Answer Set Programming and Design of Deliberative Agents

Michael Gelfond

Department of Computer Science
Texas Tech University
Lubbock, TX 79409, USA
mgelfond@cs.ttu.edu

1 Introduction

Answer set programming (ASP) (see, for instance, [22]) is a new declarative programming paradigm suitable for solving a large range of problems related to knowledge representation and search. The paradigm is rooted in recent developments in several areas of artificial intelligence. ASP starts by encoding relevant domain knowledge as a (possibly disjunctive) logic program, Π. The connectives of this program are normally understood in accordance with the answer set (stable model) semantics [12, 13]. The corresponding language is frequently referred to as A-Prolog (or ANS-Prolog). The language's ability to express defaults, i.e. statements of the form "normally, objects of class C have property P", coupled with its natural treatment of recursion, and other useful features, often leads to a comparatively concise and clear representation of knowledge. Insights on the nature of causality and its relationship with the answer sets of logic programs [14, 21, 25] allows for the description of the effects of actions which solves the frame, ramification, and qualification problems, which for a long time have caused difficulties in modeling knowledge about dynamic domains.

In the second stage of the ASP programming process, a programming task is reduced to finding the answer sets of a logic program $\Pi \cup R$ where R is normally a simple and short program corresponding to this task. The answer sets are found with the help of programming systems [23, 9, 10, 16, 20] implementing various answer set finding algorithms.

During the last few years the answer set programming paradigm seems to have crossed the boundaries of artificial intelligence and has started to attract people in various areas of computer science. The recent book [6] contains the first comprehensive introduction to the subject.

In this talk I will discuss the use of ASP for the design and implementation of software components of deliberative agents capable of reasoning, planning and acting in a changing environment. We assume that such an agent has knowledge about its domain and about its own capabilities and goals. It constantly

1. observes the world, explains the observations, and updates its knowledge base;

2. selects an appropriate goal, G;
3. looks for a plan (sequence of actions $\mathbf{a_1}, \ldots, \mathbf{a_n}$) to achieve G;
4. executes some initial part of the plan, updates the knowledge base, and goes back to step (1).

Initially we will assume [8] that

- The agent's environment can be viewed as a transition diagram whose states are sets of fluents[1] and whose arcs are labeled by actions.
- The agent is capable of making correct observations, performing actions, and remembering the domain history.
- Normally the agent is capable of observing all relevant exogenous events occurring in its environment.

The ASP methodology of design and implementation of such agents consists in

- Using A-Prolog for the description of a transition diagram representing possible trajectories of the domain, history of the agent's observations, actions, occurrences of exogenous events, agent's goals and information about preferred or most promising actions needed to achieve these goals, etc.
- Reducing the reasoning tasks of an agent (including planning and diagnostics) to finding answer sets of programs containing this knowledge.

In this talk I illustrate the basic idea of this approach by discussing its use for the development of the USA-Advisor decision support system for the Space Shuttle. The largest part of this work was done by my former and current students Dr. Monica Nogueira, Marcello Balduccini, and Dr. Richard Watson, in close cooperation with Dr. Matt Barry from the USA Advanced Technology Development Group [1, 2, 5, 24]. From the standpoint of engineering, the goal of our project 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 prototypes we modeled the Reaction Control System (RCS). The project consisted of two largely independent parts: modeling of the RCS, and the development of a planner for the RCS domain.

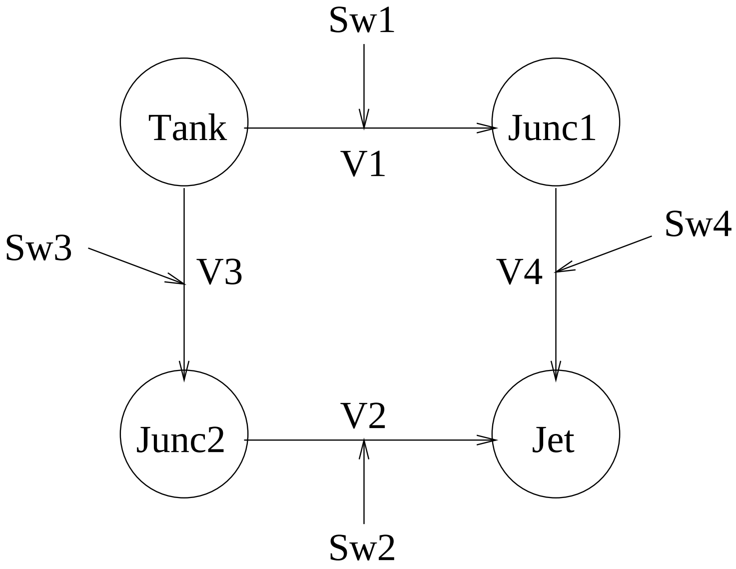
2 Modeling the RCS

The RCS is a system for maneuvering the spacecraft which consists of fuel and oxidizer tanks, valves, and other plumbing needed to deliver propellant to the maneuvering jets, electrical circuitry to control the valves, and to prepare the jets to receive firing commands. The RCS is controlled by flipping switches and issuing computer commands to open and close the corresponding valves. We are interested in a situation when a shuttle controller has the description of the RCS, and a collection of faults reported to him by the observers, e.g. switch 5

