Developing an Inference Engine for CR-Prolog with Preferences

Thesis Report

By

Loveleen Kolvekar

Oct 2004

ACKNOWLEDGEMENTS

I have been fortunate enough to have all the support I needed to finish this thesis work. I am also blessed to have support and love from my parents and my brothers and sisters.And my special thanks to my fiance Pradeep, for all his love and support and for always teaching me good CS skills. Thank you Pradeep for always being there for me.

My thanks to my professor Dr Gelfond for sharing his knowledge and guiding me in my work. It was really nice working with you. Thank you Dr. Gelfond for teaching me the basics of AI and for encouraging me in my work and for showing me a proper path in my work. I am also thankful to Dr Watson for all his support and help.

My special thanks to Marcello for all his support and help. Thank you Marcello for working with me and helping me in all the phases of my work. It would not have been possible without your help. Thanks for taking time from your work and teaching me . I learned a lot from you, including the way to write nice proofs and also good programming techniques.

Lastly I would like to thank all my labmates for providing a wonderful work environment in the lab.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ……. ii

ABSTRACT ……. v

LIST OF TABLES ……vi

LIST OF FIGURES ……vii

CHAPTER

I INTRODUCTION ……. 1

1.1 Background ……..1

1.1.2 The Need for CR-Prolog ……. 4

1.2 Goals and Contributions of this Work ……..……. 6

II SYNTAX AND SEMANTICS …… 8

2.1 Syntax ……8

2.2 Semantics ……11

III ALGORITHM ……15

3.1 The Hard Reduct ……15

3.2 Generator and Tester ……17

3.3 The Design of the Algorithm ……... 20

3.4 Description of the Algorithm ……... 22

IV PROPERTIES OF THE ALGORITHM ……... 25

4.1 Proof of Termination ……...25

4.2 Proofs of Soundness and Completeness ..…… … 26

V IMPROVING THE EFFICIENCY OF CR-MODELS …..…… …42

5.1 Lparse Merging …… 42

5.2 Own Merging ……46

5.2.1 Description of the Algorithm for Own-Merging …….. ...46

5.2.1.1 Numbering of Atoms ……. 47

5.2.1.2 Generation Phase ……. ….. 50

5.2.1.3 Testing Phase …….. …. 53

VI EXPERIMENTAL RESULTS ……... 53

6.1 Comparing Lparse and Own Merging …… …….. 53

6.2 Comparing CR-Models and CR-Old ……. …….. 55

VII CONCLUSIONS …….. 61

7.1 Future Work ……. …… 62

REFERENCES ……63

APPENDIX …….. …66

ABSTRACT

In recent years, A-Prolog with answer set semantics was shown to be a useful tool for knowledge representation and reasoning. A-Prolog is a declarative language based on stable models of logic programs. It allows the encoding of defaults and various other types of knowledge contained in dynamic domains.

It seems however that A-Prolog lacks the ability to gracefully perform the reasoning needed for certain types of conflict resolution, e.g finding the best explanations of unexpected observations. To solve this problem CR-Prolog – an extension of A-Prolog by consistency restoring rules with preferences was introduced. The most intuitive solutions correspond to those models that best satisfy the preferences expressed, and minimize the application of cr-rules.

The goal of this work is to develop an inference engine for computing the answer sets of CR-Prolog program automatically. The inference engine handles preferences efficiently.

LIST OF TABLES

6.1 Experimental Results - Own-Merging and Lparse Merging …….. 57

6.2 Experimental Results- CR-Models and CR-Old …….. 60

LIST OF FIGURES

1.1 Electrical Circuit 1 …… 3

1.2 Electrical Circuit 2 …….. 4

3.1 Algorithm CR-Models …… ……. 22

6.1 Blocks World Domain ……. ……. 53

6.2 Algorithm CR-Old ……... ….. 57

6.3 Comparing CR-Models and CR-Old for RCS Planning Module …….. 60

CHAPTER I

INTRODUCTION

The main goal of this chapter is to give a background information necessary to understand the subject , and to explain the importance of developing an inference engine for CR-Prolog. We explain the limitations that A-Prolog has with performing reasoning for certain types of conflict resolution and the way CR-Prolog can be used to solve this problem. Finally an outline of the work done in this thesis is presented.

1.1 Background

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. This set of statements is 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, 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 [1,2,3].

Declarative languages need to meet certain requirements. Some of these requirements are[1]:

1) The syntax should be simple and there should be a clear definition of the meaning of the language.
2) 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 knowledge base. [4].
3) 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 languages and the desired efficiency.

One such declarative language is *A-Prolog*[1], a logic programming language under the answer set semantics[5] which was shown to be a useful tool for knowledge representation and reasoning. It allows encoding defaults and various other types of knowledge of the dynamic domains (e.g the representation of actions and their effects). The language is expressive and has a well understood methodology of representing defaults, causal properties of actions and fluents, various types of incompleteness, etc. The development of efficient computational systems[14,15] has allowed the use of A-Prolog to be used for a diverse collection of applications. Some of these applications include: a decision support system for space shuttle flight controllers[6], planning[7,8], product configuration[9], bounded model checking etc. In recent years the development of several reasoning systems for A-Prolog led to the emergence of answer set

programming [1], a new programming paradigm. Currently the most efficient inference engines for A-Prolog are Smodels[10,14] and DLV[11,15] .

The syntax of *A-Prolog* is similar to Prolog. The following is an example of an A-Prolog program.

Example 1.1: Consider the following scenario in figure 1.1

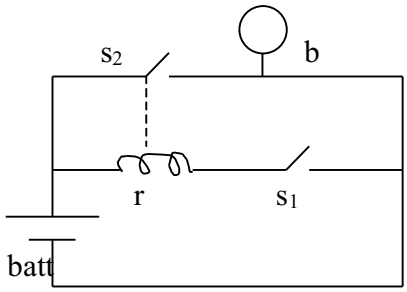

Figure 1.1 Electrical Circuit 1

The functioning of the above circuit is as follows : when the battery is okay and the switch s_1 is closed, the relay r is active.When relay r is active it causes switch s_2 to be closed and hence the bulb glows.

This knowledge can be represented using the following A-Prolog program Π_0.

$$\begin{cases} active(r) \leftarrow closed(s_1), batt_ok. & (1) \\ closed(s_2) \leftarrow active(r). \\ lit(b) \leftarrow closed(s_2), batt_ok. \\ batt_ok \leftarrow not \neg batt_ok. & (2) \end{cases}$$

The rule(1) is read as " if s_1 is closed and batt is ok then r is active", active(r) is called the head of the rule and { closed(s_1), batt_ok } is called the body of the rule.

The rule(2) says that if there is no reason to believe that batt is not ok then the battery is ok.

If we initially know that closed(s_1) is true then the answer set of $\Pi_0 \cup \{ closed(s_1) \}$ is $\{ closed(s_1), batt_ok, active(r), closed(s_2), lit(b) \}$.

Different problems of planning and diagnostics are reduced to computing the answer sets of A-Prolog program.

1.1.2 The Need for CR-Prolog

It seems however that A-Prolog lacks the ability to gracefully perform the reasoning needed for certain types of conflict resolution e.g finding the best explanations of the unexpected observations.

Lets look at the following example .

Example 1.2: Consider the following scenario in fig 1.2:

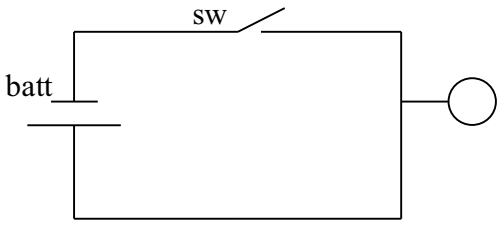

Figure 1.2 Electrical Circuit 2

In the above circuit when the battery, (batt), is ok and the switch(sw) is closed the bulb,(b), glows. This knowledge can be represented by the following A-Prolog program Π_1

% State Constraints

lit(b) ← closed(sw), batt_ok.

¬lit(b) ← ¬batt_ok.

¬lit(b) ← ¬closed(sw).

%CWA(Closed World Assumption)

closed(sw) ← not ¬closed(sw).

batt_ok ← not ¬batt_ok.

Given the above representation, if we are told that the bulb is not lit then the above program becomes inconsistent. This is because the closed world assumption forces us to conclude that closed(sw) and batt_ok are true and hence the bulb should be lit. Intuitively one would want to conclude that either the bulb is not ok or the switch sw is not closed.

Given such situations A-Prolog lacks the ability to find the best explanations.To solve this problem CR-Prolog[13] – an extension of A-Prolog by *consistency restoring rules*(cr-rules) with preferences - was introduced. In CR-Prolog the programmer gives a set of rules (cr-rules) that may possibly be applied in addition to the regular A-Prolog rules. The programmer can also specify the preferences between the cr-rules.

The inconsistency of the program in Example 1.2 can be resolved by adding the following diagnostic module Π_D

% Diagnostic Module

r1: $\neg$closed(sw) $\leftarrow+$.

r2: $\neg$batt_ok $\leftarrow+$.

where rule r1 says that " the switch may possibly be open". Hence this possibility defeats the default of the closed world assumption and one can conclude that the switch is open and hence the bulb is not lit. Therefore this module helps in restoring the consistency.

Events described using cr-rules are rare events. They are used only if the agent has no way to obtain a consistent set of beliefs using only regular rules. We always minimize the application of the number of cr-rules.

Further if we know that the battery is more reliable then we can add the following preference relation prefer(r1, r2), which says that " it is more likely that the switch is not closed than the battery not being ok"

The answer set of $\Pi_1 \cup \Pi_D \cup \{$ prefer(r1, r2) $\}$ is:

$\{ \neg$closed(sw), prefer(r1, r2), $\neg$lit(b) $\}$.

Hence the most intuitive solution correspond to the answer sets of the CR-Prolog program. The answer sets of CR-Prolog:

1) satisfy the binding preferences (i.e prefer)
2) the corresponding sets of cr-rules are set theoretically minimal.

Marcello Balduccini and Dr Gelfond showed that cr-rules can be used to improve the quality of plans and to do diagnostics. The idea is similar to what we had in A-Prolog, but a more subtle information is encoded by addition of cr-rules and planning and diagnostics are reduced to finding answer sets of CR-Prolog programs.

1.2 Goals and Contributions of this work

The *most intuitive* solutions for problems modeled in CR-Prolog correspond to those answer sets that best specify the preferences expressed, and minimize the number of applications of cr-rules. Different problems of planning and diagnostics are reduced to computing answer sets of CR-Prolog programs. Therefore, an efficient inference engine that would find the answer sets of the CR-Prolog program with preferences was necessary.

The work presented in this thesis consists of developing an efficient inference engine for CR-Prolog with preferences. An algorithm called CR-Models was designed and implemented. CR-Models handles preferences efficiently . A generate and test approach is used where generation and testing part are essentially reduced to computation of answer sets of A-Prolog programs. We investigate different criteria that can used to select the possible explanations. The algorithm can be implemented using the existing answer set solvers[10, 14, 11]. This approach makes it possible to directly exploit constantly improving performances of answer set solvers. The current implementation is based on Smodels[10].

