An A-Prolog decision support system for the Space Shuttle

M. Nogueira
Department of Computer Science
The University of Texas at El Paso
monica@cs.utep.edu

M. Balduccini, M. Gelfond, R. Watson
Department of Computer Science
Texas Tech University
{*balduccini, mgelfond, rwatson*}@*cs.ttu.edu*

M. Barry
Advanced Technology Development Lab
United Space Alliance
Matthew.R.Barry@USAHQ.UnitedSpaceAlliance.com

Abstract

The goal of this paper is to test if a programming methodology based on the declarative language A-Prolog and the systems for computing answer sets of such programs, can be successfully applied to the development of medium size knowledge-intensive applications. We report on a successful design and development of such a system controlling some of the functions of the Space Shuttle.
Keywords: answer set programming, logic programming, planning.

1 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 [12, 13] and on formalization of common-sense reasoning [25, 23] led to the development of the declarative language A-Prolog, used in this paper to encode the domain knowledge, and to an A-Prolog based methodology for representing defaults. Insights on the nature of causality and its relationship with answer sets of logic programs [14, 21, 26] 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 [24, 6, 5] implementing these algorithms. Last, but not least, we build on earlier work on applications of answer set programming to planning [8, 20].

The *goal of this paper is to test if these methodologies, algorithms, and systems can be successfully applied to the development of medium size knowledge-intensive applications*. We build on previous work [27, 4, 16] in which the authors developed a prototype of a system, M_0, capable of checking correctness of plans and finding plans for the operation of the Reaction Control System (RCS) of the Space Shuttle. 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.

The RCS is computer controlled during takeoff and landing. While in orbit, however, astronauts have the primary control. When an orbital maneuver is required, the astronauts must perform whatever actions are necessary to prepare the RCS. These actions generally require flipping switches, which are used to open or close valves or to energize the proper circuitry. In more extreme circumstances, such as a faulty switch, the astronauts communicate the problem to the ground flight controllers, who will come up with a sequence of computer commands to perform the desired task and will instruct the shuttle's computer to execute them.

During normal shuttle operations, there are pre-scripted plans that tell the astronauts what should be done to achieve certain goals. The situation changes when there are failures in the system. The number of possible sets of failures is too large to pre-plan for all of them. Continued correct operation of the RCS in such circumstances is necessary to allow for the completion of the mission and to help ensure the safety of the crew. An intelligent system to verify and generate plans would be helpful. It is within this context that this work fits.

The system presented here, as well as the previous M_0, can be viewed as a part of a decision support system for shuttle flight controllers.

In this work we expand M_0 to produce a substantially more detailed model of the RCS. In particular, we

1. substantially simplify the model of the part of the RCS represented by M_0 without loss of detail,
2. include information about electrical circuits of the RCS, which was missing in M_0,
3. include a new type of action – computer commands controlling the position of valves,

4. include a planning module(s) containing a large amount of heuristic information (this substantially improves quality of the plans and efficiency of the search),
5. include a Java interface to simplify the use of the system by a flight controller and by the system designers.

The resulting system, M, seems to be suitable for practical applications. The work on its deployment at United Space Alliance is scheduled to start in December of the year 2000.

To understand the functionality of M let us imagine a shuttle 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. M can serve as a tool facilitating 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. The system can also be used to automatically find such a plan. In the next section we give a brief introduction into the design of the system.

2 System's Design

The system, M, consists of a collection of largely independent modules, represented by lp-functions[1], and a graphical Java interface, J. 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. 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 goal, G, and finding a plan for G of a length not exceeding some number of steps, N. Based on this information, J verifies if 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 a reasoning system for computing stable models (In M this role is currently played by SMODELS[2], however we also plan to investigate performance of other systems.) 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 generate a set of possible plans the user is interested in and the correctness theorem guarantees that there is a one-to-one correspondence between the plans and the set of stable models of the program. 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 the rest of this section we give a slightly more detailed description of particular modules.

[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.

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

2.1 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, 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.

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.

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 standard Prolog is problematic, since the corresponding graph can contain cycles. (This fact is partially responsible for the relative complexity of this module in M_0.) The ability of A-Prolog to express and to reason with recursion allows us to use the following (slightly simplified) 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 Plumbing Module consists of approximately 40 rules.

2.2 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.

2.2.1 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 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 $\mathcal{B}$ [15], 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 [11]. (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*[3]) 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.

2.2.2 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.

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 [2, 3], 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.

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

While at the basic VCM 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).
```

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.

2.3 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, 7]. 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 allows 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.

2.4 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 M allows for the creation of a variety of such modules.

The structure of the Planning Module (PlM) follows the generate and test approach described in [8, 20]. 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. 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. Notice that the head of this rule has the form $L\{p(\bar{X}) : q(\bar{X})\}U$. It defines a subset $p \subseteq q$ of terms such that $L \leq |p| \leq U$. Normally, there are many possible sets satisfying these conditions. Hence, a program containing this type of rules has multiple answer sets, corresponding to possible choices of p.

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).
```

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. For instance, finding a plan of 6 actions takes 2.14 seconds (Ex3 from Table 1), as opposed to hours required when the representation of the RCS is not partitioned in subsystems. Notice that, since there are some dependencies between some subsystems, a very small number of extremely rare (and undesirable) plans can be missed. It's possible to modify the Planning module in order to find these plans, but this issue was not investigated in this paper.

