USA-Smart: Improving the Quality of Plans in Answer Set Planning

Marcello Balduccini

Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
phone: +1 806 742 1191
fax: +1 806 742 3519
marcello.balduccini@ttu.edu

Abstract. In this paper we show how CR-Prolog, a recent extension of A-Prolog, was used in the successor of USA-Advisor (USA-Smart) in order to improve the quality of the plans returned. The general problem that we address is that of improving the quality of plans by taking in consideration statements that describe "most desirable" plans. We believe that USA-Smart proves that CR-Prolog provides a simple, elegant, and flexible solution to this problem, and can be easily applied to any planning domain. We also discuss how alternative extensions of A-Prolog can be used to obtain similar results.

Keywords: planning, answer set programming, preferences.

1 Introduction

In recent years, A-Prolog – the language of logic programs with the answer set semantics [11] – was shown to be a useful tool for knowledge representation and reasoning [10]. 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 [15,6,14,18] has allowed the use of A-Prolog for a diverse collection of applications [12,17,19,16].

In previous papers [17,2], we have shown how A-Prolog was used to build a decision support system for the Space Shuttle (USA-Advisor). USA-Advisor is capable of checking the correctness of plans and of finding plans 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.

Under normal conditions, pre-scripted plans exist that tell the astronauts what should be done to achieve certain goals. However, failures in the system may render those plans useless, and the flight controllers have to come up with alternative sequences that allow the completion of the mission and ensure the safety of the crew. USA-Advisor is designed to help in this task by ensuring that plans meet both criteria. Moreover, its ability to quickly generate plans allows the controllers to concentrate on higher-level tasks.

In this paper we show how CR-Prolog [1,3], a recent extension of A-Prolog, was used in the successor of USA-Advisor (USA-Smart) in order to improve the quality of the plans returned. The general problem that we address here is that of improving the quality of plans by taking in consideration statements that describe "most desirable" plans. We believe that USA-Smart proves that CR-Prolog provides a simple, elegant, and flexible solution to this problem, and can be easily applied to any planning domain.

The present work builds on the ability of CR-Prolog to return "most reasonable" solutions to a problem encoded by a CR-Prolog program. Besides regular A-Prolog rules, the programmer specifies a set of rules (cr-rules) that may possibly be applied – although that should happen as rarely as possible – as well as a set of preferences on the application of the cr-rules. The "most reasonable" solutions correspond to those models that best satisfy the preferences expressed, and minimize the applications of cr-rules.

The paper is structured as follows. We start with a brief, informal, presentation of CR-Prolog. Next, we describe the Reaction Control System and the design of USA-Smart. In the following two sections we present the planner used in USA-Smart. Finally, we discuss related work, summarize the paper, and draw conclusions.

2 CR-Prolog

CR-Prolog is an extension of A-Prolog that consists of the introduction of consistency-restoring rules (cr-rules) with preferences.

CR-Prolog programs consist of regular rules and cr-rules. A *regular rule* is a statement:

$$r : h_1 \text{ or } h_2 \text{ or } \ldots \text{ or } h_k :- l_1, \ldots, l_m, \\ \text{not } l_{m+1}, \ldots, \text{not } l_n \tag{1}$$

where r is the name of the rule, h_i's and l_i's are literals, h_1 or $\ldots$ or h_k is the head, and $l_1, \ldots, l_m,$ not $l_{m+1}, \ldots,$ not l_n is the body. The intuitive reading of (1), 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."[1] In order to increase the readability of the programs, we allow regular rules with *choice atoms* in the head [15]:

$$r : L\{p(\bar{X}) : q(\bar{X})\}U :- l_1, \ldots, l_m, \\ \text{not } l_{m+1}, \ldots, \text{not } l_n \tag{2}$$

Intuitively, the head of this rule defines subset $p \subseteq q$, such that $L \leq |p| \leq U$. Although this form can be translated in rules of type (1), it allows for more concise programs, in particular when writing planners.