This thesis work includes:

1) Designing an efficient algorithm, CR-Models, to find the answer sets of CR-Prolog program.
2) Implementation of the inference engine CR-Models.
3) Proof that the algorithm terminates.
4) Proofs that the algorithm used by the inference engine is sound and complete.
5) Improving the efficiency of the inference engine.
6) Experimental investigation of the performance of the inference engine.

The thesis is organized in the following manner. Chapter 2 presents the syntax and semantics of CR-Prolog. Chapter 3 gives the description of the algorithm. Chapter 4 gives the properties of the alogrithm . Chapter 5 gives the way in which efficiency of CR-Models was improved . Chapter 6 gives the experimental results and finally Chapter 7 gives the conclusion and future work.

CHAPTER II

SYNTAX AND SEMANTICS

CR-Prolog[13, 17] – an extension of A-Prolog with consistency restoring rules with preferences was introduced by Dr.Michael Gelfond, Marcello Balduccini and Veena Mellarkod. CR-Prolog can be used to represent preferences intended as strict preferences and desires. The "most reasonable" solutions correspond to those models that best satisfy the preferences expressed, and minimize the application of cr-rules. Let us now look at the syntax and semantics of CR-Prolog.

2.1 Syntax

Let Σ be a signature containing symbols for constants, variables, functions, and predicates (denoted by *const*(Σ) , *var*(Σ), *func*(Σ) and *pred*(Σ), respectively). Terms, atoms, and literals are defined as usual. Literals and terms not containing variables are called *ground*. The sets of ground terms, atoms and literals over Σ will be denoted by *terms*(Σ), *atoms*(Σ), and *lit*(Σ). CR-Prolog program consists of both regular rules and cr-rules.

Definition 2.1: A regular rule of CR-Prolog is a statement of the form:

$r : h_1$ or … or $h_k \leftarrow l_1,\ldots l_m,$

not $l_{m+1},\ldots,$ not l_n .

where h_i's and l_i's are literals, h_1 or…or h_k, is the head, and $l_1,\ldots l_m$, not $l_{m+1},\ldots,$ not l_n is the body, and r is a term representing the name of the rule. The intuitive reading of this

rule , in terms of the beliefs that a rational agent complying with the rule should have , is: " if the agent believes $l_1,\ldots l_m$ and does not believe $l_{m+1},\ldots, l_n$, then it must believe one element of the head of the rule."

(Names of regular rules are not needed for the definition of semantics of programs with cr-rules and can thus be safely dropped).

Definition 2.2: A cr-rule is a statement of the form:

$r : h_1$ or … or $h_k \leftarrow^{+} l_1,\ldots l_m,$

not $l_{m+1},\ldots,$ not l_n .

The intuitive reading of this rule in terms of beliefs that a rational agent complying with the rule should have is: " if the agent believes $l_1,\ldots l_m$ belong to a set of agent's beliefs and none of $l_{m+1},\ldots, l_n$ belong to it then the agent "may possibly" believe one element of the head of the rule ". This possibility is used only if the agent has no way to obtain a consistent set of beliefs using regular rules only.

Consider for instance program Π_0 :

$a \leftarrow$ not b.

$r_1 : b \leftarrow^{+}.$

Π_0 has an answer set {a}, computed without the use of cr-rule r_1.

Now consider $\Pi'_0 = \Pi_0 \cup \{\neg a\}$. If r_1 is not used , Π'_0 is inconsistent. Consistency can be restored by using r_1 which will allow the reasoner to believe in b, leading to the answer set $\{\neg a, b\}$.

Defiinition 2.3: *Preferences between cr-rules* are expressed by atoms of the form *prefer(r_1, r_2)* where r_1, r_2 are names of cr-rules. The intuitive reading of the atom is "do not consider sets of beliefs obtained using r_2 unless you have excluded the existence of belief sets obtained using r_1 ." If all preferences in a program are expressed as facts, we say that the program employs *static preferences*. Otherwise, preferences are *dynamic*. To better understand the use of preferences, consider program Π_1:

r_1 : a $\leftarrow^+$.

r_2 : b $\leftarrow^+$.

r_3 : *prefer*(r_1, r_2).

Π_1 has one answer set: {*prefer*(r_1, r_2)}. Notice that cr-rules are not applied, and hence the preference atom has no effect. Now consider program $\Pi_2 = \Pi_1 \cup$ {r_4 : $\leftarrow$ not *a,* not b}. Now cr-rules must be used to restore consistency. Since r_1 is preferred to r_2, the answer set is: {a *, prefer*(r_1, r_2)}. Finally, consider $\Pi_3 = \Pi_2 \cup \{ r_5 : \leftarrow a\}$. Its answer set is: {b, *prefer*(r_1, r_2)}.

Definition 2.4: A *CR-Prolog program*, Π, is a pair $<\Sigma, R>$ consisting of signature Σ and a set R of rules .

Signature Σ is denoted by *sig*(Π); *const*(Π), *func*(Π), *pred*(Π), *atoms*(Π) and lit(Π) are shorthands for const(*sig*(Π)), *func*(*sig*(Π)), *pred*(*sig*(Π)), *atoms*(*sig*(Π)) and *lit*(*sig*(Π)), respectively. Let P be a set of predicate symbols from Σ . By *atoms*(Π , P) we denote the set of all atoms from *atom*s(Π) formed by predicate symbols from P. (Whenever possible we drop the first argument and simply write *atoms(*P)). The set of

rules of Π is denoted by *rules*(Π). If ρ is a rule of Π then *head*(ρ) = H, and *body*(ρ) = $\{l_1,\ldots, l_m, \text{not } l_{m+1},\ldots,\text{not } l_n\}$.

2.2 Semantics

In the following discussion Π_1 denotes a CR-Prolog program, Π_0 the regular part of Π_1, and R the cr-rules in Π_1. Also, for every $R' \subseteq R$, $\alpha(R')$ denotes the set of regular rules obtained from R' by replacing every connective $\leftarrow^+$ with $\leftarrow$.

Let us now define the relation $pref_S$.

Definition 2.5: For every set of literals, S, from the signature of Π_1, and every r_1, r_2 from R, $pref_S(r_1, r_2)$ is true iff

$prefer(r_1, r_2) \in S$, or

$\exists r_3 \in R \;\; prefer(r_1, r_3) \in S \wedge pref_S(r_3, r_2)$.

To see how the above definition works consider the following example

Example 2.1: Let $S = \{ prefer(r_1, r_2), prefer(r_2, r_3), a, q, p \}$

and $R = \{ r_1, r_2, r_3 \}$.

- $pref_S (r_1, r_2)$ holds (since $prefer(r_1, r_2) \in S$).
- $pref_S (r_2, r_3)$ holds (since $prefer(r_2, r_3) \in S$).
- $pref_S (r_1, r_3)$ holds (since $prefer(r_1, r_2) \in S \wedge pref_S (r_2, r_3)$ holds).

Definition 2.6**:** <S, R> is a *view* of Π_1 if:

1. *S* is an answer set of $\Pi_0 \cup \alpha(R_1)$, and

2. for every r_1, r_2 such that $pref_S (r_1, r_2)$, $\{r_1, r_2\} \not\subset R$, and

3.$\forall$ r $\in R$, body(r) is satisfied by *S*.

Example 2.2: Consider the following program Π_2 :

r_0: t $\leftarrow^+$.

r_1: p $\leftarrow^+$ q.

r_2: s $\leftarrow^+$.

r_3: q $\leftarrow^+$.

$\leftarrow$ not t, not p, not s.

prefer (r_0, r_2).

The views of the program Π_2 are:

$<S_1, R_1> : <\{t\}, \{r_0\}>$.

$<S_2, R_2> : <\{ t ,q\}, \{r_0, r_3 \}>$.

$<S_3, R_3> : <\{s\}, \{r_2 \}>$.

$<S_4, R_4> : <\{s,q\}, \{r_2, r_3\}>$.

$<S_5, R_5>: <\{p,q\},\{ r_1, r_3\}>$.

Since the regular part is inconsistent, the cr-rules are applied.

Notice that we cannot apply r_1 alone as the body of the rule is not satisfied(item 3 of Definition 2.6), but it can be applied along with r_3 , which makes its body true. Also, r_0 and r_2 cannot be applied together as there is a preference between them (item 2 of Definition 2.6).

Definition 2.7: A view $<S_1,R_1>$ *dominates* a view $<S_2,R_2>$, if there exist r_1, r_2 such that $r_1 \in R_1$, $r_2 \in R_2$, and $pref_{S1 \cap S2}$ (r_1, r_2).

To better understand the definition let us apply it to the views of the above program Π_1. First note that $<S_1, R_1>$ *dominates* $<S_3, R_3>$. In fact, $r_0 \in R_1$ and $r_2 \in R_2$, and $pref_{S1 \cap S3}$ (r_0, r_2). In a similar way $<S_2, R_2>$ *dominates* $<S_4, R_4>$.

If a view is dominated by another, it means that it is not as "good" as the other w.r.t preference contained in the program. Consider $<S_4, R_4>$, for example: since it is dominated by $<S_2, R_2>$, the intuition suggests that $<S_4, R_4>$ should be excluded from the set of beliefs of the agent . Views that are equally acceptable w.r.t preferences are called candidate answer sets, as stated in the next definition.

Definition 2.8: A view, $<S_1,R_1>$, is a candidate answer set of Π_1 if, for every view $<S_2,R_2>$ of Π_1,

$<S_2,R_2>$ does not dominate $<S_1,R_1>$.

Hence $<S_3, R_3>$ and $<S_4, R_4>$ above are not candidate answer sets of Π_1, while $<S_1,R_1>$, $<S_2,R_2>$ and $<S_5,R_5>$ are.

Now let us compare $<S_1,R_1>$ and $<S_2,R_2>$. Candidate answer set $<S_1,R_1>$ is obtained by applying r_0 while candidate answer set $<S_2,R_2>$ is obtained by applying r_0 and r_3. Intuitively application of rule r_3 seems unnecessary, which makes $<S_2,R_2>$ less acceptable than $<S_1,R_1>$. We discard belief sets such as $<S_2,R_2>$, by applying a minimality criterion based on set theoretic inclusion on rules present in each set. The remaining sets are answer sets of the program.

Definition 2.9: A set of literals, S_1, is an *answer set* of Π_1 if:

1. there exists $R_1 \subseteq R$ such that $<S_1,R_1>$

is a candidate answer set of Π_1, and

2. for every candidate answer set, $<S_2,R_2>$, of Π_1, $R_2 \not\subset R_1$.

Let us apply the above definition to candidate answer sets of program Π_1.

From Definition 2.8, we know that $<S_1,R_1>$, $<S_2,R_2>$ and $<S_5,R_5>$ are candidate answer sets.

Since $R_2 \subset R_1$, S_2 is not an answer set of program Π_1. The only answer sets are S_1 and S_5.

CHAPTER III

ALGORITHM

In this chapter, we present CR-Models, the algorithm for computing the answer sets of a CR-Prolog program. The algorithm uses a generate and test approach.

