Model-Based Reasoning for Complex Flight Systems

Marcello Balduccini and Michael Gelfond
Computer Science Department
Texas Tech University
{balduccini, mgelfond}@cs.ttu.edu

This paper gives an overview of the design of a decision support system for the Space Shuttle that has the ability to find (usually in a matter of seconds) provably correct plans to achieve a given goal in the presence of single or multiple failures in the Reaction Control System (RCS). This tool includes a complete model of the RCS, including wiring and plumbing diagrams. Both the models and the reasoning modules were designed in the context of a project aimed at demonstrating the applicability of the knowledge representation declarative language A-Prolog, and of the answer set programming methodology in particular, to medium-size, knowledge-intensive applications. The project also demonstrated that A-Prolog allows a modular organization of knowledge, enabling knowledge module reuse, as well as testing (and debugging) of single knowledge modules, rather than of the entire knowledge base as a whole. The techniques used in the development of our decision support system are general enough to allow for the modeling of many other flight systems, and for the execution of reasoning tasks other than planning, including fault detection and diagnosis.

I. Introduction

The research presented in this paper is rooted in recent developments in several areas of AI. Advances in the work on semantics of negation in logic programming[1,2] and on formalization of common-sense reasoning[3,4] led to the development of the knowledge representation language A-Prolog, an entirely declarative language used in this paper to encode the domain knowledge, and to the A-Prolog based programming methodology called answer set programming (ASP).[5,6] Insights on the nature of causality and its relationship with answer sets of logic programs[7–9] determined the way we characterize effects of actions and solve the frame, ramification, and qualification problems which, for a long time, caused difficulties in modeling reasoning about dynamic domains. Work on propositional satisfiability influenced the development of algorithms for computing answer sets of A-Prolog programs and programming systems[10–12] implementing these algorithms. Last, but not least, we build on earlier work on applications of answer set programming to planning.[13,14]

The goal of this paper is to show an overview of the design of a decision support system for the Space Shuttle (USA-Advisor) that has the ability to find (usually in a matter of seconds) provably correct plans to achieve a given goal in the presence of single or multiple failures in the Reaction Control System (RCS). The RCS is the system that is primarily used to perform rotational and translational movements during flight. It is a rather complex physical system, that includes 12 tanks, 44 jets, 66 valves, 33 switches, and around 40 computer commands (computer-generated signals).

USA-Advisor contains a complete model of the RCS, including wiring and plumbing diagrams. Both model and reasoning modules were designed in the context of a project aimed at demonstrating the applicability of A-Prolog, and of ASP in particular, to medium-size, knowledge-intensive applications. The project also demonstrated that A-Prolog allows a modular organization of knowledge, enabling knowledge module reuse, as well as testing (and debugging) of single knowledge modules, rather than of the entire knowledge base as a whole. The techniques used in the development of our decision support system are general enough to allow for the modeling of many other flight systems, and for the execution of reasoning tasks other than planning, including fault detection and diagnosis.

To understand the functionality of USA-Advisor, let us imagine a Shuttle's flight controller who is considering how to prepare the Shuttle for a maneuver when faced with a collection of faults present in the RCS (for example, switches and valves can be stuck in various positions, electrical circuits can malfunction in various ways, valves can be leaking, jets can be damaged, etc). In this situation, the controller needs to find a sequence of actions (a plan) to set the Shuttle ready for the maneuver. USA-Advisor is designed to facilitate this task. The controller can use it to test if a plan, which he came up with manually, will actually be able to prepare the RCS for the desired maneuver. Most importantly, USA-Advisor can be used to automatically find such a plan. It is worth stressing that, because of the possibility of mathematically demonstrating that the reasoning modules work correctly, USA-Advisor (and its successors) can also provide immediate on-board support for future unmanned and human space missions, not only as a decision support system, but also as a control system.

In the next section we give a brief introduction to the design of the system.

II. System's Design

USA-Advisor consists of a collection of largely independent A-Prolog modules, represented by lp-functions[a], and a graphical Java interface. The interface gives a simple way for the user to enter information about the history of the RCS, its faults, and the task to be performed. The A-Prolog modules are organized in knowledge modules and reasoning modules. Each knowledge module contains a different part of the knowledge about the domain of the RCS, while each reasoning module is responsible for performing a different reasoning task. At the moment there are two possible types of tasks:

- checking if a sequence of occurrences of actions in the history of the system satisfies a given goal, G;
- finding a plan for G of a length not exceeding some number of steps, N.

