therefore, we have $pos(r_g) \not\subseteq Y$ or $neg(r_g) \cap Y \neq \emptyset$. In this case, rule r_g satisfies property (a) or (b) of lemma 8.3.2.

(g) Case(ii): If $Y \models r\_lits(r)$ then

i. Suppose $head(r) \in Y$. Since $head(r_g) = head(r)$, $head(r_g) \in Y$. In this case, rule r_g satisfies property (c) of lemma 8.3.2.

ii. Suppose $head(r) \notin Y$. We show that there exists an m-atom m in $body(r_g)$, such that $m \notin X$.

iii. By transformation tr_2, every $r \in \Pi_M$, contains only one d-literal. Let $d = d\text{-}lit(r)$. Note that d is r-ground.

iv. By construction of $query(\Pi, S)$, $\neg d \in Q$. Since $c\_solve(\Pi_D, S, Q, A)$ returned $true$, we have $\Pi_D \cup S \cup X \cup M_c \models Q$. The answer constraint A consists of constraints on $\bar{V}$ such that

$\Pi_D \cup S \cup X \cup M_c \models$ *d-lits*(Q). Therefore, A consists of constraints on $\bar{V}$ such that $\Pi_D \cup S \cup X \cup M_c \models \neg d$.

v. Let the c-variables in r be $\bar{V}_r$. Let θ_r be the substitution for variables in $\bar{V}_r$ such that $r_g = ground(r)^{\bar{V}_r}_{\theta_r}$. Let $d_0 = T \cap body(r_g)$. By definition of ground, we get $d_0 = ground(d)^{\bar{V}_r}_{\theta_r}$.

vi. Each c-variable in $\bar{V}_r$ occurs in some m-atom $m_1, \ldots, m_k \in body(r)$. We get $m_{i_g} = ground(m_i)^{\bar{V}_r}_{\theta_r}$ such that $m_{i_g} \in body(r_g)$.

vii. By definition of T and (2), $\Pi_D \cup S \cup X \cup M_c \models d_0$. By (iv), for any solution substitution θ of A, $d_0 \neq d^{\bar{V}_r}_{\theta}$.

viii. By *rename* procedure in step (7) of $\mathcal{P}(\Pi)$, we get $\bar{V}_r \subseteq \bar{V}$. Let $\theta(\bar{V}_r)$ be the substitution of variables in $\bar{V}_r$ from solution θ.

ix. For any solution substitution θ of A, $\theta_r \neq \theta(\bar{V}_r)$. Therefore, $\forall\theta$ of A, $\exists V_i \in \bar{V}_r$, such that $(V_i = v) \in \theta$ and $(V_i = u) \in \theta_r$ and $v \neq u$.

x. Let V_i occur in m-atom $m \in body(r)$. Let $a = m^{\bar{V}_r}_{\theta}$ and $b = m^{\bar{V}_r}_{\theta_r}$. We know by definition of X, $a \in X$ and $b \notin X$. By definition of b, $b \in body(r_g)$.

xi. Therefore, by definition of X, for any solution θ of A, $\exists m_i \in body(r_g)$ such that $m_i \notin X$.

xii. Therefore, property (a) of lemma 8.3.2 is satisfied by r_g.

(h) Therefore, $P \cup T$ satisfy the properties from lemma 8.3.2.

7. Since Y is an answer set of P, and $P \cup T$ satisfies properties of lemma 8.3.2, we get $Y \cup T$ is an answer set of $P \cup T$.

8. Therefore, M is an asp-answer set of $\Pi_R \cup \Pi_M \cup D_m \cup X \cup M_c$.

9. Now we show that M is an asp-answer set of $\Pi_R \cup \Pi_M \cup \Pi_D \cup X \cup M_c$ using

lemma 8.3.1.

10. The set $U = m\text{-}atoms(\Pi) \cup M_c$ splits $W_1 = \Pi_R \cup \Pi_M \cup D_m \cup X \cup M_c$ with $bot_U(W_1) = X \cup M_c$. The answer set of $bot_U(W_1) = X \cup M_c$.

11. By (7) and splitting set theorem, $S \cup D_m$ is an answer set of $eval_U(\Pi_R \cup \Pi_M \cup D_m, X \cup M_c)$. Let $P_1 = eval_U(\Pi_R \cup \Pi_M, X \cup M_c)$. By definition of eval, $eval_U(\Pi_R \cup \Pi_M \cup D_m, X \cup M_c) = P_1 \cup D_m$. Therefore, $S \cup D_m$ is an answer set of $P_1 \cup D_m$.

12. By (2), M is an answer set of $W_2 = \Pi_D \cup S \cup X \cup M_c$. The set $U = m\text{-}atoms(\Pi) \cup M_c$ splits W_2. Therefore by splitting set theorem, $S \cup D_m$ is an answer set of $eval(\Pi_D \cup S, X \cup M_c)$. Let $P_2 = eval_U(\Pi_D, X \cup M_c)$. Since $eval(\Pi_D \cup S, X \cup M_c) = P_2 \cup S$, we get $S \cup D_m$ is an answer set of $P_2 \cup S$.

13. The set $U = m\text{-}atoms(\Pi) \cup M_c$ splits $W_3 = \Pi_R \cup \Pi_M \cup \Pi_D \cup X \cup M_c$ with $bot_U(W_3) = X \cup M_c$. The answer set of $bot_U(W_3) = X \cup M_c$.

14. By splitting set theorem, M is an answer set of W_3 iff $S \cup D_m$ is an answer set of $eval_U(\Pi_R \cup \Pi_M \cup \Pi_D, X \cup M_c)$. By definition of eval, $eval_U(\Pi_R \cup \Pi_M \cup \Pi_D, X \cup M_c) = P_1 \cup P_2$. Therefore, M is an answer set of $\Pi_R \cup \Pi_M \cup \Pi_D \cup X \cup M_c$ iff $S \cup D_m$ is an answer set of $P_1 \cup P_2$.

15. We show that $P_1 = eval_U(\Pi_R \cup \Pi_M, X \cup M_c)$ and $P_2 = eval_U(\Pi_D, X \cup M_c)$ satisfy the properties of lemma 8.3.1.

16. Let $A = r\text{-}lits(\Pi)$ and $B = d\text{-}lits(\Pi)$. By definition of signature, $A \cap B = \emptyset$.

17. By definition of A and B, $Lit(P_1) \subseteq A \cup B$ and $Lit(P_2) \subseteq A \cup B$. Condition (a) is satisfied by P_1 and P_2.

18. Since heads of rules from $\Pi_R \cup \Pi_M$ are subset of $r\text{-}lits(\Pi)$, $head(P_1) \subseteq A$. Similarly, $head(P_2) \subseteq B$. Condition (b) and (c) are satisfied by P_1 and P_2.

19. By syntactic restrictions on program Π, condition (d) and (e) are satisfied by P_1 and P_2. All conditions from lemma 8.3.1 are satisfied by P_1 and P_2.

20. By definition of S and D_m, $S \subseteq A$ and $D_m \subseteq B$.

21. By (10), $S \cup D_m$ is an answer set of $P_1 \cup D_m$. By (12), $S \cup D_m$ is an answer set of $P_2 \cup S$.

22. Therefore by lemma 8.3.1, $S \cup D_m$ is an answer set of $P_1 \cup P_2$.

23. By (14), $M = S \cup D_m \cup X \cup M_c$ is an answer set of $\Pi_R \cup \Pi_M \cup \Pi_D \cup X \cup M_c$.

24. Therefore, M is an answer set of $ground(\Pi) \cup X \cup M_c$.

25. By definition of answer set, M is an $\mathcal{AC}(C)$ answer set of Π and $S \cup X$ is the simplified part of M.

8.4 $\mathcal{AC}(C)$_solver

To prove proposition, we use the following lemmas.

Lemma 8.4.1. *Let* Π *be a program and* B *be a set of extended r-literals. Let* $S = expand(\Pi_R \cup \Pi_M, B)$. *If* S *is consistent and covers all r-atoms of* Π *then* $pos(S)$ *is closed under* $\Pi_R^{pos(S)}$.

Proof:

1. Let $r \in \Pi_R^{pos(S)}$ such that $body(r) \subseteq pos(S)$. We show that $head(r) \in pos(S)$.
2. By definition of reduct, $\exists\ r_0 \in \Pi_R$ such that $r = r_0^{pos(S)}$ and $r\_neg(r_0) \cap pos(S) = \emptyset$. Since S covers all r-atoms of Π, we get $r\_neg(r_0) \subseteq neg(S)$.
3. By construction of r, $body(r) = r\_pos(r_0)$. By (1), $r\_pos(r_0) \subseteq pos(S)$.
4. By (2) and (3), $S \models body(r_0)$.
5. By construction of S and inference rule (1) of definition of lower closure, S is closed under rules of Π_R. Therefore, $\forall r \in \Pi_R$, if $body(r) \subseteq S$ then $head(r) \in S$.
6. By (4) and (5), $head(r_0) \in S$. Since $head(r) = head(r_0)$, we get $head(r) \in pos(S)$.
7. Therefore, $pos(S)$ is closed under $\Pi_R^{pos(S)}$.

