Noname manuscript No.
(will be inserted by the editor)

Marcello Balduccini · **Michael Gelfond** ·
Monica Nogueira

Answer Set Based Design of Knowledge Systems

Received: date / Accepted: date

Abstract The aim of this paper is to demonstrate that A-Prolog is a powerful language for the construction of reasoning systems. In fact, A-Prolog allows to specify the initial situation, the domain model, the control knowledge, and the reasoning modules. Moreover, it is efficient enough to be used for practical tasks and can be nicely integrated with programming languages such as Java. An extension of A-Prolog (CR-Prolog) allows to further improve the quality of reasoning by specifying requirements that the solutions should satisfy if at all possible. The features of A-Prolog and CR-Prolog are demonstrated by describing in detail the design of USA-Advisor, an A-Prolog based decision support system for the Space Shuttle flight controllers.

Keywords Knowledge Representation, Answer Set Programming, Reasoning, Planning, Diagnosis, Logic Programming

PACS 68T30 · 68T35 · 68T20

1 Introduction

In recent years, A-Prolog – a knowledge representation language based on the answer set semantics [28] – was shown to be a useful tool for knowledge rep-

M. Balduccini · M. Gelfond
Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
E-mail: marcello.balduccini@ttu.edu,mgelfond@cs.ttu.edu

M. Nogueira
Center for Logistics and Digital Strategy
Kenan Institute of Private Enterprise
Kenan-Flagler Business School
The University of North Carolina at Chapel Hill
Chapel Hill, NC 27599 USA
E-mail: monica.nogueira@unc.edu

resentation and reasoning [42, 25, 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.

In this paper we describe an A-Prolog based methodology for modeling of and reasoning about complex dynamic systems. We show that A-Prolog can be used to specify all the elements of the model: the specification of the initial situation, the causal laws that rule the evolution of the domain, the reasoning modules, and the control knowledge used to guide the reasoning processes.

We also show how our methodology can be enhanced, using a recently developed extension of A-Prolog called CR-Prolog [7, 9], to allow the specification of requirements that the solutions found by the reasoning modules should satisfy if at all possible. The addition of such requirements substantially improves the quality of reasoning.

We describe our methodology by showing its application to building USA-Advisor, a decision support system for the Reaction Control System (RCS) of the Space Shuttle. This application builds on a previous investigation [46, 14, 30], where a substantially smaller part of RCS was represented in Prolog and used to check correctness of plans.

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 fuel lines and to prepare the jets to receive firing commands. Overall, the system is rather complex, in that it includes 12 tanks, 44 jets, 66 valves, 33 switches, and around 160 computer commands (computer-generated signals).

When an orbital maneuver is required, the astronauts must configure the RCS accordingly. This involves changing the position of several switches, which are used to open or close valves or to energize the proper circuitry. Normally, the sequences of actions required to configure the RCS are pre-determined before the beginning of the mission and the astronauts simply have to search for the sequence in a manual. However, faults (e.g. the inability to move a switch) may make these pre-scripted sequences of actions inapplicable. The number of possible sets of failures is too large to plan in advance for all of them. In this case, the astronauts communicate the problem to the ground flight controllers, who come up with a new sequence of actions to perform the desired task. The main challenge of this step is to find plans that achieve the desired results without causing any possibly dangerous side effect.

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

The main issues involved in building the USA-Advisor are:

– Modeling the RCS as a dynamic domain: this includes representing information at very different levels of detail. For instance, on one level we need to describe the effects of the valve positions on the plumbing system. In another level we specify the electrical circuits used to control the valves.
– Representing knowledge in several separate modules and combining the appropriate modules depending on the task given to the system – notice that one of the modules had been independently developed before the start of the USA-Advisor project.
– Developing a planning module containing a large amount of heuristic information (which substantially improves quality of the plans and efficiency of the search).

The research on USA-Advisor has been partially funded by United Space Alliance (USA), the company responsible for managing various systems of the Space Shuttle, including the RCS. USA-Advisor is currently being developed by the programmers at USA, who are working on formalizing models of other systems of the Shuttle, as well as creating a graphical interface suitable for use by the flight controller.

2 A-Prolog

A-Prolog is a knowledge representation language with roots in the research on the semantics of logic programming languages and non-monotonic reasoning [27, 28]. Over time, several extensions of the original language have been proposed [18,39,38,25,20,17]. In this work, we used an extension of A-Prolog along the lines of [25] and [39].

By the term *basic A-Prolog* we identify the language introduced in [27], and later extended with epistemic disjunction [28,26]. The term basic A-Prolog programs used later is intended as a synonym of *disjunctive program*.

The syntax of A-Prolog is determined by a typed signature Σ consisting of types, typed object constants, and typed function and predicate symbols. We assume that the signature contains symbols for integers and for the standard functions and relations of arithmetic. Terms are built as in first-order languages.

By *simple arithmetic terms* of Σ we mean its integer constants. By *complex arithmetic terms* of Σ we mean terms built from legal combinations of arithmetic functions and simple arithmetic terms (e.g. $3+2\cdot 5$ is a complex arithmetic term, but $3+\cdot\ 2\ 5$ is not).

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

Literals are atoms and negated atoms, i.e. expressions of the form $\neg p(t_1,\ldots,t_n)$. Literals $p(t_1,\ldots,t_n)$ and $\neg p(t_1,\ldots,t_n)$ are called *complementary*. By $\bar{l}$ we denote the literal complementary to l.

Definition 1 A *basic rule r* (of A-Prolog) is a statement of the form:

$$h_1 \text{ OR } h_2 \text{ OR } \ldots \text{ OR } h_k \leftarrow l_1, l_2, \ldots l_m, \text{not } l_{m+1}, \text{not } l_{m+2}, \ldots, l_n. \qquad (1)$$

where $l_1, \ldots, l_m$ are literals, and h_i's and $l_{m+1}, \ldots, l_n$ are plain literals. We call h_1 OR h_2 OR ... OR h_k the *head* of the rule (*head*(r)); $l_1, l_2, \ldots l_m, \text{not } l_{m+1}, \text{not } l_{m+2}, \ldots, l_n$ is its *body* (*body*(r)), and *pos*(r), *neg*(r) denote, respectively, $\{l_1, \ldots, l_m\}$ and $\{l_{m+1}, \ldots, l_n\}$.

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

A rule such that $k = 0$ is called *constraint*, and is considered a shorthand of:

$$\bot \leftarrow \text{not } \bot, l_1, l_2, \ldots l_m, \text{not } l_{m+1}, \text{not } l_{m+2}, \ldots, l_n.$$

Definition 2 A *basic A-Prolog program* is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of basic rules.

Whenever possible, we denote programs by their second element. The corresponding signature is denoted by $\Sigma(\Pi)$. The terms, atoms and literals of a program Π are denoted respectively by *terms*(Π), *atoms*(Π) and *literals*(Π).

Notice that the definition of the syntax of basic A-Prolog does not allow the use of variables. Rules containing variables (denoted by capital letters) are thus viewed as shorthands for the set of their ground instantiations, obtained by substituting the variables with all the terms of appropriate type from the signature of the program. The approach is justified for the so called closed domains, i.e. domains satisfying the domain closure assumption [44] that all objects in the domain of discourse have names in the language of the program. The semantics of basic A-Prolog for open domains can be found in [11, 33].

The semantics of basic A-Prolog is defined in two steps. The first step consists in giving the semantics of programs that do not contain default negation. We will begin by introducing some terminology.

An atom is in *normal form* if it is an arithmetic atom or if it is a plain atom and its arguments are either non-arithmetic terms or simple arithmetic terms. Notice that atoms that are not in normal form can be mapped into atoms in normal form by applying the standard rules of arithmetic. For example, $p(4+1)$ is mapped into $p(5)$. For this reason, in the following definition of the semantics of basic A-Prolog, we assume that all literals are in normal form unless otherwise stated.

A literal l is *satisfied* by a consistent set of plain literals S (denoted by $S \models l$) if:

- l is an arithmetic literal and is true according to the standard arithmetic interpretation;
- l is a plain literal and $l \in S$.

If l is not satisfied by S, we write $S \not\models l$. An expression not l, where l is a plain literal, is satisfied by S if $S \not\models l$. A set of literals is satisfied by S if each element of the set is satisfied by S.

We say that a consistent set of plain literals S is *closed under a program Π not containing default negation* if, for every rule

$$h_1 \text{ OR } h_2 \text{ OR } \ldots \text{ OR } h_k \leftarrow l_1, l_2, \ldots l_m$$

of Π such that the body of the rule is satisfied by S, $\{h_1, h_2, \ldots, h_k\} \cap S \neq \emptyset$.

Definition 3 (Answer Set of a program without default negation) A consistent set of plain literals, S, is an *answer set of a program Π not containing default negation* if S is closed under all the rules of Π and S is set-theoretically minimal among the sets satisfying the first property.

Programs without default negation and whose rules have at most one literal in the head are called *definite*. It can be shown that definite programs have at most one answer set. The answer set of a definite program Π is denoted by $ans(\Pi)$.

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

Definition 4 (Reduct of a basic A-Prolog program) Let Π be an arbitrary basic A-Prolog program. For any set S of plain literals, let Π^S be the program obtained from Π by deleting:

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

Definition 5 (Answer Set of a basic A-Prolog program) A set of plain literals, S, is an *answer set of a basic A-Prolog program Π* if it is an answer set of Π^S.

An interesting extension [25] of basic A-Prolog consists in the introduction of constructs that simplify representation and reasoning with sets of terms and with functions from such sets to natural numbers.

In this paper, we extend basic A-Prolog by adding to it *s-atoms* from [25], which allow to concisely represent subsets of sets of atoms. The resulting language will be called *A-Prolog*. Its syntax is defined as follows.

Definition 6 A *s-atom* is a statement of the form:

$$\{\overline{X} \ : \ p(\overline{X})\} \subseteq \{\overline{X} \ : \ q(\overline{X})\} \tag{2}$$

where $\overline{X}$ is the list of all free variables occurring in the corresponding *plain* atom.

Informally, the statement says that p is a subset of q. In A-Prolog, literals and s-atoms are disjoint sets. Literals and s-atoms are called *extended literals*. Rules are defined as follows.

Definition 7 A *rule* (of A-Prolog) is a statement of the form (1), where l_i's are as before, and either (1) $k = 1$ and h_1 is a s-atom, or (2) all h_i's are plain literals.

The reader may have noticed that, like in [25], negated atoms, $\neg p$, are not allowed to occur in s-atoms. However, differently from there, we allow negated atoms to be used everywhere else in the program.

Notice that the combination of sets with classical and default negations introduces some subtleties. Consider the following informal argument. Suppose we are given a statement $\{\overline{X} : p(\overline{X})\} \subseteq \{\overline{X} : q(\overline{X})\}$ and we know $q(a)$ and $\neg q(b)$, but have no information about $q(c)$. Clearly, $p(a)$ satisfied the condition. But can we about $\neg p(b)$? And what about $p(c)$ or $\neg p(c)$?

To restrict ourselves to cases in which the meaning of s-atoms is unambiguous, we give the following definition of A-Prolog program.

Definition 8 An *A-Prolog program* is pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature, Π is a set of A-Prolog rules, and *for every atom* $r(\overline{X})$ *that occurs in the scope of an s-atom,* Π *contains the rule:*

$$\neg r(\overline{X}) \leftarrow \text{not } r(\overline{X}).$$

(which encodes the Closed World Assumption on $r(\overline{X})$).

Thanks to this restriction, the meaning of s-atoms in our programs is unambiguous. Going back to the previous example, and assuming the Close World Assumption for p and q is part of the program, it can be shown that, for every x, $p(x)$ if $q(x)$ and $\neg p(x)$ otherwise.

