$\mathcal{SPARC}$ – Sorted ASP with Consistency Restoring Rules

Evgenii Balai[1], Michael Gelfond[2], and Yuanlin Zhang[2]

[1] Kyrgyz-Russian Slavic University
[2] Texas Tech University, USA
iensen2@mail.ru, michael.gelfond@ttu.edu, y.zhang@ttu.edu

Abstract. This is a preliminary report on the work aimed at making CR-Prolog – a version of ASP with consistency restoring rules – more suitable for use in teaching and large applications. First we describe a sorted version of CR-Prolog called $\mathcal{SPARC}$. Second, we translate a basic version of the CR-Prolog into the language of DLV and compare the performance with the state of the art CR-Prolog solver. The results form the foundation for future more efficient and user friendly implementation of $\mathcal{SPARC}$ and shed some light on the relationship between two useful knowledge representation constructs: consistency restoring rules and weak constraints of DLV.

1 Introduction

The paper continues work on design and implementation of knowledge representation languages based on Answer Set Prolog (ASP) [1]. In particular we concentrate on the extension of ASP called CR-Prolog – Answer Set Prolog with consistency restoring rules (CR-rules for short) [2]. The language, which allows a comparatively simple encoding of indirect exceptions to defaults, has been successfully used for a number of applications including planning [3], probabilistic reasoning [4], and reasoning about intentions [5]. This paper is a preliminary report on our attempts to make CR-Prolog (and hence other dialects of ASP) more user friendly and more suitable for use in teaching and large applications. This work goes in two different, but connected, directions. First we expand the syntax of CR-Prolog by introducing sorts. Second, we translate a basic version of the CR-Prolog into the language of DLV with weak constraints [6] and compare the efficiency of the resulting DLV based CR-Prolog solver with the CR-Prolog solver implemented in [7]. The original hope for the second part of the work was to obtain a substantially more efficient inference engine for CR-Prolog. This was a reasonable expectation – the older engine is built on top of existing ASP solvers and hence does not fully exploit their inner structure. However this didn't quite work out. Each engine has its strong and weak points and the matter requires further investigation. But we believe that even preliminary results are of interest since they shed some light on the relationship between two useful knowledge representation constructs: CR-rules and weak constraints. The first goal requires a lengthier explanation. Usually, a program of an Answer Set Prolog based language is understood as a pair, consisting of a signature and a collection of logic programming rules formed from symbols of this signature.

The syntax of the language does not provide any means for specifying this signature – the usual agreement is that the signature consists of symbols explicitly mentioned in the programming rules. Even though in many cases this provides a reasonable solution there are also certain well-known (e.g., [8, 9]) drawbacks:

1. Programs of the language naturally allow unsafe rules which
   - May lead to change of the program behavior under seemingly unrelated updates. A program $\{p(1).\quad q \leftarrow not\ p(X).\quad \neg q \leftarrow not\ q.\}$ entails $\neg q$, but this conclusion should be withdrawn after adding seemingly unrelated fact $r(2)$. (This happens because of the introduction of a new constant 2 which leads to a new ground instance of the second rule: $q \leftarrow not\ p(2)$.)
   - Cause difficulties for the implementation of ASP solvers. That is why most implementations do not allow unsafe rules. The corresponding error messages however are not always easy to decipher and the elimination of errors is not always an easy task.
2. The language is untyped and therefore does not provide any protection from unfortunate typos. Misspelling $john$ in the fact $parent(jone, mary)$ will not be detected by a solver and may cost a programmer unnecessary time during the program testing.

There were several attempts to address these problems for ASP and some of its variants. The $\#domain$ statements of input language of $lparse$ [9] — a popular grounder used for a number of ASP systems — defines sorts for variables. Even though this device is convenient for simple programs and allows to avoid repetition of atoms defining sorts of variables in the bodies of program's rules it causes substantial difficulties for medium size and large programs. It is especially difficult to put together pieces of programs written at different time or by different people. The same variable may be declared as ranging over different sorts by different $\#domain$ statements used in different programs. So the process of merging these programs requires renaming of variables. This concern was addressed by Marcello Balduccini [10] whose system, $RSig$, provided an ASP programmer with means for specifying sorts of parameters of the language predicates[3]. $RSig$ is a simple extension of ASP which does not require any shift in perspective and involves only minor changes in existing programs. Our new language, $\mathcal{SPARC}$, can be viewed as a simple modification of $RSig$. In particular *we propose to separate definition of sorts from the rest of the program and use this separation to improve the type checking and grounding procedure*.