Lemma 8.4.2. *Let* $\Pi = \mathcal{T}(\Pi_0)$ *be a program input to* $\mathcal{AC}(C)$_solver *and* S *be a set of ground extended r-literals that covers all r-atoms from* Π. *Let* $Q = query(\Pi, S)$ *and* $\bar{V} = vars(Q)$. *Let c_solve(*Π_D*,S,Q,A) return true and* θ *be* a_solution(A). *Let* $D = d\text{-}lits(Q)|_\theta^{\bar{V}}$ *and* $X = mcv\_set(\Pi)|_\theta^{\bar{V}}$. *Let*

$r \in ground(\Pi_M)$. *If* $S \models$ *r-lits*(r), $X \models$ *m-lits*(r) *and* $D \models$ *d-lits*(r) *then* $head(r) \in S$.

Proof:

1. Consider the case when transformations tr_1 and tr_2 are performed after computing $\mathcal{P}(\Pi)$.

2. By transformation tr_2, for every rule $r_0 \in \Pi_M$, there is only one d-literal in the body of r_0. Therefore, for every $r \in ground(\Pi_M)$, there is only one d-literal in the body of r. (Note that Π_M is r-ground).

3. Let $r_0 \in \Pi_M$ and $d =$ *d-lits*(r_0). By tr_2 and (1), the d-predicate d_p of d is unique in Π_M. Therefore, in $ground(\Pi_M)$, d_p occurs only in rules of $ground(r_0)$.

4. Let $r \in ground(\Pi_M)$ such that $S \models$ *r-lits*(r), $X \models$ *m-lits*(r) and $D \models$ *d-lits*(r). By (2), there exists exactly one d-literal in *d-lits*(r). Let it be l, we get $l \in D$. Let d_p be d-predicate of l.

5. By construction of l, we know $\exists\ d_p(\bar{t}_r, \bar{V}_c) \in$ *d-lits*(Q) such that $l = d_p(\bar{t}_r, \bar{V}_c)|_\theta^{\bar{V}}$.

6. By construction of Q, we know that $d_p(\bar{t}_r, \bar{V}_c) \in$ *d-lits*(Q) iff $\exists\ r_0 \in \Pi_M$ such that $S \models$ *r-lits*(r_0) and $head(r_0) \in S$ and $d_p(\bar{t}_r, \bar{V}_c) \in$ *d-lits*(r_0).

7. Since $l \in body(r)$ and l is formed by d-predicate d_p, by construction of r and (3), we get that $r \in ground(r_0)$. Therefore $head(r) = head(r_0)$.

8. By (6), $head(r) \in S$.

The lemma 8.4.2 assumes the construction of $\mathcal{T}(\Pi_0)$ as described in chapter 5. In this construction, the transformations tr_1 and tr_2 are performed after computing

$\mathcal{P}(\Pi_0)$. However, for efficiency during implementation, the transformations are performed before computing $\mathcal{P}(\Pi_0)$. Let us call this construction $\mathcal{T}_1$. The next lemma is proven for the implemented case.

Lemma 8.4.3. *Let* $\Pi = \mathcal{T}_1(\Pi_0)$ *be a program input to* $\mathcal{AC}(C)$_solver *and* S *be a set of ground extended r-literals that is consistent and covers all r-atoms from* Π. *Let* $Q = query(\Pi, S)$ *and* $\bar{V} = vars(Q)$. *Let c_solve(*Π_D*,S,Q,A) return true and* $\theta = a\_solution(A)$. *Let* $D = d\text{-}lits(Q)|^{\bar{V}}_{\theta}$ *and* $X = mcv\_set(\Pi)|^{\bar{V}}_{\theta}$. *Let* $r \in ground(\Pi_M)$. *If* $S \models r\text{-}lits(r)$, $X \models m\text{-}lits(r)$ *and* $D \models d\text{-}lits(r)$ *then* $head(r) \in S$.

Proof:

Consider the case when transformations tr_1 and tr_2 are performed before computing $\mathcal{P}(\Pi)$.

1. By transformation tr_2, for every rule $r_0 \in \Pi_M$, there is only one d-literal in the body of r_0. Therefore, for every $r \in ground(\Pi_M)$, there is only one d-literal in the body of r. (Note that Π_M is r-ground).

2. Let $r \in ground(\Pi_M)$ such that $S \models r\text{-}lits(r)$, $X \models m\text{-}lits(r)$ and $D \models d\text{-}lits(r)$.

   We show that $head(r) \in S$ by proof of contradiction.

3. Suppose $head(r) \notin S$.

4. By (1), there exists exactly one d-literal l in $d\text{-}lits(r)$.By (2), $l \in D$. Let d_p be d-predicate of l.

5. By construction of r, $\exists\ r_0 \in \Pi_M$ such that $r \in ground(r_0)$. Since r_0 is r-ground, we have $head(r) = head(r_0)$ and $r\text{-}lits(r) = r\text{-}lits(r_0)$. Therefore, we get $head(r_0) \notin S$ and $S \models r\text{-}lits(r_0)$.

6. By construction of Q and (5), there is a literal $\neg d_p(\bar{t}_r, \bar{V}_0) \in$ *d-lits*(Q) such that $d_p(\bar{t}_r, \bar{V}_0) =$ *d-lits*(r_0).

7. By construction of l, there is a literal $d_p(\bar{t}_r, \bar{V}_1) \in$ *d-lits*(Q) such that $l = d_p(\bar{t}_r, \bar{V}_1)|_\theta^{\bar{V}}$.

8. Since c_solve(Π_D,S,Q,A) returned true, the constraints in A on $\bar{V}$ are such that for any solution θ of A, we get $\Pi_D \cup S \cup X \cup M_c \models$ *d-lits*$(Q)|_\theta^{\bar{V}}$.

9. By (8), for any $\theta = a\_solution(A)$, we get that $\Pi_D \cup S \cup X \cup M_c \models \neg d_p(\bar{t}_r, \bar{V}_0)|_\theta^{\bar{V}}$ and $\Pi_D \cup S \cup X \cup M_c \models d_p(\bar{t}_r, \bar{V}_1)|_\theta^{\bar{V}}$.

10. By (7) and (9), we get for any θ, $l \neq d_p(\bar{t}_r, \bar{V}_0)|_\theta^{\bar{V}}$.

11. Let $\bar{C}$ be the variables in r_0. Let ζ be the substitution such that $r = r_0|_\zeta^{\bar{C}}$. Hence, $l = d_p(\bar{t}_r, \bar{V}_0)|_\zeta^{\bar{C}}$.

12. By construction of Q, $\bar{C} \subseteq \bar{V}$. Let $\theta(\bar{C})$ be the restriction of θ on $\bar{C}$. That is, $\theta(\bar{C})$ be the substitution such that it consists of only variable substitutions from θ for variables in $\bar{C}$.

13. By (10) and (11), we get that for any θ, $\theta(\bar{C}) \neq \zeta$.

14. Therefore, $\exists\, \nu \in \bar{C}$ such that $\theta(\bar{C})$ and ζ differ in their substitutions for ν. By syntactic restrictions in chapter 4, variable ν occurs in some m-atom m in r_0.

15. Let $a = m_{\theta(\bar{C})}^{\bar{C}}$ and $b = m_\zeta^{\bar{C}}$. By definition of $mc\nu\_set$, we get $m \in mc\nu\_set(\Pi)$, hence $a \in X$. By construction of r, $b \in body(r)$. By definition of X, $b \notin X$.

16. Therefore, $X \not\models$ *m-lits*(r). We reach a contradiction.

Lemma 8.4.4. *Let* Π *be a program and* B *be a set of ground extended r-literals. Let* $S = expand(\Pi_R \cup \Pi_M, B)$ *be consistent and cover all r_atoms of* Π. *Then for every literal* $l \in pos(S)$, *there exists a rule* $r \in \Pi_R \cup \Pi_M$ *with* $head(r) = l$ *such that*

- $r\_pos(r) \subseteq pos(S)$.
- $r\_neg(r) \subseteq neg(S)$.

Proof:

1. By definition of $expand$, we get $expand(\Pi_R \cup \Pi_M, B) = expand(\Pi_R \cup \Pi_M, S)$. So we use $expand(\Pi_R \cup \Pi_M, S)$ in this proof.
2. Since S is consistent, by definition of $expand$, we get $pos(S) \subseteq atmost(\Pi_R \cup \Pi_M, S)$.
3. Let $l \in pos(S)$. By (2), $l \in atmost(\Pi_R \cup \Pi_M, S)$.
4. By definition $atmost(\Pi_R \cup \Pi_M, S)$ is the deductive closure of $\alpha(\Pi_R \cup \Pi_M, S)$.
5. By (3) and (4), there is a rule $r \in \alpha(\Pi_R \cup \Pi_M, S)$ such that $l = head(r)$ and $body(r) \subseteq atmost(\Pi_R \cup \Pi_M, S)$.
6. By construction of r, $body(r) \cap neg(S) = \emptyset$. Since S covers all r-atoms, we get $body(r) \subseteq pos(S)$.
7. By construction of r, we know $\exists\ r_0 \in \Pi_R \cup \Pi_M$ such that $r = \alpha(r_0, S)$.
8. By construction of r, we get $body(r) = r\_pos(r_0)$. By (6), $r\_pos(r_0) \subseteq pos(S)$.

9. By construction of r, we also know that r_0 is not falsified by S. That is $r\_pos(r_0) \cap neg(S) = \emptyset$ and $r\_neg(r_0) \cap pos(S) = \emptyset$.