To define the semantics of A-Prolog, we introduce the following terminology. Let Σ be a signature and S be a set of plain literals from Σ. A s-atom (2) from Σ is true in S if, for any sequence $\bar{t}$ of ground terms from Σ, either $p(\bar{t}) \notin S$ or $q(\bar{t}) \in S$.

The following definition is similar to the notion of reduct introduced earlier.

Definition 9 (Set-Elimination) Let Π be an arbitrary A-Prolog program. For any consistent set S of plain literals, the set-elimination of Π with respect to S (denoted by $se(\Pi, S)$) is the program obtained from Π by:

- removing from Π all the rules whose bodies contain s-atoms not satisfied by S;
- removing all remaining s-atoms from the bodies of the rules;
- replacing rules of the form $l_s \leftarrow \Gamma$, where l_s is an s-atom not satisfied by S, by rules $\leftarrow \Gamma$;
- replacing each remaining rule

$$\{\overline{X} : p(\overline{X})\} \subseteq \{\overline{X} : q(\overline{X})\} \leftarrow \Gamma$$

  by a set of rules of the form $p(\bar{t}) \leftarrow \Gamma$ for each $p(\bar{t})$ from S.

We are now ready to define the notion of answer set of an A-Prolog program.

Definition 10 (Answer Set of an A-Prolog program) A consistent set of plain literals S from the signature of program Π is an *answer set* of Π if it is an answer set of $se(\Pi, S)$.

A-Prolog rules of the form

$$\{\overline{X} \ : \ p(\overline{X})\} \subseteq \{\overline{X} \ : \ q(\overline{X})\} \leftarrow \Gamma \tag{3}$$

are called *selection rules*. It can be noted that selection rules are closely related to the choice rules

$$m\{p(\overline{X}) : q(\overline{X})\}n \leftarrow \Gamma \tag{4}$$

introduced in [45,39]. Proposition 5 of [25] makes this connection precise. Adapted to the language used here, the proposition states the following.

Proposition 1 *For every program Π such that:*

1. *Π contains a rule*
$$\{p(\overline{X}) : q(\overline{X})\} \leftarrow \Gamma;$$
2. *no other rule of Π contains p in the head,*

let Π^{++} be the program obtained from Π by replacing the choice rule with selection rule (3). Then, S is an answer set of Π iff S is an answer set of Π^{++}.

One limitation of our definition of A-Prolog with respect to the language of [45, 39] is that it does not allow the specification of bounds, i.e. of the lower and upper number of elements of the subset defined by $\{\overline{X} \ : \ p(\overline{X})\} \subseteq \{\overline{X} \ : \ q(\overline{X})\}$. For simple bounds such as those used in this paper (we use only an upper bound of 1) we will use simple constraints, and avoid the introduction of the f-atoms from [25]. For example, imposing a maximum limit of 1 on the cardinality of the set (assuming that the arity of p is 1) can be achieved by means of the constraint:

$$\leftarrow p(X_1), p(X_2), X_1 \neq X_2.$$

To simplify the notation, from now on we use the statement:

$$m\{p(\overline{X}) : q(\overline{X})\}n \leftarrow \Gamma.$$

where m and n, if present, are 0 *and* 1 *respectively*, as an abbreviation of (3) together with the appropriate constraints to limit the cardinality of p.

3 The Reaction Control System

The RCS is the system used to maneuver the Space Shuttle while it is in orbit. The RCS is viewed as composed of three subsystems: the Forward RCS, the Left RCS, and the Right RCS.

The propellants for the RCS jets, or thrusters, are stored in fuel and oxidizer tanks, pressurized with helium, and are distributed through several different types of pressure regulation and relief valves, distribution lines (here called plumbing) and filling and draining connections, called junctions. The only physical connection among the subsystems of the RCS is an interconnection between the Left and Right subsystems, called *crossfeed*. This provision is part of the redundancy capabilities added to the Space Shuttle to ensure the safety of its operation.

In order for the Space Shuttle to perform a given maneuver, a set of jets, belonging to the correct subsystems and pointing in the correct directions, must be prepared to fire. Preparing a jet to fire involves providing an open, non-leaking path for the fuel to flow from pressurized fuel tanks to the jet. The flow of fuel is controlled by opening and closing pressure regulation and relief valves. Valves are opened and closed by either having an astronaut flip a switch or by instructing the on-board computer to issue special commands. In a very simplified form, the RCS can be viewed as a directed graph of the type shown in Figure 2, whose nodes are tanks, jets and pipe junctions, and whose arcs are labeled by valves. Switches are connected to valves through fairly complex electrical circuits.

4 USA-Advisor System's Design

The USA-Advisor system consists of a collection of largely independent modules, represented by lp-functions[1][24], and a graphical Java interface, *J*. The interface gives a simple means for the user to enter information about the history of the RCS, its faults, and the task to be performed.

USA-Advisor can perform two tasks: (1) checking if a plan satisfies a goal, *G*, and (2) finding a plan for *G* of a length not exceeding some number of steps. Based on this information, *J* verifies that the input is complete, selects an appropriate combination of modules, assembles them into an A-Prolog program, Π, and passes Π as an input to an answer set solver (SMODELS) for computing stable models In this approach the task of checking a plan *P* is reduced to checking if there exists a model of the program $\Pi \cup P$. A planning module is used to describe a set of possible plans the user is interested in. Planning is reduced to finding such models. Finally, the Java interface extracts the appropriate answer from the SMODELS output and displays it in a user-friendly format.

In our design, the RCS is described at two levels of detail, the appropriate level being selected depending on the task to be performed. At the highest level, electrical circuits are assumed to be working correctly. Thus, their internal functioning can be ignored, and the function they compute is described explicitly in terms of the effects that switches and computer commands have on the corresponding valves. At the lowest level of abstraction, used when electrical circuits contain faulty components, circuits are represented explicitly.

The RCS is decomposed in four main modules: the Plumbing Module, the Valve Control Module, the Circuit Theory Module, and the Planning Module. The Plumbing Module models the plumbing system of the RCS. The Valve Control Module describes how switches and computer commands affect the position of valves. The Circuit Theory Module describes the behavior of standard combinatorial digital circuits, augmented with other components, like delay units, power units, switches, and valves. The Planning Module is responsible for generating plans achieving the desired goal, and contains a large number of heuristics aimed

[1] By an lp-function we mean program Π of A-Prolog 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.

at improving both the quality of plans and the efficiency of the planner. Additional modules provide the description of the schematics of each electrical circuit.

The model of the RCS is based on the research on action languages [29]. As the reader will probably notice, the rules used in the model are a straightforward translation of the causal laws of action language $\mathcal{AL}$ [12].

The connections among the modules are depicted in Figure 1. In the rest of this section we give a detailed description of each module.

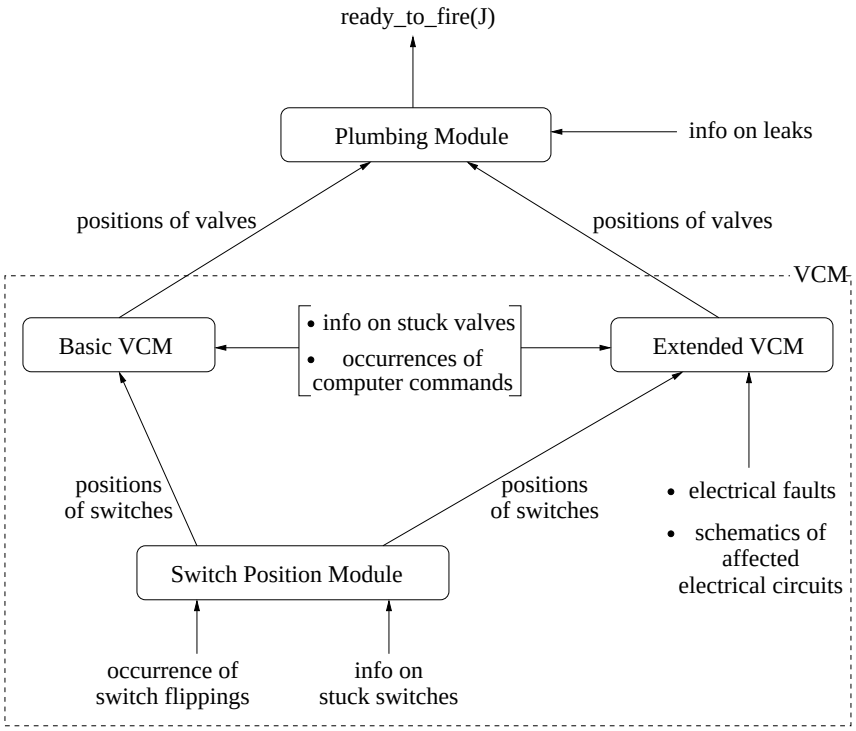

Figure 1 Modular structure of the model of the RCS.

4.1 Plumbing module

The Plumbing Module (*PM*) models the plumbing system of the RCS, which consists of a collection of tanks, jets and pipe junctions connected through pipes. The flow of fluids through the pipes is controlled by valves. The system's purpose is to deliver fuel and oxidizer from tanks to the jets needed to perform a maneuver. The structure of the plumbing system is described by a directed graph G of the type shown in Figure 2, whose nodes are tanks, jets and pipe junctions, and whose

arcs are labeled by valves. The possible faults of the system at this level are leaky valves, damaged jets, and valves stuck in some position. *The purpose of PM is to describe how faults and the position of valves affect the pressure of tanks, jets and junctions.* This is accomplished by means of state constraints alone.

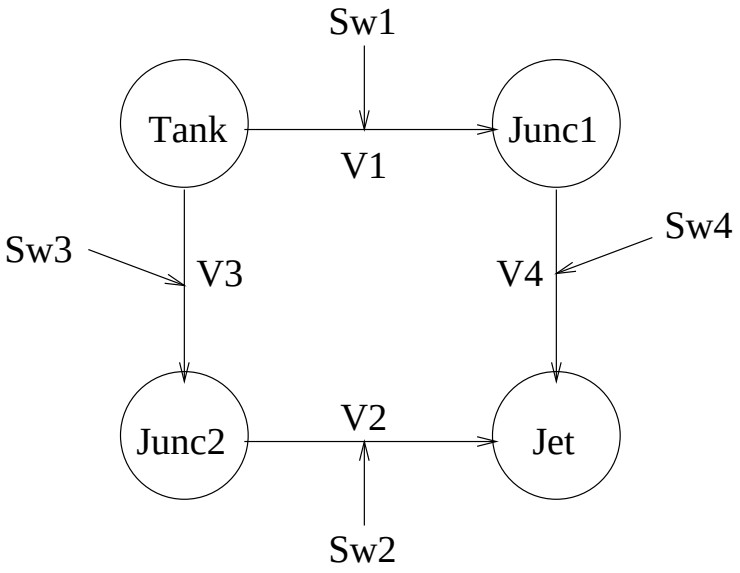

Figure 2 A simplified view of the RCS.

In particular, when fuel and oxidizer flow at the right pressure from the tanks to a properly working jet, the jet is considered ready to fire. In order for a maneuver to be started, all the jets it requires must be ready to fire. Pressurization of fuel and oxidizer tanks is obtained by releasing helium from the helium tanks connected to the fuel and oxidizer tanks. The necessary condition for a fluid to flow from a tank to a jet, and in general to any node of G, is that there exists a path without leaks from the tank to the node and that all valves along the path are open.

The rules of *PM* define a function which takes as input the structural description, G, of the plumbing system, its state including position of valves and the list of faulty components, and determines: the distribution of pressure through the nodes of G; which jets are ready to fire; which maneuvers are ready to be performed.