2 The Syntax and Semantics of $\mathcal{SPARC}$

In this section we define a simple variant of $\mathcal{SPARC}$ which contains only one predefined sort $\mathbf{nat}$ of natural numbers. Richer variants may contain other predefined sorts with precise syntax which would be described in their manuals. The discussion will be sufficiently detailed to serve as the basis for the implementation of $\mathcal{SPARC}$ reasoning system.

[3] In addition, $RSig$ provides simple means for structuring a program into modules which we will not consider here.

Let $\mathcal{L}$ be a language defined by the following grammar rules:

```
<identifier> :- <small_letter> | <identifier><letter> |
        <identifier><digit>
<variable> :- <capital_letter> | <variable><letter> |
        <variable><digit>
<non_zero_digit> :- 1|...|9
<digit> :- 0 | <non_zero_digit>
<positive_integer> :- <non_zero_digit> |
        <positive_integer><digit>
<natural_number> :- 0 | <positive_integer>
<op> :- + | - | * | mod
<arithmetic_term> :- <variable> | <natural_number> |
        <arithmetic_term> <op> <arithmetic_term> |
        (<arithmetic_term>)
<symbolic_function> :- <identifier>
<symbolic_constant> :- <identifier>
<symbolic_term> :- <variable> | <symbolic_constant> |
        <symbolic_function>(<term>,...,<term>)
<term> :- <symbolic_term> | <arithmetic_term>
<arithmetic_rel> :-  = | != | > | >= | < | <=
<pred_symbol> :- <identifier>
<atom> :- <pred_symbol>(<term>,...,<term>) |
        <arithmetic_term> <arithmetic_rel> <arithmetic_term> |
        <symbolic_term> = <symbolic_term> |
        <symbolic_term> != <symbolic_term>
```

Note that relations = and != are defined on pairs of arithmetic and pairs of non-arithmetic terms. The first is a predefined arithmetic equality, i.e. $2 + 3$=5, $2 + 1$!=1, etc. The second is an identity relation[4]. By a *ground term* we mean a term containing no variables and no symbols for arithmetic functions [11].

From now on we assume a language $\mathcal{L}$ with a fixed collection of symbolic constants and predicate symbols. A $\mathcal{SPARC}$ program parametrized by $\mathcal{L}$ consists of three consecutive parts:

```
<program> :-
        <sorts definition>
        <predicates declaration>
        <program rules>
```

The first part of the program starts with the keywords:

$sorts\ definition$

and is followed by the sorts definition:

[4] In the implementation non-arithmetic identity should be restricted to comply with the syntax of lparse and other similar grounders.

Definition 1. By *sort definition* in $\mathcal{L}$ we mean a collection Π_s of rules of the form

$$a_0 \leftarrow a_1, ..., a_m, not\ a_{m+1}, ..., not\ a_n.$$

such that

- a_i are atoms of $\mathcal{L}$ and a_0 contains no arithmetic relations;
- Π_s has a unique answer set $\mathcal{S}$[5].
- For every symbolic ground term t of $\mathcal{L}$ there is a unary predicate s such that $s(t) \in \mathcal{S}$.
- Every variable occurring in the negative part of the body, i.e. in at least one of the atoms $a_{m+1}, \ldots, a_n$, occurs in atom a_i for some $0 < i \leq m$.

Predicate s such that $s(t) \in \mathcal{S}$ is called a *defined sort* of t. The language can also contain *predefined sorts*, in our case $\mathbf{nat}$. Both, defined and predefined sorts will be referred to simply as *sorts*. (Note that a term t may have more than one sort.)

The last condition of the definition is used to avoid unexpected reaction of the program to introduction of new constants (see example in the introduction). The condition was introduced in [8] where the authors proved that every program Π satisfying this condition has the following property, called language independence: for every sorted signatures Σ_1 and Σ_2 groundings of Π with respect to Σ_1 and Σ_2 have the same answer sets. This of course assumes that every rule of Π can be viewed as a rule in Σ_1 and Σ_2.

