The USA-Advisor: A Case Study in Answer Set Planning

Marcello Balduccini, Michael Gelfond, Richard Watson[1], and Monica Nogueira[2]

[1] Department of Computer Science
Texas Tech University
Lubbock, TX 79409, USA
{balduccini,mgelfond,rwatson}@cs.ttu.edu
[2] Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968, USA
monica@cs.utep.edu

Abstract. The idea of using control knowledge to improve planning has been widely advocated. In this work we show how such knowledge was used to improve planning in the USA-Advisor decision support system for the Space Shuttle. The USA-Advisor is a medium size, real-world planning application for use by NASA flight controllers. This system contains over a dozen domain dependent and domain independent heuristics. A number of experimental results are presented here, illustrating how this control knowledge helps improve both the quality of plans as well as overall system performance.

1 Introduction

This paper is a report on the development of a medium size, real-world application, the USA-Advisor[1] - a decision support system for the Space Shuttle flight controllers.

Our goals in creating the USA-Advisor were two-fold. From a scientific standpoint the goals were to test if the rapidly developing answer set programming methodologies, algorithms, and systems could be successfully applied to the creation of medium size, knowledge intensive applications. From the standpoint of engineering, the goal was to design a system to help flight controllers plan for correct operations of the shuttle in situations where multiple failures have occurred. While the methods used in this work are general enough to model any of the subsystems of the shuttle, for our initial prototype we modeled the Reaction Control System (RCS).

[1] The USA-Advisor was created with the support of, United Space Alliance under Research Grant 26-3502-21 and Contract COC6771311. The authors would like to thank Matt Barry of the USA Advanced Technology Development Group for his technical support.

The project consisted of two largely independent parts: modeling of the RCS and development of a planner for the RCS domain. In this paper we mainly concentrate on the latter. More details of the modeling of the RCS can be found in [3].

In section 2 of the paper, the Reaction Control System of the Space Shuttle is discussed. Section 3 gives a general description of the USA-Advisor system. In section 4 we describe the basic version of the planner. Section 5 explains how the the basic planner was extended using control knowledge. Section 6 gives an overview of the results obtained in our experiments. Conclusions are given in section 7.

2 The Reaction Control System

The RCS is the system used to maneuver the Space Shuttle while it is in orbit. It consists of jets, fuel tanks, pipes, and valves used to deliver fuel to the jets, and the associated circuitry required to control the system.

The RCS is divided into three subsystems: the forward RCS, the left RCS, and the right RCS. 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 valves. Valves are opened and closed by either having an astronaut flip a switch or by instructing the computer to issue special commands. In a very simplified form, the RCS can be viewed as the directed graph in figure 1 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.

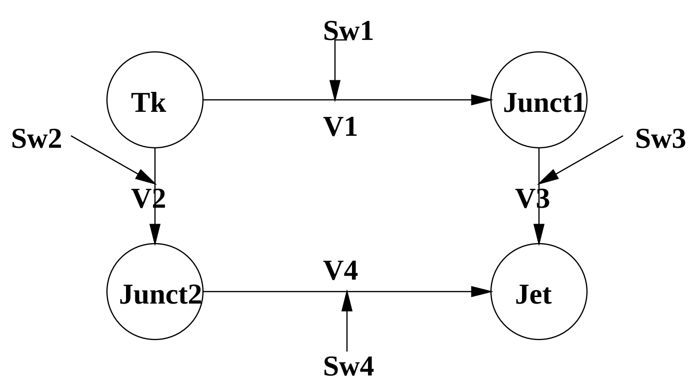

Fig. 1. A simplified view of the RCS

When everything is operating correctly, there are pre-scripted plans for each maneuver. When some components of the system fail, the situation becomes more difficult. There are many simple, single failures that plans have been created for, but in general it is impossible to create plans for every possible situation. Continued correct operation of the RCS in such circumstances is necessary to allow for the completion of the mission and to ensure the safety of the crew. An intelligent system to assist in the verification and generation of plans would be helpful. It is within this context that the USA-Advisor fits.

3 The USA-Advisor

The USA-Advisor consists of a collection of largely independent modules, represented by lp-functions[2], 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 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[3] program, Π, and passes Π as an input to a reasoning system for computing stable models (In the USA-Advisor this role is currently played by SMODELS[4], 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 and a 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.