The elements of the plumbing system are represented in *PM* as follows. The arcs of graph G are described by relation *link(N1,N2,V)* which holds iff G contains a directed arc from node $N1$ to $N2$ and this arc is labeled by the valve V. For instance, a statement *link(ffh,ff,ffha)* says that fuel helium tank *ffh* is connected to fuel propellant tank *ff* by valve *ffha*. Relation *jet_of(J,R)* identifies jets and the subsystem they belong to. The subsystems of the RCS are identified by statements: *system(fwd_rcs)*, *system(left_rcs)*, and *system(right_rcs)*. Relation *direction(J,D)* specifies the direction of jets. There are six different possible directions different jets

point to: up, down, left, right, forward, and aft. Relation *tank_of(T,R)* links each tank to the subsystem it belongs to. For instance, a statement *tank_of(ffh,fwd_rcs)* says that the forward fuel helium tank belongs to the forward subsystem. There are twelve possible maneuvers to be performed by firing jets of Shuttle, encoded by atoms of the form *maneuver*(M).

The initial state of the plumbing module is mainly characterized by fluent *in_state(V,S)*, specifying that valve V is in state S (open or closed), and a collection of faulty components described by atoms of the form *has_leak(V)*, *damaged(J)* and *stuck(V,S)* (valve V is stuck in position S). *The role of defaults is essential for a compact description of the initial state.* For example, it is assumed that all helium tanks are pressurized in the initial state and that normally functioning valves are initially closed. This statement can be nicely expressed using the default:

```
holds(in_state(V,closed),0) :-
                          ¬holds(in_state(V,open),0).
```

Here and in the rest of the discussion variable V denotes a valve.

Important fluents in the characterization of the current state of the RCS are *pressurized_by(N,TK)*, stating that fluid under pressure is flowing from tank TK to node N (in short, "N is pressurized by TK"), and *ready_to_fire(J)*, saying that jet J is pressurized by the correct type of propellants and thus ready to fire (jets need to be pressurized with both fuel and oxidizer).

The Shuttle is ready for a maneuver M when an appropriate set of jets is ready to fire. To increase the efficiency of reasoning, we partition such a set of jets based on the subsystem the jets belong to. Fluent *maneuver_ready(M,R)* says that the jets of subsystem R involved in maneuver M are ready. For example, the following rule determines when the left subsystem is ready for the maneuver called "$+x$", which only requires one aft-pointing jet in the left subsystem.

```
holds(maneuver_ready(plus_x,left_rcs),T) :-
                          jet_of(J,left_rcs),
                          direction(J,aft),
                          holds(ready_to_fire(J),T).
```

To further illustrate the issues involved in the construction of *PM*, let us consider the definition of fluent *pressurized_by*(N,Tk). Helium tanks are treated as special nodes and presently assumed to be always pressurized. Hence, the definition for these tanks is trivial. For other nodes, the definition is recursive. It says that any non-tank node $N1$ is pressurized by a tank Tk if $N1$ is not leaking and is connected by an open valve to a node $N2$ which is pressurized by Tk.

```
holds(pressurized_by(N1,Tk),T) :-
                          link(N2,N1,V),
                          ¬holds(leaking(N1),T),
                          holds(in_state(V,open),T),
                          holds(pressurized_by(N2,Tk),T).
```

In the RCS, a node is considered leaking if propellant flow toward it is regulated by a leaking, open valve, and there is a path from such valve to the node along which all valves are open. This can be nicely formalized by combining recursive

definitions and *defined fluents*,i.e. fluents that are considered false (resp., true) *by default*. The next rules show the formalization of defined fluent $leaking(N)$:

```
holds(leaking(N1),T) :-
                        link(N1,N2,V), has_leak(V),
                        holds(in_state(V,open),T).

¬holds(leaking(N),T) :-
                        not holds(leaking(N),T).
```

Notice that fluent *leaking* is non-inertial, and the key step in its representation is the use of the Closed World Assumption, encoded above as a default. The recursive step is obtained by:

```
holds(leaking(N1),T) :-
                        link(N1,N2,V),
                        holds(in_state(V,open),T),
                        holds(leaking(N2),T).
```

The high level of abstraction of A-Prolog is confirmed by the relatively small number of rules present in the knowledge modules of USA-Advisor. For example, the Plumbing Module consists of approximately 40 rules.

As usual, default rules are used to represent the *inertia axiom*. All the modules share the same inertia axiom: (we have a similar rule for $\neg holds(L,T)$)

```
holds(L,T+1) :-
               not non_inertial(L),
               holds(L,T),
               not ¬holds(L,T+1).
```

In the rule, variable L ranges over all fluent literals. Relation $non\_inertial(L)$ is defined for non-inertial fluents such as *leaking*, and allows to stop the inertia axiom from being applied to them.

4.2 Valve control module

The flow of fuel and oxidizer propellants from tanks to jets is controlled by opening/closing valves along the path connecting these nodes. The state of valves can be changed either by manipulating mechanical switches or by issuing computer commands. Switches and computer commands are connected to the valves, they control, by electrical circuits.

In some specific phases of operation of the Shuttle, such as launch and landing, the on-board general purpose computers, GPCs, is in charge of opening/closing valves and will achieve this objective by sending computer commands. If the Shuttle is in orbit, or the computer system is malfunctioning, an astronaut can normally override these commands by manually flipping the switches that control the valves to be opened/closed.

The Switch Position Module, *SPM*, describes how the actions of flipping switches and the faults present in the system affect the position of switches. The only type of

fault considered in the *SPM* is switches being stuck. Throughout the model of the RCS, this type of mechanical malfunctioning is represented by relation *stuck(D,S)*, stating that device *D* (in the model, *D* ranges over switches and valves) is stuck in state *S*. Similarly to the plumbing module, the state of devices is described by the fluent *in_state(D,S)* meaning that device *D* is in state *S*. A device is always in a state *S* if it is stuck in state *S*. The input to the *SPM* contains information on stuck switches and the occurrences of switch flippings. The output of the module consists of the position of switches resulting from the execution the specified actions. The effect of the actions performed on normally functioning switches is defined by the *dynamic causal law* below. The law says that flipping a working switch *Sw* to state *S* causes it to move to that state.

```
holds(in_state(Sw,S),T+1) :-
                          occurs(flip(Sw,S),T),
                          not stuck(Sw,S').
```

Notice the use of default negation in the rule to express the Closed World Assumption on the information on stuck switches. A more common approach would consist in replacing default negation by classical negation and in encoding separately the Closed World Assumption on relation *stuck*. Our choice to use default negation directly is motivated by performance considerations.

The fact that a switch *Sw* is always in state *S* if it stuck in *S*, is formalized by the rule:

```
holds(in_state(Sw,S),0) :- stuck(Sw,S).
```

The Valve Control Module, *VCM*, describes how computer commands and changes in the position of switches affect the state of valves. Intuitively, if a switch *Sw* is in position “open” or “closed”, the valve(s) it controls are *normally* in that state state. Computer commands issued when the appropriate switch is in special position “gpc” cause the corresponding valve(s) to either open or close (depending on type of computer command). There are, however, two types of possible failures: valves can be stuck in some position, and electrical circuits can malfunction in various ways.

A substantial simplification of the *VCM* module is achieved by dividing it in two parts, called *basic* and *extended VCM* modules. At the basic level, it is assumed that all electrical circuits are working properly and therefore are not included in the representation. The extended level includes information about electrical circuits and is normally used when some of the circuits are malfunctioning. In that case, the position of switches and the occurrence of computer commands may produce results that cannot be predicted by the basic representation.

4.2.1 Basic valve control module

At this level, the *VCM* deals with a set of switches, computer commands and valves, and connections among them. The input of the basic *VCM* consists of the positions of switches, the faults of valves, and the collection of computer commands issued. The module implements an lp-function that, given this input, returns positions of valves at the current moment of time. This output is used as

input to the plumbing module. The class of faults of the system considered at this level consists of valves being stuck in some position.

Connections between devices (i.e. switches and valves) are described by relation *controls(Sw,V,C)* meaning that switch *Sw* controls the state of valve *V* through circuit *C* (circuits are reified). The connection between computer commands and valves is modeled by atoms of the form *commands(CC,V,S)* ("computer command *CC* moves valve *V* to position *S*") and *commands(cc(CC1,CC2),V,S)* ("computer commands *CC*1 and *CC*2 *used together* move *V* to *S*").

An electrical malfunctioning of the circuitry controlling valve *V* is represented by statement of the form *bad_circuitry(V)* (in the RCS, each valve is controlled by no more than one circuit).

The dynamic behavior of the basic *VCM* is described by a set of fluents and actions. Actions are represented as follows:

- *action_of(flip(Sw,S),R)* - flipping switch *Sw* to state *S* is an action of the *R* subsystem of the RCS.
- *action_of(cc(CC1,CC2),R)* - issuing a pair of computer commands *CC*1 and *CC*2 is an action of the *R* subsystem of the RCS.
- *action_of(CC,R)* - issuing computer command *CC* is an action of the *R* subsystem of the RCS.

The input of the basic *VCM* consists of:

1. a collection of statements of the form *holds(in_state(D,S),T)* describing the states of switches and valves;
2. the description of the faults affecting the valves;
3. the set of occurrences of computer commands.

The effect of the occurrence of computer commands is described by a dynamic causal law stating that, if switch *Sw* controlling valve *V* is in state *gpc*[2], *V* is working properly, and the computer command required to move *V* to some state *S* were issued at time *T*, then *V* will be in state *S* at the next moment of time.

```
holds(in_state(V,S),T+1) :-
                          controls(Sw,V,C),
                          holds(in_state(Sw,gpc),T),
                          occurs(CC,T), commands(CC,V,S),
                          not stuck(V,S'), not bad_circuitry(V).
```

The condition on *bad_circuitry(V)* is used to stop this rule from being applied when the circuit connecting *Sw* and *V* is not working properly.

The static connection between switches and valves is expressed by a static causal law. It says that, under normal conditions, if switch *Sw* controlling valve *V* is in some state *S*, different from gpc, then *V* is also in state *S*.

```
holds(in_state(V,S1),T) :-
                          controls(Sw,V,C),
```

[2] A switch can be in one of three positions: open, closed, or gpc. When it is in gpc, it does not affect the position of the valve.

```
                        holds(in_state(Sw,S1),T),
                        state_of(S,v_switch), neq(S1,gpc),
                        not stuck(V,S2), not bad_circuitry(V).
```

It is assumed that a valve *V* is always in state *S* if it stuck in *S*, as defined by rule: `holds(in_state(V,S),0) :- stuck(V,S).`

Impossibility conditions are described by constraints. The *VCM* description includes such a constraint to express that it is not possible to move a switch to a state it is already in.

```
:- holds(in_state(Sw,S),T),
   state_of(S,v_switch),
   occurs(flip(Sw,S),T).
```

This constraint eliminates any models where an action $flip$ tries to move a switch *Sw*, which is in state *S*, to the same state *S*. Constraints of this type play an important role in increasing efficiency of the module by reducing the search space for plans.

The output of the *VCM* is a description of the state of valves and switches at the current moment of time.

4.2.2 Extended valve control module

The extended *VCM* encompasses the basic *VCM* and also includes information about electrical circuits, power and control buses, and the wiring connections among all the components of the system.

This module, too, defines an lp-function. It takes as input the same information as the basic *VCM*, together with faults on power buses, control buses and electrical circuits. *The extended VCM returns positions of valves at the current moment of time, exactly like the basic VCM*.

Since (possibly malfunctioning) electrical circuits are part of the representation, it is necessary to compute the signals present on all wiring connections, in order to determine the positions of valves. The signals present on the circuit's wires are generated by the Circuit Theory Module (CTM), included in the extended *VCM*. Large part of this module was developed independently to address a different collection of tasks [8]. The part of the CTM used by the USA-Advisor is described in the next section. Figure 3 below shows the connection between the Extended Valve Control Module and the Circuit Theory Module.