The second part of a $\mathcal{SPARC}$ program starts with a keyword

$predicates\ declaration$

and is followed by statements of the form

$pred\_symbol(sort, \ldots, sort)$

We only allow one declaration per line. Predicate symbols occurring in the declaration must differ from those occurring in sorts definition. Finally, multiple declarations for one predicate symbol with the same arity are not allowed.

The third part of a $\mathcal{SPARC}$ program starts with a keyword

$program\ rules$

and is followed by a collection Π_r of regular and consistency restoring rules of $\mathcal{SPARC}$ defined as follows:

regular rule:

$$l_0 \vee \ldots \vee l_m \leftarrow l_{m+1}, \ldots, l_k, not\ l_{k+1} \ldots not\ l_n \quad (1)$$

[5] As usual by $\mathcal{S}$ we mean answer set of a ground program obtained from Π by replacing its variables with ground terms of $\mathcal{L}$. We assume that the program has non-empty Herbrand universe

CR-rule:

$$l_0 \stackrel{+}{\leftarrow} l_1, \ldots, l_k, not\ l_{k+1} \ldots not\ l_n \quad (2)$$

where l's are literals[6] of $\mathcal{L}$. Literals occurring in the heads of the rules must not be formed by predicate symbols occurring in Π_s. In this paper, $\leftarrow$ and :- are used interchangeably, so are $\stackrel{+}{\leftarrow}$ and :+.

As expected, program Π_r is viewed as a shorthand for the set of all its ground instances which *respect the sorts defined by* Π_s. Here is the precise definition of this notion.

Definition 2. Let $gr(r)$ be a ground instance of a rule r of Π_r, i.e. a rule obtained from r by replacing its variables by ground terms of $\mathcal{L}$. We'll say that $gr(r)$ respects sorts of Π_s if every occurrence of an atom $p(t_1, \ldots, t_n)$ of $gr(r)$ satisfies the following condition: if $p(s_1, \ldots, s_n)$ is the predicate declaration of p then $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$ respectively. By $gr(\Pi_r)$ we mean the collection of *all* ground instances of rules of Π_r which respect sorts of Π_s.

Note that according to our definition $gr(r)$ may be empty. This happens, for instance, for a rule which contains atoms $p_1(X)$ and $p_2(X)$ where p_1 and p_2 require parameters from disjoint sorts.

Let us now define answer sets of a ground $\mathcal{SPARC}$ program Π. We assume that the readers are familiar with the definition of answer sets for standard ASP programs. Readers unfamiliar with the intuition behind the notion of consistency restoring rules of CR-Prolog are referred to the Appendix.

First we will need some notation. The set of regular rules of a $\mathcal{SPARC}$ program Π will be denoted by R; the set of cr-rules of Π will be denoted by CR. By $\alpha(r)$ we denote a regular rule obtained from a consistency restoring rule r by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$; α is expanded in a standard way to a set X of cr-rules, i.e. $\alpha(X) = \{\alpha(r) \ : \ r \in X\}$.

Definition 3. *(Abductive Support)*
A collection X of cr-rules of Π such that

1. $R \cup \alpha(X)$ is consistent (i.e. has an answer set) and
2. any R_0 satisfying the above condition has cardinality which is greater than or equal to that of R

is called an *abductive support* of Π.

Definition 4. *(Answer Sets of $\mathcal{SPARC}$ Programs)*
A set A is called an *answer set* of Π if it is an answer set of a regular program $R \cup \alpha(X)$ for some abductive support X of Π.

[6] By a literal we mean an atom a or its negation $\neg a$. Note in this paper, we use $\neg$ and - interchangeably.

To complete the definition of syntax and semantics of a $\mathcal{SPARC}$ program we need to note that though such program is defined with respect to some language $\mathcal{L}$ in practice this language is extracted from the program. We always assume that terms of $\mathcal{L}$ defined by a $\mathcal{SPARC}$ program P are arithmetic terms and terms defined by the sorts definition[7]; predicate symbols are those occurring in sorts definition and predicate declaration. Now we are ready to give an example of a $\mathcal{SPARC}$ program.