The modules in this system are the plumbing module, the basic and extended valve control modules, the circuit theory module, and a planning module. The plumbing module models the plumbing system of the RCS, which connects fuel tanks to the jets. The basic valve control module describes the effects of manipulating switches and issuing computer commands on the valves under the assumption that the corresponding circuits are not faulty. Otherwise these effects are described by the extended valve control module. The circuit theory describes the flow of signals through the circuits.

A planning model of the RCS consists of a collection of fluents and actions, and a transition diagram defining the effects of these actions. Since various fluents of the RCS are highly interrelated, defining this diagram becomes a non-trivial

[2] 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.

[3] The language of logic programs under the answer set semantics.

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

task. We solved this problem by using the techniques developed in theory of actions and change and the power of A-Prolog rules. To illustrate our approach let us consider a single action,

$flip(S, P)$

which flips switch S in position P and three fluents:

$pressurized\_by(N, Tnk)$ - node N is pressurized by a tank Tnk;

$in\_state(V, P)$ - valve V is in valve position P;

$in\_state(S, P)$ - switch S is in switch position P.

The effects of action $flip(Sw, P)$ on fluents of the form in_state are defined by causal laws expressed by A-Prolog rules:

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

h(in_state(V,S),T) :-
            controls(Sw,V),
            h(in_state(Sw,S),T),
            not eq(S,auto),
            not stuck(V),
            not bad_circuitry(V).
```

The first rule states that if an action is performed to flip a switch, Sw, to a position, S, at time T, then, as long as the switch is not stuck in its current position, it will be in the new position at the next moment of time. This is a typical dynamic causal law expressing causal relationship between actions and fluents. In the second rule, the position, S, of a valve, V, which is controlled by a switch, Sw, is caused to be in the same position as the switch as long as the valve is not stuck and the circuitry controlling the valve had no faults. If the circuitry had faults, the position of the valve would be determined by the extended valve control module in accordance with the behavior of the damaged circuit. This law expresses causal relationship between fluents and is more difficult to express in traditional planning languages. Defining effects of the flipping action on the fluent *pressurized* is even more involved. It requires a recursive rule

```
h(pressurized_by(N1,Tnk),T) :-
            not tank(N1),
            link(N2,N1,V),
            h(in_state(V,open),T),
            h(pressurized_by(N2,Tnk),T).
```

The rule states that if there is a link between nodes $N1$ and $N2$ labeled by valve V in the graph describing the structure, the valve is open, and node $N2$ is being pressurized by a tank, Tnk, then node $N1$ is also being pressurized by the same tank.

The complete program, T, describing the transition diagram of our system contains about 130 rules. Other rules are used to describe the initial state of the system including position of valves, the list of faulty components, etc.

4 The Basic Planner

In this section we will give a brief description of the Basic Planning Module of the USA-Advisor. 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 the USA-Advisor allows for the creation of a variety of such modules.

The structure of the Basic Planning Module described in this section follows the generate and test approach from [10, 17]. The following rules form the heart of the planner. The first rule states that, for each moment of time from a given finite interval, if the goal has not been reached for one of the RCS subsystems, then an action should occur at that time.

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

The second rule states that the overall goal has been reached if the goal has been reached on each subsystem at some time.

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

:- not goal.
```

Finally, the last rule above is a constraint that states that for a model to exist, the overall goal must be achieved.

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. We refer to the Basic Planner expanded by such heuristics as Smart Planner.

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 in the framework of logic programming has been investigated in [1, 10, 16, 15, 2, 6]. Such knowledge can be classified into two categories: domain dependent and domain independent knowledge. 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 undesired plans.

Some of the control knowledge used in the USA-Advisor could 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, such as the RCS, 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),
   time(T1),
   time(T2),
   neq(T1,T2),
   occurs(A,T1),
   occurs(A,T2).
```

Next consider the following statement: "Do not perform two different types of actions which achieve the same effect." While the general idea expressed in this statement is similar to the one above, the encoding is quite different:

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

This is due to the fact that in the RCS domain, the only actions which have the same effects are those of using either a switch or a computer command to change the position of a valve. It is much easier to encode the domain specific instance of the general rule than to write the general rule itself. However we found that the understanding of the general nature of this heuristic makes the encoding much easier.

There are a number of domain specific heuristics in the USA-Advisor. 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.