The set of A-Prolog modules that are used depends on the particular task being performed (e.g. detailed knowledge about electrical circuits is included only in presence of electrical faults). Based on the information provided by the user, the graphical interface verifies that the input is complete, selects an appropriate combination of modules, assembles everything into an A-Prolog program, Π, and passes Π as an input to an inference engine for computing its answer sets. (The inference engine used in USA-Advisor is SMODELS[b].)

The results of the reasoning task are then extracted from the answer sets of Π. The interpretation of the contents of the answer sets depends, again, on the reasoning task being performed. In our approach, the task of verifying the correctness of a sequence of actions is reduced to checking if program Π has at least an answer set. On the other hand, the planning module is designed so that there is a one-to-one correspondence between the plans and the answer sets of Π. Extraction and displaying of the results is performed by the Java interface.

In the rest of this section we give a more detailed description of particular knowledge modules. These modules consist of A-Prolog rules of the form

$$l_0 : -l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n.$$

where l_i's are atoms following essentially the same syntax as Prolog atoms. The intuitive reading of such a rule is "if $l_1, \ldots, l_m$ are true and there is no reason to believe that $l_{m+1}, \ldots, l_n$ hold, then l_0 is true." For a more thorough description of the syntax and semantics of A-Prolog and (some of) its extensions, the reader can refer to.[1,2,15]

A. 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, Gr, 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 changes in the position of valves affect the pressure of tanks, jets and junctions. 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. The necessary condition for a fluid to flow from a tank to a jet, and in general to any node of Gr, is that there exists a path without leaks from the tank to the node and that all valves along the path are open.

[a]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.

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

The rules of PM define a function which takes as input the structural description, Gr, of the plumbing system, its current state, including position of valves and the list of faulty components, and determines: the distribution of pressure through the nodes of Gr; which jets are ready to fire; which maneuvers are ready to be performed. In our approach, the state of the plumbing system (as well as of the electrical system shown later) consists of the set of fluents (properties of the domain whose truth depends on time) which are true in that state.

To illustrate the issues involved in the construction of PM, let us consider the definition of fluent $pressurized\_by(N, Tk)$, describing the pressure obtained on a node N by a tank Tk. Some special nodes, the helium tanks, are always pressurized. For all other nodes, the definition is recursive. It says that any 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.

Representation of this definition in most logic programming languages, including Prolog, is problematic, since the corresponding graph can contain cycles. The ability of A-Prolog to express and to reason with recursion allows us to use the following concise definition of pressure on non-tank nodes.

```
h(pressurized_by(N1,Tk),T) :-
            not tank_of(N1,R),
            not h(leaking(N1),T),
            link(N2,N1,V),
            h(in_state(V,open),T),
            h(pressurized_by(N2,Tk),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.

B. Valve control module

The flow of fuel and oxidizer propellants from tanks to jets is controlled by opening/closing valves along the path. 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.

The action of flipping a switch Sw to some position S normally puts a valve controlled by Sw in this position. Similarly for computer commands. There are, however, three types of possible failures: switches and valves can be stuck in some position, and electrical circuits can malfunction in various ways. 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, flipping switches and issuing computer commands may produce results that cannot be predicted by the basic representation.

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 initial positions and faults of switches and valves, and the sequence of actions defining the history of events. 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 possible faults of the system at this level are valves and switches stuck at some position(s).

Effects of actions in the basic VCM are described in a variant of action language $\mathcal{B}$,[16] which contains both static and dynamic causal laws, as well as impossibility conditions. Our version of $\mathcal{B}$ uses a slightly different syntax to avoid lists and nesting of function symbols, because of limitations of the inference engines currently available. The use of $\mathcal{B}$ allows to prove correctness of logic programming implementation of causal laws.[17] (Of course, it does not guarantee correctness of the causal laws *per se*. This can only be done by domain experts.) The complexity of this representation makes it hard to employ STRIPS-like formalisms.

The following rules show an example of syntax and use of our version of $\mathcal{B}$. The first is a dynamic causal rule stating that, if a properly working switch Sw is flipped to state S at time T, then Sw will be in this state at the next moment of time.

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

A static connection between switches and valves is expressed by the next rule. This static law says that, under normal conditions, if switch Sw controlling a valve V is in some state S (different from *gpc*[c]) at time T, then V is also in this state at the same time.