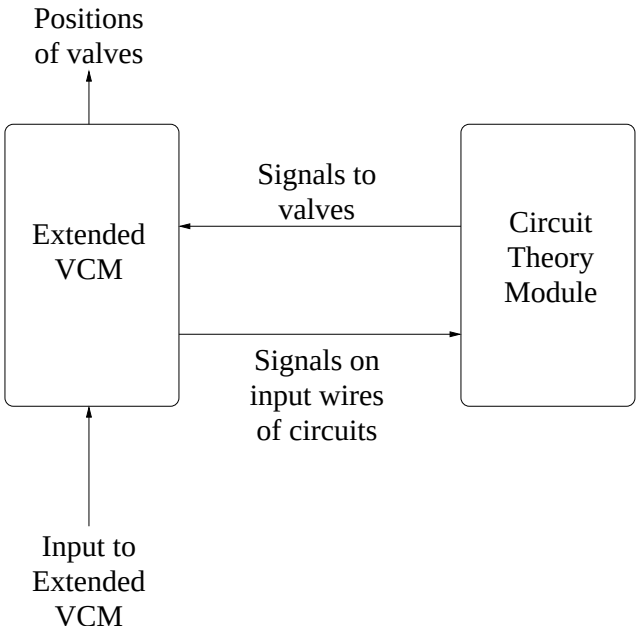

Figure 3 Connection between Extended VCM and Circuit Theory Module.

The state of a valve in the extended *VMC* is determined by the signals present on its two input wires, labeled *open* and *closed*. If the *open* wire is set to 1 and the *closed* wire is set to 0, the valve moves to state open. Similarly for the state closed. The following static causal law defines this behavior. Relation $input\_of\_type(W,S)$ is used to indicate whether W is an *open* wire or a *closed* wire.

```
holds(in_state(V,S1),T) :-
                          input(W1,V), input(W2,V),
                          input_of_type(W1,S1),input_of_type(W2,S2),
                          neq(S1,S2),
                          holds(value(W1,1),T),
                          holds(value(W2,0),T),
                          not stuck(V).
```

The output signals of switches, valves, power buses and control buses are also defined by means of static causal laws, to be discussed shortly.

At this level, the representation of a switch is extended by a collection of its input and output wires. Each input wire is associated to one and only one output wire, and every input/output pair is linked to a position of the switch. There are a few different types of switches in the RCS system. Those that control valves are called *v_switches* and represented by relation *of_type(Sw,v_switch)*. Possible states for v_switches are expressed by relation *state_of(S,v_switch)*, and include *open*, *closed*, and *gpc*. When a switch *Sw* is in position (or state) *S*, an electrical

connection is established between input *Wi* and output *Wo* of the pair(s) corresponding to *S* and represented in A-Prolog by statement *connects(S,Sw,Wi,Wo)*. This relation says that "state *S* of switch *Sw* connects input wire *Wi* to output wire *Wo*." Therefore the signal present on *Wi* is transferred to *Wo*, as expressed by the following rule.

```
holds(value(Wo,X),T) :-
                        holds(in_state(Sw,S),T),
                        connects(S,Sw,Wi,Wo),
                        holds(value(Wi,X),T).
```

Output wires *Wo* of all pairs corresponding to states different from *S* will have value 0 at time *T*, as defined by rule

```
holds(value(Wo,0),T) :-
                        holds(in_state(Sw,S1),T),
                        connects(S2,Sw,Wi,Wo), neq(S1,S2).
```

We will of course also need a more detailed representation of valves. There are two types of valves in the RCS: solenoid and motor controlled valves. However, a motor controlled valve can operate in one of three ways depending on the type of electrical circuit connected to it. So, in our representation, valves can be of four types. In all cases, wires coming from an electrical circuit control the state of the valves. The present state of a valve *V* and the value present on its input wire connected to a power bus control the value of signals on the output wires of *V*.

Valves have a set of input pins, one power pin, and two output pins. Valves are classified according to their physical properties and to the number of input pins they have, as follows: (a) solenoid valves (which have two input pins), (b) two-pin motor-controlled (*MC*) valves, (c) three-pin *MC* valves, and (d) four-pin *MC* valves. The number of input pins determines the way valves are controlled. Two-pin valves have one "open" and one "closed" pin. When a signal 1 is sent to an input pin, while the other is set to 0, the valve moves to the state associated with the pin set to 1. This behavior is captured by rule

```
holds(in_state(V,S),T) :-
                        v_twopin(V), input(W1,V),
                        input_of_type(W1,S), neq(S,power_bus),
                        input(W2,V), input_of_type(W2,S1),
                        neq(S1,power_bus), neq(S,S1),
                        holds(value(W1,1),T),
                        holds(value(W2,0),T),
                        not stuck(V,S1).
```

In these rules, the type, *Y*, of a valve, *V*, is given by statement *type_of_valve(V,Y)*. For instance, valve *ffha* is identified as a solenoid by statement *type_of_valve(ffha,solenoid)*. An input/output pin of a valve has a specific function associated with it. Wires connected to the input pins of valves are represented by the two relations *input(W,V)* and *input_of_type(W,Y)*, where *Y* is chosen in order to be able to distinguish among the different pins.[3]

[3] The actual naming depends on the type of valves.

Rules describing the behavior of three-pin and four-pin valves are similar.

Power and output pins work in the same way for all types of valves. Of the two valve output pins one is labeled “open”, and the other “closed”. When a valve is in state “open”, an electrical connection is established between the power pin and the “open” output pin, while the “closed” output pin is disconnected. Wires connected to the output pins are represented by statements *output(W,V)*, which says that wire *W* is an output wire of valve *V*, and *output_of_type(W,S)*, stating that output wire *W* corresponds to state *S*. Values on output wires of both solenoid and motor controlled valves are determined by rule

```
holds(value(W,1),T) :-
                        of_type(V,valve),
                        output(W,V), output_of_type(W,S),
                        input(Wp,V), input_of_type(Wp,power_bus),
                        holds(in_state(V,S),T),
                        holds(value(Wp,1),T).
```

This rule expresses that if valve *V* is in state *S* at time *T*, then the value on the output wire (corresponding to *S*) of *V* is 1 at *T* when *V* is powered.

Values on output wires of a valve *V* indicate the state of *V*, and are therefore mutually exclusive under normal behavior. If an output wire has value 1 at time *T*, then the value on the other output wire is 0 at *T*. This behavior is defined by rule

```
holds(value(W2,0),T) :-
                        of_type(V,valve),
                        output(W1,V), output(W2,V), neq(W1,W2),
                        holds(value(W1,1),T).
```

If a valve has no power (abnormal condition) then all its output wires have value 0, which is specified by rule

```
holds(value(W,0),T) :-
                        of_type(V,valve), output(W,V),
                        input(Wp,V), input_of_type(Wp,power_bus),
                        holds(value(Wp,0),T).
```

The behaviors described for switches and valves are valid provided that no faults are involved. If a switch is stuck in some position, flipping has no effect. If a valve is stuck in some position, signals on the input pins are not effective. If a power or control bus is faulty, its output is constantly 0. Stuck devices are represented by *stuck(D,S)* as in the basic valve control module. Faulty power buses and control buses are described by statement *bad_device(B)*.

Given the type of a valve *V*, values on input wires of *V* at time *T*, malfunctioning conditions expressed by *stuck(V,S)*, and the state of *V* at time $T-1$, the program determines the state of *V* and the values present on its output wires at moment *T*.

The electrical circuits of the RCS are composed of both analog and digital components. Circuits are named through statements of the form *elec_circ(C)*. In the extended level of the *VCM*, a digital gate or component, *G*, can malfunction if its input/output wire *W* is stuck at a value *X* (0 or 1), defined by statement

stuck_at(W,G,X). If this is the case, the representation of the electrical circuit(s) these gates belong to, are also included as part of the module. However, it is not necessary to add the representation of circuits that are working properly. To indicate that circuit *C* connected to a valve *V* is malfunctioning we add rule

```
bad_circuitry(V) :-
                          bad_circuitry(C),
                          controls(Sw,V,C).
```

The behavior of different components of electrical circuits is described within the circuit theory module.

The Space Shuttle flight computer software is contained in its five general purpose computers (GPCs) which control the vehicle during specific phases of a flight. This software allows control of all RCS activity being responsible for transmitting commands for valve configuration and jet firings. If a switch is placed in GPC state, computer commands can be output to *open* or *close* the affected valves. Issuing a computer command is represented as an action that will affect a target device *D* by setting *D* to a new state. At the extended level of the *VCM*, issuing computer commands is expressed by a dynamic causal law that asserts value 1 on the wire *W* that connects the computer to a component of an electrical circuit. The rule defining this behavior is

```
holds(value(W,1),T+1) :-
                          commands(CC,V,S), output(W,CC),
                          occurs(CC,T).
```

Normally, i.e. in the absence of computer commands, a signal value 0 is assigned to the wire that connects a component of an electrical circuit to the computer, as follows

```
holds(value(W,0),T) :-
                          commands(CC,V,S), output(W,CC),
                          ¬holds(value(W,1),T).
```

Wires connected to the output pins of computer commands, as well as power buses and control buses, are identified by *output(W,E)*, where *E* is either a computer command, a power bus or a control bus.

The extended *VCM*, without the Circuit Theory module, consists of 36 rules.

4.3 Circuit theory module

The Circuit Theory Module (*CTM*) is a general description of normal and faulty behavior of components of electrical circuits with possible propagation delays and 3-valued logic. It can also be used as a stand-alone application for simulation, computation of the maximum delay of a circuit, detection of glitches, and other tasks.

A large portion of the *CTM* was independently developed as part of the A-Circuit project [8]. Because of the modularity of our design, it has been possible to directly include the *CTM* in the USA-Advisor system. Some additions were necessary to account for more complex components used in the RCS. More importantly,

we extended the model to allow the representation of faulty components. The description of the *CTM* is beyond the scope of this paper, but it is important to stress the central role of recursion in the state constraints of the *CTM*. The interested reader can refer to [41] for an in-depth discussion.

Next, we analyze the planning module used in USA-Advisor. For simplicity of presentation we start our discussion by describing the basic structure of the module. Section 4.5 contains an elaboration of the basic module obtained by adding control knowledge. Section 5 describes a further improvement based on an extension of A-Prolog.

4.4 The Basic Planner

The Basic Planning Module of the USA-Advisor establishes a simple search criteria used by the program to find a plan. The structure of the Basic Planning Module described in this section follows the generate and test approach from [21,36,42]. The main idea of this approach consists in establishing a one-to-one correspondence between plans for achieving a goal G in at most a given number, *lasttime*, of steps and answer sets of a logic program P_G. This program normally consists of (a) a large part describing our knowledge about the corresponding dynamic system, and (b) a smaller part containing specification of a goal, a special rule "generating" actions needed to achieve this goal, and possibly some other rules describing properties of the desired plans. The following discussion illustrates this idea. Notice that we differ from the standard answer set planning approach in that we take advantage of the fact that the RCS consists of three, largely independent, subsystems. A plan for the RCS is viewed as the composition of three separate plans that can operate in parallel.

The following rules form the heart of the planner. The first rule, which is responsible for the generation of occurrences actions, states that, for each time point, T, in a given finite interval, if the goal has not been achieved for subsystem R, then an action controlling subsystem R may occur at T.

```
0{occurs(A,T):action_of(A,R)}1 :-
                              T < lasttime, subsystem(R),
                              not goal(T,R).
```

Informally, not $goal(T,R)$ means "if the goal has not been achieved at step T for subsystem R."

The goal of preparing for such a maneuver is also split into subgoals, each preparing a particular subsystem. The first rule below states that the overall goal has been achieved if every subsystem is ready for the current maneuver.

```
goal :-
        selected_maneuver(M),
        holds(maneuver_ready(M,left_rcs),T1),
        holds(maneuver_ready(M,right_rcs),T2),
        holds(maneuver_ready(M,fwd_rcs),T3).

:- not goal.
```

The second rule above is a constraint that states that the overall goal must be achieved in every model.