10. Since S covers all r-atoms, we get $r\_neg(r_0) \subseteq neg(S)$.

11. Therefore, $r_0 \in \Pi_R \cup \Pi_M$ with $head(r_0) = l$ such that $r\_pos(r_0) \subseteq pos(S)$ and $r\_neg(r_0) \subseteq neg(S)$.

■

Lemma 8.4.5. *Let* Π *be a program and* B *be a set of ground extended r-literals. Let* $S = expand(\Pi_R \cup \Pi_M, B)$ *be consistent and cover all r-atoms of* Π. *Let* $Q = query(\Pi, S)$ *and* $\bar{V} = vars(Q)$ *and let c_solve(*Π_D*,S,Q,A) return true and* θ *be* $a\_solution(A)$. *Let* $D = d\text{-}lits(Q)|_\theta^{\bar{V}}$ *and* $X = mcv\_set(\Pi)|_\theta^{\bar{V}}$. *Then for every literal* $l \in pos(S)$ *there exists a rule* $r \in \Pi_R \cup ground(\Pi_M)$ *such that* $l \in head(r)$ *and* $pos(S) \cup X \cup D \models body(r)$.

Proof:

1. Let $l \in pos(S)$.

2. By Lemma (8.4.4), we know that $\exists r \in \Pi_R \cup \Pi_M$ such that $l \in head(r)$ and $S \models r\text{-}lits(r)$. That is $r\_neg(r) \cap pos(S) = \emptyset$, $r\_pos(r) \cap neg(S) = \emptyset$ and $r\_pos(r) \subseteq pos(S)$.

3. Case(a): If this r belongs to Π_R, then $pos(S) \cup X \cup D \models body(r)$.

4. Case(b): If this r belongs to Π_M then we have to show that $\exists r_0 \in ground(r)$ such that $pos(S) \cup X \cup D \models body(r_0)$. (Note that Π_M is r-ground).

5. By tr_2, there exists exactly one d-literal $d \in body(r)$. Since $S \models r\text{-}lits(r)$ and $head(r) = l$ and $l \in pos(S)$, by construction of Q, $d \in d\text{-}lits(Q)$.

6. By definition of $\mathrm{mcv\_set}(\Pi)$, $m\text{-}lits(r) \subseteq \mathrm{mcv\_set}(\Pi)$.

7. Let $\bar{C}$ be the variables in r. we know that $\bar{C}$ are all c-variables. Further by construction of Q, we get $\bar{C} \subseteq \bar{V}$.

8. Let $r_0 = r|_\theta^{\bar{C}}$, where θ is a solution of A as defined above. We know $r_0 \in \mathrm{ground}(r)$.

9. By (6) and construction of r_0 and X, we get $m\text{-}lits(r_0) \subseteq X$.

10. By (5) and construction of r_0 and D, we get $d\text{-}lits(r_0) \subseteq D$.

11. By (2), (9) and (10), $\mathrm{pos}(S) \cup X \cup D \models \mathrm{body}(r_0)$.

■

Lemma 8.4.6. *Let* Π *be a program input to* $\mathcal{AC}(C)$*_solver and* B *be a set of ground extended r-literals. Let* $S = \mathrm{expand}(\Pi_R \cup \Pi_M, B)$ *be consistent. Let* L *be a set of ground r-literals such that* $\forall r \in \Pi_R \cup \Pi_M$ *with* $\mathrm{head}(r) \in L$*, if*

(a) $\mathrm{r\_pos}(r) \cap \mathrm{neg}(S) \neq \emptyset$ *or*

(b) $\mathrm{r\_neg}(r) \cap \mathrm{pos}(S) \neq \emptyset$ *or*

(c) $\mathrm{head}(r) \in \mathrm{neg}(S)$ *or*

(d) $\mathrm{r\_pos}(r) \cap L \neq \emptyset$

Then $L \cap \mathrm{atmost}(\Pi_R \cup \Pi_M, S) = \emptyset$.

Proof:

Recall that $\mathrm{atmost}(\Pi_R \cup \Pi_M, S)$ is the deductive closure of the positive program $\alpha(\Pi_R \cup \Pi_M, S)$.

1. By construction of α, we get $\alpha(r, S) = \emptyset$ for rules $r \in \Pi_R \cup \Pi_M$ which satisfy conditions (a) or (b) or (c).

2. Let R be the set of rules $r \in \alpha(\Pi_R \cup \Pi_M, S)$ with $head(r) \in L$. By (1) and conditions on L and construction of α, $\forall r \in R$, condition (d) is true.

   The program $P = \alpha(\Pi_R \cup \Pi_M, S)$ is positive. We can compute the deductive closure using iterated fix point approach [10]. Recall, $T_P \uparrow \omega$ is the deductive closure of P where $T_P \uparrow 0 = \emptyset$, $T_P \uparrow k = T_P(T_P \uparrow k-1)$ and $T_P(I) = \{l \mid P$ contains a rule r such that $head(r) = l$ and $body(r) \subseteq I\}$.

3. Suppose $L \cap atmost(\Pi_R \cup \Pi_M, S) \neq \emptyset$. Therefore, there exists a literal $l \in L$ which was first added in the computation of $T_P \uparrow \omega$. Let us say it was added at $T_P \uparrow k$.

4. Since l is the first literal from L added to $T_P \uparrow \omega$, we have $L \cap T_P \uparrow k-1 = \emptyset$.

5. By (3) and definition of $T_P \uparrow k$, $\exists r \in P$ such that $head(r) = l$ and $body(r) \subseteq T_P \uparrow k-1$.

6. By construction of R, the rule r is in R. By (2), condition (d) is satisfied by r. That is $r\_pos(r) \cap L \neq \emptyset$.

7. By (5) and (6), we get $L \cap T_P \uparrow k-1 \neq \emptyset$.

8. By (4), we reach a contradiction. Hence, $L \cap atmost(\Pi_R \cup \Pi_M, S) = \emptyset$.

■

Lemma 8.4.7. *Let* Π *be a program input to* $\mathcal{AC}(C)$*_solver and* B *be a set of ground extended r-literals. Let* $S = expand(\Pi_R \cup \Pi_M, B)$ *be consistent and cover all r-atoms of* Π*. Let* $Q = query(\Pi, S)$ *and* $\bar{V} = vars(Q)$ *and let*

c_solve(Π_D*,S,Q,A) return true and* θ *be* $a\_solution(A)$. *Let* $D = d\text{-}lits(Q)|_\theta^{\bar{V}}$ *and* $X = mcv\_set(\Pi)|_\theta^{\bar{V}}$. *Let* L *be a set of ground r-literals such that* $\forall r \in \Pi_R \cup ground(\Pi_M)$ *with* $head(r) \in L$, *if*

(a_1*)* $S \cup X \cup D \cup M_c \not\models body(r)$ *or*

(a_2*)* $r\_pos(r) \cap L \neq \emptyset$

Then $L \cap atmost(\Pi_R \cup \Pi_M, S) = \emptyset$.

Proof:

We use lemma 8.4.6 to prove this lemma. It suffices to show that the conditions in lemma 8.4.6 for program $\Pi_R \cup \Pi_M$ are satisfied. We use the conditions (a_1) and (a_2) satisfied by program $\Pi_R \cup ground(\Pi_M)$ to show this.

1. Let $r \in \Pi_R$ with $head(r) \in L$.
   (a) Case(i): Let condition (a_1) be satisfied by r. Since r contains only r-literals, we get $S \not\models body(r)$. Hence, $r\_pos(r) \cap neg(S) \neq \emptyset$ or $r\_neg(r) \cap pos(S) \neq \emptyset$. Therefore condition (a) or (b) in lemma 8.4.6 is satisfied by r.
   (b) Case(ii): Let condition (a_2) be satisfied by r. Condition (d) in lemma 8.4.6 is satisfied by r.
2. Let $r \in \Pi_M$ with $head(r) \in L$. Let R be the set of all groundings of r.
   (a) Suppose rule r satisfies condition (d) from lemma 8.4.6, then we are done.
   (b) Suppose rule r does not satisfy condition (d) from lemma 8.4.6. we show that (a) or (b) or (c) is satisfied by r.

(c) By (2b) we know $\forall r_0 \in R$, (a_2) is not satisfied. Since $head(R) \in L$ and $r_0 \in ground(\Pi_M)$, $\forall r_0 \in R$, (a_1) is satisfied. That is $S \cup X \cup D \cup M_c \not\models r\text{-}lits(r_0) \cup m\text{-}lits(r_0) \cup d\text{-}lits(r_0)$.

(d) Case(i): Let $S \not\models r\text{-}lits(r)$. Since S covers all r-atoms of Π, we get $r\_pos(r) \cap neg(S) \neq \emptyset$ or $r\_neg(r) \cap pos(S) \neq \emptyset$. Therefore condition (a) or (b) in lemma 8.4.6 is satisfied by r.

(e) Case(ii): Let $S \models r\text{-}lits(r)$. By definition of R, we get $\forall r_0 \in R$, $S \models r\text{-}lits(r_0)$.

(f) By (2c) and (2e), we get $\forall r_0 \in R$, $X \cup D \cup M_c \not\models m\text{-}lits(r_0) \cup d\text{-}lits(r_0)$.

(g) By construction of r and r_0, conditions (a) or (b) or (d) of lemma 8.4.6 are not satisfied by r. Now we show that condition (c) of lemma 8.4.6 is satisfied by r by a proof of contradiction.