A *cr-rule* is a statement of the form:

$$r : \quad h_1 \text{ or } h_2 \text{ or } \ldots \text{ or } h_k \stackrel{+}{-} l_1, \ldots, l_m, \\ \text{not } l_{m+1}, \ldots, \text{not } l_n \tag{3}$$

The cr-rule intuitively says that, if the agent believes $l_1, \ldots, l_m$ and does not believe $l_{m+1}, \ldots, l_n$, then it "may possibly" believe one element of the head. This possibility is used only if there is no way to obtain a consistent set of beliefs using regular rules only. (For the definition of the semantics of CR-Prolog, see [3].)

Let us see how cr-rules work in practice. Consider the following program:

$$r_1 : p \text{ or } q \stackrel{+}{-} \text{not } t. \\ r_2 : s.$$

Since the program containing only r_2 is consistent, r_1 need not be applied. Hence, there is only one answer set: $\{s\}$. On the other hand, program

$$r_1 : p \text{ or } q \stackrel{+}{-} \text{not } t. \\ r_2 : s. \\ r_3 : \ :- \text{not } p, \text{not } q.$$

has two answer sets: $\{s, p\}$ and $\{s, q\}$. (An empty head means that the body of the rule must never be satisfied.)

Preferences between cr-rules are encoded by atoms of the form $prefer(r_1, r_2)$, where r_1 and 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." We call this type of preference *binding*.

[1] As usual with the semantics of epistemic disjunction, the rule forces the agent to believe only *one* literal, bu he may be forced to believe also other elements of the head by other rules in the program.

To better understand the use of preferences, consider program Π_1:

$$\begin{array}{l} r_1 : p \stackrel{+}{\leftarrow} \text{not } t. \\ r_2 : q \stackrel{+}{\leftarrow} \text{not } t. \\ r_3 : prefer(r_1, r_2). \end{array}$$

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 $\Pi_2 = \Pi_1 \cup \{r_4 : \ :- \text{not } p, \text{not } q\}$. Now cr-rules must be used to restore consistency. Since r_1 is preferred to r_2, the answer set is: $\{p, prefer(r_1, r_2)\}$. Finally, consider $\Pi_3 = \Pi_2 \cup \{r_5 : \ :- p\}$. Its answer set is: $\{q, prefer(r_1, r_2)\}$.

In the rest of the discussion, we will omit rule names whenever possible. Now we describe in more detail the RCS and present the design of USA-Smart.

3 The RCS and the Design of USA-Smart

The RCS is the Shuttle's system that has primary responsibility for maneuvering the aircraft while it is in space. It consists of fuel and oxidizer tanks, valves and other plumbing needed to provide propellant to the maneuvering jets of the Shuttle. It also includes electronic circuitry: both to control the valves in the propellant lines and to prepare the jets to receive firing commands.

The RCS is divided in three subsystems: the forward RCS, the left RCS, and the right RCS. Each subsystem controls jets located in different parts of the craft. For most maneuvers, two or more subsystems have to be used concurrently. Each subsystem has its own propellant tanks, plumbing, circuitry, and jets. There is almost no connection between the subsystems, with the only important exception of the crossfeed, which connects the plumbing of the left and right subsystems. 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 propellant. It is NASA's policy to use the crossfeed as sparingly as possible, in order to keep the level of propellant in the two subsystems balanced.

The RCS is computer controlled during takeoff and landing. While in orbit, however, astronauts have the primary control. When an orbital maneuver is required, the astronauts must perform whatever actions are necessary to prepare the RCS. These actions generally require flipping switches, which are used to open or close valves, or to activate the proper circuitry. Acting on the valves will allow propellant to reach the jets that are involved in the maneuver. When the operation is complete, the jets are "ready for the maneuver." In emergency situations, such as when some switches are faulty, the astronauts communicate the problem to the ground flight controllers, who will come up with a sequence of computer commands to perform the desired task and will instruct the Shuttle's computer to execute them. At the same time, they will send to the astronauts a sequence of operations on the switches that must be combined with the computer commands. Instructing the computer to operate the valves is quite complex, since it requires modifying the computer's software and uploading it to the Shuttle. For this reason, flight controllers prefer the use of switches, when possible.

During normal Shuttle operations, there are pre-scripted plans that tell the astronauts which switches should be flipped to achieve certain goals. The situation changes when there are failures in the system. The number of possible sets of failures is too large to pre-plan for all of them. Continued correct operation of the RCS in such circumstances is necessary to allow for the completion of the mission and to help ensure the safety of the crew.