```
h(in_state(V,S),T) :-
            controls(Sw,V),
            h(in_state(Sw,S),T),
            neq(S,gpc),
            not h(ab_input(V),T),
            not stuck(V),
            not bad_circuitry(V).
```

The condition *not bad_circuitry(V)* is used to stop this rule from being applied when the circuit connecting Sw and V is not working properly. (Notice that the previous dynamic rule, instead, is applied independently of the functioning conditions of the circuit, since it is related only to the switch itself.) If the switch is in a position, $S1$, different from *gpc*, and a computer command is issued to move the valve to position $S2$, then there is a conflict in case $S1 \neq S2$. This is an abnormal situation, which is expressed by fluent $ab\_input(V)$. When this fluent is true, negation as failure is used to stop the application of this rule. In fact, the final position of the valve can only be determined by using the representation of the electrical circuit that controls it. This will be discussed in the next section.

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.

The lp-function defined by this module takes as input the same information accepted by the basic VCM, together with faults on power buses, control buses and electrical circuits. It returns the 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. Since this module was developed independently to address a different collection of tasks,[18] its use in this system is described in a separate section.

There are two main types of valves in the RCS: solenoid and motor controlled valves. Depending on the number of input wires they have, motor controlled valves are further divided in 3 sub-types. While at the basic VCM level there is no need to distinguish between these different types of valves, they must be taken into account at the extended level, since the type determines the number of input wires of the valve. In all cases, the state of a valve is normally determined by the signals present on its input wires.

For the solenoid valve, its two input wires are 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 law defines this behavior.

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

[c]A switch can be in one of three positions: open, closed, or gpc. When it is in gpc, the state of the valve is determined by input from the on-board computer.

The state of all other types of valves is determined in much the same way. The only difference is in the number of wires that are taken into consideration.

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

At this level, the representation of a switch is extended by a collection of 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. When a switch is in position S, an electrical connection is established between input Wi and output Wo of the pair(s) corresponding to S. Therefore, the signal present on Wi is transferred to Wo, as expressed by the following rule.

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

The VCM consists of 36 rules, not including the rules of the Circuit Theory Module.

C. Circuit theory module

The Circuit Theory Module (CTM) is a general description of components of electrical circuits. It can be used as a stand-alone application for simulation, computation of the topological delay of a circuit, detection of glitches, and abduction of the circuit's inputs given the desired output.

The CTM is employed in this system to model the electrical circuits of the RCS, which are formed by digital gates and other electrical components, connected by wires. Here, we refer to both types of components as *gates*. The structure of an electrical circuit is represented by a directed graph E where gates are nodes and wires are arcs. A gate can possibly have a propagation delay D associated with it, where D is a natural number (zero indicates no delay). All signals present in the circuit are expressed in 3-valued logic (0, 1, u). If no value is present on a wire at a certain moment of time T then it is said to be unknown (u) at T.

This module describes the normal and faulty behavior of electrical circuits with possible propagation delays and 3-valued logic.

In CTM, *input wires* of a circuit are defined as the wires coming from switches, valves, computer commands, power buses and control buses. *Output wires* are those that go to valves. The CTM is an lp-function that takes as input the description of a circuit C, the values of signals present on its input wires, the set of faults affecting its gates, and determines the values on the output wires of C at the current moment of time.

We allow for standard faults from the theory of digital circuits.[19,20] A gate G malfunctions if its output, or at least one of its input pins, are permanently stuck on a signal value. The effect of a fault associated to a gate of the direct graph E only propagates forward.

CTM contains two sets of static rules. One of them is used for the representation of the normal behavior of gates, while the other expresses their faulty behavior. To illustrate how the normal behavior of gates is described in the CTM, let us consider the case of the Tri-State gate. This type of component has two input wires, of which one is labeled *enable*. If this wire is set to 1, the value of the other input is transferred to the output wire. Otherwise, the output is undefined. The following rule describes the normal behavior of the Tri-State gate when it is enabled.

```
h(value(W,X),T+D) :-
           delay(G,D),
           input(W1,G),
           input(W2,G),
           type_of_wire(W2,G,enable),
           neq(W1,W2),
           h(value(W1,X),T),
           h(value(W2,1),T),
           output(W,G),
           not is_stuck(W,G).
```

It is interesting to discuss how faults are treated when they occur on the input wire of a gate. Let us consider the case of a gate G with an input wire stuck at value X. This wire is represented as two unconnected wires, W and $stuck\_wire(W)$, corresponding to the normal and faulty sections of the wire. The faulty part is stuck at value X, while the value of W is computed by normal rules depending upon its connection to the output of other gates. Rules for gates with faulty inputs use $stuck\_wire(W)$ as input wire. The example below is related to a Tri-State gate with the non-enable wire stuck to X.