(h) Suppose (c) of lemma 8.4.6 is not satisfied. That is $head(r) \notin neg(S)$. We show that this case is not possible by constructing a rule r' in R such that $S \cup X \cup D \cup M_c \models body(r')$ thus reaching a contradiction from (2c).

(i) By (2e) $S \models r\text{-}lits(r)$. Since S covers all r-atoms, by (2h), we get $head(r) \in pos(S)$.

(j) By construction of Q and tr_2, since $head(r) \in S$ and $S \models r\text{-}lits(r)$ then $\exists d \in d\text{-}lits(Q)$ such that $d = d\text{-}lits(r)$.

(k) Let $\bar{C}$ be the variables in r. By construction $\bar{C} \subseteq \bar{V}$. Consider the rule $r' = r|_{\theta}^{\bar{C}}$.

(l) By construction of r' and D, we get $d\text{-}lits(r') \subseteq D$.

(m) Since $m\text{-}lits(r) \subseteq mcv\_set(\Pi)$, by construction of r' and X, we get $m\text{-}lits(r') \subseteq X$.

(n) Since $S \models$ *r-lits*(r), we get $S \models$ *r-lits*(r').

(o) By (2l), (2m) and (2n), we get $S \cup X \cup D \cup M_c \models body(r')$.

(p) Since $r' \in R$, by (2c) we reach a contradiction.

3. We showed that $\forall r \in \Pi_R \cup \Pi_M$, with $head(r) \in L$, at least one of the condition from (a) to (d) in lemma 8.4.6 is satisfied.

4. By lemma 8.4.6, we get $L \cap atmost(\Pi_R \cup \Pi_M, S) = \emptyset$.

Proposition 5.5.6: *Let a program* Π *and a set of extended r-literals* B *be inputs to* $\mathcal{AC}(C)$*_solver. Let*

(s_1) $S = expand(\Pi_R \cup \Pi_M, B)$ *is consistent and covers all r-atoms of* Π

(s_2) $Q = query(\Pi, S)$, $\bar{V} = vars(Q)$ *and* $c\_solve(\Pi_D, S, Q, A)$ *returns* true *at step (d) of* $\mathcal{AC}(C)$*_solver*

(s_3) $\theta = a\_solution(A)$, $D =$ *d-lits*$(Q)|^{\bar{V}}_{\theta}$, $X = mcv\_set(\Pi)|^{\bar{V}}_{\theta}$ *and* M_c *is the intended interpretation of c-atoms(*Π*).*

Then $pos(S) \cup D \cup X \cup M_c$ *is an* asp*-answer set of* $\Pi_R \cup \Pi_M \cup D \cup X \cup M_c$ *agreeing with* B*.*

Proof:

1. Let $Y = pos(S) \cup D \cup X \cup M_c$.

2. By statement (s_1), $pos(S)$ agrees with B. By definition of Y, r-atoms(Y) $= pos(S)$. Therefore by definition of *agrees*, Y agrees with B.

3. Now we show that Y is an asp-answer set of $\Pi_R \cup \Pi_M \cup D \cup X \cup M_c$.

4. (3) is true iff Y is an asp-answer set of ground program $P = \Pi_R \cup ground(\Pi_M) \cup D \cup X \cup M_c$.

5. (4) is true iff Y is the minimal set closed under the rules of P^Y.

6. First, we show that Y is closed under P^Y.

   (a) By definition of reduct, Π_R, Π_M and Y,
   $P^Y = \Pi_R^{pos(S)} \cup ground(\Pi_M)^{pos(S)} \cup D \cup X \cup M_c$.

   (b) By definition of Y, we get Y is closed under rules of $D \cup X \cup M_c$.

   (c) Since statement s_1 is true, we use lemma 8.4.1 to conclude that $pos(S)$ is closed under rules of $\Pi_R^{pos(S)}$. By construction of Y and $\Pi_R^{pos(S)}$, Y is closed under rules of $\Pi_R^{pos(S)}$.

   (d) Now using lemma 8.4.2, we show that Y is closed under rules of $ground(\Pi_M)^Y$.

      i. Let $r \in ground(\Pi_M)^{pos(S)}$ such that $body(r) \subseteq Y$. We need to show that $head(r) \in Y$.

      ii. By definition of reduct, $\exists r_0 \in ground(\Pi_M)$ such that $r = r_0^{pos(S)}$ and $r\_neg(r_0) \cap pos(S) = \emptyset$.

      iii. Since S covers all r-atoms of Π, we get $r\_neg(r_0) \subseteq neg(S)$.

      iv. Since $body(r) = r\_pos(r_0) \cup$ *m-lits*$(r_0) \cup$ *d-lits*(r_0), by (i) we get $r\_pos(r_0) \subseteq pos(S)$, *m-lits*$(r_0) \subseteq X$ and *d-lits*$(r_0) \subseteq D$.

      v. By (iii) and (iv), $S \models$ *r-lits*(r_0), $X \models$ *m-lits*(r_0) and $D \models$ *d-lits*(r_0).

      vi. By lemma (8.4.2), $head(r_0) \in S$. Since $head(r_0) = head(r)$, by construction of Y, we get $head(r) \in Y$.

   (e) Y is closed under P^Y.

7. Now we show that Y is minimally closed under P^Y.

8. Let $Z \subset Y$ be minimally closed under P^Y.

9. Since $X \cup D \cup M_c$ are facts in P^Y and they do not appear as heads of any other rule in P^Y, $X \cup D \cup M_c \subseteq Z$.

10. Therefore we can represent Z as $Z = Z_0 \cup X \cup D \cup M_c$ such that $Z_0 \subset pos(S)$.

11. Let $L = pos(S) \setminus Z_0$. L is a set of ground r-literals.

12. Now we show that L satisfies conditions of lemma 8.4.7.

    (a) Suppose r is a rule from $\Pi_R \cup ground(\Pi_M)$ with $head(r) \in L$.

    (b) We show that r satisfies at least one of the conditions (a_1) or (a_2) of lemma 8.4.7.

    (c) Suppose $r^Y = \emptyset$. By definition of reduct, $r\_neg(r) \cap pos(S) \neq \emptyset$. Therefore, $pos(S) \cup X \cup D \cup M_c \not\models body(r)$. Condition (a_1) of lemma 8.4.7 is satisfied by r.

    (d) Suppose $r_0 = r^Y \neq \emptyset$. Then $r_0 \in P^Y$.

    (e) Case(i): Suppose $body(r_0) \not\subseteq Y$. Then $Y \not\models body(r)$. Therefore, $pos(S) \cup X \cup D \cup M_c \not\models body(r)$. Condition (a_1) of lemma 8.4.7 is satisfied by r.

    (f) Case(ii): Suppose $body(r_0) \subseteq Y$. We will show that r satisfies condition (a_2) of lemma 8.4.7.

    (g) Since $head(r_0) \in L$, we get $head(r_0) \notin Z$. By (8), $body(r_0) \not\subseteq Z$.

    (h) Since $body(r_0) \subseteq Y$, $\exists\, l \in Y \setminus Z$ such that $l \in body(r_0)$. Since r_0 is from a reduct, $l \in pos(S)$.

    (i) Since $l \in pos(S)$ and $l \notin Z$, by construction of L, $l \in L$. Hence, $r\_pos(r_0) \cap L \neq \emptyset$. Therefore, $r\_pos(r) \cap L \neq \emptyset$. Hence, r satisfies condition (a_2) of lemma 8.4.7.

(j) We can conclude from lemma 8.4.7 that $L \cap atmost(\Pi_R \cup \Pi_M, S) = \emptyset$.

13. By definition of expand, if $l \notin atmost(\Pi_R \cup \Pi_M, S)$ then *not* $l \in S$. Therefore $L \subseteq neg(S)$, but we know $L \subseteq pos(S)$. By (s_1), we reach a contradiction.

14. Therefore, Y is minimally closed under P^Y. BY (3), (4) and (5), Y is an answer set of $\Pi_R \cup \Pi_M \cup X \cup D \cup M_c$.

■

Proposition 5.2.1: *Let* Π *be a program and* B *be a set of ground extended literals input to ACengine. If ACengine returns* true *and a set* A *then* A *is a simplified answer set of* Π *agreeing with* B.

The proof of this proposition follows from Propositions 5.5.5 and 5.5.6.

■

CHAPTER 9

RELATED WORK

In this chapter we are interested in describing related works that integrate different reasoning techniques to compute answer sets of programs in ASP languages, in particular integration to CLP techniques.

9.1 Language ASP-CLP

In [37], the authors introduce a framework called ASP-CLP, which is an extension of *A-Prolog* to generic constraint domains. The syntax and semantics are a natural extension of *A-Prolog*. The main objective of their work was to address the aggregation capabilities within ASP. Towards this, they develop and study an instance of ASP-CLP called ASP-CLP(Agg) specialized to constraint theory of aggregates.

An ASP-CLP clause (or rule) in ASP-CLP is of the form:

$$A :- C \mid B_1, \ldots, B_k \tag{9.1}$$

where A is an asp atom (regular atom), $B_1, \ldots, B_k$ are asp (regular) literals and C is an arbitrary conjunction of primitive constraints and their negation (called C-constraint) over a constraint domain $\mathcal{C}$.