Splitting the RCS into subsystems allowed us to substantially improve the efficiency of the module because of the reduction in the length of plans. For instance, in some cases, it allowed us to reduce the time to find a plan of 5 steps from a few hours to a few seconds. Notice that, since there actually are some dependencies between some subsystems, a very small number of extremely rare (and undesirable) plans can be missed. It is possible to extend the planning module in order to find these plans. The interested reader may refer to [4] to see how this is accomplished.

Since the RCS contains more than 200 actions, with rather complex effects, and may require long plans, the standard planning approach described above can still be too slow, and needs to be substantially improved. This is done by addition of various forms of heuristic, domain-dependent[4], information. We refer to the Basic Planner expanded by such heuristics as Smart Planner.

4.5 Smart Planner: adding the control knowledge

In this section we will discuss the expansion of the basic planner by useful heuristic information, including control knowledge. The usefulness of control knowledge for planning has been investigated in [1, 34, 32, 3], but comparatively little is known about the influence of heuristics in answer set planning (see however [13]). Such knowledge can be classified into two categories: domain dependent and domain independent. Both types of heuristics work by either limiting the combinations of actions that can occur or by declaring that certain situations are illegal. In either case the heuristics help prune the search space, leading to increased efficiency, and improving plan quality by eliminating unwanted plans.

Some of the control knowledge used in the USA-Advisor can easily be included for planning in other domains. An example of such domain independent knowledge is the statement “Do not repeat actions already performed.” Note that, while this rule does not apply in all domains, in many an optimal plan will never include the same action twice. This rule can be easily encoded in A-Prolog as the following constraint:

```
:- action_of(A,R), neq(T1,T2),
   occurs(A,T1), occurs(A,T2).
```

USA-Advisor contains also a number of domain specific heuristics. The first example shown here states that a switch should not be moved to the gpc (general purpose computer) position unless the following action is to issue a computer command to the valve related to that switch.

```
:- controls(Sw,V),
    occurs(flip(Sw,gpc),T),
    not issued_commands(V,T+1).
```

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

Note that while there are valid plans for the operation of the RCS which do not obey this rule, for each of them there is a plan containing exactly the same actions which does obey it. This allows us to further prune the search space.

More domain-dependent rules embody common-sense knowledge of the type "do not pressurize nodes which are already pressurized." In the RCS, some nodes can be pressurized through more than one path. Clearly, performing an action in order to pressurize a node already pressurized will not invalidate a plan, but this involves an unnecessary action. Although we do not claim the plans computed are optimal, the shortest sequence of actions to achieve the goal is a good candidate as the optimal plan(s). The following constraint eliminates models where more than one path to pressurize a node $N2$ is open.

```
:- link(N1,N2,V1), link(N1,N2,V2), neq(V1,V2),
    holds(in_state(V1,open),T),
    holds(in_state(V2,open),T),
    not stuck(V1,open), not stuck(V2,open).
```

The Planning Module contains approximately 20 rules of which 15 are heuristics.

Next, we discuss the lessons learned from the development of USA-Advisor described in the previous sections. In Section 5, we explain how the use of an extension of A-Prolog later allowed us to substantially improve the quality of reasoning carried out by the system.

4.6 Discussion

The Smart Planner is to the best of our knowledge the largest and most sophisticated answer set planner in existence. Below are some lessons we learned from its design and implementation.

- Since a single action of an astronaut changes the values of many interrelated fluents of the RCS the description of effects of this action becomes a non-trivial task. To solve it we need to find solutions to frame, ramification, and qualification problems [31,23,37]. We solved these problems by using the techniques developed in theory of actions and change and the power of A-Prolog rules. The frame problem was solved by encoding the inertia axiom by a "non-monotonic", default rule of A-Prolog. Qualification was addressed by the use of constraints. And finally, the most difficult ramification problem was solved by the use of static causal laws. It is not clear to us how and if the effects of the RCS actions could be accurately represented by more traditional STRIPS-like action languages like ADL [43].
- A-Prolog proved to be a language capable of specifying the initial situation, causal and other relations of the domain, as well as the heuristic information limiting the search space and improving quality of plans. This contrasts with some of the other representational approaches which require separate languages for each of these classes of statements. For instance, the encoding of heuristic information in [1,2,3] required a fairly sophisticated use of temporal logic.

– The *same* formalization of the domain can be used for other reasoning tasks than planning. All that is needed is replacing the reasoning module, as shown later in Section 6 and in more detail in [5].
– The heuristics used in the Smart Planner were easy to encode and to use. Moreover, our experiments show that they significantly improve both, quality of plans and efficiency of search.
– The planner's ability to mix parallel and sequential plans[5] and to efficiently search for them are the key ingredients in the success of the project.

Overall, answer set planning proved to be a good tool for our purpose. We are not aware of any other tool which would allow us to deal with the complex effects of actions of the RCS.

Experiments show that the system is also quite efficient and meets the criteria for use by NASA stating that a plan should be found in at most 20 minutes. In fact, in our experiments the threshold has been exceeded in only 2 cases out of 2000, while the average time to find a plan has been about 11 seconds – far lower than the 20 minute threshold. Partly this is due to non-numerical nature of the problem. The fact that despite a large number of concurrent actions involved, the plans were comparatively short also contributed to the efficiency. To expand the applicability of answer set planning and reasoning to hybrid systems, i.e. systems involving "continuous" time and numerical computations we need to substantially extend the existing answer set solvers.

Let us now look in more detail at how the experiments were performed and at the results. To assess efficiency, we have randomly generated 2000 problem instances, each specifying a set of faults and a maneuver to be performed. The instances are partitioned in 10 sets of 200 elements, according to the number and type of faults in them. Every set is denoted by a pair $\langle mech, elec \rangle$, where *mech* and *elect* are respectively the number of mechanical and electrical faults present in the instances of the set. Table 1 shows the sets of instances used for our experiments (for further details on the generation of the instances, the reader can refer to [41]).

Recall that our planner takes a parameter, $lasttime$, specifying the maximum allowed length of the plans. In the experiments, we used an algorithm that, given a problem instance, iteratively runs the planner, increasing $lasttime$ by 1 if no plan is found. When a plan is found the procedure terminates and the plan is returned. The value of $lasttime$ ranges between 3 and 10. We also included a timeout of 600 seconds for each call to the planner: if no plan is found within that time, the planner is interrupted and a new iteration is performed. Notice that, if no timeout occurs, this approach is guaranteed to find shortest plans, in terms of the corresponding value of $lasttime$[6].

The average time (including all the calls to the planner that occur during the iterations over $lasttime$) to find a plan of up to 10 steps or determine that none exists are shown in Table 2. The table also includes the number of instances that do not

[5] As we discussed earlier, the plans found by our planner consist of sequences of compound actions, each containing at most one action per subsystem. The elements of each compound action are to be executed concurrently.

[6] But not necessarily in terms of number of actions, as we will see later.

Table 1 Sets of instances used in the experiments.

Set Name	Mechanical Faults	Electrical Faults
ins-3-0	3	0
ins-5-0	5	0
ins-8-0	8	0
ins-10-0	10	0
ins-3-2	3	2
ins-5-3	5	3
ins-8-5	8	5
ins-10-3	10	3
ins-10-5	10	5
ins-10-7	10	7

have a solution if 10 steps or less. As the numbers show, some sets of instances were quite hard.

Figures 4–8 give a graphical representation of the times to find a solution for each instance. All the tests in this paper were performed on a Pentium 4 3.2GHz with 1.5GB RAM running NetBSD 3.99.7, *lparse* 1.0.13, and SMODELS 2.26.

Table 2 Average times, grouped by set of instances.

Set Name	Average Time (sec)	No solution
ins-3-0	4.6443	7
ins-5-0	4.5658	28
ins-8-0	11.4887	63
ins-10-0	22.4169	96
ins-3-2	4.1187	60
ins-5-3	13.8207	102
ins-8-5	8.3934	138
ins-10-3	20.2484	177
ins-10-5	10.1764	162
ins-10-7	11.0357	143
Average	11.0909	976

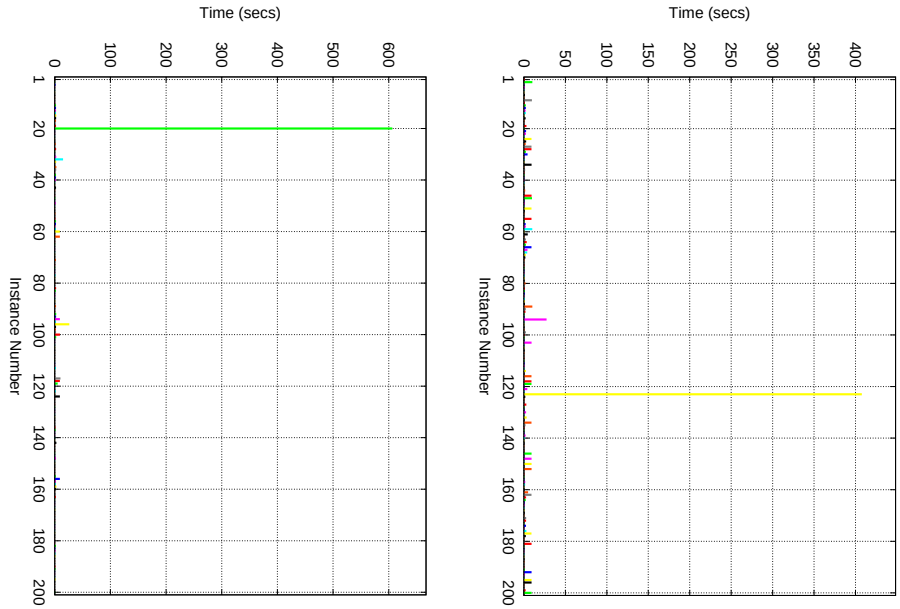

Figure 4 Times for set ins-3-0 (left) and ins-5-0 (right).

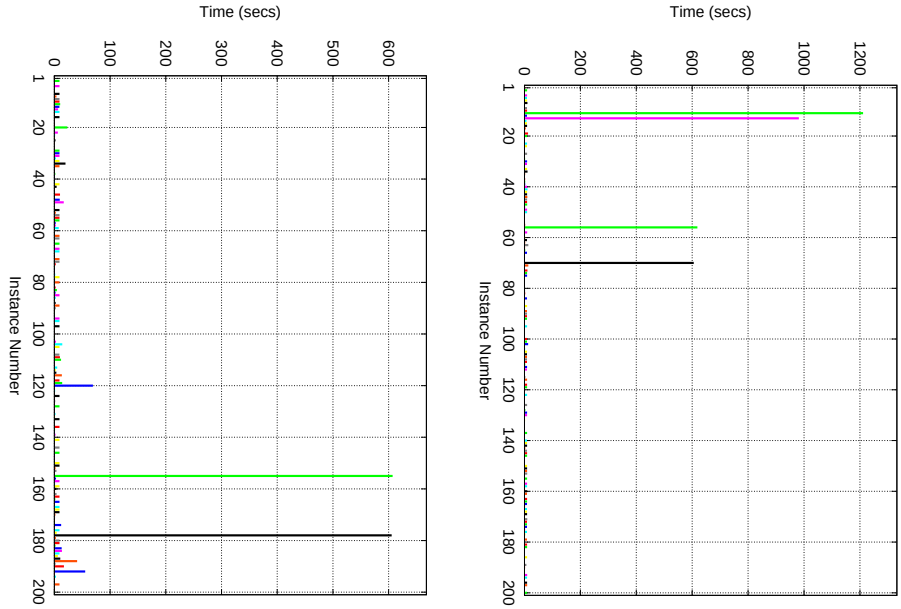

Figure 5 Times for set ins-8-0 (left) and ins-10-0 (right).