3.1 The Hard Reduct

The algorithm, CR-Models makes use of the hard reduct, hr(Π), of program Π. A hard reduct is a translation of a CR-Prolog program into an A-Prolog program. This translation is used because there exists one-to-one correspondence, β, between the answer sets of hr(Π) and the views of Π. The translation introduces some new predicate and function symbols. In the following discussion, we will assume that such symbols do not belong to the signature of Π. More precisely, we require that, for any input program Π:

1) *func*(Π) does not contain *choice*.

2) *pre*d(Π) contains *prefer* and does not contain *appl, fired*, and *is_preferred*.

Definition 3.1 : The hard reduct hr(Π) consists of:

1) Every regular rule of Π .

2) For every cr-rule r, hr(Π) contains:

a) head(ρ)←body(ρ), applcr(r) , where r is the name of cr-rule ρ.

b) crname(r).

c) bodytrue(r) ← body(ρ) .

3) Rules:

% prohibit the application of rule when the body of the rule is not satisfied

← not bodytrue(R) , applcr(R), crname(R).

% The generator rule

{applcr(R) : crname(R) }

4) hr(Π) also contains the following set of rules denoted by Πp:

% transitive closure of predicate prefer

*m*1*a* : *is_preferred*(*R*1,*R*2) ← *prefer*(*R*1,*R*2).

*m*1*b* : *is_preferred*(*R*1,*R*2) ← *prefer*(*R*1,*R*3), *is_preferred*(*R*3,*R*2).

% no circular preferences

*m*2 : ← *is_preferred*(*R*,*R*).

% prohibit application of *R*1 and *R*2 if

% *R*1 is preferred to *R*2

*m*3 : ← *applcr*(*R*1), *applcr*(*R*2), *is_preferred*(*R*1,*R*2).

Example 3.1: Consider the following program Π_3 , and let us compute its hard reduct $hr(\Pi_3)$:

r1: p ←+ not q.

r2: s ←+ .

:- not p, not s.

prefer(r1, r2).

The hr(Π_3) consists of Πp and:

r1 : p ← not q, applcr(r1).

crname(r1).

bodytrue(r1) ← not q.

r2 : s ← applcr(r2).

crname(r2).

bodytrue(r2).

:- not p, not s.

prefer(r1, r2).

:- not bodytrue(R), applcr(R), crname(R).

{applcr(R) : crname(R)}.

3.2 The Generator and Tester

The algorithm to compute answer sets for CR-Prolog program is based on a generate-and-test approach. Generation of candidate solutions and their testing are reduced to the computation of the answer sets of two smodels programs, the i-generator, $G_i(\Pi)$ and tester, $T(\Pi,M)$. The generator generates the views of the CR-Prolog program and the tester checks if this view is a candidate answer set.(For simplicity, we consider only ground CR-Prolog programs).

a) The i-generator, $G_i(\Pi)$, whose answer sets correspond to the answer sets of hr(Π) obtained using i cr-rules, where $0<i<n$, n=total number of cr-rules. Note: answer set, M of $G_i(\Pi)$ is an answer set of hr(Π).

b) The tester, $T(\Pi,M)$, where

   1) answer set K, of $T(\Pi,M)$ is an answer set of hr(Π).

2) K is an answer set of T(Π,M) iff β(K) dominates β(M).

3) T(Π,M) is inconsistent if there exists no β(K) that dominates β(M).

The i generator program is described in the following definition.

Definition 3.2: Let P be a CR-Prolog program, and i an integer. The i-generator, $G_i(P)$, consists of:

1) hr(P).

2) ok ← i{applcr(R) : crname(R)}i.

3) ← not ok.

It is easy to see that answer sets of $G_i(P)$ correspond to the answer sets of hr(P) obtained using i cr-rules.

Example 3.2: $G_1(\Pi)$ for the program from Example 3.1, consists of the following

a) hr(Π)

b) ok ← 1{applcr(r1), applcr(r2)}1.

c) ← not ok.

The answer sets of $G_1(\Pi)$ are:(some atoms have been omitted)

1) { applcr(r1), p, bodytrue(r1), crname(r1), prefer(r1, r2) }.

2) { applcr(r2), s, bodytrue(r2), crname(r2), prefer(r1, r2) }.

The Tester program is decribed as follows:

Definition 3.3: Let P be a CR-Prolog program and M an answer set of $G_i(P)$, for some i. The tester program, T(P,M) consists of the following:

a) The facts.

1) o_applcr(r), for each r such that applcr(r) ∈ M

2) o_is_preferred(r1,r2), for each r1,r2, such that is_preferred(r1, r2) ∈ M.

b) The rules

1) dominates ← applcr(I), o_applcr(J), is_preferred(I,J), o_is_preferred(I,J).

2) ← not dominates.

c) hr(P).

Example 3.3: Consider the program Π from Example 3.1. The views of Π are

M1: {applcr(r1), p}.

M2: {applcr(r2), s }.

The tester program T(Π, M2), which checks if M2 is a candidate answer sets consists of:

% the facts

o_applcr(r2).

o_is_preferred(r1, r2).

% the rules

dominates ← applcr(r1), o_applcr(r2), is_preferred(r1, r2), o_is_preferred(r1,r2).

← not dominates.

% hard reduct

hr(Π).

The answer sets are: {dominates, applcr(r1), p}.

In fact, <{p},{r1}> dominates <{s},{r2}>. Hence <{s},{r2}> is not a candidate a.s

Note: T(Π, M1) is inconsistent. i.e there are no views of Π that dominate <{p},{r1}>.

Hence M1 is a candidate answer set.

3.3 Design of the Algorithm

In order to describe the algorithm, we need the following terminology.

Definition 3.4: Given a set of literals, M, CONSTR(M) denotes the constraint $\leftarrow$ M.

Definition 3.5: Given a collection of sets of literals, S, CONSTR(S) denotes the set of constraints obtained by applying CONSTR(M) to each element, M, of S, i.e.

CONSTR(S) = { :- M : M $\in$ S }.

$CONSTR_P^*(S)$ = {:- M, not N | M $\in$S and N= lit(P)\ M }

Example 3.4: Let us consider set lit(P) ={a, b, c} and S = {a}

Then $CONSTR_P^*(S)$ = {:- a, not b, not c. }.

Definition 3.6: Let M be a set of literals, and R a set of cr-rules. R(M) denotes the subset of R such that

r $\in$ R(M) iff applcr(r) $\in$ M .

Definition 3.7: Let M be a set of literals, S(M) = M \ atoms({applcr, is_preferred, bodytrue, crname}).

The algorithm to generate answer sets of CR-Prolog program P is as follows:

Function CR-Models (P)

Input : A CR-Prolog program P.

Output : The answer sets of P .

var

i : number of cr - rules applied during current iteration.

A : set of answer sets of P.

F : collection of sets of cr-rules in P.

L : collection of sets of literals.

C : set of constraints.

M : set of literals.

done : boolean.

1 i := 0; { Start finding the answer sets obtained by applying no cr -rules }

2 C := Ø;

3 { Let n be the number of cr-rules in P}

4 While(i ≤ n) do

5 L := Ø , F := Ø , done = false;

6 {Loop invariant for inner while loop is:

7 answer_sets ($G_i(P)$ + C + $CONSTR_P$*(L)) = answer_sets ($G_i(P)$ + C) \ L }

8 While (not done) do

9 If G_i (P) + C + $CONSTR_P$*(L) is inconsistent then

10 done := true;

11 else

12 M := one answer set of G_i (P) + C + $CONSTR_P$*(L) ;

13 L := L U {M}

14 If T(P, M) is inconsistent then

15 A := A U {S(M)};

16 F := F U {R(M)};

17 end

18 end {if}

```
19          end { inner while loop }
20      { each element of F has cardinality i }
21        for every R(M) ∈ F do
22          C := C U  CONSTR(R(M));
23        i := i + 1;
24      end  { outer while loop }
25      return A;
```

Figure 3.1 : Algorithm CR-Models

3.4 Description of the algorithm

The algorithm to compute answer sets of a CR-Prolog program with preferences is shown in Figure 3.1. CR-Models(P) returns the answer sets of the CR-Prolog program P.

This is achieved as follows:

- Recall that the answer sets of $G_i(P)$ correspond to the views of P with exactly i cr- rules applied.
- We exploit the sequentiality of the computation to prune the generation of views of P, so that we only compute candidate answer sets that are guaranteed to be answer sets of P.

To achieve this, we enumerate the views of P by increasing number of cr-rules applied. We also keep track of which sets of cr-rules were used to obtain the answer sets computed so far.

When a new view is computed, it must satisfy constraints ensuring that it does not use any set of cr-rules already applied in an answer set.

Set C contains such constraints.

When we compute the answer sets with i cr-rules, the corresponding sets of cr-rules are intitially added only to set F . They are converted to constraints and added to C when we move to the answer sets with i + 1 cr-rules.

This approach ensures a proper handling of programs where multiple answer sets are obtained from a fixed choice of cr-rules, e.g:

r_1 : a or b ←+ .

← not a, not b.

- In order to enumerate the answer sets of $G_i(P) + C$, we maintain a set, L, of the answer sets of $G_i(P) + C$ that have been already computed (with a given i).

a) We start by finding the answer sets without the application of any rules.(i.e, i = 0)

b) If $G_i(P)$ is inconsistent with C and $CONSTR_P^*(L)$, then, obviously there are no more answer sets to be enumerated.

c) Otherwise, we first of all add the new answer set, M, to L.

d) Next , we use the tester T(P,M) to check if M is a candidate answer set.

e) If T(P,M) is inconsistent, then there are no views of P dominating <S(M), R(M)>. This guarantees that S(M) is an answer set of P.

Hence, do the following

1) Add M to A, the collection of answer sets of P.

2) Add the set of cr-rule names applied in M to F, the collection of sets of cr- rule names.

- Otherwise, we discard M and consider the next answer set of $G_i(P) + C$ (the algorithm goes back to step 2)
- When there are no more answer sets that can be obtained using i cr-rules :
  1) We add the sets of cr-rule names from F to C as constraints.
  2) We iterate, to compute the views of P with i + 1 cr-rules.
- When all the views of P have been considered ($i < n$), A contains the answer sets of P

CHAPTER IV

PROPERTIES OF CR-MODELS

In this chapter we discuss the properties of the algorithm, CR-Models. We prove that the algorithm terminates for any CR-Prolog program. We also prove that it is sound and complete.

4.1 Proof of Termination

In the following theorem we prove that the algorithm terminates.

Theorem 4.1: Let $N = |\ answer_sets\ (G_i(P) + C)\ |$. (4.1)

The inner loop performs at most $N + 1$ iterations.

Proof : It is easy to see that the invariant of the inner loop is:

$$answer_sets\ (\ G_i(P) + C + CONSTR_P^*(S)) = answer_sets\ (G_i(P) + C) \setminus S \quad (4.2)$$

Assume that the algorithm didn't halt before $N + 1$th iteration . Then at the beginning of this iteration, $|\ S\ | = N$, (4.3)

since a new element is added to S at each iteration.