Example 1. [$\mathcal{SPARC}$ programs]
Consider a $\mathcal{SPARC}$ program P_1:

```
sorts definition
s1(1).
s1(2).
s2(X+1) :-
          s1(X).
s3(f(X,Y)) :-
          s1(X),
          s1(Y),
          X != Y.
predicates declaration
p(s1)
q(s1,s3)
r(s1,s3)
program rules
p(X).
r(1,f(1,2)).
q(X,Y) :-
        p(X),
        r(X,Y).
```

The sort declaration of the program defines ground terms 1, 2, 3, $f(1,2)$, $f(2,1)$ with the following defined sorts:

$s_1 = \{1, 2\}$
$s_2 = \{2, 3\}$
$s_3 = \{f(1,2), f(2,1)\}$

Of course, 1, 2, and 3 are also of the sort $\mathbf{nat}$. The sort respecting grounding of the rules of Π is

```
p(1).
p(2).
r(1,f(1,2)).
q(1,f(1,2)) :-
        p(1),
        r(1,f(1,2)).
```

[7] A term t is defined by Π_s if for some sort s, $s(t)$ belongs to the answer set of Π_s

```
q(2,f(1,2)) :-
       p(2),
       r(2,f(1,2)).
q(1,f(2,1)) :-
       p(1),
       r(1,f(2,1)).
q(2,f(2,1)) :-
       p(2),
       r(2,f(2,1)).
```

The answer set of the program is $\{p(1), p(2), r(1, f(1,2)), q(1, f(1,2))\}$. (We are not showing the sort atoms.)

Consider now a $\mathcal{SPARC}$ program P_2:

```
sorts definition
t(a,b).
t(c,1).
s1(X) :- t(X,Y).
s2(Y) :- t(X,Y).
s3(a).
predicates declaration
p(s1,s2).
program rules
p(X,Y) :- s3(X),t(X,Y).
```

The sort respecting grounding of the program is

```
p(a,b) :- s3(a),t(a,b).
```

Its answer set is $\{p(a,b), t(a,b)\}$.

Another example can be obtained by restating the CR-Prolog program from Example 4 in the Appendix by adding sort definitions $s_1(a)$ and $s_2(d(a))$ and predicates declarations $p(s_1)$, $q(s_1)$, $c(s_1)$ and $ab(s_2)$. One can easily check that, as expected, the answer set of the resulting program is $\{\neg q(a), c(a), \neg p(a))\}$.

3 Translation of $\mathcal{SPARC}$ Programs to DLV Programs

DLV [12] is one of the well developed solvers for ASP programs. We select DLV as the target language mainly because of its *weak constraints* [6] which can be used to represent cr-rules. A *weak constraint* is of the form

$$:\sim l_1, \ldots, l_k, not\ l_{k+1} \ldots not\ l_n.$$

where l_i's are literals. (Weak constraints of DLV allow preferences which we ignore here.) Informally, weak constraints can be violated, but as many of them should be

satisfied as possible. The *answer sets* of a program P with a set W of weak constraints are those of P which minimize the number of violated weak constraints.

We first introduce some notations before presenting the translation algorithm.

Definition 5. *(DLV counterparts of $\mathcal{SPARC}$ programs)*
A DLV program P_2 is *a counterpart* of $\mathcal{SPARC}$ program P_1 if answer sets of P_1 and P_2 coincide on literals from the language of P_1.

Definition 6. Given a $\mathcal{SPARC}$ program P, we associate a unique number to each of its cr-rules. The *name of a cr-rule* r of Π is a term $rn(i, X_1, ..., X_n)$ where rn is a new function symbol, i is the unique number associated with r, and $X_1, ..., X_n$ is the list of distinct variables occurring in r.

For instance, if rule $p(X, Y) \leftarrow q(Z, X, Y)$ is assigned number 1 then its name is $rn(1, X, Y, Z)$.
In what follows we describe a translation of $\mathcal{SPARC}$ programs into their DLV counterparts.