USA-Smart is designed to help achieve this goal by generating plans for emergency situations, and by verifying the correctness, and the safety, of the plans proposed by the flight controllers.

Like its predecessor, USA-Smart consists of a collection of largely independent modules, represented by lp-functions[2], and a graphical Java interface. The interface provides a simple way for the user to enter information about the history of the RCS, its faults, and the task to be performed. The two tasks possible are: checking if a sequence of occurrences of actions satisfies goal G, and finding a plan for G of a length not exceeding some number of steps, N. Based on this information, the graphical interface verifies if the input is complete, selects an appropriate combination of modules, assembles them into a CR-Prolog program, Π, and passes Π as input to a reasoning system for computing answer sets (in USA-Smart this role is played by CRMODELS[3], which performs the underlying computations using SMODELS[4][15]). In this approach the task of checking a plan P is reduced to checking if there exists a model of the program $\Pi \cup P$.

Plan generation is performed by the planning module; the corresponding correctness theorem [16] guarantees that there is a one-to-one correspondence between the plans and the set of answer sets of the program. Finally, the Java interface extracts the appropriate answer from the CRMODELS output and displays it in a user-friendly format.

The modules used by USA-Smart are:

[2] By lp-function we mean a CR-Prolog program Π with input and output signatures $\sigma_i(\Pi)$ and $\sigma_o(\Pi)$ and a set $dom(\Pi)$ of sets of literals from $\sigma_i(\Pi)$ such that, for any $X \in dom(\Pi)$, $\Pi \cup X$ is consistent, i.e. has an answer set.

[3] http://www.krlab.cs.ttu.edu/Software

[4] http://www.tcs.hut.fi/Software/smodels

– the plumbing module;
– the valve control module;
– the circuit theory module;
– the planning module.

The first three modules describe the behavior of the RCS, and are examined in detail in [17,2]. The planning module establishes the search criteria used by the program to find a plan.

4 Planning in USA-Smart

The structure of the planning module follows the *generate, (define) and test* approach described in [7,13,9]. Since the RCS contains more than 200 actions, with rather complex effects, and may require very long plans, this standard approach needs to be substantially improved. This is done by adding various forms of heuristic, domain-dependent information[5]. In particular, the generation part takes advantage of the division of the RCS in three, largely independent, subsystems. A plan for the RCS can therefore be viewed as the composition of three separate plans that can operate in parallel.

Plan generation is implemented using the following rule, $AGEN$:

```
0{occurs(A,T): action_of(A,R)}1 :- subsystem(R),
                                    involved(R,T).
```

The intuitive reading of `involved(R,T)` is "subsystem R is involved at time T in the maneuver being performed", and `action_of(A,R)` means "A is an action that operates on subsystem R." Overall, $AGEN$ selects at each time step, T, at most one action, A, for each subsystem, R, that is involved in the maneuver. (To save space, we omit from the rules the specification of the domains of variables.)

The precise definition of `involved(R,T)` is given by the following two rules. The first rule says that subsystem R is involved in the maneuver at time T if the goal for that subsystem has not yet been achieved.

```
involved(R,T) :- subsystem(R),
                 not goal(T,R).
```

The second rule says that subsystem $R1$ is involved in the maneuver at time T if the crossfeed must be used, and if $R1$ is connected through the crossfeed to another subsystem, $R2$, whose goal has not yet been achieved.

[5] Notice that the addition does not affect the generality of the algorithm.

```
involved(R1,T) :- subsystem(R1), has_crossfeed(R1),
                  subsystem(R2), has_crossfeed(R2),
                  neq(R1,R2),
                  not goal(T,R2).
```

In our approach, the test phase of the search is the one that most directly controls the quality of plans. Tests are expressed by constraints, so that, when a sequence of actions is not a desirable solution according to some test, the body of the corresponding constraint is satisfied. This guarantees that only "desirable plans" are returned.

The first step is ensuring that the models of the program contain valid plans. This is obtained by the constraint:

```
:- not goal.
```

The definition of `goal` is:

```
goal :-
           goal(T1,left_rcs),
           goal(T2,right_rcs),
           goal(T3,fwd_rcs).
```

The rationale for this definition is that the goal of preparing the Shuttle for a maneuver is split into several subgoals, each setting some jets, from a particular subsystem, ready to fire. The overall goal is stated as a composition of the goals of the individual subsystems.

