1 Introduction

Dijkstra viewed programming as the process of the "refinement of specification." This applies not only to imperative programming, but in the myriad fields of Artificial Intelligence Research. It is especially made clear given that the levels of refinement we are able to obtain in our various formalisms, programs, etc. are inherently tied to the languages we use to reason about our problem domains.

This is also a central view taken by many members of what has been termed: the "Knowledge Representation and Reasoning Community." Within this community one of the central goals has been to understand and formalize the reasoning of intelligent agents. A number of languages have developed such *action languages*, and the language of *answer set prolog* (denoted throughout as A-Prolog). A rather significant body of work has been done regarding various reasoning tasks such as planning, diagnosis, prediction, counter-factual reasoning, etc., all from the perspective of them being simply one of many reasoning tasks an agent may perform. This has lead to a framework for the design and development of intelligent agents based on a *portfolio* of languages, corresponding roughly to individual levels of refinement about an agent's knowledge of the domain.

This central view does not appear to be shared by the "Classical Planning" community, given its use of a single language, *PDDL*, which was "originally developed by the AIPS-98 Competition Committee for use in defining problem domains."

This difference in first principles leads to some interesting consequences which we hope to explore to some extent in this paper. The rest of this paper is structured as follows: an overview of the classical planning approach will be given in the context of four different planning domains, The Blocks World, Towers of Hanoi, Lin's Briefcase, and an Electrical Circuit. Once this has been, a similar overview of the knowledge representation and reasoning approaches, including descriptions of the action language $\mathcal{AL}$, followed by the logic programming language of A-Prolog. Once this has been covered, representations of the aforementioned domains will be presented.

2 The Knowledge Representation and Reasoning Approach

As was stated earlier, in the knowledge representation and reasoning community, the general approach is centered around the use of a portfolio of various knowledge representation languages, among which are the action language $\mathcal{AL}$, and the logic programming language A-Prolog. In keeping with the notion of programming as the refinement of specification we:

1. Construct a mental model of the transition system described by the domain
2. Describe the transition system via an *action description* in some action language (such as $\mathcal{AL}$)
3. Refine the action description further by translation into A-Prolog

In this section we take a brief look at the languages $\mathcal{AL}$ and A-Prolog, and show their application towards the aforementioned domains.

2.1 The Action Language $\mathcal{AL}$

Simply put, action languages are a family of declarative languages designed for the purpose of describing the effects of actions. They have a simple syntax and semantics, while remaining powerful enough to represent many complex reasoning domains [3]. Collections of statements are called *action descriptions*, and define transitions whose nodes correspond to *possible states of the world*[1], and whose arcs are labeled by actions. An arc (σ, α, τ) states that if action α occurs in state σ, the resulting state will be τ. In addition to specifying what changes due to the occurrence of a particular action, it is also vital to be able to specify what does not change. This is know as the *frame problem* [4], and its solution lies in an elegant and precise representation of what is called the *inertia axiom*. The beauty and utility of such languages stems from the ability to compactly represent huge transition systems, and in their graceful approach towards solving the frame problem.

Action descriptions in the language of $\mathcal{AL}$ are collections of three statement types: *dynamic causal laws*, *static causal laws*, and *impossibility conditions*. Dynamic causal laws are statements of the form:

$$a \textbf{ causes } f \textbf{ if } l_0, \ldots, l_n$$

where a is an action, and f, and $l_0, \ldots, l_n$ are fluent literals. Laws of this form a read as: "action a causes f to be come true if $l_0, \ldots, l_n$." The corresponding arc in the transition diagram is shown in Figure 1.

Static causal laws have the form:

$$f \textbf{ if } l_0, \ldots, l_n$$

where f, and $l_0, \ldots, l_n$ are fluent literals. Such laws are read as: "f is true whenever $l_0, \ldots, l_n$ are true." Unlike dynamic causal laws, static laws are used to describe the properties of states, as well as the *indirect effects* of actions.

[1] A difference we believe to be significant from the conception of a state in the classical view as simply an interpretation of the fluents present in a domain.

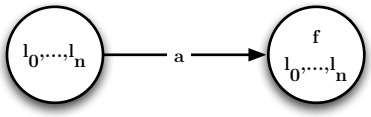

Figure 1: transition defined by a simple causal law

Example 1. Consider the following action description AD_1:

$$a \textbf{ causes } f$$
$$g \textbf{ if } f$$

and let the state, $\sigma = \{\neg f, \neg g\}$. If a is executed in σ, we have the transition depicted in Figure 2.