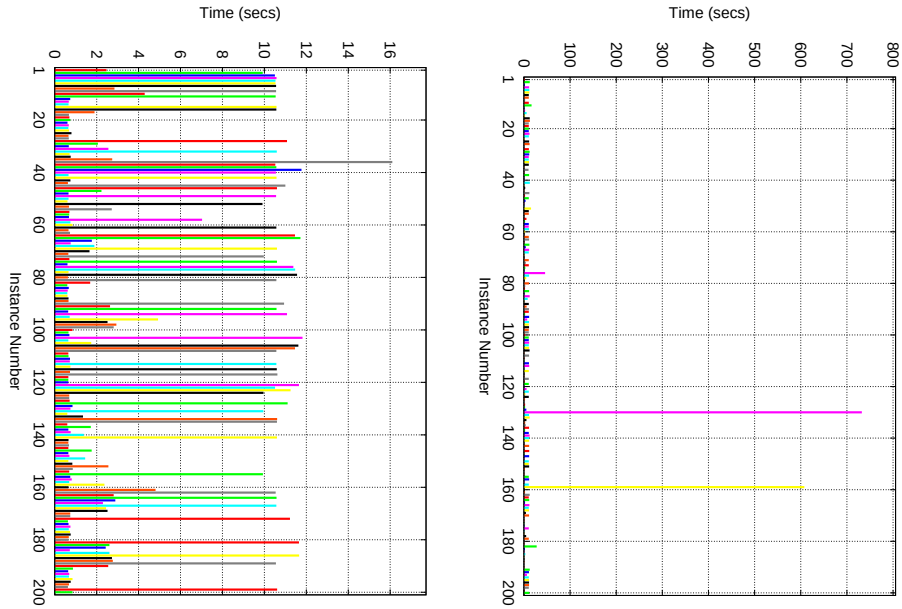

Figure 6 Times for set ins-3-2 (left) and ins-5-3 (right).

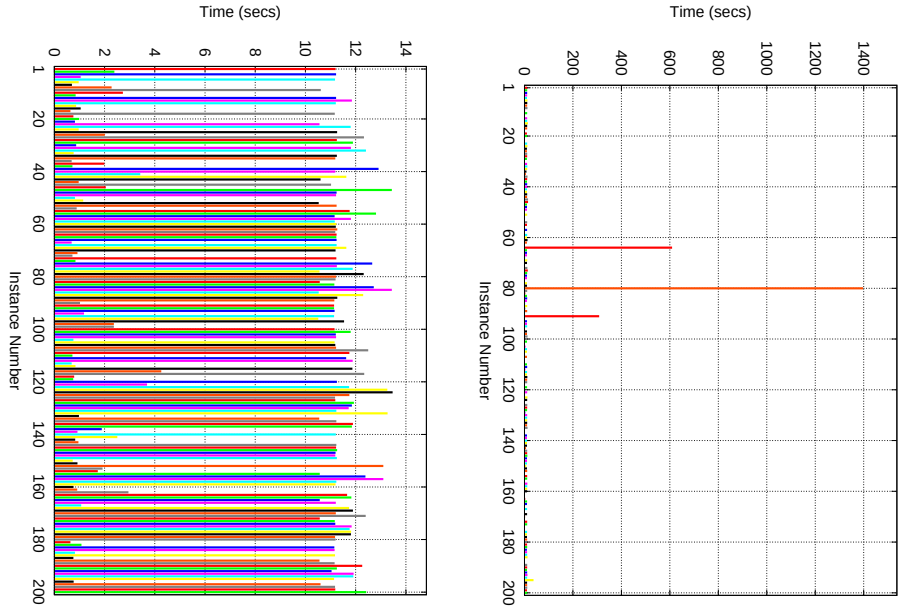

Figure 7 Times for set ins-8-5 (left) and ins-10-3 (right).

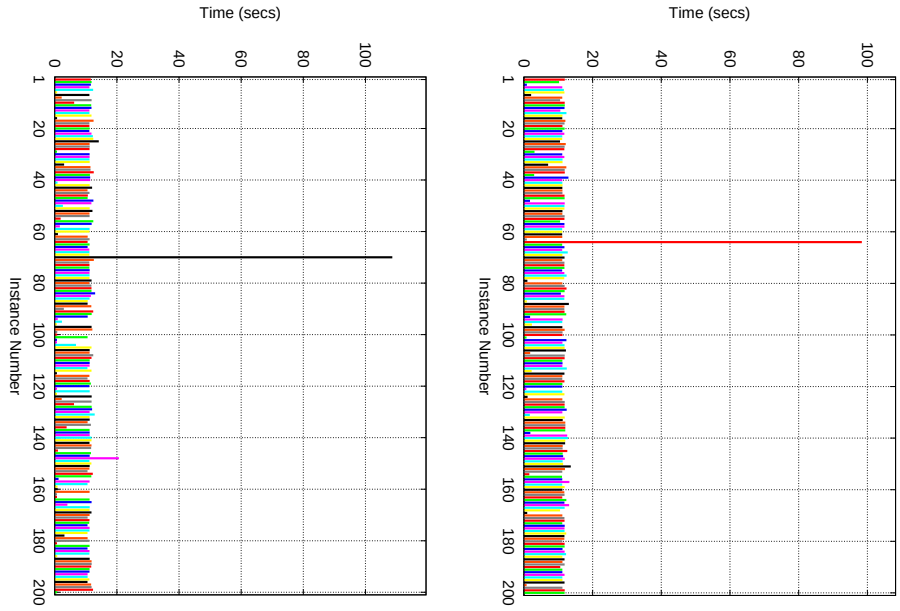

Figure 8 Times for set ins-10-5 (left) and ins-10-7 (right).

5 Improving the Quality of Reasoning

The previous sections showed that USA-Advisor is capable of fairly sophisticated planning in presence of complex faults.

On the other hand, although the plans found are all reasonable, some of them may be preferable to the others. For example, plans that do not involve the use of the crossfeed in the RCS are preferable to those that do, because they allow to maintain a better balance of the level of propellant in the tanks. In this section, we show how answer set programming techniques can be extended to allow the specification of preferences on plans and the computation of such preferred plans. We begin by extending the syntax and semantics of A-Prolog.

5.1 CR-Prolog

CR-Prolog is obtained from A-Prolog by adding consistency-restoring rules (cr-rules) with preferences. Rules of A-Prolog are called *regular rules*. A *cr-rule* is a statement of the form:

$$r: \ h_1 \text{ or } h_2 \text{ or } \ldots \text{ or } h_k \stackrel{+}{\leftarrow} l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n \tag{5}$$

where r is the name of the rule (in the rest of the discussion, we will omit rule names whenever possible). 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.

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

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

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

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

has two answer sets: $\{s, p\}$ and $\{s, q\}$, obtained by applying r_1.

The semantics of the fragment of CR-Prolog described so far can be concisely defined as follows. Let Π^r denote the set of regular rules of program Π and let Π^{cr} denote the set of cr-rules of Π. By $\alpha(r)$ we denote the regular rule obtained from a consistency restoring rule r by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$; α is expanded in a standard way to an arbitrary set R of cr-rules.

A minimal (with respect to set theoretic inclusion) collection R of cr-rules of Π such that $\Pi^r \cup \alpha(R)$ is consistent (i.e. has an answer set) is called an *abductive support* of Π.

A set A of literals is called an *answer set* of Π if it is an answer set of the regular program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.

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*.

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

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

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 $\Pi_2 = \Pi_1 \cup \{r_4 : \leftarrow \text{not } p, \text{not } q\}$. 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 $\Pi_3 = \Pi_2 \cup \{r_5 : \leftarrow p\}$. Its answer set is: $\{q, prefer(r_1, r_2)\}$.

For the definition of the semantics of CR-Prolog, refer to [5].

5.2 CR-Prolog and Soft Requirements

In several cases, "best" plans are selected based on some minimization criteria. An interesting case is when we are given a set of requirements that plans should satisfy *if at all possible* (e.g., "if at all possible, do not skip lunch"). Such requirements are referred to as *soft* (or *defeasible*). In our approach, the satisfaction of soft requirements is checked for in the test phase of the search.

In its simplest form, a soft requirement is encoded by a constraint and a cr-rule. The body of the constraint contains:

- the encoding of the condition that plans should satisfy, according to the soft requirement; the encoding is such that, if the requirement is not met, the body of the constraint is *satisfied*;
- a condition (the *inhibitor*) that allows to stop the application of the constraint, in case the soft requirement has to be violated.

For example, a possible constraint for the soft requirement "if at all possible, do not skip lunch" is:

$$\leftarrow skip(lunch), \text{not } allowed(skip(lunch)).$$

The cr-rule is used to say that, under some conditions, the constraint *may possibly* be inhibited, but its inhibition should be a rare occurrence. The cr-rule for the soft requirement above is:

$$allowed(skip(lunch)) \stackrel{+}{\leftarrow}.$$

which intuitively says that one may be possibly allowed to skip lunch.

If plans exist that do not violate the requirement, the cr-rule is not used. However, if no such plan exists, the cr-rule is used to conclude that skipping lunch is allowed. This inhibits the constraint, and allows the computation of plans violating the requirement.

For another example, consider the encoding of the soft requirement "if possible do not skip lunch; however, if you had a big breakfast, you are allowed to skip lunch," which consists of the rules:

$$\leftarrow skip(lunch), \text{not } allowed(skip(lunch)).$$
$$allowed(skip(lunch)) \stackrel{+}{\leftarrow} had(big\_breakfast).$$

The cr-rule informally says that, if one had a big breakfast, he may possibly be allowed to skip lunch.

When several soft requirements are specified, one is often interested in ranking them in order of preference, so that the most preferred soft requirements are the ones that are less likely to be violated. Preferences statements of CR-Prolog provide a convenient way to encode such preferences. For example, consider the two soft requirements:

- if at all possible, do not skip lunch;
- if at all possible, do not skip dinner;

together with the preference "skipping lunch is preferred over skipping dinner." The soft requirements can be encoded as before:

$$\leftarrow skip(lunch), \text{not } allowed(skip(lunch)).$$
$$skip_l : allowed(skip(lunch)) \stackrel{+}{\leftarrow} .$$

$$\leftarrow skip(dinner), \text{not } allowed(skip(dinner)).$$
$$skip_d : allowed(skip(dinner)) \stackrel{+}{\leftarrow} .$$

The preference is encoded by the following rule:

$$prefer(skip_l, skip_d).$$

which says that (if one has to skip either dinner or lunch) skipping dinner should be considered only if skipping lunch is not possible. It is important to stress that preference statements of CR-Prolog allow to encode more complex criteria than the one above, e.g. dynamic preferences such as "if you had a big breakfast, it is better for you to skip lunch than skipping dinner; otherwise, skipping dinner is preferred." Such preference can be encoded in CR-Prolog with the rules:

$$prefer(skip_l, skip_d) \leftarrow had(big\_breakfast).$$
$$prefer(skip_d, skip_l) \leftarrow \text{not } had(big\_breakfast).$$

5.3 CR-Prolog Based Planner

The structure of the A-Prolog based planners, such as the one shown in Section 4.4, can be easily extended to take defeasible requirements into account. Let *PLANPROB* be a set of A-Prolog rules containing the encoding of a domain description as well as the specification of the goal and the planning module. Let also *SOFTREQ* be the encoding of a set of defeasible requirements. The plans that best satisfy the requirements can be found by computing the answer sets of:

$$PLANPROB \cup SOFTREQ.$$

Soft requirements and preferences over them have an immediate application in an extended planner for USA-Advisor (CR-Plan). For example, recall that the left and right subsystems of the RCS are actually connected by the *crossfeed*, which allows to share propellant between the two subsystems. The crossfeed is intended to be used when one of the two subsystems is affected by faults preventing the use of the propellant from its own tanks. Use of the crossfeed should normally be avoided, to keep the level of propellant in the two subsystems balanced. This statement can be seen as the soft requirement "*avoid the use of the crossfeed if at all possible.*" Following the approach outlined above, a possible encoding of such requirement in CR-Prolog is:

$$r_{xf}(R,T) : allowed(xfeed(R,T)) \stackrel{+}{\leftarrow} subsystem(R).$$

$$\begin{array}{l} \leftarrow subsystem(R), action\_of(A,R), \\ \quad occurs(A,T), \\ \quad opens\_xfeed\_valve(A), \\ \quad \text{not } allowed(xfeed(R,T)). \end{array}$$

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.