Several other constraints that are used to encode heuristic, domain-dependent information are described in [17,2].

In order to improve the quality of plans with respect to the results obtained with USA-Advisor, the planner of USA-Smart must be able to:

1. avoid the use of the crossfeed if at all possible;
2. avoid the use of computer commands if at all possible;
3. avoid the generation of irrelevant actions.

Notice that these requirements are in some sense defeasible. The planner is allowed to return a solution that does not satisfy some of the requirements, if no better solution exists.

The A-Prolog based planner used in USA-Advisor is unable to cope with requirements of this type. In fact, A-Prolog lacks the expressive power necessary to compute *best* or *preferred* solutions.

The adoption of CR-Prolog solves the problem. The key step in encoding a defeasible test is the introduction of a cr-rule that determines whether the corresponding constraints must be applied. Since cr-rules are used as rarely as possible, the test will be ignored only when strictly necessary. Moreover, preferences on cr-rules allow to specify which tests are more important.

Consider requirement 1 above. The corresponding test is encoded by:

```
r1(R,T): xfeed_allowed(R,T) +- subsystem(R).

:- subsystem(R), action_of(A,R),
   occurs(A,T),
   opens_xfeed_valve(A),
   not xfeed_allowed(R,T).
```

The cr-rule says that the use of the crossfeed may possibly be allowed at any time step T. The constraint says that it is impossible for action A of subsystem R to occur at T if A opens a crossfeed valve, and the use of the crossfeed is not allowed in R at time step T.

Requirement 2 is encoded in a similar way.

```
r2(R,T): ccs_allowed(R,T) +- subsystem(R).

:- subsystem(R), action_of(A,R),
   occur(A,T),
   sends_computer_command(A),
   not ccs_allowed(R,T).
```

The cr-rule says that computer commands may possibly be allowed at any time step T. The constraint says that it is impossible for action A of subsystem R to occur at T if A sends a computer command and computer commands are not allowed in R at time step T.

As we mentioned above, CR-Prolog also allows to express the relative importance of defeasible tests. For example, if the flight controllers decide that modifying the software of the Shuttle's computer is preferable to losing the balance of the propellant between the left and right subsystems, the following rule can be added to the planner:

```
prefer(r2(R2,T2),r1(R1,T1)).
```

Notice that preferences are not restricted to occur as facts. The rule

```
prefer(r2(R2,T2),r1(R1,T1)) :- computer_reliable.
```

says that the use of computer commands is preferred to the use of the crossfeed only if the on-board computer is reliable. In this case, if we want to make sure that computer commands are used only as a last resort, we can add:

```
prefer(r1(R1,T1),r2(R2,T2)) :- -computer_reliable.
```

(Here “-” is classical negation.)

Avoiding the generation of irrelevant actions (requirement 3 above) is obtained by a test that ensures that all non-empty time steps in the plan for a subsystem are strictly necessary. The test is encoded as:

```
r3(R,T): non_empty(R,T) +- subsystem(R).

:- subsystem(R), action_of(A,R),
   occurs(A,T),
   not non_empty(R,T).
```

The cr-rule says that any time step T of the plan for subsystem R may possibly be non-empty. The constraint says that it is impossible for action A of subsystem R to occur at time step T if T is empty in the plan for R.

Experimental results confirm that the plans generated by USA-Smart are of a significantly higher quality than the plans generated by USA-Advisor.

We have applied USA-Smart to 800 problem instances from [16], namely the instances with 3, 5, 8, and 10 mechanical faults, respectively, and no electrical faults. For these experiments, we did not include in the planner the preference statements previously discussed.[6]

The planning algorithm iteratively invokes the reasoning system with maximum plan length L, checks if a model is returned, and iterates after incrementing L if no model was found. If no plans are found that are 10 or less time steps long, the algorithm terminates and returns no solution. This approach guarantees that plans found by the algorithm are the shortest (in term of number of time steps between the first and the last action in the plan). Notice that the current implementation of CRMODELS returns the models ordered by the number of (ground) cr-rules used to obtain the model, with the model that uses the least cr-rules returned first. Hence, the plan returned by the algorithm is both the shortest and the one that uses the minimum number of cr-rules.