By (4.2),

$$|\ answer_sets\ (G_i(P) + C + CONSTR_P^*(S))\ | = |\ answer_sets\ (G_i(P) + C) \setminus S\ | \quad (4.4)$$

By (4.4) and (4.1) ,

$$|\ answer_sets\ (G_i(P) + C + CONSTR_P^*(S))\ | = N - |\ S\ | \quad (4.5)$$

Finally, from (4.5) and (4.3)

$$|\ answer_sets\ (G_i(P) + C + CONSTR_P^*(S))\ | = N - N = 0$$

Hence, *done* is set to false and the loop terminates.

Theorem 4.2: CR-Models(hr(P)) terminates for any CR-Prolog program P.

Proof : (sketch) The conclusion follows by considering that:

If hr(P) is consistent, the algorithm terminates.

The outer loop clearly performs n iterations (where n is the number of cr-rules in P)

By Theorem 4.1 the inner loop terminates.

4.2 Proofs of Soundness and Completeness

To discuss the properties of the algorithm, we need first to introduce some terminology

- atoms(P,{$p_{1, ..., p_n}$}) denotes the set of atoms, from the signature of P, built with predicate names $p_{1, ..., p_n}$. Whereever possible, we omit the first argument, and write atoms({$p_{1, ..., p_n}$}).

Definition 4.1: Let M be a set of literals, and R a set of cr-rules. R(M) denotes the subset of R such that $r \in R(M)$ iff $appl_{cr}(r) \in M$.

Definition 4.2: Let M be a set of literals, S(M) = M \ atoms({bodytrue, is_preferred, crname, $appl_{cr}$}).

Lemma 4.1: If M is an answer set of $G_i(P)$, then <S(M), R(M)> is a view.

Lemma 4.2: If <Z, Q> is a candidate a.s of P, then ∃M, i, s.t

1) S(M) = Z, R(M) = Q . 2) M is a.s of $G_i(P)$.

Lemma 4.3: If M is a.s of $G_i(P)$ and T(P,M) is inconsistent, then <S(M), R(M)> is a candidate a.s of P.

Lemma 4.4: If <Z, Q> is a candidate a.s of P, then ∃M, i , s.t

1) S(M) = Z and R(M) = Q,
2) M is a.s of $G_i(P)$,
3) T(P,M) is inconsistent.

Lemma 4.5: If M is an a.s of $G_i(P)$, then $|R(M)|$ is i.

Theorem 4.3: (Soundness) If $J \in$ CR-Models(P), then J is an a.s of P.

Proof: In the following discussion, V^i will denote the value of a variable V, from the algorithm, at iteration i of the outer while loop.

Let $J \in$ CR-Models(P) . From step 25 of the algorithm, $J \in \mathtt{A}$. From steps 15 and 12, $\exists i$, J* s.t 1) $J = S(J^*)$, and

2) J* is a.s of $G_i(P) + C^i + CONSTR_P^*(\mathtt{L}^i)$.

Since $CONSTR_P^*(\mathtt{L}^i)$ and C^i are sets of constraints, from item (2) above, it follows that J* is a.s of $G_i(P)$. Here we use the following fact:

Let $\Pi = \Pi_0 + R$, where R is a set of constraints, then A is an answer set of Π iff A is an answer set of Π_0 satisfying the constraints from R.

Also, from $J \in \mathtt{A}$. and steps 14 and 15 of the algorithm, it follows that T(P,J*) is inconsistent.

Hence, we can apply Lemma 4.3, obtaining that $<S(J^*), R(J^*)>$ is a candidate a.s of P.

Now, to conclude that J is an a.s of P, we need to show that, for every candidate a.s , $<S_2, R_2>$ of P, $R_2 \not\subset R(J^*)$.

Proving by contradiction, let M be a set of literals, s.t $<S(M), R(M)>$ is a candidate a.s of P and $R(M) \subset R(J^*)$. By Lemma 4.4, $\exists$ M*,i′ s.t M* is an a.s of $G_{i'}(P)$, and T(P,M*) is inconsistent and $S(M) = S(M^*)$, $R(M) = R(M^*)$.

From $R(M) \subset R(J^*)$ and $R(M) = R(M^*)$, it follows that

$$|R(M^*)| < |R(J^*)| \tag{4.6}$$

By Lemma 4.5, and the fact that M* is a.s of $G_{i'}(P)$

$i' < i$. (4.7)

From steps 22, 16, 12 and 9 of the algorithm and from the fact that M* is obtained at the i′th iteration, we can see that

$$C^{i'+1} \supseteq \{ \text{CONSTR}(R(M^*)) \}. \quad (4.8)$$

From (4.8) and from steps 22 and 16 of the algorithm, we can see that,

$C^i \supseteq C^{i'+1}$. Hence, from (4.8) we can see that

$$C^i \supseteq \{ \text{CONSTR}(R(M^*)) \}. \quad (4.9)$$

Since J* is a.s of $G_i(P) + C^i + \text{CONSTR}_P^*(L^i)$, by (4.9) (4.10)

J* satisfies CONSTR(R(M*)). (4.11)

Recall that CONSTR(R(M*)) is a constraint whose body consists of all atoms $\text{appl}_{cr}(r)$ for every $r \in R(M^*)$.

Hence, from (4.11) we have that:

$$R(M^*) \not\subset R(J^*). \quad (4.12)$$

Since R(M) = R(M*),

$R(M) \not\subset R(J^*)$.

This contradicts our initial assumption.

Hence the proof. □

Theorem 4.4: (Completeness) If J is an answer set of P, then $J \in$ CR-Models(P).

Proof: Since J is an answer set of P, from Definition 2.9, we know that

$\exists R_J \subseteq R$, s.t $<J, R_J>$ is a candidate a.s of P.

Since <J, R_J> is a candidate a.s, from Lemma 4.4, we know that $\exists$ M*, i, s.t S(M*) = J, R(M*) = R_J, M* is a.s of $G_i(P)$ and T(P, M*) is inconsistent.

Now let us prove that M* is also a.s of $G_i(P) + C^i$, where C^i is the value of variable C, from the algorithm, at iteration i of the outer while loop.

Proving by contradiction.

Let us assume that M* is not an a.s of $G_i(P) + C^i$. Then from the algorithm (steps 22, 16, 14, 12 and 9) and from Lemma 4.5, we can see that

$\exists$M', i', s.t i' < i, and

CONSTR(R(M')) $\subseteq C^i$, and

R(M') $\subset$ R(M*), and (4.13)

M' is a.s of $G_{i'}(P) + C^{i'}$, and

T(P, M') is inconsistent.

From steps 22, 16 and 15 of the algorithm, we can see that

S(M') $\in A^{i'}$. Hence, from Theorem 4.3, S(M') is a.s of P.

(4.14)

Therefore, from (4.13), (4.14) and item 2 of Definition 2.9, we can see that J is not an a.s of P.

Hence the contradiction. Therefore, M* is a.s of $G_i(P) + C^i$.

From this it can be shown that there exists some iteration, j, of the inner loop, s.t M* is a.s of $G_i(P) + C^i + L^{i,j}$ (where $L^{i,j}$ is the value of the variable L at iteration i of the outer while loop and iteration j of the inner loop). Hence $M^{i,j}$ = M* because of step 12 of the algorithm.

Since T(P,M*) is inconsistent, from step 15 and 14 of the algorithm, we can see that $J \in \mathtt{A}^i$. Theorem 4.2 guarantees that the algorithm terminates after some iteration, k, of the outer loop. Since variable $\mathtt{A}$ grows monotonically, $\mathtt{A}^i \subseteq \mathtt{A}^k$ = CR-Models(P).

Hence J ∈ CR-Models(P). □

Theorem 4.5: Algorithm CR-Models is sound and complete w.r.t the semantics of CR-Prolog.

Proof: From Theorem 4.3, we know that

If J ∈ CR-Models(P), then J is an a.s of P. (4.15)

From Theorem 4.4, we know that

If J is an answer set of P, then J ∈ CR-Models(P). (4.16)

From (4.15) and (4.16), we can see that,

J ∈ CR-Models(P), iff J is an a.s of P.

Hence the algorithm is sound and complete .□

Lemma 4.5: If M is an a.s of $G_i(P)$, then | R(M)| is i.

Proof: By the definition of a.s, we know that if M is an a.s of $G_i(P)$, M is closed under the rules of $G_i(P)$. Hence, M is closed under:

$$i\{\ appl_{cr}(r_1), ..., appl_{cr}(r_n)\}i \quad (4.17)$$

By the semantics of choice rules, it follows that $|\ atoms(appl_{cr}, G_i(P)) \cap M\ | = i$. By Definition 4.1 , $|R(M)| = i$

Therefore , $|R(M)| = |\ atoms(appl_{cr}, G_i(P)) \cap M\ |$

Hence the conclusion. □

Definition 4.3: Let is_preferred(r_1, r_2) $\in$ M . The set-length of is_preferred(r_1,r_2), denoted by $\bar{l}_M$ (is_preferred(r_1,r_2)), is the set defined as follows:

$0 \in \bar{l}_M$(is_preferred(r_1,r_2)) if prefer(r_1, r_2) $\in$ M,

$n+1 \in \bar{l}_M$(is_preferred(r_1,r_2)) if

$\exists$ r_3 s.t prefer(r_1, r_3) $\in$ M , and

is_preferred(r_3, r_2) $\in$ M, and

$n \in \bar{l}_M$(is_preferred(r_3, r_2)).

Definition 4.4: The length of is_preferred(r_1, r_2) w.r.t M, denoted by l_M(is_preferred(r_1, r_2)), is defined as the minimum element of $\bar{l}_M$(is_preferred(r_1,r_2)).

Lemma 4.1: If M is a.s of G_i(P), then <S(M), R(M)> is a view of P.

Proof: Let us start by proving that item (1) of Definition 2.6 holds, i.e M is an a.s of

$P_0 + \alpha(R(M))$.

M is a.s of G_i(P), where

G_i(P) consists of:

(1) $\prod_0$, the regular part of P.

(2) crname(r), where r is a cr-rule name.

(3) bodytrue(r) $\leftarrow$ body(ρ), where ρ is a cr-rule and r is its rule name.

(4) A constraint, $\leftarrow$ not bodytrue(R), $appl_{cr}$(R), crname(R).

(5) The choice rule {$appl_{cr}$(R): crname(R)}.

(6) α'(P), set of rules obtained from P by replacing every cr-rule, ρ with a rule head(ρ)$\leftarrow$ body(ρ), $appl_{cr}$(r), where r is its rule name.

(7) $\prod_p$, rules describing transitive closure of predicate prefer.

Let $G_i^1(P)$ be the program obtained from $G_i(P)$ by removing the constraint described in item (4). It can be shown that M is also an a.s of $G_i^1(P)$.

Consider now the program $G_i^2(P)$ consisting of:

(1) $\prod_0$,

(2) crname(r),

(3) The choice rule,

(4) $\alpha'(P)$,

(5) $\prod_p$.

By Splitting Set Lemma and the fact that M is an a.s of $G_i^1(P)$ it follows that M\ atoms({bodytrue}) is an a.s of $G_i^2(P)$.