To see how the introduction of this requirement affects planning, consider a situation in which the RCS is functioning correctly and we need to perform a maneuver that involves the use of the left and right subsystems.

Because of the absence of faults, the design of the RCS guarantees that the maneuver can be performed without the use of the crossfeed. On the other hand, the design also guarantees that the crossfeed *can* be used to achieve the goal. Hence, the set of plans found by the planner from Sections 4.4 and 4.5 contains both plans that use the crossfeed and plans that do not use it.

If the soft requirement described above is used, then the planner will return only plans that *do not* use the crossfeed, as these are the "best" plans according to the requirement. It is worth stressing the non-monotonic behavior of the planner: if faults are later added to the description of the initial situation, so that the goal can only be achieved with the use of the crossfeed, then the planner will be forced to violate the soft requirement and to return plans that involve the crossfeed.

Another example of the use of soft requirements for USA-Advisor is the encoding of the policy that "*computer commands should be avoided if at all possible.*" (This policy is motivated by the fact that, normally, issuing a computer command requires preparing and uploading a patch of the software of the on-board computer.) The CR-Prolog encoding of the requirements is:

$$r_{ccs}(R,T) : allowed(ccs(R,T)) \stackrel{+}{\leftarrow} subsystem(R).$$

$$\leftarrow subsystem(R), action\_of(A,R),$$
$$\quad occur(A,T), sends\_computer\_command(A),$$
$$\quad \text{not } allowed(ccs(R,T)).$$

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.

It is of course possible to state preferences between the two soft requirements. For example, if modifying the software of the Shuttle's computer is considered preferable to losing the balance of the propellant between the left and right subsystems, the following rule can be added to the planner:

$$prefer(r_{ccs}(R2,T2), r_{xf}(R1,T1)). \tag{6}$$

It is also possible (and often important) to use dynamic preferences. For example, the rules:

$$\begin{array}{l} prefer(r_{xf}(R1,T1), r_{ccs}(R2,T2)) \leftarrow computer\_unreliable. \\ prefer(r_{ccs}(R2,T2), r_{xf}(R1,T1)) \leftarrow \text{not } computer\_unreliable. \end{array} \tag{7}$$

say that the use of the crossfeed is preferred to computer commands only if the on-board computer is *known*[7] to be unreliable. Otherwise, computer commands are preferred.

Notice once again the non-monotonic nature of the planner: if the preference statement(s) are not satisfiable, they can be violated. For example, if the computer is unreliable, but the goal still cannot be achieved after allowing the use of the crossfeed, then the use of computer commands will be allowed.

It is interesting to notice that soft requirements can also be used to avoid the generation of irrelevant actions, typical of planning domains in which the goal is divided in independent subgoals, and the execution of parallel actions is allowed. Consider what happens in USA-Advisor if the goal requires that some jets in the forward and left subsystems be set ready to fire, and achieving the subgoal for the forward subsystem takes n_f steps, while achieving the subgoal for the left subsystem takes n_l steps, with $n_f < n_l$. By inspecting the selection rule used in the planning module, one can see that, even if `lasttime` is set to the lowest possible value of n_l, a plan in which the subgoal for the forward subsystem is achieved at step n_f is considered equivalent to one in which the same subgoal is achieved at $n_f + 1$. For this reason, a plan in which an extra, irrelevant action is performed on the forward

[7] Notice the use of default negation to encode the Closed World Assumption.

subsystem at some $n' < n_f + 1$ is as likely to be returned as the plan that achieves the subgoal at step n_f.

A soft requirement can be written so that, if a plan of length $n_f + 1$ is generated for the forward subsystem, it is possible to guarantee that no extra action will occur at step n' above (i.e. the plan for that subsystem contains an empty step). The soft requirement for irrelevant actions states that "performing actions should be avoided if at all possible.", and is encoded by the rules:

$$r_{short}(R,T) : allowed(execute\_action(R,T)) \stackrel{+}{\leftarrow} subsystem(R).$$

$$\leftarrow subsystem(R), action\_of(A,R),$$
$$\quad occurs(A,T), \text{not } allowed(execute\_action(R,T)).$$

The cr-rule says that, at any step T of the plan for subsystem R, the agent may be possibly allowed to perform actions. The constraint says that it is impossible for action A of subsystem R to occur at step T if the agent is not allowed to execute actions on subsystem R at step T.

Experimental results confirm that the plans returned by CR-Plan are of a significantly higher quality than the plans generated by the basic planner described in Sections 4.4 and 4.5.

We have applied CR-Plan to the problem instances from Section 4.6. The iteration over the maximum plan length has been performed using the algorithm described there. For these experiments, we have used CRMODELS 1.5[8], an inference engine for CR-Prolog recently developed [35].

The experiments have been performed in two sessions. In the first sessions, we have removed from CR-Plan all the preference statements (see (6)–(7) above). The resulting planner, called CR-Plan$^-$, was tested on the 2000 problem instances. In the second session, we added to CR-Plan$^-$ the preference statements (7) and tested the resulting planner, called CR-Plan$^+$, on the same 2000 instances.

The use of CR-Plan$^-$ substantially increased the quality of plans with respect to the A-Prolog based planner. Overall, computer commands and crossfeed were used 119 times, as opposed to 3089 times by the A-Prolog planner, with an improvement of 96.15%. Moreover, in 569 cases, CR-Plan$^-$ returned plans that contained less actions than the plans found by the A-Prolog planner (in no occasion they were longer). The total number of irrelevant actions avoided by CR-Plan$^-$ was 1595, corresponding to a reduction of 19.69% on the total number of actions used (8102 for the A-Prolog planner and 6507 for CR-Plan$^-$).

Although the experiments were mainly aimed at assessing the quality improvement, we found that the speed of CR-Plan$^-$ was still largely acceptable, in spite of the substantial increase in the complexity of the task performed. Out of 2000 runs, CR-Plan$^-$ exceeded NASA's 20 minute threshold only 43 times, corresponding to 2.15% of the instances. The average time to complete one instance was 238 seconds, far below the threshold. If we discard the 43 outliers, the average time goes down to 156 seconds. The times for CR-Plan$^-$ are shown in Table 3.

[8] Available from `http://www.krlab.cs.ttu.edu/Software`.

Table 3 Average times for CR-Plan$^-$ and the A-Prolog planner, grouped by set of instances.

Set Name	Average Time (sec)	A-Prolog Avg. (sec)
ins-3-0	140.5569	4.6443
ins-5-0	161.3922	4.5658
ins-8-0	340.4750	11.4887
ins-10-0	236.3294	22.4169
ins-3-2	155.8627	4.1187
ins-5-3	316.4596	13.8207
ins-8-5	221.4760	8.3934
ins-10-3	282.9618	20.2484
ins-10-5	238.2858	10.1764
ins-10-7	288.1211	11.0357
Average	238.1920	11.0909

The results of the experiments on CR-Plan$^+$ are equally satisfactory. Because of the introduction of preference statements (7) in CR-Plan$^+$, the number of times the crossfeed was used throughout the 2000 instances went down from 86 for CR-Plan$^-$ to 56 for CR-Plan$^+$. The speed of CR-Plan$^+$ was quite good: out of 2000 instances, only 46 times NASA's 20 minute threshold was exceeded (compare to 43 times for CR-Plan$^-$), and the average time was 245.92 seconds (154 seconds if the 46 outliers are discarded). These numbers show that, overall, the introduction of preferences didn't affect significantly the computation time. A comparison of the average times, grouped by set of instances, is shown in Table 4. It is interest-

Table 4 Average times for CR-Plan$^+$ and comparison with the other planners

Set Name	CR-Plan$^+$	CR-Plan$^-$	A-Prolog
ins-3-0	72.2124	140.5569	4.6443
ins-5-0	83.7280	161.3922	4.5658
ins-8-0	368.8101	340.4750	11.4887
ins-10-0	250.4088	236.3294	22.4169
ins-3-2	156.8805	155.8627	4.1187
ins-5-3	351.7330	316.4596	13.8207
ins-8-5	241.5304	221.4760	8.3934
ins-10-3	264.3492	282.9618	20.2484
ins-10-5	265.1365	238.2858	10.1764
ins-10-7	404.4173	288.1211	11.0357
Average	245.9206	238.1920	11.0909

ing to note that in some cases is average time for CR-Plan$^+$ is significantly smaller than that of CR-Plan$^-$. A possible explanation of the phenomenon is that the introduction of preferences adds several constraints on the application of cr-rules, and in some experiments this can help to reduce the search space.

5.4 Discussion

Various extensions of A-Prolog have been recently developed, providing constructs that allow the specification of preferences.

In its simplest form, the `minimize` statement of SMODELS [40] instructs the reasoning system to look for one model that minimizes 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. The fact that the `minimize` statement allows to find only one model limits its applicability, as one may be interested in finding multiple, equally good solutions to a problem. Moreover, in the presence of preferences, an encoding of defeasible requirements based on `minimize` is likely to be less elaboration tolerant than the CR-Prolog equivalent, because of the need to find suitable weights to be assigned to the atoms in the `minimize` statement.

The language of Logic Programs with Ordered Disjunction LPOD [15, 16] is an extension of A-Prolog that allows the specification, in the head of the rules, of a list of alternative literals in order of preference (this is similar to epistemic disjunction, with the difference that in epistemic disjunction all the alternatives are considered equivalent). If the body of the rule is satisfied, one alternative must be selected following the preference order. A possible, rather straightforward, encoding of soft requirements using LPOD consists in writing the constraint part of the requirement as shown earlier, and replacing the corresponding cr-rule by an LPOD rule:

$$\neg allowed(req) \times allowed(req).$$

where $allowed(req)$ is the inhibitor used in the constraint. If multiple requirements need to be specified, and a total preference order exists over them, the constraints are written as usual, and their inhibitors are listed in a single rule:

$$none\_violated \times allowed(req_1) \times allowed(req_2) \times \ldots \times allowed(req_n).$$

Because of the use of a single rule to list all the inhibitors, this type of encoding is less elaboration tolerant than ours. Another difference in CR-Prolog and LPOD lies in the different definition of the preference relation. If conflicts arise among preferences, the Pareto-style preference of LPOD simply ignores the conflicting preferences. On the other hand, our binding preference discards any solutions that are involved in the conflict of preferences. In particular, when preferences are static, the program becomes inconsistent. This behavior derives from our view that programs should contain a small, clearly specified set of preferences. Having inconsistency is a way to alert the user (e.g. the flight controllers) that the preferences were not clearly specified. We believe that our more conservative definition of preferences can be especially useful when the consequences of making the wrong choice are serious (e.g. loss of valuable equipment, loss of lives). For a more detailed discussion, refer to [4].

Finally, the *weak constraints* of DLV [19] (also used in the approaches that rely on a translation to this language, such as DLV$^{\mathcal{K}}$ [22]) provide an elegant way to encode defeasible requirements. Unfortunately, the current implementation of the DLV inference engine does not allow function symbols, which complicates the

development of a complex system such as USA-Advisor. With respect to the encoding of defeasible requirements, notice that preferences on weak constraints are encoded with numerical weights. This is likely to limit the elaboration tolerance of the approach, as the addition of a new soft requirement may require re-assigning most of the weights in the program. It is also important to notice that the definition of the preference relation in DLV is in the style of Pareto preference. Conflicting preferences are ignored in a way similar to LPOD, which may cause problems if the making wrong choice may have negative consequences.