A $program$ in ASP-CLP is a set of ASP-CLP clauses. The language ASP-CLP over a constraint domain $\mathcal{C}$ does not classify predicates as in language $\mathcal{AC}(\mathcal{C})$. ASP-CLP rules have syntactic similarities to the mixed rules of $\mathcal{AC}(\mathcal{C})$: the C-constraints in the body of ASP-CLP rules are conjunction of primitive constraints over a constraint domain and c-atoms of mixed rules of $\mathcal{AC}(\mathcal{C})$ are conjunctions of primitive constraints over a constraint domain. The asp literals in

the bodies of ASP-CLP rules are like regular literals in the body of mixed rules of $\mathcal{AC}(\mathcal{C})$. The marked difference between the two is the presence of mixed atoms in the body of $\mathcal{AC}(\mathcal{C})$ mixed rules. These mixed atoms act like connections of relations between regular and constraint predicates. The C-constraints in the body of ASP-CLP rule, directly connects with the asp literals in the body. Therefore, for a generic constraint domain, this direct connection forces the grounding of constraints along with the other asp literals in the body of the rules (unless asp literals are also not ground).

Now we study semantic relation between arbitrary ASP-CLP and $\mathcal{AC}(\mathcal{C})$ languages defined over a same constraint domain. Let L_1 be an $\mathcal{AC}(\mathcal{C})$ language over a constraint domain $\mathcal{C}$. Let L_2 be a language of ASP-CLP over the same constraint domain $\mathcal{C}$. Let Π be an L_1 program, and Π_a be the translation of Π to *A-Prolog*. Program Π_a is a valid program of L_2 (Π_a contains choice rules but authors in [37] allow such rules in their language and view them as short hand for normal rules). Note that the program Π_a contains constraint atoms from the constraint domain $\mathcal{C}$. Let M_c be the intended interpretation of the constraint predicates in Π. For instance, if $>$ is a constraint relation over a domain of integers then M_c contains $\{1 < 2, 1 < 3, \ldots 2 < 3, \ldots\}$. *The answer sets of* $\Pi_a \cup M_c$ *with respect to asp semantics are in one to one correspondence with answer sets of* Π_a *in ASP-CLP semantics.*

A particular instance that is studied in [37] is language ASP-CLP(Agg), an extension of *A-Prolog*, where the constraint domain is for aggregates (like sum, $count$, min, max functions over sets and multi-sets). Aggregates have been shown to significantly improve the compactness and clarity of programs in logic programming and there is a lot of work in including aggregates in ASP [29, 95, 102, 30, 48].

A solver is implemented for computing ASP-CLP answer sets of ASP-CLP(Agg) programs. The solver tightly integrates ASP reasoning techniques of ASP solver $Smodels$ [75] and finite domain constraint solving techniques of CLP Solver ECLiPSe [2]. The ASP-CLP(Agg) system consists of two levels of computation. The first level is a preprocessor which uses $lparse$ to ground all the literals in the program except the aggregate literals. The aggregate literals are rewritten as constraints understood by the language of ECLiPSe. The rewriting of aggregates depends on the ground atoms involved with the aggregate. The second level is the inference engine which computes ASP-CLP answer sets of the ground ASP-CLP(Agg) program. For each ground atom a, the engine introduces a new boolean domain variable X_a in ECLiPSe. For each aggregate literal, a new boolean domain variable is introduced and is used to describe the aggregate as a constraint in ECLiPSe. For instance, if extension of p is $p(3)$, $p(5)$, $p(7)$, then an aggregate $count(\{ X : p(X) \}) \geq 2$ is written as:

$$X_3 : 0..1,\ X_5 : 0..1,\ X_7 : 0..1,\ X_3 + X_5 + X_7 \geq 2$$

where variables X_3, X_5 and X_7 denote atoms $p(3)$, $p(5)$ and $p(7)$ respectively. The information exchange between Smodels and ECLiPSe is two ways. The Smodels system is capable of posting constraints to the ECLiPSe system. The ECLiPSe system communicates Smodels for either sending the truth values of a posted completed aggregate constraint (truth values of the dependent atoms of the aggregate literal are decided and therefore the truth value of aggregate literal can be computed) or for sending back values of labeled variables appearing in a constraint corresponding to non-completed aggregates.

Systems $\mathcal{AD}solver$ and ASP-CLP(Agg) allow different constraint domains (difference constraints vs aggregates) and work with different constraint solvers

($\mathcal{D}$solver vs ECLiPSe) but the underlying inference engine of $\mathcal{AD}$*solver* and system ASP-CLP($\mathcal{A}$gg) tightly couple ASP reasoning techniques with constraint solving techniques. A higher level difference between the two implementations is that ASP-CLP($\mathcal{A}$gg) system has the advantage of returning truth values for atoms other than aggregate literals. This allows for computing more consequences at each step. The disadvantage of this approach is that when there is a conflict in Smodels (ASP engine), then backtrack occurs in two places: (a) Smodels chosen atoms's truth value or (b) ECLiPSe, the solver needs to find another solution. In $\mathcal{AD}$*solver*, the *expand_dc* function also uses $\mathcal{D}$solver to compute consequences of r-literals but $\mathcal{D}$solver is not used to guess or return truth values of r-literals. This method has the advantage that backtracking occurs only at one place: ASP engine. During backtracking the constraint store is just relaxed, and there is no need to find a different answer as we know there exists a solution for the relaxed constraint store. Another difference is that, the constraints in ASP-CLP($\mathcal{A}$gg) are always on boolean variables but can be complex constraints. The constraints of $\mathcal{AD}$*solver* are all difference constraints but on variables ranging over large numerical domains. The solver $\mathcal{AC}$*solver* is more general and allows different types of constraints. Even in $\mathcal{AC}$*solver* backtracking occurs only on the ASP reasoning side.

9.2 Language CASP

Consider an instance of $\mathcal{AC}(\mathcal{C})$ with the following properties:

- ▷ there are no defined predicates in the language and
- ▷ for any rule r, if mixed atoms or constraint atoms are in the body of r then head of r is empty.

This language is called CASP [12]. There are two different semantics for CASP called (a) strong semantics and (b) weak semantics. Strong semantics is similar to our semantics for $\mathcal{AC}(\mathcal{C})$. Let Π be a program in language CASP. Let X be a candidate mixed set and M_c be the intended interpretation of constraint predicates in Π. *The* $\mathcal{AC}(\mathcal{C})$ *answer sets of* Π *have one to one correspondence with CASP strong semantic answer sets of* Π.

An answer set under strong semantics of CASP is an answer set under weak semantics of CASP. In strong semantics, the set of mixed atoms in any answer set is a candidate mixed set, that is, for every r-ground mixed atom of Π, there exists exactly one ground mixed atom in the answer set. In weak semantics, this condition is relaxed and the set of mixed atoms in any answer set is a candidate mixed set or its superset.

Both $\mathcal{AD}$*solver* and CASP algorithms integrate ASP reasoning techniques with constraint solving techniques. The CASP algorithm for computing answer sets of CASP programs uses constraint solving techniques to check for consistency of the constraint store. The $\mathcal{AD}$*solver* algorithm for computing answer sets of $\mathcal{AC}_0$ programs uses constraint solving techniques to check for consistency of the constraint store and also to compute new consequences (r-literals) in the $expand\_dc$ function. This allows for deciding on the truth values of some r-atoms earlier in computation of an answer set.

Unlike the tightly coupled $\mathcal{AD}$*solver*, CASP solver loosely couples an off the shelf ASP inference engine Smodels [75] and finite domain constraint solver GNU Prolog [31]. Given a program Π, the CASP solver first computes all the answer sets of regular part of Π using Smodels and then for each of the answer set, the constraints are fed to the constraint solver to check for consistency and to generate a solution. If there is no solution for the set of constraints then the system uses a

different answer set and feeds the constraint solver with a new set of constraints. The advantage of CASP solver is that it allows more general constraints unlike $\mathcal{AD}solver$ which only allows difference constraints. Further, $\mathcal{AD}solver$ has the advantage of using incremental constraint solving techniques. A new CASP solver is now being built by tightly coupling ASP and constraint solving engines.

9.3 ASP and SAT solvers

ASP took a new turn with work in developing algorithms and building solvers that use an underlying satisfiability (SAT) solver to compute answer sets for *A-Prolog* programs. There are two solvers that compute answer sets of *A-Prolog* programs using SAT solvers: $ASSAT$ [68] and $Cmodels$ [66].

Given a *A-Prolog* program Π, these solvers ground Π using an intelligent grounder like $lparse$, then $ground(\Pi)$ is transformed to a satisfiability problem whose solutions have one to one correspondence with answer sets of Π. The solutions to the satisfiability problem are computed using SAT solvers. Since SAT solvers have been studied well, this method of computing answer sets for *A-Prolog* programs is quite efficient [66].

This approach has the disadvantage that *A-Prolog* programs need to be fully grounded before computing answer sets. Therefore, for programs containing variables ranging over large numerical domains, the solvers might behave the same way traditional ASP solvers do.

Towards building complex applications, SAT solvers have been integrated with other theorem solvers to handle arithmetic (constraint solving) and other decidable theories. Such solvers built are called *Satisfiability Modulo Theorem* (SMT) solvers.