```
h(value(W,X),T+D) :-
            delay(G,D),
            input(stuck_wire(W1),G),
            input(W2,G),
            type_of_wire(W2,G,enable),
            neq(W1,W2),
            h(value(stuck_wire(W1),X),T),
            h(value(W2,1),T),
            output(W,G),
            not is_stuck(W,G).
```

Notice that condition *not is_stuck(W,G)* prevents the above rules from being applied when the output wire is stuck. Whenever an output wire is stuck at X, the corresponding rule guarantees that its signal value is always X.

The behavior of a circuit is said *normal* if all its gates are functioning correctly. If one or more gates of a circuit malfunction then the circuit is called *faulty*.

The description of faulty electrical circuit(s) is included as part of the RCS representation. However, it is not necessary to add the description of normal circuits controlling a valve(s) since the program can reason about effects of actions performed on that valve through the basic VCM. This allows for an increase in efficiency when computing models of the program.

The Circuit Theory Module contains approximately 50 rules.

D. Planning module

This module establishes the search criteria used by the program to find a plan, i.e. a sequence of actions that, if executed, would achieve the goal. The modular design of USA-Advisor allows to create of a variety of such modules.

The structure of the Planning Module (PlM) follows the generate and test approach described in.[13,14] Since the RCS contains more than 200 actions, with rather complex effects, and may require very long plans, this standard approach needs to be substantially improved. This is done by addition of various forms of heuristic, domain-dependent information[d]. In particular, the generation part takes advantage of the fact that the RCS consists of three, largely independent, subsystems. A plan for the RCS can therefore be viewed as the composition of three separate plans that can operate in parallel. Generation is implemented using the following rule:

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

This rule states that exactly one action for each subsystem of the RCS should occur at each moment of time, until the goal is reached for that subsystem.

In the RCS, the common task is to prepare the Shuttle for a given maneuver. The goal of preparing for such a maneuver can be split into several subgoals, each setting some jets, from a particular subsystem, ready to fire. The overall goal can therefore be stated as a composition of the goals of individual subsystems containing the desired jets, as follows:

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

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

The plan testing phase of the search is implemented by the following constraint

```
:- not goal.
```

which eliminates the models that do not contain plans for the goal.

Splitting into subsystems allows us to improve the efficiency of the module substantially.

The module also contains other domain-dependent as well as domain-independent heuristics. The reasons for adding such heuristics are two-fold: first, to eliminate plans which are correct but unintended, and second, to increase efficiency. A-Prolog allows a concise representation of these heuristics as constraint rules. This can be demonstrated by means of the following examples.

Some heuristics are instances of domain-independent heuristics. They express common-sense knowledge like "under normal conditions, do not perform two different actions with the same effect." In the RCS, there are two different types of actions that can move a valve V to a state S: a) flipping to state S the switch, Sw, that controls V, or b) issuing the (specific) computer command CC capable of moving V to S. In A-Prolog we can write this heuristic as follows

```
:- occurs(flip(Sw,S),T),
   controls(Sw,V),
   occurs(CC,T1),
   commands(CC,V,S),
   not bad_circuitry(V).
```

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 discuss optimality of plans in this paper, 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),
   h(in_state(V1,open),T),
   h(in_state(V2,open),T),
   not stuck(V1,open),
   not stuck(V2,open).
```

As mentioned before, some heuristics are crucial for the improvement of the planner's efficiency. One of them states that "a normally functioning valve connecting nodes $N1$ and $N2$ should not be open if $N1$ is not pressurized." This heuristic clearly prunes a significant number of unintended plans. It is represented by a constraint that discards all plans in which a valve V is opened before the node, preceding it, is pressurized.

```
:- link(N1,N2,V),
   h(in_state(V,open),T),
   not h(pressurized_by(N1,Tk),T),
   not has_leak(V),
   not stuck(V).
```

The efficiency improvement offered by domain-dependent heuristics has not been studied mathematically. However, experiments showed impressive results. In the case of tasks involving a large number of faults, for example, the introduction of some of the most effective heuristics reduced the time required to find a plan from hours to seconds.