Now consider the program $G_i^3(P)$, consisting of :

(1) $\prod_0$,

(2) $\alpha'(P)$,

(3) $\{a \mid a \in M \cap atoms(appl_{cr})\}$,

(4) $\prod_p$.

By Splitting Set Lemma and the fact that M\atoms({bodytrue}) is an a.s of $G_i^2(P)$ it follows that M\ atoms({bodytrue, crname}) is an a.s of $G_i^3(P)$.

Now consider the program $G_i^4(P)$, consisting of

(1) $\prod_0$,

(2) $\alpha'(P)$,

(3) $\{a \mid a \in M \cap atoms(appl_{cr})\}$,

By Splitting Set Lemma and the fact that M\atoms({bodytrue, crname}) is an a.s of $G_i^3(P)$ it follows that M\ atoms({bodytrue, crname, is_preferred}) is an a.s of $G_i^4(P)$.

Finally, consider $G_i^5(P)$, consisting of

(1) $\prod_0$,

(2) $\alpha(R(M))$.

Again by Splitting Set Lemma and the fact that M\atoms({bodytrue, crname, is_preferred}) is an a.s of $G_i^4(P)$, it follows that M\ atoms({bodytrue, crname, is_preferred, $appl_{cr}$}) is an a.s of $G_i^5(P)$.

By Definition 4.2, M\ atoms({bodytrue, crname, is_preferred, $appl_{cr}$}) = S(M).

Hence S(M) is a.s of $\prod_0 \cup \alpha(R(M))$. This proves that item (1) for the definition of view holds.

Now let us prove that item (2) of Definition 2.6 holds.

Here we need to show that if M is a.s of $G_i(P)$, then for every r_1, r_2 s.t $pref_{S(M)}(r_1, r_2)$, $\{r_1, r_2\} \not\subset R(M)$.

To prove the above statement first we need to prove that

If is_preferred(r_1,r_2) $\in$ M , then $pref_{S(M)}(r_1, r_2)$ holds. (4.18)

Let us prove (4.18) by induction on l_M(is_preferred(r_1, r_2)).

Base Case: l_M(is_preferred(r_1, r_2)) = 0.

From Definition 4.4 we can see that $0 \in \bar{l}_M$(is_preferred(r_1, r_2)).

Hence from Definition 4.3 , prefer(r_1, r_2) $\in$ M. By Definition 4.2,

prefer(r_1, r_2) $\in$S(M).

Therefore by Definition 2.5, $pref_{S(M)}(r_1, r_2)$, holds.

Inductive step:We assume that (4.18) holds for any r_1, r_2 s.t

$l_M(is_preferred(r_1, r_2)) \leq n$ and show that (4.18) holds if $l_M(is_preferred(r_1, r_2)) = n+1$.

If $l_M(is_preferred(r_1, r_2)) = n+1$ then from Definition 4.4 ,

$n+1 \in \overline{l}_M(is_preferred(r_1, r_2))$.

Therefore by Definition 4.3,

$\exists\ r_3$ s.t (a) $prefer(r_1, r_3) \in M$, and

(b) $is_preferred(r_3, r_2) \in M$, and

(c) $n \in \overline{l}_M(is_preferred(r_3, r_2))$.

From item (a) above , and by Definition 4.2, we obtain

$prefer(r_1, r_3) \in S(M)$.

By Definition 2.5, we obtain

$pref_{S(M)}(r_1, r_3)$. (4.19)

From item (c) above and from definition 4.4, we conclude that

$l_M\ (is_preferred(r_3, r_2)) \leq n$ is true.

By Inductive Hypothesis, $pref_{S(M)}(r_3, r_2)$ holds. (4.20)

Hence, by (4.19), (4.20) and Definition 2.5

$pref_{S(M)}(r_1, r_2)$ holds. (4.21)

Now let us prove that for every r_1, r_2 s.t $pref_{S(M)}(r_1, r_2)$, $\{r_1, r_2\} \not\subset R(M)$ holds.

Consider the following constraint in $G_i(P)$

$\leftarrow appl(R_1), appl(R_2), is_preferred(R_1, R_2), crname(R_1), crname(R_2)$.

Since M is closed under this constraint, its body cannot be satisfied by M.

Hence, if is_preferred(r_1, r_2) $\in$ M then

{appl(r_1), appl(r_2) } $\not\subset$ M. (4.22)

By (4.22) , Definitions 4.1 and 4.2,

if is_preferred(r_1, r_2) $\in$ S(M), $\neg$({r_1, r_2}$\subseteq$ R(M)) . (4.23)

Therefore, from(4.23), (4.18), it follows that for every r_1, r_2 s.t $pref_{S(M)}(r_1, r_2)$, {r_1,r_2} $\not\subset$ R(M) holds.

This proves that item (2) of the definition of view holds.

Now let us prove that item (3) of Definition 2.6 holds.

Consider the following rules in G_i(P):

$\leftarrow$ not bodytrue(R), $appl_{cr}$(R) (4.24)

bodytrue(r) $\leftarrow$ body(ρ), where ρ is a cr-rule and r is its rule name . (4.25)

Since M is a.s of G_i(P) , M is closed under (4.24) and (4.25).

Since M is closed under the constraint (7), its body cannot be satisfied by M.

Hence, if $appl_{cr}$ (r) $\in$ M, then bodytrue(r) $\in$ M. (4.26)

From Definition 4.1, the above statement (4.26) can be written as

If r $\in$ R(M), then bodyture(r) $\in$ M.

Hence item(3) of Definition 2.6 holds.

Hence the conclusion of Lemma 4.1.□

Lemma 4.2: If <Z, Q> is a view of P, then $\exists$M, i, s.t

1) S(M) = Z, R(M) = Q . 2) M is a.s of G_i(P).

Proof: From item(1) of Definition 2.6, we can see that

Z is a.s of $\Pi_0 \cup \alpha(Q)$, (4.27)

where Π_0 is regular part of program P

Consider program $G^1(P)$, consisting of

1) $\Pi_0 \cup \alpha(Q)$

2) $\Pi_{p,}$ consisting of

a) is_preferred(R_1,R_2) $\leftarrow$ prefer(R_1, R_2).

b) is_preferred(R_1,R_2) $\leftarrow$ prefer(R_1, R_3), is_preferred(R_3,R_2).

c) $\leftarrow$ $appl_{cr}(R_1)$, $appl_{cr}(R_2)$, is_preferred(R_1,R2).

It can be shown that

$M_1^* = Z \cup \{\text{is_preferred}(r_1,r_2) \mid pref_Z(r_1, r_2)\}$ is a.s of $G^1(P)$

The conclusion follows from :

1) Z a.s of $\Pi_0 \cup \alpha(Q)$,
2) Definition 2.5,
3) The fact that any a.s of $G^1(P)$ must be closed under rules a) and b) above,
4) The fact that, <Z, Q> is a view, by item 2 of Definition 2.6, the body of constraint (c) above is never satisfied.

Now consider the program $G^2(P)$, consisting of

1) $\Pi_0 \cup \alpha(Q)$,
2) $\Pi_{p,}$
3) { crname(r). | r is a cr-rule name},

We can see that, $M_2^* = M_1^* \cup$ { crname(r). | r is a cr-rule name} is a.s of $G^2(P)$.

Now let i = |Q| and $G^3(P)$, consisting of

1) $\Pi_0 \cup \alpha(Q)$,

2) Π_p,

3) { crname(r). | r is a cr-rule name},

4) The choice rule, i{ $appl_{cr}(r)$: crname(r) }.

Since item(4) is a choice rule, it can be shown that

$M_3^* = M_2^* \cup \{ appl_{cr}(r) \mid r \in Q\}$ is a.s of $G^3(P)$.

Now consider program $G^4(P)$, consisting of

1) $\Pi_0 \cup \alpha(Q)$,

2) Π_p,

3) { crname(r). | r is a cr-rule name},

4) i{ $appl_{cr}(r)$: crname(r) }i,

5) bodytrue(r) $\leftarrow$ body(ρ),

6) $\leftarrow$ not bodytrue(r), $appl_{cr}(r)$.

From item(3) of definition 2.6, we can see that

$M_4^* = M_3^* \cup \{$ bodytrue(r) | ρ is a cr-rule, r is name of ρ and body(ρ) is satisfied by Z }, is a.s of $G^4(P)$.

Now consider program $G^5(P)$, consisting of

1) $\Pi_0 \cup \alpha'(Q)$,

2) Π_p,

3) { crname(r). | r is a cr-rule name},

4) i{ $appl_{cr}(r)$: crname(r) }i,

5) bodytrue(r) $\leftarrow$ body(ρ),

6) $\leftarrow$ not bodytrue(r), $appl_{cr}$(r).

where $\alpha'(Q)$, is the set of rules obtained by replacing every cr-rule, ρ, of P with a rule head(ρ) $\leftarrow$ body(ρ), $appl_{cr}$(r), where r is its rule name.

Since $M_4^* \cap atoms\{appl_{cr}\} = \{ appl_{cr}(r) \mid r \in Q\}$, it can shown that

M_4^* is an a.s of $G^5(P)$.

Since $G^5(P)$ is exactly equal to $G_i(P)$, M_4^* is a.s of $G_i(P)$. Also, $S(M_4^*) = Z$ and $R(M_4^*) = Q$ by construction. This proves that $\exists M^*$, i, s.t $S(M^*) = Z$, $R(M^*) = Q$, and M^* is a.s of $G_i(P)$.

Lemma 4.3: If M is a.s of $G_i(P)$ and T(P,M) is inconsistent, then <S(M), R(M)> is a candidate a.s of P.

Proof: Let us start by proving that <S(M), R(M)> is a view of P.

Applying Lemma 4.1 to the premises of our lemma, we get

<S(M), R(M)> is a view of P. (4.28)

Now let us prove that <S(M), R(M)> is also a candidate a.s .

By contradiction, assume that <S(M), R(M)> is not a candidate a.s.

Since <S(M), R(M)> is a view, by Definition 2.8,

$\exists$ a view <S′, R′> of P , s.t <S′, R′> dominates <S(M), R(M)> (4.29)

By Lemma 4.2, $\exists M'$, i′ , s.t $S(M') = S'$, $R(M') = R'$, and M' is a.s of $G_{i'}(P)$. (4.30)

Now, let us look at the relationship between M′ and T(P,M).

Let M′′ consist of M′ together with the following atoms:

a) $\{o_appl_{cr}(r) \mid appl_{cr}(r) \in M\}$.

b) $\{o_is_preferred(r_1,r_2) \mid is_preferred(r_1, r_2) \in M\}$

c) dominates.

Let us now prove that M′′ is a.s of T(P, M) (only the key steps are shown)

(From (4.29) and Definition 2.7, we obtain that

$\exists r_1 \in R'$, $r_2 \in R(M)$ and $pref_{S'\cap S(M)}(r_1, r_2)$. (4.31)

From (4.30) and (4.31),

$\exists r_1 \in R(M')$, $r_2 \in R(M)$, and $pref_{S(M')\cap S(M)}(r_1, r_2)$. (4.32)

By Definition 4.1, $appl_{cr}(r_1) \in M'$ and $appl_{cr}(r_2) \in M$.