Overall, computer commands were used 27 times, as opposed to 1831 computer commands generated by USA-Advisor. The crossfeed was used 10 times by

[6] This decision is due to the fact that the current version of CRMODELS handles preferences very inefficiently. Work is well under way in the implementation of a new, efficient algorithm that will be able to deal with preferences efficiently.

USA-Smart, and 187 times by USA-Advisor. Moreover, in 327 cases over 800, USA-Smart generated plans that contained less actions than the plans found by USA-Advisor (as expected, in no occasion they were longer). The total number of irrelevant actions avoided by USA-Smart was 577, which is about 12% of the total number of actions used by USA-Advisor (4601).

In spite of the improvement in the quality of plans, the time required by USA-Smart to compute a plan (or prove the absence of a solution) was still largely acceptable. Many plans were found in seconds; most were found in less than 2 minutes, and the program almost always returned an answer in less than 20 minutes (the maximum that the Shuttle experts consider acceptable). The only exception consists of about 10 cases, when planning took a few hours. These outliers were most likely due to the fact that CRMODELS is still largely unoptimized.

5 Advanced Use of Preferences

The quality of plans is significantly influenced by the set of preferences included in the planner. We have shown in the previous section a simple example of preferences used in USA-Smart. Now we examine more complex preferences, the way they interact with each other, and the effect on the solution returned by the planner.

The first preference that we show deals with the source of the propellant that is delivered to the jets. In the Space Shuttle, the RCS can optionally be powered with fuel coming from the three main jets of the craft, that are controlled by the Orbital Maneuvering System (OMS). However, the propellant cannot be delivered back from the RCS to the OMS if later needed. Since the OMS jets are critical for safe re-entry, the use of the OMS propellant for the RCS is avoided unless there is no other choice. Summing up, either the crossfeed or the OMS-feed may possibly be used to deliver propellant to the jets, but *the OMS-feed should not be used unless no plan can be found, that uses the crossfeed*. This statement can be encoded by the following rules: (to simplify the presentation, we will make the decisions independent of time and of subsystem – e.g. if the use of the crossfeed is allowed, it may occur at any time step in any subsystem)

```
xfeed: xfeed_allowed +-.
oms:   omsfeed_allowed +-.

prefer(xfeed,oms).
```

A similar preference can be included in the planner if we model the capability of the crew to repair damaged switches in the control panels of the RCS. Since such repairs may take a very long time, and short-circuits may occur during the process, either computer commands or switch repairs may be possibly included in the plan, but switch repairs should be included only if no plans that use computer commands are found. The statement is encoded by:

```
ccs: ccs_allowed +-.
rep: repair_allowed +-.

prefer(ccs,rep).
```

It is interesting to examine the interaction between the two preferences above. Suppose that we are given an initial situation in which:

- jets in the left subsystem must be used;
- leaking valves prevent the use of the propellant in the tanks of the left subsystem;
- the wires connecting the on-board computer to the valves that control the crossfeed are damaged;
- the switches that enable the OMS-feed are stuck.

Clearly the only reasonable choices available to deliver propellant to the jets are:

1. via the OMS-feed using computer commands, or
2. via the crossfeed after repairing the switches.

Let us imagine the reasoning of a flight controller trying to decide between the two alternatives. Should the plan use the propellant in the OMS tanks ? Since it is quite risky, that is normally done only if the crossfeed cannot be used, and alternative 2 allows the use of the crossfeed. On the other hand, alternative 2 requires the repair of the switches. That, again, is dangerous, and is normally done only is there is no way to use computer commands. Hence, he is back to alternative 1. It is reasonable to expect that, after some thinking, the flight controller would discuss the problem with his colleagues in order to consider all the relevant aspects of the remaining part of the mission (notice that these data are *not* available to USA-Smart). Only after taking all these elements into account, he would finally be able to make a decision.