```
:- next(T,T1),
   occurs(flip(Sw,gpc),T),
   controls(Sw,V),
   not issued_commands(V,T1).
```

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.

The next rule, which is the only one we show here which does not directly discuss the performance of an action, states that it is not allowed for a valve to be open if there is no pressure above it unless it is stuck.

```
:- time(T),
   link(N1,N2,V),
   h(in_state(V,open),T),
   not h(pressurized(N1),T),
   not stuck(V),
   not h(in_state(V,open),0).
```

The reason for this rule is not a physical requirement but rather a preference on types of plans.

6 Experiments

In this section we give an overview of our experiments with the two planners used by the USA-Advisor. We used a 933 Mhz Pentium III computer with 128 MB of RAM, running the NetBSD 1.5 Operating System; SMODELS version 2.26 with input from Lparse version 1.0.2 were used to find the plans.

By a *test instance* we mean a collection of system faults together with a maneuver to be performed by the shuttle. In the first series of experiments we:

(a) randomly generated a collection of test instances with a given number of mechanical and electrical faults
(b) run the basic and the smart planners in a loop with *lasttime* ranging from 3 to 10. The duration of each iteration of the loop was limited to 10 minutes of time.

Overall, about 500 test instances were generated in this manner. Figure 2 shows the performance of both planners for 60 instances containing three mechanical and two electrical faults (the most interesting situation from the standpoint of the USA experts). As we can see the Smart Planner was able to find the plans or discover their absence in less than 22 seconds. The Basic Planner required substantially more time. In some cases the difference exceeded 2 orders of magnitude. At average the Smart Planner was about 10 times faster. We were surprised to discover that the number of steps used by both planners did not exceed 5 and that the size of the grounded version of our program was not large. Other random experiments run on tests with numbers of faults between 3 and 8 did not produce any new insights.

The plans produced by the Smart Planner were of reasonably good quality. They were minimal in the number of steps and satisfied many requirements of the USA experts which were incorporated in heuristics of the planner. The unnecessary actions sometimes produced by the planner were easily detectable. The basic planner did substantially worse. In fact we discovered that only one plan produced by this planner in the experiment from figure 2 satisfies the USA experts criteria for a reasonable plan.

The second series of experiments dealt with our deliberate attempt to crash our system. We selected a number of test instances which seemed to correspond to especially difficult situations. The table 1 gives outcomes of running the Smart Planner on 6 of such instances. Even though the size of the grounded program, the length of plans, and the number of actions involved are substantially larger than those in the initial experiments, the time is still quite acceptable (i.e. less than 15 minutes according to the USA requirements). In contrast, the basic planner was not able to find solutions to any of these problems - we stopped the planner after 24 hours of work.