III. Conclusion

In this paper we described a medium size decision support system written in A-Prolog. This application requires modeling of the operation of a fairly complex subsystem of the Space Shuttle at a level suitable for use by the Shuttle's flight controllers. Work for the deployment of this tool at United Space Alliance is under way (most of the refinement is expected to regard the user interface).

The techniques used to design USA-Advisor are general enough to be applicable to many other flight systems, and to allow for the execution of reasoning tasks other than planning, including fault detection and diagnosis. To verify that, we have already implemented a simple diagnostic module based on techniques from.[21] The diagnostic module takes in input a history of occurrences of actions and a set of observations on the state of the RCS after such actions have been performed, and returns the smallest collection of faults that would cause the observed behavior (other types of minimization are possible, e.g. preference-based). It is important to stress that the use of the diagnostic module does not require the modification of the model of the RCS. In fact, our diagnostic module is run in conjunction with the same knowledge modules used for planning. We believe that this is an important property of our approach, resulting from both the modularity of the design and the expressive power of A-Prolog. This property allows a high degree of knowledge reuse, which is essential in cases where a substantial effort has to be made to verify that the model is correct with respect to the physical system.

This project showed the advantages of A-Prolog with respect to standard Prolog, evident even in the case of plan checking. An important methodological lesson we learned from this exercise is the importance of careful initial design. For instance, introduction of junction nodes in the model of the Plumbing Module of the RCS substantially simplified the resulting program. We are also satisfied with our use of the Java interface for selecting modules necessary for solving a given problem, and integrating these modules into a final A-Prolog program. Structuring most modules as lp-functions contributed to the reusability and proof of correctness of the integration[e]. Such proof is especially important due to the critical nature of the RCS.

From the point of view of the research on planning, this work shows a system of substantial size built on theory of actions and change. In particular, it is worth stressing the key role of static causal laws in our model. It is unclear whether the use of STRIPS-like languages containing only dynamic causal laws is sufficient for a concise representation of the RCS, and especially of the extended VCM.

The use of A-Prolog allowed us to deal with recursive causal laws, which may pose a problem to more classical planning methods. (Partial solution to this problem is suggested in,[22] where the authors use CCALC[23] to reduce the computation of answer sets to the computation of models of some propositional formula. They give a sufficient condition of the correctness of such transformation. Unfortunately, the idea does not apply here, since the corresponding graph is not acyclic.)

Recent work in planning drew attention to the problem of finding a language which would allow a declarative and efficient representation of heuristic information.[24–27] We believe that this paper demonstrates that a large amount of such information can be naturally expressed in A-Prolog. Moreover, its use dramatically improves efficiency of the planner (which is not always the case for satisfiability based planners.)

Finally, this paper demonstrates that the concept of modularity can be applied to A-Prolog programs, and how the use of modules allows planning to be easily and naturally performed at different levels of abstraction (i.e. if no electric faults are present, the detailed behavior of electrical circuits can be abstracted upon by just selecting the proper modules). The modular organization of knowledge made it possible for each knowledge module to be developed and tested, to a large extent, separately from the other modules. This has substantially simplified the next stage of the development, when we have begun using the modules together.

IV. Acknowledgements

This project was partially supported by United Space Alliance under contract NAS9-20000.

[e]To give an example of what we learned here, let us consider the following situation: suppose you have lp-functions f and g correctly implementing the plumbing and basic VCM modules of the system; integration of these modules leads to the creation of new lp-function $h = f \circ g$. It is known that, due to non-monotonicity of A-Prolog, logic programming representation of this function cannot always be obtained by combining together rules of f and g. In our case, however, a general theorem[17] can be used to check if this is indeed the case. We are currently working on formulating and proving the correctness of the complete integration.

References

[1] Gelfond, M. and Lifschitz, V., "The stable model semantics for logic programming," *Proceedings of ICLP-88*, 1988, pp. 1070–1080.

[2] Gelfond, M. and Lifschitz, V., "Classical negation in logic programs and disjunctive databases," *New Generation Computing*, 1991, pp. 365–385.

[3] Moore, R. C., "Semantical considerations on nonmonotonic logic," *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, Aug 1983, pp. 272–279.

[4] Reiter, R., "A Logic for Default Reasoning," *Artificial Intelligence*, Vol. 13, No. 1–2, 1980, pp. 81–132.

[5] Lifschitz, V., "Answer set programming and plan generation," *Artificial Intelligence*, Vol. 138, 2002, pp. 39–54.