Given the above encoding of the preferences, and an appropriate encoding of the initial situation, USA-Smart would reason in a way that mimics the flight controller's thoughts. Alternative 1 uses the OMS-feed, and there is another alternative that uses the crossfeed, while rule `prefer(xfeed,oms)` says that the OMS-feed can be used only if there is no way to use the crossfeed. Hence, alternative 1 cannot be used to generate a plan. Similarly, alternative 2 cannot be used to generate a plan. Therefore, the problem has no solution, with the given information.[7]

The behavior of the planner is due to the use of binding preferences. According to the informal semantics described in Section 2, rule `prefer(xfeed,oms)` is best seen as an order that describes how reasoning must be performed. When conflicts arise on the specification of how reasoning should be performed, the reasoner does not return any of the conflicting belief sets, following the intuition that such conflicts must be considered carefully.

On the other hand, there are cases when it is desirable to specify weaker preferences, that can be violated if conflicts arise. This typically happens when the situation is not particularly dangerous. The example that follows describes the use of weaker preferences in the domain of the RCS.

In the RCS, it is possible in principle to allow the propellant to reach (lightly) damaged jets, as well as go through valves that are stuck open. None of the options is particularly dangerous: a specific command must be sent to turn on the jets, so propellant can be safely allowed to reach damaged jets[8]; stuck valves can be safely traversed by the propellant without any leaks (unless they are also leaking). Nonetheless, severe problems may occur in an (unlikely) emergency in which it is necessary to shut off quickly the flow of propellant, if the only valve that is in the path is stuck. For this reason, it seems reasonable to prefer the delivery of propellant to damaged jets over the use of stuck valves. This idea is formalized in CR-Prolog by:

```
d: dam_jets_allowed +-.
v: stuck_valves_allowed +-.

prefer(d,v) :- not -prefer(d,v).
p1: -prefer(d,v) +-.
```

The last two rules encode a weaker type of preference. The first is a default saying that, *normally*, cr-rule `v` cannot be considered unless there are no solutions that use cr-rule `d`. The second rule encodes a strong exception to the default, saying that the preference between `d` and `v` may be possibly violated.

To see how weak preferences work, let us consider the interaction of the previous rules with:

[7] With the addition of a few other cr-rules, it is actually possible to allow USA-Smart to return a model whose literals give details on the problem encountered. We do not describe this technique here, because it is out of the scope of the paper.

[8] We are assuming that the firing command is working correctly.

```
s: repair_switches_allowed +-.
c: repair_ccs_allowed +-.

prefer(s,c) :- not -prefer(s,c).
p2: -prefer(s,c) +-.
```

These rules express the fact that the crew can either repair the switches of the control panel, or repair the wires that give the on-board computer control of the valves of the RCS. The former repair is preferred to the latter (as working on the computer command wires requires shutting down the on-board computer).

Now let us consider a situation in which both switches and computer commands are damaged, and we cannot avoid delivering propellant either through a stuck valve or to a damaged jet (without firing it). The damages to the switches are such that, even after repairing them, the goal can be achieved only by delivering the propellant through the stuck valve. The two reasonable solutions are: repairing the computer commands and delivering the propellant to the damaged jet, or repairing the switches and delivering the propellant to the stuck valve. Intuitively, since there are no major risks involved, both solutions are viable. Because of the use of defaults, and of cr-rules `p1` and `p2`, USA-Smart would consider both solutions equivalent, and return indiscriminately one plan associated with them. [9]

6 Related Work

The amount of literature on planning with preferences is huge. Because of space constraints, we will restrict the attention only to those logical approaches to planning in which the language allows the representation of *state contraints*[10]. This capability is crucial to model most of the RCS, e.g. the electrical circuits.

In its simplest form, the `minimize` statement of SMODELS [15] instructs the reasoning system to look for models that minimize the number of atoms, from a given set, that are present in the model. In its complete form, the statement allows to minimize the sum of the weights associated with the specified atoms. Encoding defeasible tests using `minimize` seems non-trivial because of the possibility to specify only one statement in the program. Moreover, it is not entirely clear how preferences on tests could be encoded. The *weak constraints* of DLV [6] provide an elegant way to encode defeasible tests. A weak constraint is a constraint that can be violated if necessary. A numerical weight can be

[9] The conclusion can be formally proven from the semantics of CR-Prolog.
[10] Also called *static causal laws* in the context of action languages.