Hence $M'' \supseteq \{appl_{cr}(r_1), o_appl_{cr}(r_2)\}$. (4.33)

By (4.32) and Definition 2.5, and by the fact that $G_{i'}(P)$ includes

a) $is_preferred(R_1, R_2) \leftarrow prefer(R_1, R_2)$.

b) $is_preferred(R_1,R_2) \leftarrow prefer(R_1, R_3), is_preferred(R_3, R_2)$.

we can see that $is_preferred(r_1,r_2) \in M'$, and that

$M'' \supseteq \{ is_preferred(r_1, r_2), o_is_preferred(r_1,r_2) \}$. (4.34)

By (4.33) and (4.34), the body of rule that defines atom dominates in T(P,M) is satisfied.

This concludes the proof that M′′ is a.s of T(P,M).

Since M′′ is a.s of T(P,M), T(P,M) is consistent.

This contradicts the premise of the Lemma. The contradiction was obtained by

assuming that <S(M), R(M)> is not a candidate a.s of P.

Hence <S(M), R(M)> is a candidate a.s of P. □

Lemma 4.4: If <Z, Q> is a candidate a.s of P, then ∃M, i , s.t

1) S(M) = Z and R(M) = Q,

2) M is a.s of $G_i(P)$,

3) T(P,M) is inconsistent.

Proof: We know that <Z, Q> is a candidate a.s of P. From Definition 2.8, we know that every candidate a.s is also a view.

Therefore <Z, Q> is a view.

From Lemma 4.2, we know that if <Z, Q> is a view, then $\exists$M, i , s.t S(M) = Z, and

R(M) = Q, and

M is a.s of $G_i(P)$.

Now let us prove that T(P,M) is inconsistent.

Since <Z, Q> is a candidate a.s, from Definition 2.8, we know that for every view <Z', Q'> of P, <Z', Q'> does not dominate <Z, Q>.

Therefore from Definition 2.7, we know that

$$\neg\exists\ r_2, r_1 \text{ s.t } r_2 \in Q',\ r_1 \in Q, \text{ and } pref_{Z' \cap Z}(r_2, r_1) \tag{4.35}$$

Since <Z', Q'> is a view , from Lemma 4.2, we know that

$\exists$ M', i', s.t S(M') = Z', R(M') = Q', M' a.s of $G_{i'}(P)$.

From Definition 4.1 and Definition 2.5, (4.35) can be written as

$\neg\exists\ appl_{cr}(r_1) \in M, appl_{cr}(r_2) \in M'$, s.t is_preferred$(r_2, r_1) \in M$, and

is_preferred$(r_2, r_1) \in M'$. (4.36)

Recall that T(P, M) consists of :

dominates $\leftarrow$ $appl_{cr}(I)$, $o_appl_{cr}(J)$, is_preferred(I, J), o_is_preferred(I, J) (4.37)

$\leftarrow$ not dominates. (4.38)

Where $o_appl_{cr}(r)$ holds for each r such that $appl_{cr}(r) \in M$, and

o_is_preferred(r_1, r_2) holds, for r_1, r_2, s.t is_preferred$(r_1, r_2) \in M$.

Notice that (4.36) guarantees that the body of (4.37) is never satisfied. Since atom dominates is only present in the head of (4.37), dominates is never true.

Therefore, due to constraint (4.38), T(P,M) is inconsistent.

Hence the proof. □

CHAPTER V

IMPROVING THE EFFICIENCY OF CR-MODELS

In this chapter we describe the way in which we improved the efficiency of CR-Models. In CR-Models, we have a large program which is the hard reduct, hr(P) and we repeatedly add new rules(ex : C and $CONSTR_P$*(L)) to this program(at steps 9 and 14 of the algorithm the generator is extended by constraints) . If both these programs have to be grounded each time then, the process is very slow. So, instead we ground hr(P) once and merge it with the non grounded new rules.

There are two different ways to do this kind of merging. One approach is to use lparse's –g option and the other is to use our New Merging Algorithm. Now we describe and compare these two approaches.

To understand these approaches we first need to understand the structure of Lparse output and the details of the different sections in this output. This explanation is given in the Appendix.

5.1 Lparse Merging

Lparse is the front end that transforms programs with variables into grounded programs that Smodels understands. SMODELS consists of two parts, the first part, called Lparse, takes a program P as a parameter and outputs a program g(P), s.t g(P) contains no variables and A is an a.s of g(P) iff A is a.s of P. g(P) serves as an input to answer set finding algorithm(also called smodels), which computes its answer sets. When a program P is expanded by a new program R, the answer sets of P+R are computed by grounding P+R and applying smodels to ground(P + R). Even if as in case of step 9 of the algorithm, the ground(hr(P)) is already computed, this information is lost and all the work of grounding is repeated . To avoid this situation Lparse allows option –g. This option allows the user to read in a previously grounded program and add new rules to it.

The following example shows the usage of –g option.

Example 5.1: Consider the following program test

```
const max_a = 2.
a( 1 … 2).
```

b(X) :- a(X), not c(X).

Suppose now we want to add constraint, C, to program test, we first need to ground test To ground this we use a command line :

% lparse test > g(test)

This command grounds test and stores the output in the file g(test). Suppose now we would like to compute a.s of test $\cup$ C, we can use the command line:

$ lparse –g g(test) C | smodels 0

The output of the command line is

Smodels version 2.26. Reading ….. done

Answer: 1

Stable Model: b(1), a(2).

Lparse reads the ground program in, recognizing and storing the atom names in the process. Then it simply parses the rules, unmodified, to Smodels and starts to read constraint.pl and process.

We ran some experiments using this –g option. During our experiments we found that its implementation had some errors. We fixed these errors and ran experiments using their version of Lparse with our fixes.

In our experiments, we found that Lparse ran very slow compared to the rest of the algorithm and hence this lowered the overall efficiency of the inference engine. So, in order to increase the efficiency we devised our own algorithm to merge the ground and non-ground programs.

5.2 New Merging Algorithm (NMA)

To understand this algorithm we first need to understand the structure of Lparse output. This explanation is given in the Appendix.

The NMA algorithm is specific to the generator and tester programs of our algorithm . The algorithm is based on the following conditions.

1) The atoms from the grounded program, Π_G , appear only in the bodies of the rules in the non-ground program, Π_{NG}.
2) The heads of the rules of $\Pi_{NG,}$ are new atoms of arity zero.

Here to ground the rules we make use of the atom numbers present in the symbol table[See Smodels Internal Format in the Appendix] of the lparse output and for new atoms we extend the atom section of the symbol table

Let gr(P), be a function that takes a non-grounded program P, returns the grounding of the program P.

In NMA we:

1) ground P.
2) extend symbol table of P by the new rules R
3) compute the answer set of the new ground progam.

Proposition 5.1: For any set of rules Π and R, if they satisfy conditions 1) and 2),

gr(Π and R) = gr(gr(Π) + R).

Example 5.2: Consider the following non-ground rule from the Tester program.

dominates :- appl(I), o_appl(J), is_preferred(I, J), o_is_preferred(I,J).

The rule can be grounded as follows: (let *domind* be the index for the new atom dominates).

Step 1: For every possible instantiation of o_appl(J), where J is replaced by some v1, retrieve from the symbol table the index of atom o_appl(v1). Let i_1 be such index.

Step 2: For each instantiation of o_is_preferred(I, v1), let v2 denote the constant that replaces I . Let i_2 be the index of o_is_preferred(v2, v1)

Step 3: Retrive the index, i_3, of appl(v1) .

Step 4: Retrive the index, i_4, of is_preferred(v2, v1) .

Step 5: Output the line: 1 *domind* 4 0 i_1 i_2 i_3 i_4

We conducted several experiments and found that this algorithm was more efficient compared to lparse merging technique.

5.2.1 Description of the New Merging Algorithm

CR-Models makes use of a generate and test approach [see section 3.2]. We make use of the merging technique during both the generation and testing phases. Let us look at the algorithm in each of these phases seperately.

In the following discussion we assume that the reader is familiar with the structure of the Lparse ground output. Unfamiliar readers can refer to Appendix.

5.2.1.1 Numbering of Atoms

During grounding lparse transforms the complex rules to those accepted by Smodels. Internally Smodels uses integers as atoms and the atom names are stored in a separate symbol table(see Appendix).

In our algorithm, we keep a marker to indicate the end of the symbol table. This marker atom is given the next highest unique atom number. Now, when we want to add new atoms to this table, we increment the numbering by 1 and assign it to the new atom. Let GetCreate_NewAtomnumber(atom), be a function that returns such unique new atom number.

And during the computation we often need to find the atom number for an atom from the symbol table. Let Get_AtomNumber(atom) be such a function which takes an atom name and returns the atom number. Let Get_AtomNumbers(set of atoms) be a function which takes as input a set of atoms and returns a set of atom numbers from the symbol table by repeatedly invoking Get_AtomNumber(atom) on each of the atoms in the set.

5.2.1.2 Generation Phase

In section 3.2 , we saw that during the generation phase we add two sets of constraints $CONSTR_P^*(L)$ and CONSTR(C), to the Generator. Therefore, the generation phase may be viewed as generating the answer set of the program :

$$\text{Generator} + CONSTR_P^*(L) + CONSTR(C). \quad (5.1)$$

1) hr(P).

2) ok :- i { applcr(R) : crname(R) } i.

3) :- not ok.

In the above program the hard reduct is a large program and we repeatedly add some rules and new constraints to it.

If both the hard reduct and the rest of the programs have to be grounded each time then the process is very slow. So, instead we :

1)Ground the hard reduct once using Lparse ;

2) Ground the new constraints and rules using our own algorithm ;

3) Merge the hard reduct with the new constraints and rules.

Let us look at the algorithm to ground each of the rules above.

1) Grounding of the rule: ok :- i { applcr(R) : crname(R) } i.

The above rule follows the syntax of Lparse but, it is not in the standard rule type format(See Appendix), that lparse uses for grounding.

So for grounding, we convert it into the standard constraint type rule by replacing it with the following 3 rules:

a) ok_L :- i { applcr(R) : crname(R) } . (5.2)

b) bad_U :- i+1 { applcr(R) : crname(R) }. (5.3)

c) ok :- ok_L, not bad_U. (5.4)

Following are the steps for grounding the rule(5.2)

Step 1: Find the total number of applcr(R) atoms present in the atom section(See Appendix) of the grounded hard reduct and store it in variable *total*.

Step 2: Find the atom number for the new atom ok_L by invoking the function GetCreate_NewAtomNumber(ok_L) and let index $ok_L ind$ be returned by this function.

Step 3: For every possible instantiation of applcr(R) , where R is replaced by some value v1, retrieve from the atom section the index of atom

applcr(v1) by calling function Get_AtomNumber(applcr(v1)). Let

{ind_i ... ind_n }be the set of indices obtained.