9.3.1 Satisfiability Modulo Theories

Satisfiability modulo theories (SMT) can be seen as an extended form of propositional satisfiability, where propositions are either simple boolean propositions or constraints in a specific theory [15]. SMT is a decision problem for logical formulas with respect to combination of background theories and finding whether such formulas are satisfiable. For instance, an example of SMT formula is: $X \vee Y \wedge 3 * A + 4 * B \leq 7$ where X and Y are boolean variables and A, B are variables ranging in real domain.

An SMT solver based on theory T would tightly integrate a SAT solver with a theory-based solver for T. For instance, a SMT solver based on difference logic would tightly integrate a SAT solver with a difference constraint solver. The SAT solver interacts with the theory solver through a well defined interface. The SAT solver sends conjunctions of atoms and the theorem solver checks for feasibility (consistency) of these atoms put together. The theorem solver then returns $true$ or $false$ to the SAT solver. If the conjunction of the predicates is satisfiable then the theorem solver also returns a valid solution for the set of variables in the predicates. The SAT solver continues searching for a solution or backtracks depending on the return value of theorem solver. For SMT solvers to be efficient, the underlying theorem solver should be incremental and backtrackable.

Some current SMT solvers are MathSAT [15] and Ario [89]. MathSAT is a DPLL-based decision procedure for the SMT problem for various theories, including those of Equality and Uninterpreted Function (EUF), Separation Logic (SEP), Linear Arithmetic over the Reals (LA(R)) and Linear Arithmetic over the Integers (LA(Z)).

We believe SMT solvers can be used to compute answer sets of $\mathcal{V}(\mathcal{C})$ programs by integrating answer set reasoning techniques of ASP with satisfiability and

constraint solving techniques of a SMT solver.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

This dissertation work investigates the integration of different reasoning techniques to compute the answer sets of programs in ASP paradigm. In particular, we investigate the integration of *answer set reasoning* from *A-Prolog* solvers [75, 72], *a form of abduction* from $CR-Prolog$ solvers [4], *resolution* from CLP [58] and *constraint solving* techniques from CLP. Our work is a significant step to declaratively solve problems that cannot be solved by pure ASP or CLP solvers for example the USA-Advisor extension with time constraints. To our knowledge, the solvers built are the first to *tightly* integrate different reasoning techniques to compute answer sets from *partially ground* programs. The CASP solver [12] loosely couples a ASP reasoning system with a constraint solver. The ASP-CLP(Agg) system [37] tightly couples ASP reasoning with constraint reasoning but needs to compute the whole ground instantiation of atoms in the program.

The contribution of this work is enumerated as follows:

1. We developed a collection of languages $\mathcal{AC}(\mathcal{C})$ parameterized over $\mathcal{C}$.
   - ▷ Syntax of $\mathcal{AC}(\mathcal{C})$ is an extension to syntax of *A-Prolog*.
   - ▷ Semantics of $\mathcal{AC}(\mathcal{C})$ is a natural extension of semantics of *A-Prolog*.
2. We designed an algorithm $\mathcal{AC}solver$, to compute answer sets of a class of programs in $\mathcal{AC}(\mathcal{C})$ and prove its correctness. The algorithm computes answer sets of programs from their partial ground instantiations.
   - ▷ Partial grounder uses intelligent grounding mechanisms from Lparse [99].

- ▷ The algorithm tightly couples answer set reasoning, a form of abduction and CLP's resolution and constraint solving techniques.

3. We implemented $\mathcal{AC}solver$ for the constraint domain of real numbers.
   - ▷ We used a CLP(R) solver to tightly integrate resolution and constraint solving techniques.
4. We designed a collection of languages $\mathcal{V}(\mathcal{C})$ parameterized over $\mathcal{C}$. These languages allow consistency restoring capabilities of *CR-Prolog* [3].
5. We study an instance of $\mathcal{V}(\mathcal{C})$ called $\mathcal{AC}_0$ and an instance of $\mathcal{AC}(\mathcal{C})$, $\mathcal{AC}(\mathcal{R})$. The study concentrates on knowledge representation methodologies and reasoning capabilities of $\mathcal{AC}_0$ and $\mathcal{AC}(\mathcal{R})$.
6. We designed an algorithm $\mathcal{AD}solver$, to compute answer sets of programs in $\mathcal{AC}_0$ and prove correctness. The algorithm computes answer sets of $\mathcal{AC}_0$ programs from their partial ground instantiations.
   - ▷ The partial grounder uses intelligent grounding mechanisms from Lparse.
   - ▷ The algorithm tightly couples answer set reasoning, a form of abduction and difference constraint solving techniques.
   - ▷ The implemented difference constraint solver $\mathcal{D}$solver is incremental.
7. We implemented $\mathcal{AD}solver$ to tightly integrate answer set reasoning mechanisms and constraint solving techniques.
8. We showed efficiency of $\mathcal{AD}solver$ over classical *ASP* and *CR-Prolog* solvers for planning with temporal constraints using an extension of system USA-Advisor.

10.1 Future Work

There are several directions the current work can be expanded. Some of these future works are implementation related, some are research related and some are experimental studies. We discuss and enumerate these accordingly.
Some of the implementation related future works are as follows:

1. *$\mathcal{AC}$solver* can be improved by changing the underlying CLP solver to be incremental. Though the implementation is tight, the current solver queries a set of goals every time from scratch. The time needed to search for a successful derivation sequence starting from the beginning will be much greater when compared to searching only for new goals added to the query by using a previous solution, especially when resolution techniques are used in the sequence.

2. An improvement of *$\mathcal{AC}$solver* would be to send or input the answer constraints to a constraint solver and get solutions. As the underlying CLP engine in *$\mathcal{AC}$solver* returns only answer constraints (primitive constraints on c-variables), *$\mathcal{AC}$solver* outputs these constraints. Though the answer constraints ensure that there is a solution, it would be nice to see some solution substitutions to these variables. This can be done by simply forwarding the output answer constraints to any constraint solver and getting a result.

3. *$\mathcal{AD}$solver* and *$\mathcal{AC}$solver* use Surya as an underlying inference engine. Surya was built to support *Extended Evaluation Rule* (EER), which is an inference mechanism [72] to deduct inconsistencies. EER efficiently retrieves information spread across several rules of a program. For the current version of the solvers the EER mechanism is switched off since it needs some

additional implementation to include it. Adding EER would improve the efficiency of $\mathcal{AD}$*solver* and $\mathcal{AC}$*solver* whenever EER is used and would perform well otherwise too.

4. $\mathcal{AC}$*solver* can be modified to allow r-literals in the body of defined rules. This increases the expressiveness allowed by defined rules. For instance, the continuous time problem in TAG [46] can be naturally represented using $\mathcal{AC}(\mathcal{R})$ but $\mathcal{AC}$*solver* needs to be extended to allow r-literals in the body of defined rules to be able to compute answer sets. This requires more modification of the underlying CLP system of $\mathcal{AC}$*solver*.

5. Currently, $\mathcal{AC}$*solver* allows only variables as arguments in the place of c-terms in mixed atoms. This restriction can be removed by allowing c-terms.

Some of the research related future works are as follows:

1. $\mathcal{AC}$*solver* algorithm does not allow a cycle between definition of a defined literal and a regular literal. In a standard algorithm [75, 72] to compute answer sets of *A-Prolog* programs, cycles between r-literals are reasoned using function $atmost$. Function $atmost$ computes the upper closure of the program with respect to the partial answer set built so far. If two r-literals are in a cycle and get support from each other, then such r-literals do not belong to the upper closure and therefore their negations are added to the partial answer set. If there is a cycle between definitions of d-literals then the CLP operational semantics goes into an infinite loop. If there are cycles between d-literals and r-literals in a program, then neither the $atmost$ function detects it or the function $csolve$ detects its. This requires constant check whether the d-literals and r-literals are supporting each other through

cycles. For this, there needs to be an efficient mechanism. Consider the following example:

Example 10.1.1. *Let* q *and* r *be regular predicates,* at *be a mixed predicate and* d *be a defined predicate. Let* Π *be a program as follows:*

```
q(a).     q(b).
#csort time(0..1000).
r(X) :- q(X), at(X, T), d(X, T).
d(X,T) :- r(X), 3 * T < 1000.
```

Notice that there is a cycle between definitions of r *and* d.

The current algorithm would return four simplified answer sets:

(a) $\{q(a), q(b), at(a, V1), at(b, V2), 0 \leq V1 \leq 1000, 0 \leq V2 \leq 1000\}$

(b) $\{q(a), q(b), at(a, V1), at(b, V2), r(a), 0 \leq V1 < 333.333, 333.333 \leq V2 \leq 1000\}$

(c) $\{q(a), q(b), at(a, V1), at(b, V2), r(b), 333.333 \leq V1 \leq 1000, 0 \leq V2 < 333.333\}$

(d) $\{q(a), q(b), at(a, V1), at(b, V2), r(a), r(b), 0 \leq V1 < 333.333, 0 \leq V2 < 333.333\}$

The only simplified answer set of the above program is (a). The others are supported but not minimal. The wrong answer sets are returned because of the cycle between regular and defined rules. One trivial way to check for these cycles and a literal's support is by using function atmost also on defined rules. This requires grounding the defined rules and is not what we would like to have. Alternate method is to use a dependency graph to find

such supportedness. Notice that a dependency graph for predicates in the program is not enough. For instance, if we add $r(a)$ to the above program as a fact then just a dependency graph on predicates is not enough anymore. we need a dependency graph for the whole ground program which is not feasible when constraint variables have a large domain. It seems like we need something in the middle, a dependency graph between predicate symbols and then a further check to ensure literal support. We are not sure where such type of rules will be required for knowledge representation. More investigation is required.