Algorithm 1 *($\mathcal{SPARC}$ program translation)*
Input*: a $\mathcal{SPARC}$ program P_1.*
Output*: a DLV counterpart P_2 of P_1.*

1. Set variable P_2 to $\emptyset$, and let `appl`/1 be a new predicate not occurring in P_1.
2. Add all rules of the sorts definition part of P_1 to P_2.
3. For any program rule r of P_1,
   3.1. Let

$$s = \{s_1(t_1), ..., s_n(t_n) \mid p(t_1, ..., t_n) \text{ occurs in } r \text{ and } p(s_1, ..., s_n) \in P_1\},$$

   and let rule r' be the result of adding all elements of s to the body of r.
   3.2. If r' is a regular rule, add it to P_2.
   3.3. If r' is a cr-rule of the form

$$q \stackrel{+}{\leftarrow} body.$$

   add to P_2 the rules
   `appl(`$rn(i, X_1, ..., X_n)$`)` $\vee$ $\neg$`appl(`$rn(i, X_1, ..., X_n)$`)` `:-` *body*`.`
   `:~` `appl(`$rn(i, X_1, ..., X_n)$`),` *body*`.`
   q `:-` `appl(`$rn(i, X_1, ..., X_n)$`),` *body*`.`
   where $rn(i, X_1, ..., X_n)$ is the name of r.

The intuitive idea behind the rules added to P_2 in 3.3. is as follows: `appl(`$rn(i, X_1, ..., X_n)$`)` holds if the cr-rule r is used to obtain an answer set of the $\mathcal{SPARC}$ program; the first rule says that r is either used or not used; the second rule, a weak constraint, guarantees that r is not used if possible, and the last rule allows the use of r when necessary.

The correctness of the algorithm is guaranteed by the following theorem whose complete proof can be found at http://www.cs.ttu.edu/research/krlab/pdfs/papers/sparc-proof.pdf.

Theorem 1. *A DLV program P_2 obtained from a $\mathcal{SPARC}$ program P_1 by Algorithm 1 is a DLV counterpart of P_1.*

The translation can be used to compute an answer set of $\mathcal{SPARC}$ program P.

Algorithm 2 *(Computing an answer set of a $\mathcal{SPARC}$ program)*
Input*: a $\mathcal{SPARC}$ program P.*
Output*: an answer set of P.*

1 Translate P into its DLV counterpart P'.
2 Use DLV to find an answer set S of P'.
3 Drop all literals with predicate symbol `appl` from S and return the new set.

Example 2. To illustrate the translation and the algorithm, consider the following program.

```
sorts definition
s(a).
predicates declaration
p(s)
q(s)
program rules
p(X) :- not q(X).
-p(X).
q(X) :+ .
```

After step 2 of Algorithm 1 , P' becomes:

```
s(a).
```

After the execution of the loop 3 of this algorithm for the first and second program rule, P' becomes

```
s(a).
p(X) :- not q(X),s(X).
¬p(X):- s(X).
```

Assuming the only cr-rule is numbered by 1, after the algorithm is applied to the third rule, P' becomes

```
s(a).
p(X) :- not q(X),s(X).
¬p(X):- s(X).
appl(rn(1,X)) ∨ ¬appl(rn(1,X)) :- s(X).
:∼ appl(rn(1,X)), s(X).
q(X) :- appl(rn(1,X)), s(X).
```

Given the program P', DLV solver returns an answer set

$$\{s(a), appl(rn(1,a)), q(a), \neg p(a)\}$$

After dropping `appl(`$rn(1,a)$`)` from this answer set, we obtain an answer set

$$\{s(a), q(a), \neg p(a)\}$$

for the original program.

4 Experimental Results

We have implemented a $\mathcal{SPARC}$ program solver, called *crTranslator* (available from the link in [13]), based on the proposed translation approach. CRModels2 [7] is the state of the art solver for CR-prolog programs. To compare the performance of the DLV based solver to CRModels2, we use the classical benchmark of the reaction control system for the space shuttle [3] and new benchmarks such as representing and reasoning with intentions [5], and the shortest path problem.

Clock time, in seconds, is used to measure the performance of the solvers. Since the time complexity of translation is low, the recorded problem solving time does not include the translation time.

In this experiment, we use DLV build BEN/Dec 21 2011 and CRModels2 2.0.12 [14] which uses ASP solver Clasp 2.0.5 with grounder Gringo 3.0.4 [15]. The experiments are carried out on a computer with Intel Core 2 Duo CPU E4600 at 2.40 Ghz, 3GB RAM, and Cygwin 1.7.10 on Windows XP.