It is interesting to note that to achieve this performance we need all of the Smart Planner heuristics. Even though removal of some of them gave us small improvements on a few test instances, on others the performance was worsened by more than an order of magnitude.

Table 1. Hard test cases run with Smart Planner

Inst	RCSs	Steps	Actions	Rules	Atoms	Time
1	3	7	20	129950	33147	24.030
2	3	7	20	130105	33143	34.710
3	3	8	23	156500	37215	71.870
4	3	8	24	156463	37214	52.110
5	3	8	23	139047	29138	81.110
6	3	8	24	156437	37215	88.200

In the table, the first column is the test instance number, the second gives the number of RCS subsystems involved in the maneuver (1, 2, or 3), the third is the number of time steps needed, the fourth is the total number of actions performed during the time steps, the fifth and sixth are the number of rules and atoms used by SMODELS in the grounded code for that test case, and the seventh column is the time, in seconds, needed to find a plan.

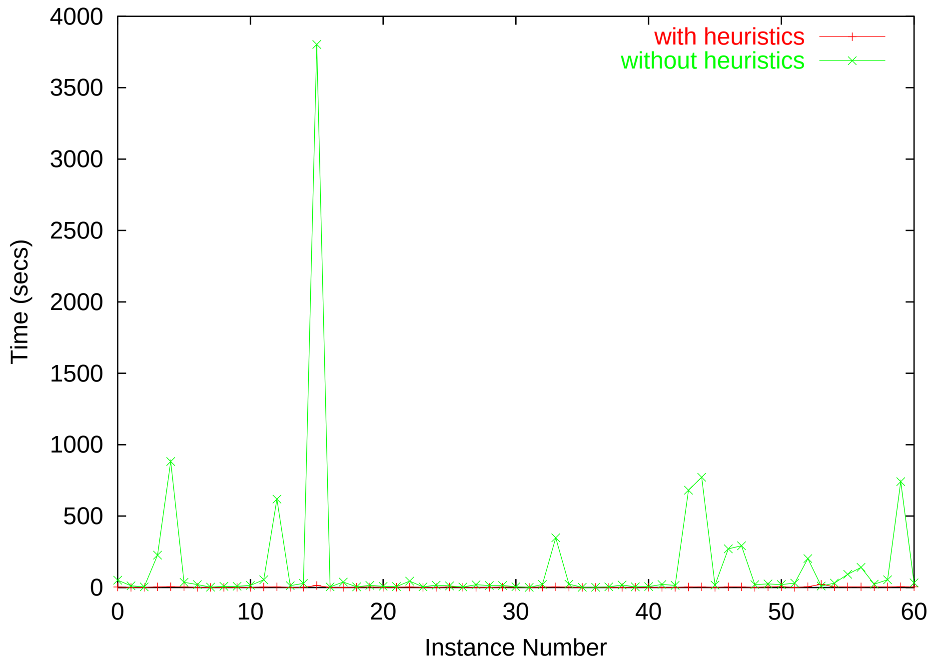

Fig. 2. Times for randomly generated tests, Basic vs. Smart Planner

7 Conclusion

In this paper we described the planner used by a medium size decision support system[5] written in A-Prolog. The domain of the planner and its construction can be of interest to the reader from several different standpoints.

- 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. We solved this problem by using the techniques developed in theory of actions and change and the power of A-Prolog rules. It is not clear to us how these effects could be accurately represented by more traditional STRIPS like action languages.
- 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.
- Answer set planning proved to be a good tool for our purpose. Partly this is due to non-numerical nature of the problem. But the planner's ability to

[5] The code for the USA-Advisor is available on request from the authors.

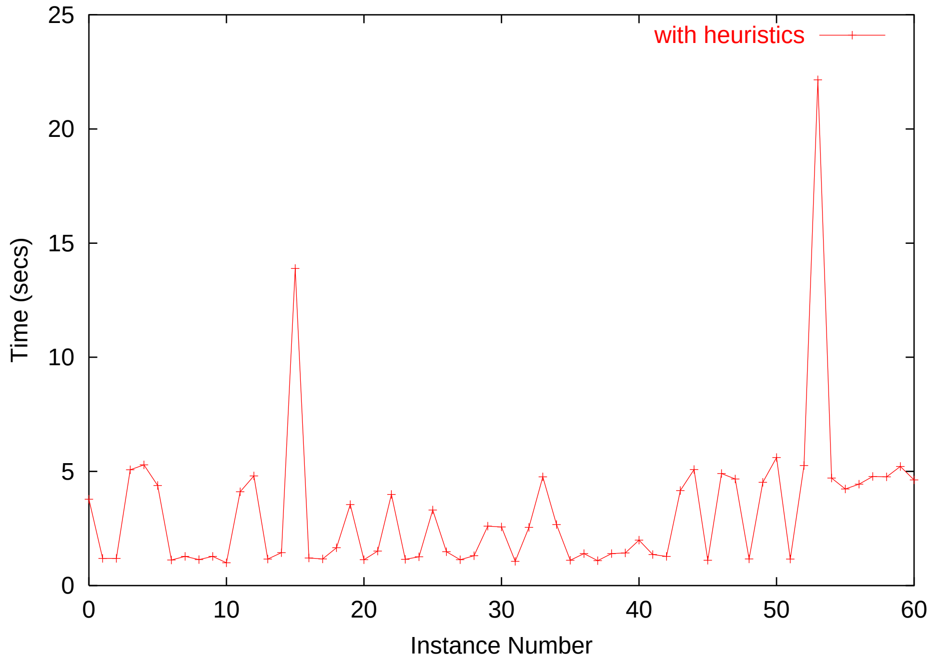

Fig. 3. Times for randomly generated tests, Smart Planner only

mix parallel and sequential plans and to efficiently search for them are the key ingredients in the success of the project.

- The heuristics used in the Smart Planner were easy to encode and to use. Our experiments show that they significantly improve both, quality of plans and efficiency of search.
- It was interesting to notice that many fluents of the RCS domain had natural recursive definitions, easily expressible in A-Prolog. This simplified the representation but precluded the immediate use of CCALC[18] style planning with satisfiability solvers. It will be interesting to see if such solvers could be used after some modifications of the representation. It is probably also worth mentioning that non-monotonicity of A-Prolog played an important role in the formalization of the domain, e.g. in specifying the inertia axiom, closed world assumptions used for describing the initial situation, and other typical default knowledge.

References

1. Baccus, F. and Kabanza, F.: Using Temporal Logic to Control Search in a Forward Chaining Planner. In *New Dircetions in Planning*, M. Ghallab and A. Milano (Eds.), IOS Press, (1996), 141–153

2. Baccus, F. and Kabanza, F.: Using Temporal Logics to Express Search Control Knowledge for Planning, *Artificial Intelligence*, (2000), Vol. 16, 123–191