[6] Subrahmanian, V. S., "Relating Stable Models and AI Planning Domains," *Proceedings of ICLP-95*, 1995.

[7] Gelfond, M. and Lifschitz, V., "Representing Action and Change by Logic Programs," *Journal of Logic Programming*, Vol. 17, No. 2–4, 1993, pp. 301–321.

[8] McCain, N. and Turner, H., "A causal theory of ramifications and qualifications," *Artificial Intelligence*, Vol. 32, 1995, pp. 57–95.

[9] Turner, H., "Reprenting Actions in Logic Programs and Default Theories: A Situation Calculus Approach," *Journal of Logic Programming*, Vol. 31, No. 1-3, Jun 1997, pp. 245–298.

[10] Calimeri, F., Dell'Armi, T., Eiter, T., Faber, W., Gottlob, G., Ianni, G., Ielpa, G., Koch, C., Leone, N., Perri, S., Pfeifer, G., and Polleres, A., "The DLV System," *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, edited by S. Flesca and G. Ianni, Sep 2002.

[11] Cholewinski, P., Marek, V. W., and Truszczynski, M., "Default Reasoning System DeReS," *International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, 1996, pp. 518–528.

[12] Niemela, I. and Simons, P., "Smodels - an implementation of the stable model and well-founded semantics for normal logic programs," *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, Vol. 1265 of *Lecture Notes in Artificial Intelligence (LNCS)*, 1997, pp. 420–429.

[13] Dimopoulos, Y., Koehler, J., and Nebel, B., "Encoding planning problems in nonmonotonic logic programs," *Proceedings of the 4th European Conference on Planning*, Vol. 1348 of *Lecture Notes in Artificial Intelligence (LNCS)*, 1997, pp. 169–181.

[14] Lifschitz, V., *Action Languages, Answer Sets, and Planning*, The Logic Programming Paradigm: a 25-Year Perspective, Springer Verlag, Berlin, 1999, pp. 357–373.

[15] Niemela, I. and Simons, P., *Extending the Smodels System with Cardinality and Weight Constraints*, Logic-Based Artificial Intelligence, Kluwer Academic Publishers, 2000, pp. 491–521.

[16] Gelfond, M. and Lifschitz, V., "Action languages," *Electronic Transactions on AI*, Vol. 3, No. 16, 1998, pp. 193–210.

[17] Gabaldon, A. and Gelfond, M., "From Functional Specifications to Logic Programs," *Proceedings of the International Logic Programming Symposium (ILPS'97)*, 1997.

[18] Balduccini, M., Gelfond, M., and Nogueira, M., "A-Prolog as a tool for declarative programming," *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000)*, 2000, pp. 63–72.

[19] Kohavi, Z., *Switching and Finite Automata Theory*, McGraw–Hill CS Series, 1978.

[20] Micheli, G. D., *Synthesis and Optimization of Digital Circuits*, McGraw–Hill Series in Electrical and Computer Engineering, 1994.

[21] Balduccini, M. and Gelfond, M., "Diagnostic reasoning with A-Prolog," *Journal of Theory and Practice of Logic Programming (TPLP)*, Vol. 3, No. 4–5, Jul 2003, pp. 425–461.

[22] Erdem, E. and Lifschitz, V., "Transitive closure, answer sets and predicate completion," *Working Notes of AAAI Spring Symposium*, 2001, pp. 60–65.

[23] McCain, N., *Causality in commonsense reasoning about actions*, Ph.D. thesis, University of Texas, 1997.

[24] Bacchus, F. and Kabanza, F., "Planning for Temporally Extended Goals," *Annals of Mathematics and Artificial Intelligence*, Vol. 22, No. 1-2, 1998, pp. 5–27.

[25] Finzi, A., Pirri, F., and Reiter, R., "Open World Planning in the Situation Calculus," *Proceedings of the 17th National Conference of Artificial Intelligence (AAAI'00)*, 2000, pp. 754–760.

[26] Huang, Y., Kautz, H., and Selman, B., "Control Knowledge in Planning: Benefits and Tradeoffs," *Proceedings of the 16th National Conference of Artificial Intelligence (AAAI'99)*, 1999, pp. 511–517.

[27] Kautz, H. and Selman, B., "The Role of Domain-Specific Knowledge in the Planning as Satisfiability Framework," *Proceedings of AIPS'98*, 1998.