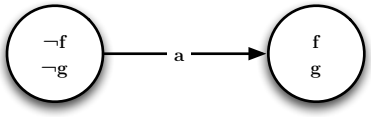

Figure 2: transition involving both direct and indirect effects

Impossibility conditions have the form:

$$\textbf{impossible } a \textbf{ if } l_0, \ldots, l_n$$

where as before, a is an action, and $l_0, \ldots, l_n$ are fluent literals. Rules such as this are used to state that: "action a may not occur in a state which satisfies $l_0, \ldots, l_n$." From the view of a transition system, this means that there are no outgoing arcs labeled by a originating from a state which satisfies $l_0, \ldots, l_n$.

The semantics of an action description of $\mathcal{AL}$ is given by its transition diagram. For a more detailed treatment, we refer the reader to [3]. Having described the action language $\mathcal{AL}$, we now turn our attention to the language of A-Prolog.

2.2 The Language of A-Prolog

The language of A-Prolog is a logic programming language developed by Michael Gelfond and Vladimir Lifschitz in [1]. It has a simple syntax, and is focused on modeling the beliefs of a rational agent. A rather broad community has developed around it, and it has emerged as strong programming paradigm in its own right.

A logic program in the language of A-Prolog is defined as a collection of rules of the form [1, 2]:

$$l_0 \leftarrow l_1, \ldots, l_m, \mathtt{not}\ l_{m+1}, \ldots, \mathtt{not}\ l_n.$$

where $0 \leq m \leq n$, each l_i is a literal, and $\mathtt{not}$ represents *negation-as-failure* (also termed *default negation*). Rules of this form are read as: "if an agent believes $l_1, \ldots, l_m$ to be true and has no reason to believe in $l_{m+1}, \ldots, l_n$, then he believes l_0 to be true."

The left-hand side of a rule is known as the *head*, and the right-hand side is called the *body*. Rules with empty heads are known as *constraints*, while those with empty bodies are termed *facts*.

The variables of a program Π range over the set of ground terms of its signature σ. Programs which do not contain any variables are said to be *ground*. A rule r which does contain variables may be viewed as a shorthand form for the entire set of its ground instantiations. A *definite rule* is a rule which does not contain any occurrences of default negation, and programs comprised solely of definite rules are called *definite programs*.

Let S be a set of ground literals. The body of a rule is satisfied by S if $\{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$ and $\{l_1, \ldots, l_m\} \subseteq S$. A rule with a non-empty head is satisfied by S if either its body is not satisfied by S or $l_0 \in S$. A constraint is satisfied by S if its body is not satisfied by S.

Given an arbitrary program Π, and a set of ground literals, S, the *reduct* of Π with respect to S, denoted by Π^S is the definite program obtained from Π by:

1. deleting all rules in Π which $\mathtt{not}\ l$ in the body, where $l \in S$
2. removing all remaining occurrences of $\mathtt{not}\ l$ from the bodies of the remaining rules

A set of ground literals, S, is said to be an answer set of a logic program Π if it satisfies the following:

1. if Π is a definite program then S is a minimal set of literals satisfying all of the rules of Π
2. if Π is not a definite program the S is an answer set of Π if and only if S is an answer set of Π^S

The fact that a program can have multiple (or no) answer sets has given rise to an alternative method of solving problems through logic programming, called *Answer Set Programming* (ASP) [5, 6]. In this approach, we develop logic programs whose answer sets have a one-to-one correspondence with the solutions of the particular problem being modeled. Typically an ASP program consists of:

- rules for the enumeration of possible solutions to a problem as *candidate answer sets*
- constraints which remove candidates that do not correspond to solutions of the problem

In the context of planning, the "planning module" will serve to generate possible sequences of while the "goal specification" will prune those which fail to satisfy our goal.

2.3 Domains

In this section we examine a number of planning and apply the knowledge representation and reasoning approach towards their resolution. All of the listings presented in this paper are in the input language of the clingo answer set programming system [7].

2.3.1 Lin's Briefcase

Consider a domain consisting of a briefcase with two latches. Flipping a closed latch causes it to be open, while flipping an open latch causes it to become closed. How do we represent this domain using the reasoning about actions approach?

We begin by constructing a mental model of the system, which involves the transition system shown in Figure 3.