6 Other Forms of Reasoning

In the previous sections we have shown how our A-Prolog based methodology can be used to model complex domains and to perform planning tasks.

An important feature of our methodology is that *the domain model is independent of the particular type of reasoning* and can thus be shared by all the reasoning modules.

To demonstrate this point, in this section we describe a simple diagnostic module for the RCS that uses the same domain model as the planning module. The interested reader may refer to [6] for an in-depth description of answer set based diagnosis.

We view the diagnostic task as a reasoning process in which the agent explains unexpected observations by making hypotheses on faults that may be present in the system. In our approach, observations about fluents are encoded by statements of the form $obs(l,t)$, where l is a fluent literal and t is a time step. The statement informally says that l was observed to hold at step t. Possible observations on the state of the RCS are, for example, $obs(pressurized\_by(ff12j, ffh), 3)$ and $obs(in\_state(ffm1, open), 5)$. Notice the difference between relation *obs*, which encodes observations, and relation *holds*, which encodes the reasoner's beliefs or expectations.

Observations and expectations are linked in A-Prolog by the following set of axioms, *RA*:

$$holds(L,0) \leftarrow obs(L,0).$$
$$\leftarrow obs(L,T), holds(\bar{L},T).$$

The first axiom provides a simple way to describe the initial situation (and can be easily made more sophisticated, e.g. by introducing the Closed World Assumption). The second axiom, called *reality check*, ensures that the reasoner's expectations coincide with the observations.

Given a domain model M, a set H of statements of the form $occurs(A,T)$, specifying the actions performed, and a collection of observations O, the need for a diagnosis can be verified by checking the consistency of

$$\mathscr{S} = M \cup RA \cup H \cup O.$$

If $\mathscr{S}$ is consistent, the reasoner can conclude that O contains no unexpected observations. Otherwise, $\mathscr{S}$ is called a symptom, and a diagnosis needs to be found.

In this paper by diagnosis we mean a set-theoretically minimal collection, $\mathscr{D}$, of faults such that $\mathscr{S} \cup \mathscr{D}$ is consistent.

Notice that, in the context of diagnosis, faults can be viewed as unlikely events, and can thus be nicely formalized using cr-rules. For example, a cr-rule:

$$stuck(V,S) \stackrel{+}{\leftarrow} .$$

says that any valve V may be stuck in some position S, although this is unlikely. Similarly, we can write cr-rules for all the other possible faults from the model of the RCS, e.g.

$$has\_leak(V) \stackrel{+}{\leftarrow} .$$

The corresponding set of cr-rules, *DM*, constitutes a *diagnostic module* for the RCS. It is not difficult to check that, in the presence of unexpected observations, the answer sets of the program:

$$\mathscr{S} \cup DM$$

correspond to the possible diagnoses of the system. In fact, the cr-rules in *DM* are used only if $\mathscr{S}$ is inconsistent. As discussed above, this happens when *O* contains unexpected observations. The application of the cr-rules in *DM* allows the reasoner to assume the existence of faults, and the reality check axiom ensures that in every answer set the reasoner's expectations coincide with the observations. Moreover, thanks to the minimality built in the semantics of CR-Prolog, the set of faults found with this method are minimal with respect to set-theoretic inclusion, and thus constitute diagnoses according to the above definition.

7 Lessons Learned

Our methodology for representing knowledge about dynamic domains and for designing reasoning modules proved to be scalable beyond small domains. The key steps of the methodology are:

1. Identifying the relevant objects and relations in the domain.
2. Identifying the actions.
3. Describing the effects of the actions using the action language based approach.

The decisions made at step (1) heavily influence both the clarity of the model and efficiency of the resulting system. An example of this is the modeling of the junctions of the RCS, which improved to the model substantially.

The availability of state constraints also proved to be important for modeling domains of size and complexity comparable to the RCS. We believe state constraints contributed substantially to the compact definition of fluents such as $pressurized\_by(N,TK)$ and of our general theory of electrical circuits.

Besides its already known applications, we found default negation useful in selecting modules of the domain's encoding. Consider for instance the way relation $bad\_circuitry(V)$ influences the selection of the Basic and Extended Valve Control Modules: in the model, we only had to specify when the relation holds (thus

enabling the Extended Valve Control Module), while default negation was used to determine when it does not hold. Without default negation, we would have been forced to state explicitly when $bad\_circuitry(V)$ does not hold.

Control knowledge proved to be essential in improving the speed of reasoning. Very frequently, this type of information could be found in the operating procedures of the RCS.

In order to improve the speed of computation, it is also profitable to divide the actions in independent subsets (elements of which are executed concurrently) whenever possible.

The use of an external frontend is important to allow the automatical selection of the appropriate modules and avoid problems due to the large size of the grounding of the whole model.

8 Conclusions

In this paper we have described an A-Prolog based methodology for modeling dynamic domains that allows to formalize the description of rather complex domains. In this methodology, A-Prolog, or CR-Prolog in more sophisticated cases, are used to specify the initial situation, the domain model, the control knowledge, and the reasoning modules. It is important to stress that, in our approach, the domain model is shared by *all the reasoning modules*.

The resulting programs are efficient enough to be used for practical applications. We demonstrated our methodology by applying it to the development of a decision support system for the Reaction Control System of the Space Shuttle. [9] The system is intended for actual use by NASA flight controllers and its applicability is not limited to the Space Shuttle. In fact, the development of USA-Advisor is currently being continued by the programmers at United Space Alliance, who are working on formalizing models of other systems of the Shuttle and of the International Space Station, as well as creating suitable graphical interfaces.

Finally, in this paper we have also shown how CR-Prolog, the extension of A-Prolog by consistency-restoring rules and preferences, allows to substantially improve the quality of reasoning by specifying soft requirements (i.e. conditions on the solutions that the reasoning modules should satisfy if at all possible) and preference over them.

Acknowledgements The authors would like to thank United Space Alliance for their continued support. This work was partially supported by United Space Alliance under contract number NAS9-20000 and by NASA under contract number NASA-NNG05GP48G.

References

1. F. Bacchus and F. Kabanza. *Using Temporal Logic to Control Search in a Forward Chaining Planner*, pages 141–153.

[9] USA-Advisor can be downloaded from `http://www.krlab.cs.ttu.edu/Software`.

2. F. Bacchus and F. Kabanza. Planning for Temporally Extended Goals. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):5–27, 1998.
3. F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 16:123–191, 2000.
4. Marcello Balduccini. USA-Smart: Improving the Quality of Plans in Answer Set Planning. In *PADL'04*, Lecture Notes in Artificial Intelligence (LNCS), Jun 2004.
5. Marcello Balduccini. *Answer Set Based Design of Highly Autonomous, Rational Agents*. PhD thesis, Texas Tech University, Dec 2005.
6. Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)*, 3(4–5):425–461, Jul 2003.
7. Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. In Patrick Doherty, John McCarthy, and Mary-Anne Williams, editors, *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, pages 9–18, Mar 2003.
8. Marcello Balduccini, Michael Gelfond, and Monica Nogueira. A-Prolog as a tool for declarative programming. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000)*, pages 63–72, 2000.
9. Marcello Balduccini and Veena S. Mellarkod. A-Prolog with CR-Rules and Ordered Disjunction. In *ICISIP'04*, pages 1–6, Jan 2004.
10. Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Jan 2003.
11. Chitta Baral and Michael Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19(20):73–148, 1994.
12. Chitta Baral and Michael Gelfond. Reasoning Agents In Dynamic Domains. In *Workshop on Logic-Based Artificial Intelligence*, pages 257–279. Kluwer Academic Publishers, Jun 2000.
13. Chitta Baral and Le-Chi Tuan. Effect of knowledge representation on model based planning: experiments using logic programming encodings. In *Proceedings of 2001 AAAI Spring Symposium on Answer Set Programming*, pages 110–115, 2001.
14. Matthew Barry and Richard Watson. Reasoning about actions for spacecraft redundancy management. In *Proceedings of the 1999 IEEE Aerospace Conference*, volume 5, pages 101–112, 1999.
15. Gerhard Brewka. Logic programming with ordered disjunction. In *Proceedings of AAAI-02*, 2002.
16. 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.
17. Gerhard Brewka, Ilkka Niemela, and Tommi Syrjanen. Logic Programs wirh Ordered Disjunction. 20(2):335–357, 2004.
18. Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Adding Weak Constraints to Disjunctive Datalog. In *Proceedings of the 1997 Joint Conference on Declarative Programming APPIA-GULP-PRODE'97*, 1997.
19. Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Giovanbattista Ianni, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, Gerard Pfeifer, and Axel Polleres. The DLV System. In Sergio Flesca and Giovanbattista Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, Sep 2002.
20. Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerard Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 03)*. Morgan Kaufmann, Aug 2003.
21. 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.
22. 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.
23. J. J. Finger. *Exploiting Constraints in Design Synthesis*. PhD thesis, Stanford University, 1987.

24. Alfredo Gabaldon and Michael Gelfond. From Functional Specifications to Logic Programs. In *Proceedings of the International Logic Programming Symposium (ILPS'97)*, 1997.
25. 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.
26. Michael Gelfond and Nicola Leone. Knowledge Representation and Logic Programming. *Artificial Intelligence*, 138(1–2), 2002.
27. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.
28. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–385, 1991.
29. Michael Gelfond and Vladimir Lifschitz. Action Languages. *Electronic Transactions on AI*, 3(16), 1998.
30. Michael Gelfond and Richard Watson. On methodology for representing knowledge in dynamic domains. In *Proc of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, pages 57–66, 1999.
31. Patrick J. Hayes and John McCarthy. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.
32. Y. Huang, H. Kautz, and B. Selman. Control Knowledge in Planning: Benefits and Tradeoffs. In *Proceedings of the 16th National Conference of Artificial Intelligence (AAAI'99)*, pages 511–517, 1999.
33. M. Kaminski. A note on the stable model semantics of logic programs. *Artificial Intelligence*, 96(2):467–479, 1997.
34. H. Kautz and B. Selman. The Role of Domain-Specific Knowledge in the Planning as Satisfiability Framework. In *Proceedings of AIPS'98*, 1998.
35. Loveleen Kolvekal. Developing an Inference Engine for CR-Prolog with Preferences. Master's thesis, Texas Tech University, Dec 2004.
36. Vladimir Lifschitz. *Action Languages, Answer Sets, and Planning*, pages 357–373. The Logic Programming Paradigm: a 25-Year Perspective. Springer Verlag, Berlin, 1999.
37. John McCarthy. Epistemological problems of artificial intelligence. In *Proceedings of IJCAI-77*, pages 1038–1044, 1977.
38. Veena S. Mellarkod. Optimizing the Computation of Stable Models using Merged Rules. Master's thesis, Texas Tech University, May 2002.
39. Ilkka Niemela and Patrik Simons. *Extending the Smodels System with Cardinality and Weight Constraints*, pages 491–521. Logic-Based Artificial Intelligence. Kluwer Academic Publishers, 2000.
40. Ilkka Niemela, Patrik Simons, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, Jun 2002.
41. Monica Nogueira. *Building Knowledge Systems in A-Prolog*. PhD thesis, University of Texas at El Paso, May 2003.
42. 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.
43. E. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, pages 324–332, 1989.
44. Raymond Reiter. *On Closed World Data Bases*, pages 119–140. Logic and Data Bases. Plenum Press, 1978.
45. Patrik Simons. Extending the Stable Model Semantics with More Expressive Rules. In *Proceedings of the 5th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR-99)*, number 1730 in Lecture Notes in Artificial Intelligence (LNCS). Springer Verlag, Berlin, 1999.
46. Richard Watson. An application of action theory to the Space Shuttle. In *PADL-99*, volume 1551 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 290–304, 1999.