Step 4: Output the rule according to the constraint rule form i.e,

schema: 2 head #literals #negative bound *negative* *positive*

2 $ok_L ind$ *total* 0 i ind_i ... ind_n

Grounding of the rule (5.3)

It is the same as the grounding algorithm for rule (5.2) except for replacing $ok_L ind$ by $bad_U ind$, which is returned by function GetCreate_NewAtomNumber(bad_U). And the bound now is i+1. Therefore, following is the rule that is output at Step 4:

2 $bad_U ind$ *total* 0 i+1 ind_i ... ind_n

Grounding of the rule (5.4)

Step 1: Get the index *okind* for the new atom ok, by calling function

GetCreate_NewAtomNumber(ok).

Step 2: Output the rule according to the basic rule format

schema: 1 head #literals #negative *negative* *positive*

1 *okind* 2 1 $bad_U ind$ $ok_L ind$

Grounding of rule: :- not ok.

Step 1: Output the rule according to the basic rule format ,

with head being false i.e assigned the number 1.

1 1 1 1 *okind*.

Grounding of the constraints in $CONSTR_P^*(L)$.

Step 1: For every set of literals in L, say {l_1 … l_n }, find the set of all the

literals from the atom section say $\{l_{n+1} \ldots. l_m\}$ which are not present in the set $\{l_1 \ldots l_n\}$.

Step 2: Find the index for the literals present in both the sets $\{l_1 \ldots l_n\}$ and $\{l_{n+1} \ldots. l_m\}$ from the atom section. Let $\{ind_1 \ldots ind_n\}$ and $\{ ind_{n+1} \ldots. ind_m \}$ be the sets of indices for the two sets returned by function Get_AtomNumbers($\{l_1 \ldots l_n\}$) and Get_AtomNumbers($\{l_{n+1} \ldots. l_m\}$) respectively.

Step 3: Let *s1* be the size of $\{l_1 \ldots l_n\}$ and *s2*, the size of $\{l_{n+1} \ldots. l_m\}$.

Step 4: Output the regular rule with head being false as follows:

1 1 *s1+s2 s2* $ind_{n+1} \ldots ind_m$ $ind_1 \ldots ind_n$.

Grounding of the constraint CONSTR(C).

Step 1: For every set of literals say $\{l_1 \ldots l_n\}$, find the index for all the literals . Let $\{ind_1 \ldots ind_n\}$ be this set of indices returned by Get_SetOfAtomNumbers($\{l_1 \ldots. l_n\}$).

Step 2: Let *sum* be the size of set $\{ l_1 \ldots l_n \}$.

Step 3: Output the rule

1 1 *sum* 0 $ind_1 \ldots ind_n$

After grounding all the rules of the generator except for the hard reduct, which is grounded just once, we append all the ground rules to the rule section(See Appendix) of the Lparse output of the hard reduct. Finally, we give this new ground input to Smodels, which finds the a.s of the Generator.

5.2.1.3 Testing Phase

In section 3.2 we saw that the Tester consists of the following:

a) A fact

1) o_applcr(r), for each r such that applcr(r) ∈ M (5.5)

2) o_is_preferred(r1,r2), for each r1,r2, such that (5.6)

is_preferred(r1,r2) ∈ M.

b) The rules

1) dominates ← applcr(I), o_applcr(J), is_preferred(I,J), (5.7)

o_is_preferred(I,J).

2) ← not dominates. (5.8)

c) hr(P).

As we discussed earlier in the Generation phase, hr(P) is grounded once and new rules and facts are added to it.

Grounding of fact (5.5)

Step 1: For every atom applcr(r) present in the a.s returned by the Generator, create a new atom o_applcr(r). For every such new atom find the index by functiom Get_NewAtomNumber(o_applcr(r)). Let *oind* be such an index.

Step 2: For every such new fact, output the rule

1 *oind* 0 0

Grounding of fact (5.6)

This is similar to grounding of fact (5.5), except for creating new fact o_is_preferred(r1,r2) for every is_preferred(r1,r2) and finding the new index *oisind*. For every such new atom the rule is output as:

1 *oisind* 0 0

Grounding of rule (5.7)

Given in Example 5.2.

Grounding of rule (5.8)

Step 1: Let *domind* (as seen in Example 5.2) be the index of the new atom dominates.

Step 2: Output the rule, with head being false.

1 1 1 1 *domind*

Finally, similar to the Generation phase, append all the ground rules to the rule section of the Lparse ouput of the hard reduct and give this as input to the Smodels. Smodels then computes the a.s of the Tester program.

The experimental results of using Own-Merging is give in Chapter VI.

CHAPTER VI

EXPERIMENTAL RESULTS

To experimentally investigate the efficiency of CR-Models, we compared its performance with the previous CR-Prolog inference engine. We also give improved performance results achieved by replacing Lparse-Merging by New Merging.

6.1 Comparing Lparse and New Merging Algorithm

We compared the efficiency of the two implementations by using a planning program.

Blocks world domain:

This is a planning domain where the agent has to arrange blocks according to what is mentioned in the goal . We are given an initial and a goal state as below:

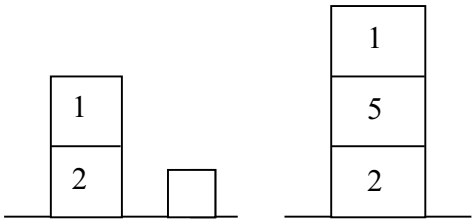

Figure 6.1: Blocks world domain

We conducted experiments on different variations of this domain. Let us look at each one of these

BW1:

This is the initial scenario where the initial state is as given above and the agent is supposed to arrange the blocks as shown in the goal state. The plan is supposed to be 3 steps long and there are no concurrent actions allowed. The actions are generated using the below cr-rule which says that if the goal is not achieved at time T, any action A may possibly occur.

r(A, T) : occurs(A,T) +- T < n, not goal_h(T).

BW2:

Next we add to the above program the following constraint which says that it is impossible to move block 1 onto the table at time zero.

← occurs(move(1, t), 0).

BW3:

Next we replace the above constraint by the following 3 rules:

1) The first rule says that it is impossible to move block 1 onto the table unless it is allowed to do so.

   ← occurs(move(1,t), T), not allowed(move(1, t), T).

2) The second rule is a cr-rule and it says that it may possibly be true that block 1 is allowed to move onto the table at time T.

   $r_m(T)$: allowed(move(1, t), T) ←+.

3) The third rule says that it is preferred to allow the possibility of moving block 1 onto the table at time T+1 than at time T. (i.e, block 1 should be moved to the table as late as possible)

   prefer($r_m(T+1)$, $r_m(T)$).

BW 4:

Next we add 5 blocks, 10-15, initially on the table . The goal remains unchanged. The table below shows the time taken to compute one plan for each variation of the blocks world domain, using both merging techniques. The experiments show an improvement of about two orders of magnitude. For the last column, the

inference engine based on Lparse- Merging terminated for lack of memory after more than 3,000 seconds.

Table 6.1: Experimental Results- New Merging and Lparse Merging

Programs	New Merging (sec)	Lparse-Merging (sec)
BW1	1.1	160.5
BW2	8.7	1931
BW3	1.4	242.4
BW4	32.1	> 3,000

6.2. Comparing CR-Models and CR-Old

We compared CR-Models with the previous inference engine for CR-Prolog. To distinguish between the two we will call the previous prototype CR-Old.

Before we show the experimental results of the comparison, let us briefly describe the algorithm on which CR-Old is based.

Function CR-Old( )

Input: A CR-Prolog program P.

Output: The answer sets of P.

Var :

n : The total number of cr-rules in P.

A : set of answer sets of P.

B : collection of sets of literals.

M_1 : set of literals.

M_2 : set of literals.

M : collection of sets of literals.

```
1    i = 0; A = Ø; B = Ø; C = Ø;
2    while( i < n)
3    {
4      M := all the models of  G_i(P) + CONSTR(C);
5       B := B U M;
6         for  M_1  in  M
7                                   C :=  C U { R(M_1) };
8         i++;
9    }
10     for each  M_1 in B do
11   {
12         if
13           ¬∃ M_2 in B s.t  M_2 dominates M_1
14         then
15           A := A  U { M_1} ;
16    }
17    return   A;
```

Figure 6.2: Algorithm CR-Old

So in CR-Old we generate all the views first and then check if each of these views is an a.s of the CR-Prolog program.

During our experiments we found that CR-Models was more efficient than CR-Old. The preferences were now handled more efficiently and always gave the most intuitive solutions.

The table 6.2 shows the time taken to compute one plan for each variation of the blocks world domain described in section 6.1, using both CR-Models and CR-Old. The experiments show an improvement of about 50% when there are no preferences and improvement of 1-2 orders of magnitude with preferences.

Table 6.2 Experimental Results- CR-Models and CR-Old

Programs	CR-Models (sec)	CR-Old (sec)
BW1	1.1	2.2
BW2	8.7	14.3
BW3	1.4	16.4
BW4	32.1	1885.9

We also compared the performance difference between CR-Models and CR-Old by conduction experiments in the RCS(Reaction Control System).

RCS is described below:

1. The RCS is the Shuttle's system that has primary responsibility for maneuvering the spacecraft while it is in space.
2. RCS is divided into three subsystems: The forward RCS, left RCS and the right RCS. Each subsystem controls jets located in different parts of the craft.
3. There is almost no connection between the subsystems, with only important exception the crossfeed pipe, which connects the plumbing of the left and right subsystems
4. The crossfeed is valve controlled and is intended to be used when one of the two subsystems is affected by faults that prevent the use of its own propellent.

The experiments were conducted for the following RCS planning module.

- Domain: Reaction Control System of the space shuttle
- Goal: Preparing the RCS for the manuever.
- Action generation:

0{ occ(A, T): act_of(A, R)}1← subsystem(R).

- Requirements:
  - avoid the use of crossfeed if at all possible;
  - avoid the use of computer commands if at all possible;

Encoding the first requirement:

← subsystem(R), act_of(A, R),

occ(A, T), opens_xfeed_valve(A),

not xfeed_allowed(R, T).

r1(R, T): xfeed_allowed(R, T) ←+ subsystem(R).

Encoding the second requirement:

← subsystem(R), action_of(A, R) ,

occur(A, T) ,

sends_computer_command(A) ,

not ccs_allowed(R, T).

r2(R, T): ccs_allowed(R, T) ←+ subsystem(R).

According to the operational manual for shuttle controllers, use of computer commands is preferred to using the crossfeed.

prefer(r2(Sy1,T1), r1(Sy2,T2)) ← system(Sy1),

system(Sy2),

time(T1), time(T2),

not stop_prefer.

rstop : stop_prefer ←+.

The figure 6.3 shows the results of the experiments that were conducted on 10 problem instances. The number of faults is the same in each instance -- 3 mechanical faults and 0 electrical. However, the components affected by the faults are different in each instance.

dels was 2-3 orders of magnitude faster than CR-Old.