specified to express the cost of violating the constraint. Unfortunately, examples show that the use of weak constraints to encode preferences for planning is affected by the same problems that we discussed in the context of diagnosis [1]. This is true also for the approaches that rely on a translation to the language of DLV, e.g. DLV$^{\mathcal{K}}$ [8]. Another alternative is the use of LPOD [4,5], which extends A-Prolog by allowing the specification of a list of alternatives in the head of rules. The alternatives are listed from the most preferred to the least preferred. If the body of the rule is satisfied, one alternative must be selected following the preference order. Moreover, preferences can be specified between rules, so that the reasoning system tries to pick the best alternatives possible for preferred rules. Preferences in LPOD are intended in the weaker meaning discussed in the previous section (in [5], the authors argue that *Pareto* preference is superior to the other types of preferences that they considered in the paper). Hence, it is definitely possible to encode in this language both defeasible tests and the weak preferences of Section 5. However, it is not clear if there is a way to encode binding preferences in LPOD. The ability to encode binding preferences is very important in USA-Smart, as it allows for a more cautious form of reasoning, which is essential in delicate situation such as the Shuttle's missions.

7 Conclusions

In this paper, we have shown how CR-Prolog was used in our decision support system for the Space Shuttle in order to improve significantly the quality of the plans returned. The general problem that we have addressed is that of improving the quality of plans by taking into consideration statements that describe "most desirable" plans. We believe that USA-Smart proves that CR-Prolog provides a simple, elegant, and flexible solution to this problem, and can be easily applied to any planning domain. We have also discussed how alternative extensions of A-Prolog can be used to obtain similar results.

The author is very thankful to Michael Gelfond for his suggestions. This work was partially supported by United Space Alliance under Research Grant 26-3502-21 and Contract COC6771311, and by NASA under grant NCC9-157.

References

1. 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.
2. Marcello Balduccini, Michael Gelfond, Monica Nogueira, and Richard Watson. The USA-Advisor: A Case Study in Answer Set Planning. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 439–442, Sep 2001.

3. 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.
4. Gerhard Brewka. Logic programming with ordered disjunction. In *Proceedings of AAAI-02*, 2002.
5. Gerhard Brewka, Ilkka Niemela, and Tommi Syrjanen. Implementing ordered disjunction using answer set solvers for normal programs. In Sergio Flesca and Giovanbattista Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, Sep 2002.
6. Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Giovanbattista Ianni, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, Gerard Pfeifer, and Axel Polleres. The dlv system. In Sergio Flesca and Giovanbattista Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, Sep 2002.
7. Yannis Dimopoulos, J. Koehler, and B. Nebel. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the 4th European Conference on Planning*, volume 1348 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 169–181, 1997.
8. Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerard Pfeifer, and Axel Polleres. Answer set planning under action costs. *Journal of Artificial Intelligence Research*, 19:25–71, 2003.
9. Selim Erdogan and Vladimir Lifschitz. Definitions in answer set programming. In *Proceedings of LPNMR-7*, Jan 2004.
10. Michael Gelfond. Representing knowledge in A-Prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408, pages 413–451. Springer Verlag, Berlin, 2002.
11. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–385, 1991.
12. K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamenta Informaticae*, 37(3):247–268, 1999.
13. Vladimir Lifschitz. *Action Languages, Answer Sets, and Planning*, pages 357–373. The Logic Programming Paradigm: a 25-Year Perspective. Springer Verlag, Berlin, 1999.
14. Fangzhen Lin and Yuting Zhao. Assat: Computing answer sets of a logic program by sat solvers. In *Proceedings of AAAI-02*, 2002.
15. Ilkka Niemela, Patrik Simons, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, Jun 2002.
16. Monica Nogueira. *Building Knowledge Systems in A-Prolog*. PhD thesis, University of Texas at El Paso, May 2003.
17. Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space Shuttle. In *PADL 2001*, pages 169–183, 2001.
18. Enrico Pontelli, Marcello Balduccini, and F. Bermudez. Non-monotonic reasoning on beowulf platforms. In Veronica Dahl and Philip Wadler, editors, *PADL 2003*, volume 2562 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 37–57, Jan 2003.
19. Timo Soininen and Ilkka Niemela. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, May 1999.