2. Consider a collection of languages $\mathcal{L}(\mathcal{C})$ obtained from $\mathcal{V}(\mathcal{C})$ by removing cr-rules. Recently ASP solvers have been built which use SAT solvers to compute answer sets. The underlying SAT solver can be replaced with an SMT solver that has constraint solving techniques built into it. Such solvers can be used to compute answer sets of $\mathcal{L}(\mathcal{C})$ programs. In $\mathcal{AD}solver$, changing the underlying constraint solver for a different constraint domain would give rise to a new solver for a language in $\mathcal{V}(\mathcal{C})$. This is true for such SMT based answer set solvers too. The partial grounder $\mathcal{P}\mathrm{ground}_{\mathrm{d}}$ can be directly used for getting the input language for such solvers. SAT based solvers for computing answer sets add loop formulas [68] to the programs in order to compute minimal supported models. Though these loop formulas do not add extra variables to the program, sometimes the number of loop formulas to be added can grow exponential. In $\mathcal{L}(\mathcal{C})$ programs, the constraint atoms are in the body of denials, therefore they do not give rise to new loop formulas as these c-rules will not form a loop. For an SMT based answer set solver for computing answer sets of $\mathcal{L}(\mathcal{C})$ programs, the loop formulas are only for the regular part of the program. This gives rise to the

reasoning that this work might just require replacing the underlying SAT solver by an SMT solver but more investigation is needed. Consider $\mathcal{AC}(C)$ languages where there are no defined predicates. Such languages can contain r-literals in the head of mixed rules. These SMT based solvers can be easily extended to such languages but new kind of loop formulas for the middle rules will be required. Building solvers for $\mathcal{AC}(C)$ languages might be more challenging because there is no resolution involved with SMT solvers and grounding the defined rules would give rise to a large ground instantiation.

3. Abductive reasoning techniques integrated from CR-Prolog solvers, act as a meta layer for the computation of answer sets. For a class of CR-Prolog languages, this reasoning can be tightly coupled with ASP reasoning techniques. This might dramatically improve the efficiency of the solvers for programs containing cr-rules. Further study is needed to build such techniques.

4. The $csolve$ function in $\mathcal{AC}solver$ algorithm can be improved by modifying the $cneg$ derivation. The current definition can be extended to capture the r-literal consequences in the derivation for partial answer sets. This would allow us to query negative literals even when the set of r-literals computed so far is just partial (does not cover all atoms in the program).

5. Notice that the current $\mathcal{AC}solver$ algorithm developed does not allow mixed atoms in the head of rules. This makes it impossible to write inertia for mixed atoms. This is okay for programs where we view mixed atoms as functions. We can look at a syntactically restricted class where we can allow mixed atoms in the head. For instance we can write a rule as

   ```
   % John takes half hour to eat.
   ```

```
at(S, end, T2) :- o(eat(john),S), at(S, start, T1), T2 = T1+30.
```

This would require the inference system to send the constraint T2 = T1+30, to the underlying constraint solver. Though, this can look simple, some of our proofs use the fact that mixed atoms do not occur in the head of rules. They need to be studied and proven again to be sure the algorithm works fine.

Some of the experimental studies that need to be done are as follows:

1. $\mathcal{AD}solver$ does not use constraint solving techniques during execution of $lookahead$ function [75]. Lookahead function is used for finding immediate inconsistencies and thus saving a lot of wasted computation time. Though lookahead is exhaustive, most times it is very helpful and gives good efficiency. $\mathcal{AD}solver$ does not use constraint solving mechanisms during the execution of lookahead because these techniques might be slower and would thereby decrease the efficiency of the solver. This might not be true. We need to investigate more by running examples and studying the effects of running lookahead with and without constraint solving.

2. A different kind of lookahead is possible for the constraint atoms in $\mathcal{AD}solver$. Just like lookahead for regular atoms which finds immediate inconsistencies, we can add a new lookahead for c-atoms which deduct immediate inconsistencies of c-atoms. Given a set of constraints C in a store, we can include a new constraint c to the store and check for feasibility. If the store $C \cup \{c\}$ is not feasible then we can conclude $\neg c$. This conclusion would trigger new consequences on regular literals. This approach may work very well just like lookahead for regular literals, but more experimental study should be done.

3. Using a new lookahead for c-atoms allows us to backjump on the c-atoms when inconsistency is found. This can dramatically increase the efficiency of the solver. More experimental study should be done to see when such backjumping helps efficiency.

BIBLIOGRAPHY

[1] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. *In J. Minker, editor, Foundations of Deductive Databases and Logic Programming*, pages 89–148, 1988.

[2] Krzysztof Apt and Mark Wallace. *Constraint Logic Programming Using ECLiPSe*. Cambridge University Press, 2006.

[3] Marcello Balduccini. *Answer Set Based Design of Highly Autonomous, Rational Agents*. PhD thesis, Texas Tech University, Dec 2005.

[4] Marcello Balduccini. CR-models: An inference engine for CR-prolog. In *Logic Programming and Nonmonotonic Reasoning*, May 2007.

[5] Marcello Balduccini, Michael Gelfond, and Monica Nogueira. A-prolog as a tool for declarative programming. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000)*, pages 63–72, 2000.

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

[7] Marcello Balduccini, Michael Gelfond, Monica Nogueira, and Richard Watson. The usa-advisor: A case study in answer set planning. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 439–442, Sep 2001.

[8] Marcello Balduccini, Michael Gelfond, Monica Nogueira, and Richard Watson. Planning with the usa-advisor. In David Kortenkamp, editor, *3rd*

NASA International workshop on Planning and Scheduling for Space, Oct 2002.

[9] Marcello Balduccini and Monica Nogueira. Usa-advisor source code. http://www.krlab.cs.ttu.edu/Software/Download/rcs/, Dec 2007.

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

[11] Chitta Baral and Michael Gelfond. Reasoning agents in dynamic domains. In *Workshop on Logic-Based Artificial Intelligence*. Kluwer Academic Publishers, Jun 2000.

[12] S. Baselice, P. A. Bonatti, and Michael Gelfond. Towards an integration of answer set and constraint solving. In *In Proceedings of ICLP*, pages 52–66, 2005.

[13] J. Bellone, A. Chamard, and C. Pradelles. Plane - an evolutive planning system for aircraft production. *In Proc. First International Conference on Practical Applications of Prolog*, 1992.

[14] F. Benhamou and W. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programmiing*, 1995.

[15] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter Rossum, Stephan Schulz, and Roberto Sebastiani. MathSAT: Tight Integration of SAT and Mathematical Decision Procedures. *Journal of Automated Reasoning*, 35(1-3):265–293, 2005.

[16] Daniel R. Brooks, Esra Erdem, Selim T. Erdogan, James W. Minett, and Don Ringe. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning*, 2007.

[17] Daniel R. Brooks, Esra Erdem, James W. Minett, and Don Ringe. Character-based cladistics and answer set programming. *Practical Aspects of Declarative Languages*, pages 37–51, 2005.

[18] Holzbaur C. Ofai clp(q,r) manual. *Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09*, 1995.

[19] Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Giovanbattista Ianni, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, Gerard Pfeifer, and Axel Polleres. The dlv system. In Sergio Flesca and Giovanbattista Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, Sep 2002.

[20] Carlson B. Carlsson M., Ottosson G. An open-ended finite domain constraint solver. *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.

[21] Pawel Cholewinski, V. Wiktor Marek, and Miroslaw Truszczynski. Default reasoning system deres. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 518–528. Morgan Kaufmann, 1996.

[22] K. P. Chow and M. Perrett. Airport counter allocation using constraint logic programming. *In Proc. PACT97*, 1997.

[23] P. Codognet and D. Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 27(3), 1996.

[24] C. Collignon. Gestion optimisee de resources humaines pour l'audiovisuel. *In Proc. CHIP users' club*, 1996.

[25] Alain Colmerauer and Philippe Roussel. The birth of prolog. *History of Programming Languages. The second ACM SIGPLAN conference on History of programming languages*, pages 37–52, 1993.

[26] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1993.

[27] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5 (7):394–397, 1962.

[28] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[29] Tina DellArmi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, Simona Perri, and Gerald Pfeifer. System description: Dlv with aggregates. *Logic Programming and Nonmonotonic Reasoning*, 2923:326–330, 2004.

[30] Tina DellArmi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate functions in dlv. *Answer Set Programming*, 2003.

[31] Daniel Diaz. The GNU Prolog website. http://www.gprolog.org, 2007.

[32] Y. Dimopoulos, J. Koehler, and B. Nebel. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the 4th European Conference on Planning*, volume 1348 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 169–181, 1997.

[33] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. *In proc. International Conference on Fifth Generation Computer Systems*, 1988.

[34] K. J. Dryllerakis. Residual sldnf in clp languages. *Technical Report*, 1995.

[35] Yu. Lierler E. Giunchiglia and M. Maratea. Cmodels-2: Sat-based answer set programming. *In Proc. of AAAI*, 2004.

[36] D. East and M. Truszczynski. More on wire routing with asp. *Technical Report in Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning 2001 AAAI Spring Symposium*, pages 39–44, 2001.