4.1 The First Benchmark: Programs for Representing and Reasoning with Intentions

Recently, CR-Prolog has been employed to represent and reason with intentions [5]. We compare crTranslator with CRModels2 on the following scenarios proposed in [5]: Consider a row of four rooms, r_1, r_2, r_3, r_4 connected by doorways, such that an agent may move along the row from one room to the next. We say that two people *meet* if they are located in the same room. Assume that initially our agent Bob is in r_1 and he intends to meet with John who, as Bob knows, is in r_3. This type of intention is frequently referred to as an intention to achieve the goal. The first task is to design a simple plan for Bob to achieve this goal: move from r_1 to r_2 and then to r_3. Assuming that as Bob is moving from r_1 to r_2, John moves from r_3 to r_2, the second task is to recognize the unexpected achievement of his goal and not continue moving to r_3. Programs to implement these two tasks are given as $\mathcal{B}_0$ and $\mathcal{B}_1$ respectively in [5].

Tasks	CRModels2	crTranslator
task 1	104	**11**
task 2	104	**101**

Fig. 1. CPU time for intention reasoning benchmark using CRModels2 and crTranslator

In this experiment, crTranslator has a clear advantage over CR-Models2 on task 1 and similar performance on task 2.

4.2 The Second Benchmark: Reaction Control System of Space Shuttle

USA-Smart is a CR-prolog program to find plans with improved quality for the operation of the Reaction Control System (RCS) of the Space Shuttle. Plans consist of a sequence of operations to open and close the valves controlling the flow of propellant from the tanks to the jets of the RCS.

In our experiment, we used the USA-Smart program with four instances: fmc1 to fmc4 [16]. The $\mathcal{SPARC}$ variant of the USA-Smart program is written as close as possible to USA-smart. The results of the performance of crTranslator and CRModels for these programs are listed in Figure 2.

Instances	CRModels2	crTranslator
fmc1	**29.0**	74.0
fmc2	**11.6**	34.0
fmc3	**6.0**	8907.0
fmc4	**30.5**	22790.0

Fig. 2. CPU time for reaction control system using CRModels2 and crTranslator

We note that these instances have small abductive supports (with sizes of the supports less than 9) and relatively large number of cr-rules (with more than 1200). This can partially explain why CRModels2 is faster because it finds the abductive support by exhaustive enumeration of the candidate supports starting from size 0 to all cr-rules in an increasing manner.

4.3 The Third Benchmark: Shortest Path Problem

Given a simple directed graph and a pair of distinct vertices of the graph, the shortest path problem is to find a shortest path between these two vertices. Given a graph with n vertices and e edges, its *density* is defined as $e/(n * (n - 1))$. In our experiment, the problem instances are generated randomly based on the number of vertices and the density of the graph. The density of the graphs varies from 0.1 to 1 so that the shortest paths involve abductive supports of different sizes. To produce graphs with longer shortest path (which needs larger abductive supports), we zoom into the density between 0 to 0.1 with a step of 0.01. To reduce the time solving the problem instances, as density increases, we use smaller number of vertices. Given a graph, we define *the distance* between a pair of vertices as the length of the shortest path between them. For any randomly generated graph, we select any two vertices such that their distance is the longest among those of all pairs of vertices. The problem is to find the shortest path between these two vertices.

The $\mathcal{SPARC}$ programs and CR-prolog programs are written separately due to the difference between these two languages, but we make them as similar as possible and

use exactly the same cr-rules in both programs. The experimental results are listed in Figure 3.

From the results, CRModels2 is faster on a majority of cases. Again, crTranslator is faster when the size of the abductive support is large. The graphs with density between 0.02 and 0.03 have support size of 16 while the other graphs (except the one of density 0.01) have support sizes not more than 12. Further investigation is needed to have a better understanding of the performance difference between the two solvers.