[1] fluent is a propositions whose truth values may change as a result of actions

and valve 8 are stuck in the open position, circuit 7 is malfunctioning, valve 3 is leaking, etc. The controller's goal is to find a sequence of actions (a plan), to perform a desired maneuver. Controller may use the USA-Advisor to test if the plan he came up with manually, actually satisfies the goal, and/or find a plan to achieve this goal. The USA-Advisor consists of a collection of largely independent modules, represented by A-Prolog programs. A Java interface asks the user about the history of the RCS, its faults, and the task to be performed. Based on this information, the interface selects an appropriate combination of modules and assembles them into an A-Prolog program. The program is passed as an input to a reasoning system for computing stable models (SMODELS). The desired plans are extracted from these models by the Java interface.

In its simplest form, the RCS can be viewed as a directed graph, with nodes corresponding to tanks, pipe junctions, and jets, and links labeled by valves, together with a collection of switches controlling the positions of valves.



This information can be encoded by facts describing objects of the domain and their connections:

```
tank_of(tank,fwd_rcs).
jet_of(jet,fwd_rcs).
link(tank,junc2,v3).
controls(sw3,v3).
```

A state of the RCS is given by fluents, including:
$pressurized\_by(N, Tk)$ - node N is pressurized by a tank Tk
$in\_state(V, P)$ - valve V is in valve position P
$in\_state(Sw, P)$ - switch Sw is in switch position P
A typical action is:
$flip(Sw, P)$ - flip switch Sw to position P

Our description of the corresponding transition diagram is based on McCain-Turner style theories of action and change [21, 18] and a number of results establishing the relationship between these theories and logic programs [14, 25]. In this approach the direct effects of actions are specified by dynamic causal laws:

$$holds(f, T+1) \text{ :- } holds(p, T), occurs(a, T)$$

static causal laws:

$$holds(f, T) \text{ :- } holds(p, T)$$

and impossibility conditions:

$$\text{:- } occurs(a, T), holds(p, T).$$

Below are several examples of such laws in the context of the RCS:

Direct effect of $flip(Sw, P)$ on *in_state*:

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

Indirect effects of $flip(Sw, P)$ on fluents *in_state* and *pressurized_by* are given by the following static causal laws

```
holds(in_state(V,P),T) :-
              controls(Sw,V),
              holds(in_state(Sw,P),T),
              not stuck(V),
              not bad_circuitry(V).
holds(pressurized_by(Tk,Tk),T) :-
            tank(Tk).
holds(pressurized_by(N1,Tk),T) :-
            link(N2,N1,V),
            holds(in_state(V,open),T),
            holds(pressurized_by(N2,Tk),T).
```

Note that the last law is recursive and that the effect of a single action propagates and affects several fluents.

Similar simple laws (including a simple theory of electrical circuits) constitutes a commonsense theory, EMD, of simple electro-mechanical devices. Program T consisting of EMD together with a collection, S, of atoms describing the RCS (which can be automatically generated from the RCS schematics) incorporates the controller's knowledge about the RCS.

3 The USA Planning Module

The structure of the USA Planning Module described in this section follows the generate and test approach from [11, 17]. The following rules, PM, form the heart of the planner. The first rule states that, for each moment of time from $[0, n)$ 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 :- T < n,
                                  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 :- goal(T1,left_rcs),
        goal(T2,right_rcs),
        goal(T3,fwd_rcs).
:- not goal.
```

Finally, the last rule states that failure to achieve a goal is not an option. Based on results from [25] one can show that, given a collection of faults, I, and a goal G, finding a plan for G of max length n can be reduced to finding an answer set of the program

$$T \cup I \cup PM$$

Since the RCS contains more than 200 actions, with rather complex effects, this standard approach needs to be substantially improved to provide the necessary efficiency.

To improve the efficiency of the planner, and the quality of plans, we will expand our planning module with control knowledge from the operating manual for system controllers. For instance, we include a statement "Normal valve, connecting node $N1$ with node $N2$, shall not be open if $N1$ is not pressurized" which can be expressed in A-Prolog as follows:

```
holds(pressurized(N),T) :-
             node(N), tank(Tk),
             holds(pressurized_by(N,Tk),T).
:- link(N1,N2,V),
   not stuck(V),
   holds(in_state(V,open),T),
   not holds(pressurized(N1),T).
```

After the basic planning module is expanded by about twenty such rules, planning becomes efficient, and the quality of the returned plans improves dramatically. We ran thousands of experiments in which faults were generated at random, and the planner was run in a loop with n ranging from 0 to the maximal possible length (easily available for the RCS system). In most cases the answers were found in seconds. We always were able to find plans or discover that the goal cannot be achieved in less than 20 minutes - the time limit given to us by the USA controllers.

It may be instructive to do some comparison of ASP planning with more traditional approaches.

– The ability to specify causal relations between fluents is crucial for the success of our project. 'Classical' planning languages cannot express such relations.
– We do not use any specialized planning algorithm. Finding a plan is reduced to finding an answer set of a program. This is of course similar to satisfiability planning [15]. In fact recent results [20] show that in many cases finding a stable model of a program can be reduced to finding classical models of its "propositional translation". However in [19] the authors show that any equivalent transformation from logic programs to propositional formula involves a significant increase in size. This confirms our experience that the ability of A-Prolog to represent the domain's transition diagram, its control knowledge, the complex initial situations, allows to naturally formalize domains for which propositional formalizations are far from obvious.
– The program is fully declarative. This means that if the causal laws correctly describe the system's behavior then the plans returned by our system are provenly correct. This also simplifies informal validation of our model and allows for simple modification and reuse.
– One can move from sequential to parallel planning by simply changing a constant.
– Finally, recent work on extensions of A-Prolog allowed us to declaratively specify preferences between plans. E.g. the system returns plans which use computer commands only if it is unavoidable, etc. (See [5])

4 Future Work

The discussion above shows that ASP planning works well in domains where actions have complex effects depending on a large body of knowledge, and where plans are comparatively short. If we are interested in long plans, or in plans which require non-trivial computation, current answer set solvers do not work well. This is due to the fact that all the solvers start their work by grounding the corresponding program. Even though the grounding algorithms are smart and economical and the current solvers can deal with hundreds of thousands of ground rules this is still a very serious limitation. Finding new algorithms with some kind of "grounding on demand" mechanism is a very interesting and important research problem.

Now a few words about other reasoning steps in the agent observe-think-act loop. In [3] we showed how the process of diagnostics (finding an explanation for an unexpected observation) can be reduced to finding answer sets of a logic program. In the context of the RCS, this program consists of the controller's knowledge T, observations on the values of fluents, and a diagnostics module DM similar to PM. The program efficiently finds collections of faults (a stuck valve, a malfunctioning electrical circuit, etc) which explain the unexpected observation. It is important to notice that the same knowledge, T, is used for diagnostics and for planning. In other knowledge representation languages this is usually not the case - each task requires its own representation. It is important to make sure that the system has enough information to first return diagnoses which are most plausible or most suitable for testing, etc. To achieve this goal we need to better understand how such information can be expressed in A-Prolog and its extensions. Some of the ideas related to this task are discussed in [4, 7] but much more work is needed to make these ideas practical.

References

1. M. Balduccini, M. Barry, M. Gelfond, M. Nogueira, and R. Watson An A-Prolog decision support system for the Space Shuttle. *Lecture Notes in Computer Science - Proceedings of Practical Aspects of Declarative Languages'01*, (2001), 1990:169–183
2. M. Balduccini,M. Gelfond, M. Nogueira, and R. Watson Planning with the USA-Advisor. In 3rd International NASA Workshop on Planning and Scheduling for Space, Sep 2002.
3. M. Balduccini and M. Gelfond Diagnostic reasoning with A-Prolog. Theory and Practice of Logic Programming, 3(4-5):425-461, Jul 2003.
4. M. Balduccini and V. Mellarkod A-Prolog with CR-Rules and Ordered Disjunction. In ICISIP'04, pages 1-6, Jan 2004.
5. M. Balduccini USA-Smart: Improving the Quality of Plans in Answer Set Planning. In PADL'04, Lecture Notes in Artificial Intelligence (LNCS), Jun 2004.
6. C. Baral. Knowledge representation, reasoning and declarative problem solving. Cambridge University Press, 2003.
7. C. Baral, M. Gelfond and N. Rushton. Probabilistic reasoning with answer sets. The Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning, pp 21-33, 2004.
8. C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In J Minker, editor, *Logic Based AI.* pp. 257–279, Kluwer, 2000.
9. 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.
10. 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.
11. Dimopoulos, Y., Nebel, B., and Koehler, J.: 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*, (1997), 1348:169–18
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. H. Kautz and B. Selman. Planning as Satisfiability. In Proc. of ECAI-92, pp. 359-363, 1992
16. Yu. Lierler and M. Maratea Cmodels-2: SAT-based Answer Sets Solver Enhanced to Non-tight Programs, In Proc. of LPNMR-7, 2004.

17. V. Lifschitz Answer set programming and plan generation Artificial Intelligence, Vol. 138, 2002, pp. 39-54.
18. V. Lifschitz and H. Turner, Representing transition systems by logic programs. In Proc. of the Fifth Int'l Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'99), pp. 92-106, 1999.
19. V. Lifschitz and A. Razborov Why are there so many loop formulas? ACM Transactions on Computational Logic, to appear.
20. F. Lin and Yu. Zhao ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers In Proc. of AAAI-02, pp. 112-117, 2002.
21. N. McCain and H. Turner. Causal theories of action and change. In *14th National Conference of Artificial Intelligence (AAAI'97)*, 460–465, 1997.
22. W. Marek, and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, 375–398, Spring-Verlag. 1999.
23. 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.
24. M. Nogueira. Building Knowledge Systems in A-Prolog. PhD thesis, University of Texas at El Paso, May 2003.
25. 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.