[37] Islam Elkabani, Enrico Pontelli, , and Tran Cao Son. Smodels with clp and its applications: A simple and effective approach to aggregates in asp. *ICLP-04*, pages 73–89, 2004.

[38] E. Erdem, V. Lifschitz, L. Nakhleh, and D. Ringe. Reconstructing the evolutionary history of indo-european languages using answer set programming. *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL 03)*, 2003.

[39] E. Erdem, V. Lifschitz, and D. Ringe. Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming*, 6(5):539–558, 2006.

[40] E. Erdem, V. Lifschitz, and M. Wong. Wire routing and satisfiability planning. *Proceedings of CL-2000*, pages 822–836, 2000.

[41] Esra Erdem. Application of logic programming to planning: Computational experiments. In *Proceedings of LPNMR-99*, Lecture Notes in Artificial Intelligence (LNCS). Springer Verlag, Berlin, 1999.

[42] Esra Erdem. *Theory and applications of answer set programming.* PhD thesis, University of Texas at Austin, 2002.

[43] Pollack Martha E. et. al. Pearl: A mobile robotic assistant for the elderly. *AAAI Workshop on Automation as Eldercare*, Aug 2002.

[44] Pollack Martha E. et. al. Autominder: An intelligent cognitive orthotic system for people with memory impairment. *Robotics and Autonomous Systems*, 44(3-4):273–282, 2003.

[45] Thomas Eiter et. al. Working group on answer set programming WASP, TUWIEN node. http://www.kr.tuwien.ac.at/, Dec 2007.

[46] V. Lifschitz et. al. TAG Discussion. http://www.cs.utexas.edu/users/vl/tag/discussions.html, Dec 2007.

[47] Boi Faltings and Marc Torrens. Using soft csps for approximating pareto-optimal solution sets. In Ulrich Junker, editor, *Preferences in AI and CP: Symbolic Approaches*, AAAI 2002 Spring Symposium Series, pages 99–106, 2002.

[48] Michael Gelfond. Representing knowledge in a-prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408, pages 413–451. Springer Verlag, Berlin, 2002.

[49] Michael Gelfond and Nicola Leone. Knowledge representation and logic programming. *Artificial Intelligence*, 138(1–2), 2002.

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

[51] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–385, 1991.

[52] Nevin C. Heintze, Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The clp(r) programmer's manual, 1992.

[53] ILOG CP. ILOG Constraint Programming. http://www.ilog.fr/products/cp, 2000.

[54] J. Jaffar and J. Lassez. Constraint logic programming. *Proceedings of the Principles of Programming Languages Conference*, pages 111–119, Jan 1987.

[55] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[56] Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.

[57] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The clp(r) language and system: an overview. *Compcon*, pages 376–381, 1991.

[58] Joxan Jaffar, Spiro Michaylov, Peter 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.

[59] Gabriele Kern-Isberner, Christoph Beierle, and Oliver Dusso. Modelling and implementing a knowledge base for checking medical invoices with dlv. In Gerhard Brewka, Ilkka Niemelä, Torsten Schaub, and Miroslaw Truszczynski, editors, *Nonmonotonic Reasoning, Answer Set Programming and Constraints*, number 05171 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http://drops.dagstuhl.de/opus/volltexte/2005/261> [date of citation: 2005-01-01].

[60] L. Khatib, P. Morris, R. Morris, and F. Rossi. Temporal reasoning about preferences, 2001.

[61] F. Kokkoras and S. Gregory. D-wms: Distributed workforce management using clp. *In Proc. PACT98*, 1998.

[62] Loveleen Kolvekar. Developing an inference engine for cr-prolog with preferences. Master's thesis, Texas Tech University, Dec 2004.

[63] Robert A. Kowalski. Predicate logic as programming language. *IFIP Congress*, pages 569–574, 1974.

[64] Robert A. Kowalski. *Logic for Problem Solving*. Prentice Hall PTR, 1979.

[65] Nicola Leone, Thomas Eiter, Wolfgang Faber, Michael Fink, Georg Gottlob, Luigi Granata, Gianluigi Greco, Edyta Ka?ka, Giovambattista Ianni,

Domenico Lembo, Maurizio Lenzerini, Vincenzino Lio, Bartosz Nowicki, Riccardo Rosati, Marco Ruzzi, Witold Staniszkis, and Giorgio Terracina. Data integration: a challenging asp application. In *Logic Programming and Nonmonotonic Reasoning*, volume 3662 of *Lecture Notes in Computer Science*, pages 379–383, 2005.

[66] Yu. Lierler and M. Maratea. Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. *In Proc. of LPNMR-7*, 2004.

[67] V. Lifschitz and H. Turner. Splitting a logic program. *In Pascal Van Hentenryck, editor, Proc. of Eleventh Int'l Conf. on Logic Programming*, pages 23–38, 1994.

[68] Fangzhen Lin and Yuting Zhao. Assat: Computing answer sets of a logic program by sat solvers. In *Proceedings of AAAI-02*, 2002.

[69] V. Wiktor Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398, 1999.

[70] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[71] John McCarthy. Elaboration tolerance, 1998.

[72] Veena S. Mellarkod. Optimizing the computation of stable models using merged rules. Master's thesis, Texas Tech University, May 2002.

[73] Veena S. Mellarkod. Acsolver. http://www.krlab.cs.ttu.edu///software/, Dec 2007.

[74] Veena S. Mellarkod. Adsolver. http://www.krlab.cs.ttu.edu///software/, Dec 2007.

[75] Ilkka Niemela and Patrik Simons. *Extending the Smodels System with Cardinality and Weight Constraints*, pages 491–521. Logic-Based Artificial Intelligence. Kluwer Academic Publishers, 2000.

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

[77] Monica Nogueira. *Building Knowledge Systems in A-Prolog*. PhD thesis, University of Texas at El Paso, May 2003.

[78] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An a-prolog decision support system for the space shuttle. In Alessandro Provetti and Son Cao Tran, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, AAAI 2001 Spring Symposium Series, Mar 2001.

[79] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An a-prolog decision support system for the space shuttle. In *PADL 2001*, pages 169–183, 2001.

[80] V. Orekov. On glivenko's classes of formulas. *Trudy Mat. Inst. Steklov*, 98:131–154, 1968.

[81] K. Heus P. Chan and G. Veil. Nurse scheduling with global constraints in chip: Gymnaste. *In Proc. PACT98*, 1998.

[82] M. Perrett. Using constraint logic programming techniques in container port planning. *ICL Technical Journal*, pages 537–545, 1991.

[83] Martha E. Pollack, Colleen E. McCarthy, Sailesh Ramakrishnan, Ioannis Tsamardinos, Laura Brown, Steven Carrion, Dirk Colbry, Cheryl Orosz, and Bart Peintner. Autominder: A planning, monitoring, and reminding assistive agent. *7th International Conf. on Intelligent Autonomous Systems*, 2002.

[84] Martha E. Pollack and Nicola Muscettola. Temporal and resource reasoning for planning, scheduling and execution. *Tutorial Forum Notes, AAAI06*, Jul 2006.

[85] G. Ramalingam, J. Song, L. Joscovicz, and R. Miller. Solving difference constraints incrementally. *Algorithmica*, 23:261–275, 1999.

[86] F. Rossi, A. Sperduti, K. Venable, L. Khatib, P. Morris, and R. Morris. Learning and solving soft temporal constraints: An experimental study, 2002.

[87] F. Rossi, K. Venable, and L. Khatib. Two solvers for tractable temporal constraints with preferences, 2002.

[88] Francesca Rossi. Constraint (logic) programming: A survey on research and applications. *K.R. Apt et al. (Eds.): New Trends in Constraints, LNAI 1865*, pages 40–74, 2000.

[89] Hossein M. Sheini and Karem A. Sakallah. A sat-based decision procedure for mixed logical/integer linear problems. *CPAIOR*, pages 320–335, 2005.

[90] H. Simonis and P. Charlier. Cobra - a system for train crew scheduling. *Proc. DIMACS workshop on constraint programming and large scale combinatorial optimization*, 1998.

[91] H. Simonis and T. Cornelissens. Modeling producer consumer constraints. *In Proc. First International Conference on Principles and Practice of Constraint Programming*, 1995.

[92] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Helsinki, Apr 2000. Research Report 58.

[93] Barbara M. Smith. Modelling. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, pages 378–406. Elsevier, 2006.

[94] Timo Soininen and Ilkka Niemela. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, May 1999.

[95] Tran Cao Son and Enrico Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, pages 355–375, 2007.

[96] Peter J. Stuckey. Constructive negation for constraint logic programming. *Logic in Computer Science, LICS 91., Proceedings of Sixth Annual IEEE Symposium*, pages 328–339, 1991.

[97] Peter J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.

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

[99] Tommi Syrjanen. Lparse user's manual, 2000.

[100] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 23(4):722–742, 1976.

[101] WASP. Working Group on Answer Set Programming. http://wasp.unime.it/, Dec 2007.

[102] Gerald Pfeifer Wolfgang Faber, Nicola Leone. Recursive aggregates in disjunctive logic programs: Semantics and complexity. *JELIA 2004*, pages 200–212, 2004.

[103] N. Yorke-Smith, K. B. Venable, and Rossi F. Temporal reasoning with preferences and uncertainty, 2003.