3. Balduccini, M., Barry, M., Gelfond, M., Nogueira, M., and Watson, R.: An A-Prolog decision support system for the Space Shuttle. *Lecture Notes in Computer Science - Procs of Practical Aspects of Declarative Languages '01*, (2001), 1990:169–183
4. Balduccini, M., Gelfond, M., and Nogueira, M.: A-Prolog as a tool for declarative programming. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000)*, (2000), 63–72
5. Balduccini, M., Gelfond, M., and Nogueira, M.: Digital Circuits in A-Prolog. *Technical Report*, University of Texas at El Paso, (2000)
6. Baral, C., Tuan, L.: Effect of knowledge representation on model based planning: experiments using logic programming encodings. In *Proc. of 2001 AAAI Spring Symposium on Answer Set Programming*, (2001), 110–115
7. Barry, M. and Watson, R.: Reasoning about actions for spacecraft redundancy management. In *Proceedings of the 1999 IEEE Aerospace Conference*, (1999), 5:101–112
8. Cholewinski, P., Marek, W., and Truszczyński, M.: Default Reasoning System DeReS. In *International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kauffman, (1996), 518–528
9. Citrigno, S., Eiter, T., Faber, W., Gottlob, G., Koch, C., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F.: The dlv system: Model generator and application frontends. In *Proc. of the 12th Workshop on Logic Programming*, (1997), 128–137
10. Dimopoulos, Y., Nebel, B., and Koehler, J.: Encoding planning problems in non-monotonic logic programs. *Lecture Notes in Artificial Intelligence - Recent Advances in AI Planning, Proceedings of the 4th European Conference on Planning, ECP'97*, (1997), 1348:169–18
11. Gelfond, M. and Lifschitz, V.: The Stable Model Semantics for Logic Programs. In *Proceedings of the 5th International Conference on Logic Programming*, (1988), 1070–1080
12. Gelfond, M. and Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, (1991), 9(3/4):365–386
13. Gelfond, M. and Lifschitz, V.: Representing Actions and Change by Logic Programs. *Journal of Logic Programming*, (1993), 17:301–323
14. Gelfond, M. and Watson, R.: 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*, (1999), 57–66
15. Huang, Y., Kautz, H., and Selman, B.: Control Knowledge in Planning: Benefits and Tradeoffs. In *Proceedings of AAAI-99*, (1999)
16. Kautz, H. and Selman, B.: The Role of Domain-Specific Axioms in the Planning as Satisfiability Framework. In *Proceedings of AIPS'98*, (1998), 181–189
17. Lifschitz, V.: Action languages, Answer Sets, and Planning. In *The Logic Programming Paradigm: a 25–Year Perspective*, Spring-Verlag, (1999), 357–373
18. McCain, N. and Turner, H.: A causal theory of ramifications and qualifications. In *Proceedings of IJCAI'95*, (1995), 1978–1984
19. Moore, R.: Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, (1985), 25(1):75–94
20. Niemelä, I. and Simons, P.: 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*, (1997), 420–429
21. Reiter, R.: A logic for default reasoning. *Artificial Intelligence*, (1980), 13(1,2):81–132

22. Turner, H.: Representing actions in logic programs and default theories: A situation calculus approach. *Journal of Logic Programming*, (1997), Vol. 31, No. 1-3, 245–298
23. Watson, R.: An application of action theory to the space shuttle. *Lecture Notes in Computer Science - Procs of Practical Aspects of Declarative Languages '99*, (1999), 1551:290–304