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 for 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 improvement offered by domain-dependent heuristics has not been studied mathematically here. 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.

Table 1 presents a summary of five experiments, showing the growth of the time required, in response to an increase of the complexity of the task. The columns of the table indicate: task name; number of *RCSs* subsystems involved in the task; number of *steps* required to reach the goal; total number of *actions* required to achieve the goal (actions of different subsystems may be executed in parallel); number of *faults* affecting the RCS; time needed to *check* a plan; time needed to find a *plan*.

Table 2 reports the timings obtained for variants of the previous experiments, where faults have been introduced in circuits of the system.

All times are expressed in seconds and were taken on a Pentium II 450MHz system, running NetBSD 1.4.1, LPARSE 0.99.59 and SMODELS 2.26.

Task	RCSs	steps	actions	faults	check	plan
Ex1	1	4	4	2	0.82	2.92
Ex2	1	5	5	2	1.01	4.74
Ex3	2	3	6	0	0.66	2.14
Ex4	2	4	8	2	0.82	5.90
Ex5	3	7	21	8	1.37	43.52

Table 1: Results of plan checking and planning on sample tasks without malfunctioning circuits.

Task	check	plan
Ex1	1.11	3.48
Ex2	1.35	5.55
Ex3	0.88	2.41
Ex4	1.11	6.45
Ex5	1.87	45.80

Table 2: Results of plan checking and planning on sample tasks with malfunctioning circuits.

3 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 shuttle flight controllers. It is expected that deployment of this system, for use in the space program, will begin in December of 2000. The system, while based on previous work, represents a substantial advance over its predecessor.

From the scientific standpoint, this work can be of interest to two groups of people, those interested in answer set programming and those interested in planning. We hope both groups will be glad to learn about the existence of a comparatively big and practical software system written in A-Prolog.

The former group can also learn about 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[4]. Such proof is especially important due to the critical nature of the RCS.

The people from planning may find it interesting to see a system of substantial size built on theory of actions and change. In particular, we were somewhat surprised by the importance of static causal laws in our model. We are not sure that 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 [9], where the authors use CCALC ([22]) 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 [1, 18, 17, 10]. 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, it may be interesting to see how modularity allows planning to be performed in different levels. It is easy, for instance, to modify our planning module to search for manual plans, i.e., those not including computer commands. The new planner will be much more efficient and, in many cases, sufficient for the flight controllers' needs. We have plans of applying these techniques to modeling other systems of the Space Shuttle.

[4]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 [11] 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.

4 Acknowledgments

This work was partially supported by United Space Alliance under Research Grant 26-3502-21 and Contract COC6771311.

References

[1] F. Bacchus and F. Kabanza. Planning for Temporally Extended Goals. *Annals of Mathematics and Artificial Intelligence*, 22:1-2, 5-27, 1998.

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

[3] M. Balduccini, M. Gelfond and M. Nogueira. Digital Circuits in A-Prolog. *Technical Report*, University of Texas at El Paso, 2000.

[4] M. Barry and R. Watson. Reasoning about actions for spacecraft redundancy management. In *Proceedings of the 1999 IEEE Aerospace Conference*, 5:101–112, 1999.

[5] P. Cholewinski, W. Marek and M. Truszczyński. Default Reasoning System DeReS. In *International Conference on Principles of Knowledge Representation and Reasoning*, 518-528. Morgan Kauffman, 1996.

[6] S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer and F. Scarcello. The dlv system: Model generator and application frontends. In *Proceedings of the 12th Workshop on Logic Programming*, 128–137, 1997.

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

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

[9] E. Erdem and V. Lifschitz. Transitive Closure, Answer Sets and Predicate Completion. Submitted for publication, 2000.

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

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

[12] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *Proceedings of the 5th International Conference on Logic Programming*, 1070-1080, 1988.

[13] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4):365-386, 1991.

[14] M. Gelfond and V. Lifschitz. Representing Actions and Change by Logic Programs. *Journal of Logic Programming*, 17:301–323, 1993.

[15] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16), 1998.

[16] M. Gelfond, and R. Watson. On methodology for representing knowledge in dynamic domains. In *Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, 57–66, 1999.

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

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

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

[20] V. Lifschitz. Action languages, Answer Sets, and Planning. In *The Logic Programming Paradigm: a 25–Year Perspective*, 357–373. Spring-Verlag, 1999.

[21] N. McCain and H. Turner. A causal theory of ramifications and qualifications. In *Proceedings of IJCAI'95*, 1978-1984, 1995.

[22] N. McCain and H. Turner. Causal theories of action and change. In *14th National Conference of Artificial Intelligence (AAAI'97)*, 460–465, 1997.

[23] R. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75-94, 1985.

[24] I. Niemelä, and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, 420–429, 1997.

[25] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81-132, 1980.

[26] H. Turner. Representing actions in logic programs and default theories: A situation calculus approach. *Journal of Logic Programming*, Vol. 31, No. 1-3, 245-298, 1997.

[27] R. Watson. An application of action theory to the space shuttle. *Lecture Notes in Computer Science - Procs of Practical Aspects of Declarative Languages '99*, 1551:290–304, 1999.