Number of vertices	Density	CRModels2	crTranslator
60	0.01	4.0	**0.2**
60	0.02	5.1	**0.4**
60	0.03	5.7	**0.8**
60	0.04	**9.1**	66.5
60	0.05	**29.2**	337.0
60	0.06	**235.7**	4451.8
40	0.07	**7.4**	19.9
40	0.08	**8.4**	154.6
40	0.09	**7.0**	32.6
30	0.1	**6.0**	16.8
30	0.2	**39.9**	9711.4
20	0.3	**7.6**	54.9
20	0.4	**9.3**	52.2
20	0.5	**16.4**	234.8
20	0.6	**9.6**	51.7
20	0.7	**14.3**	52.0
20	0.8	**17.6**	58.8
20	0.9	**22.2**	69.1
20	1.0	**5.5**	55.6

Fig. 3. CPU time for solving shortest path problem using CRModels2 and crTranslator

5 Conclusion and Future Work

This paper describes a sorted version of CR-Prolog called $\mathcal{SPARC}$, presents a translation of consistency restoring rules of the language into weak constraints of DLV, and investigates the possibility of building efficient inference engines for $\mathcal{SPARC}$ based on this translation. This is a preliminary report. There is a number of steps which should be made to truly develop $\mathcal{SPARC}$ into a knowledge representation language of choice for teaching and applications. In particular we plan the following:

– Expand $\mathcal{SPARC}$ to include a number of useful language constructs beyond the original language of ASP such as aggregates and optimization constructs. In this

expansion, instead of committing to a particular syntax, we are planning to allow users to select their favorite input language such as that of DLV or LPARSE or GRINGO and provide the final system with support for the corresponding language.
– Provide $\mathcal{SPARC}$ with means to specify different preference relations between sets of cr-rules, define and investigate answer sets minimal with respect to these preference relations, and implement the corresponding $\mathcal{SPARC}$ solvers.
– Design and implement $\mathcal{SPARC}$ grounders to directly use the sort information provided by definitions and declaration of a program. The emphasis will be on error checking and incrementality of the grounders.
– Investigate more efficient reasoning algorithms for $\mathcal{SPARC}$. DLV uses a more advanced technique of branch and bound to process weak constraints while CRModels employs a more primitive search algorithm. However, our experiments show that the latter is not necessarily slower. Further understanding of these two approaches is expected to inspire new techniques for building more efficient solvers for $\mathcal{SPARC}$ programs.
– Expand $\mathcal{SPARC}$ and its solvers to other extensions of ASP including ACC [17] and P-log [4].

Acknowledgement

The work of Gelfond and Zhang was partially supported by NSF grant IIS-1018031.

Appendix: CR-Prolog

This Appendix contains a short informal introduction to CR-Prolog. The version discussed here is less general than the standard version — in particular it omits the treatment of preferences which is a task orthogonal to the goals of this paper. One of the original goals of the CR-Prolog was to provide a construct allowing a simple representation of exceptions to defaults, sometimes referred to as **indirect exceptions**. Intuitively, these are rare exceptions that come into play only as a last resort, to restore the consistency of the agent's world view when all else fails. The representation of indirect exceptions seems to be beyond the power of "pure" ASP [1] which prompted the introduction of cr-rules. To illustrate the problem let us consider the following example.

Example 3. [Indirect Exception in ASP]
Consider an ASP representation of the default "elements of class c normally have property p":

$$p(X) \leftarrow c(X), \mathit{not}\ ab(d(X)), \mathit{not}\ \neg p(X).$$

(where $d(X)$ is used as the name of the default) together with the rule

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

and two observations:

$$c(a).$$
$$\neg q(a).$$

It is not difficult to check that this program is inconsistent. No rules allow the reasoner to prove that the default is not applicable to a (i.e. to prove $ab(d(a))$) or that a does not have property p. Hence the default must conclude $p(a)$. The second rule implies $q(a)$ which contradicts the observation.

There, however, seems to exist a commonsense argument which may allow a reasoner to avoid inconsistency, and to conclude that a is an indirect exception to the default. The argument is based on the **Contingency Axiom** for default $d(X)$ which says that *Any element of class c can be an exception to the default $d(X)$ above, but such a possibility is very rare and, whenever possible, should be ignored*. One may informally argue that since the application of the default to a leads to a contradiction, the possibility of x being an exception to $d(a)$ cannot be ignored and hence a must satisfy this rare property.