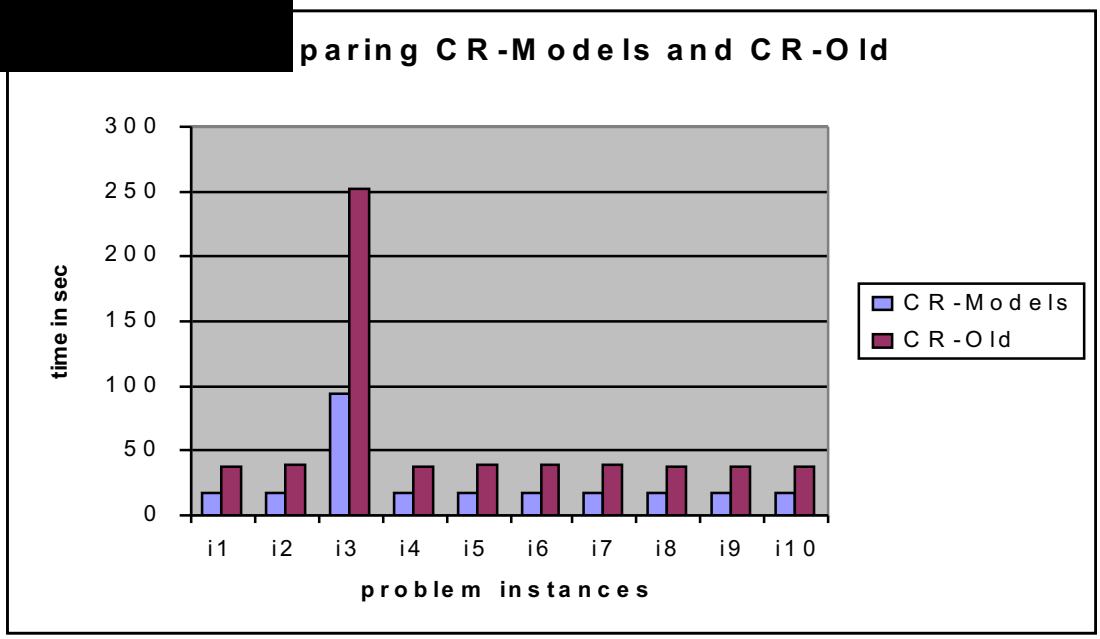

Figure 6.3 : Comparing CR-Models and CR-Old for RCS Planning Module

CHAPTER VII

CONCLUSIONS

A-Prolog lacks the ability to gracefully perform the reasoning needed for certain types of conflict resolution, e.g finding the best explanations of unexpected observations. To solve this problem CR-Prolog – an extension of A-Prolog by consistency restoring rules with preferences was introduced. The most intuitive solutions correspond to those

models that best satisfy the preferences expressed, and minimize the application of cr-rules.

An efficient inference engine was required to compute the a.s of a CR-Prolog program and which would handle the preferences efficiently.

To achieve this objective, we :

1) Designed an algorithm which could find the a.s of CR-Prolog program and deal with preferences efficiently.
2) Implemented the algorithm using C++ with Smodels as the underlying inference engine
3) Proved the soundness and completeness of the algorithm.
4) Improved the efficiency by 2-3 orders of magnitude by developing developing our New Merging technique.
5) The efficiency of the engine was evaluated by comparing it with the Lparse Merging based engine and also with the older CR-Prolog inference engine.

Based on our experiments we can conclude that CR-Models is an efficient inference engine for CR-Prolog programs. The answer sets are precisely defined. These are the "most intuitive" answer sets. It is faster than the previous engine.

7.1 Future work

The inference engine can be extended to handle CR-Prolog with ordered disjunction[13]. CR-Prolog with ordered disjunction allows ordered disjunction in the

head of both regular and consistency restoring rules. The use of ordered disjunction, when the preference order on a set of alternatives is total, allows for a more concise and easier to read, representation of knowledge. The flexibility of preference relation is such that metapreferences from LPOD[16] can be encoded using directly its preference relation, rather than requiring the definition of a new type of preference.

REFERENCES

[1] M.Gelfond. Representing knowledge in A-Prolog. To appear in Computational Logic: form logic programming to the future, collection in the papers in the honour of B. Kowalski.

[2] R.Kowalski. *Logic for problem solving*. North-Holland, Amsterdam 1979.

[3] W.Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective,* 375-398, Spring-Verlag, New York, 1999.

[4] J.McCarthy. Elaboration Tolerance. Manuscript, 1997.

[5] M.Gelfond and V.Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proc of the fifth Int'l Conf. and Symp.,* pages 1070-1080, 1988.

[6] M. Balduccini, M. Barry, M. Gelfond, M. Nogueira, and R.Watson. An A-Prolog decision support system for the Space Shuttle. *Lecture Notes in Computer Science-Proceedings of Practical Aspects of Declarative Languages '01,* (2001), 1990: 169-183.

[7] E.Erdem, V. Lifschitz, and M. Wong. Wire routing and satisfiability planning. *Proc. Of CL-2000,* 822-836, 2000.

[8] Y. Dimopoulos, N. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. *Lecture Notes in Artificial Intelligence – Recent Advances in AI Planning, Proc. Of the 4th European Conference on Planning, ECP'97,* 1348:169-181, 1997

[9] T. Soininen and I.Niemela. Developing a declarative rule language for applications in product configuration. *In practical Aspects of Declarative Languages,* LNCS 1551, pages 305-319, 1999.

[10] P. Simons. Extending and implementing the stable model semantics. Research Report A58, Helsinki University of Technology, April, 2000.

[11] T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N. Leone, S. Perri, and G. Pfeifer. System Description: DLV. In *Logic Programming and Nonmonotonic Reasoning – 6th International Conference, LPNMR'01,* in AI(LNAI), 2173:409-412, 2001.

[12] System built by M. Balduccini at Texas Tech University.

[13] Marcello Balduccini and Veena S. Mellarkod. CR-Prolog2: CR-Prolog with Ordered Disjunction. In *ASP03 Answer Set Programming: Advances in Theory and Implementation*, volume 78 of *CEUR Workshop proceedings*, Sep 2003.

[14] P. Simons. *Efficient implementation of the stable model semantics for normal logic programs*. Research Report 35, Helsinki University of Technology, September 1995.

[15] The DLV System for Knowledge Representationand Reasoning. Nicola Leone and Gerald Pfeifer and Wolfgang Faber and Thomas Eiter and Georg Gottlob and Simona Perri and Francesco Scarcello.

[16] Gerhard Brewka, Ilkka Niemela, and Tommi Syrjanen. Implementing ordered disjunction using answer set solvers for normal programs. In Sergio Flesca and Giovanbattisa Ianni, editors, *Proceedings of the 8th European Conference on Artificial Inelligence(JELIA 2002),* Sep 2002.

[17] Marcello Balduccini and Michael Gelfond. Logic programs with consistency restoring rules. In Patrick Doherty, John McCarthy, and Mary-Anne Williams, editors, *International Symposium on Logical Formalization of Commonsense Reasoning, AAAI* 2003 Spring Symposium Series, Mar 2003.

[18] T. Syrjanen. Lparse User's Manual available at http://www.tcs.hut.fi/Software/smodels/

[19] http://www.cs.utexas.edu/users/vl/tag/

[20] http://www.cs.utexas.edu/users/tag/

[21] http://www.krlab.cs.ttu.edu/

APPENDIX

Smodels Internal Format

The language that smodels 2.x accepts is much simpler than the one accepted by lparse. During grounding lparse transforms the complex rules to those accepted by smodels. There are four different rule types: basic rules, constraint

rules, choice rules, and weight rules.

Additionally, the minimize statements are internally represented by their own rule type. The maximize statements are changed into minimize statements by negating all literals in them.

Basic rules are the rules that don't use any extended features. Constraint rules correspond to lparse rules of the form

```
a :- 2 { b, c, not d }.
```

choice rules have the form

```
{ a, b, c } :- d, e, not f, not g.
```

and weight rules have the form

```
a :- 2 [ b=1, c=2, not d=3 ].
```

In all cases there may be only one special construct in one rule and there are only lower bounds for constraint and weight rules.

Internally *smodels* uses integers as atoms and the atom names are stored in a separate symbol table. *smodels* expects to read first the actual rules of the program, next the symbol table, and finally the compute statement. The different parts are separated by a line that has only a '0'.

The different sections are best introduced by an example. Consider the following program:

Program A.1

```
a :- not b.
b :- not a.
```

```
:- b.

compute { a }.
```

The internal format for this program is:

```
1 2 1 1 3

1 3 1 1 2

1 1 1 0 3

0

2 a

3 b

0

B+

2

0

B-

1

0

1
```

The first part of the listing consists of the the rules of the program:

```
1 2 1 1 3

1 3 1 1 2

1 1 1 0 3

0
```

The firrst number denotes the rule type. All of the rules are basic rules so the number is one in all cases. The next number identifies the head of the atom. In this case, the atom *a* is represented by 2 and the atom *b* by 3. The atom number 1 is an internal atom named *_false* that is true when a model candidate should be rejected.

Next comes the body definition. The first number is the total number of literals in the body and the second one is the number of negative literals. The rest of the line contains the number of literals, with negative ones being in front. The line with just 0 signals the end of the rules.

The second part is the symbol table containing the atom definitions:

2 a

3 b

0

If an atom is left out the symbol table, it is considered to be a hidden atom and it is not printed in the model. In this example, the first atom is hidden.

The third part contains the compute statement

B+

2

0

B-

1

0

1

After B+ comes the positive compute statement, that is, a list of atoms that should be true in the model. After B- comes a list of atoms that shouldn't be in the model. The last 1 signifies the number of models that should be calculated.

Table A.1: The mapping of atoms that are used in the examples

atom	number

a	1
b	2
c	3
d	4
e	5

A.1 Rule Types

Basic Rule

A basic rule has the form:

head #literals #negative negative positive

Where *head* is the atom that is the head of the rule, *#literals* is the total number of literals in the rule body, *#negative* is the number of negative literals, and *positive* is the list of positive literals.

Example A.1

Let the atoms be defined as in Table A.1. Then, the rule:

a :- b, not c, d, not e.

is represented as:

1 1 4 2 3 5 2 3

Constraint Rule

A constraint rule has the form:

2 head *#literals #negative* bound *negative positive*

where *head, #literals, #negative, negative,* and *positive* are as with basic rules and bound is the amount of literals in the body that has to be true so that the head is true .

Example A.2

Given the bindings in Table A.1, the rule

a :- 2 { b, c, not d }.

is represented as

2 1 3 1 2 4 2 3

Choice Rule

A choice rule has the form

3 #heads *heads* #literals #negative *negative positive*

Most of the entries are the same as with basic rules. The entry #heads denotes the number of the atoms in a choice rule head and *heads* is the list of the atoms in the rule head.

Example B.3

Using the usual bindings for the atoms, the rule

{ a, b, c } :- e, not d.

is represented as

3 3 1 2 3 2 1 4 5

Weight Rule

A weight rule has the form

5 head bound #lits #negative *negative positive weights*

Most of the entries are the same as in constraint rules. The entry *weights* is a list of the weights of the literals in the rule body.

Example B.4

The rule:

a :- 3 [b=1, not c=2].

is represented as:

5 1 3 2 1 3 2 2 1

Minimize Rule

A minimize rule is of the form:

6 0 #lits #negative *negative positive weights*

Note that each literal have to have an explicit weight assigned to it. Maximization can be achieved by negating all literals in the statement body.

Example B.5

The statement:

Maximize [a=5, not b = 10].

is represented as:

6 0 2 1 2 1 10 5