The CR-Prolog is an extension of ASP capable of encoding and reasoning about such rare events. In addition to regular logic programming rules the language allows *consistency restoring* rules of the form

$$l_0 \stackrel{+}{\leftarrow} l_1, \ldots, l_k, \mathit{not}\ l_{k+1}, \ldots, \mathit{not}\ l_n \tag{3}$$

where l's are literals. Intuitively, the rule says that if the reasoner associated with the program believes the body of the rule, then it "may possibly" believe its head. However, this possibility may be used only if there is no way to obtain a consistent set of beliefs by using only regular rules of the program.

The following Example shows the use of CR-Prolog for representing defaults and their indirect exceptions.

Example 4. [Indirect Exception in CR-Prolog]
The CR-Prolog representation of default $d(X)$ may look as follows

$$p(X) \leftarrow c(X), \mathit{not}\ ab(d(X)), \mathit{not}\ \neg p(X).$$
$$\neg p(X) \stackrel{+}{\leftarrow} c(X).$$

The first rule is the standard ASP representation of the default, while the second rule expresses the Contingency Axiom for default $d(X)$. Consider now a program obtained by combining these two rules with an atom $c(a)$.
Assuming that a is the only constant in the signature of this program, the program's answer set will be $\{c(a), p(a)\}$. Of course this is also the answer set of the regular part of our program. (Since the regular part is consistent, the Contingency Axiom is ignored.) Let us now expand this program by the rules

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

The regular part of the new program is inconsistent. To save the day we need to use the Contingency Axiom for $d(a)$ to form the abductive support of the program. As a result the new program has the answer set $\{\neg q(a), c(a), \neg p(a))\}$. The new information does

not produce inconsistency as in the analogous case of ASP representation. Instead the program withdraws its previous conclusion and recognizes a as a (strong) exception to default $d(a)$.

The possibility to encode rare events which may serve as unknown exceptions to defaults proved to be very useful for various knowledge representation tasks, including planning, diagnostics, and reasoning about the agent's intentions.

References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9**(3/4) (1991) 365–386
2. Balduccini, M., Gelfond, M.: Logic Programs with Consistency-Restoring Rules. In Doherty, P., McCarthy, J., Williams, M.A., eds.: International Symposium on Logical Formalization of Commonsense Reasoning. (November 2003) 9–18
3. Balduccini, M.: USA-Smart: Improving the quality of plans in answer set planning. In Jayaraman, B., ed.: Practical Aspects of Declarative Languages. Volume 3057 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2004) 135–147
4. Gelfond, M., Rushton, N.: Causal and probabilistic reasoning in P-log. In Dechter, R., Geffner, H., Halpern, J., eds.: A tribute to Judea Pearl. College Publications (2010) 337–359
5. Blount, J., Gelfond, M.: Reasoning about the intentions of agents. In: Logic Programs, Norms and Action. Volume 7360 of Lecture Notes in Computer Science. (2012) 147–171
6. Faber, W.: Disjunctive Datalog with Strong and Weak Constraints: Representational and Computational Issues. PhD thesis, Institut fur Informationssysteme, Technische Universitat Wien (1998)
7. Balduccini, M.: CR-MODELS: An inference engine for CR-Prolog. In Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the 9th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'07). Volume 3662 of Lecture Notes in Artificial Intelligence., Springer (2007) 18–30
8. McCain, N., Turner, H.: Language independence and language tolerance in logic programs. In: ICLP. (1994) 38–57
9. Syrjanen, T.: Lparse 1.0 users manual, http://www.tcs.hut.fi/software/smodels/
10. Balduccini, M.: Modules and signature declarations for a-prolog: Progress report. In: Software Engineering for Answer Set Programming Workshop (SEA07). (2007)
11. Gelfond, M.: Answer sets. Handbook of Knowledge Representation. Elsevier (2008) 285–316
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7** (2006) 499–562
13. Balai, E.: http://code.google.com/p/crtranslator
14. Balduccini, M.: Crmodels, http://marcy.cjb.net/crmodels/index.html
15. Gebser, M., Kaufman, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. In Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07). Volume 3662 of LNAI., Springer (2007) 136–148
16. USA-Smart: http://marcy.cjb.net/rcs-asp/rcs/
17. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Ann. Math. Artif. Intell. **53**(1-4) (2008) 251–287