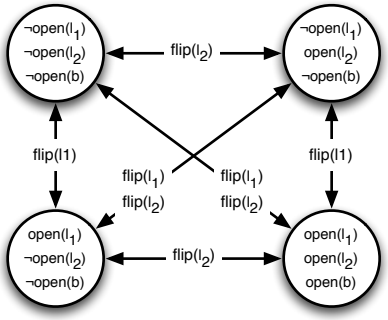

Figure 3: transition system for the Lin's Briefcase domain

Once we have the transition fixed, we represent in in $\mathcal{AL}$ as follows:

$$\begin{aligned}
&flip(l_1)\ \textbf{causes}\ open(l_1)\ \textbf{if}\ \neg open(l_1)\\
&flip(l_1)\ \textbf{causes}\ \neg open(l_1)\ \textbf{if}\ open(l_1)\\
&flip(l_2)\ \textbf{causes}\ open(l_2)\ \textbf{if}\ \neg open(l_2)\\
&flip(l_2)\ \textbf{causes}\ \neg open(l_2)\ \textbf{if}\ open(l_2)\\
&open(b)\ \textbf{if}\ open(l_1),\ open(l_2)\\
&\neg open(b)\ \textbf{if}\ \neg open(l_1)\\
&\neg open(b)\ \textbf{if}\ \neg open(l_2)
\end{aligned}$$

With our domain description in place, we translate the above laws and add some additional domain related information to obtain the following A-Prolog program:

$$language = Prolog]lstlisting$$

```
const n = 2.
time(0..n).
latch(1). latch(2).
fluent(open(Latch)) :- latch(Latch).
fluent(opened).
action(flip(Latch)) :- latch(Latch).
holds(open(Latch), T + 1) :- occurs(flip(Latch), T), -holds(open(Latch), T), latch(Latch),
time(T).
-holds(open(Latch), T + 1) :- occurs(flip(Latch), T), holds(open(Latch), T), latch(Latch),
time(T).
-holds(opened, T) :- -holds(open(Latch), T), latch(Latch), time(T).
holds(opened, T) :- not -holds(opened, T), time(T).
holds(Fluent, T + 1) :- holds(Fluent, T), not -holds(Fluent, T + 1), fluent(Fluent), time(T).
-holds(Fluent, T + 1) :- -holds(Fluent, T), not holds(Fluent, T + 1), fluent(Fluent), time(T).
-holds(open(Latch), 0) :- latch(Latch).
goal(T) :- holds(opened, T), time(T).
done :- goal(T).
:- not done.
1occurs(Action,T) : action(Action)1 :- not goal(T), time(T).
hide. show occurs(,).
```

Utilizing the clingo answer set solver we obtain the following two plans (this is due to the fact that the planning module is written in such a way as to only look for sequential plans, parallel plans may be supported through a minor modification):

```
Answer: 1
occurs(flip(2),1) occurs(flip(1),0)
Answer: 2
occurs(flip(1),1) occurs(flip(2),0)
SATISFIABLE

Models       : 2
Time         : 0.000
  Prepare    : 0.000
  Prepro.    : 0.000
  Solving    : 0.000
```

2.3.2 Electrical Circuit

We now turn our attention towards the representation of an electrical circuit domain. Such a domain is of interest due to the fact that there is essentially only a single action that an agent may perform, namely, the application of some input signals to the circuit. These signals must then be propagated throughout the entire circuit to obtain some output value.

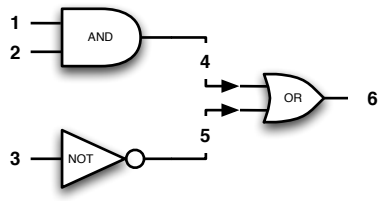

Figure 4: a simple electrical circuit

Consider the electrical circuit shown in Figure 4. Once a signal has been applied to inputs $1, 2, 3$, it must be propagated to the next level of the circuit within the same state. Static causal laws shine here.

```
% ================================
% OBJECTS
% ================================

time(0).

% ================================
% CIRCUIT STRUCTURE
% ================================

wire(1..6).
gate(1..3).
and(1).
inverter(2).
or(3).

% and gate has inputs 1 and 2

input_to(1,1..2).

% inverter has input 3

input_to(2,3).

% or gate has inputs 4 and 5

input_to(3,4..5).

% and gate has output 4

output_from(1,4).

% inverter has output 5

output_from(2,5).

% or gate has output 6

output_from(3,6).

% ================================
% GENERAL DESCRIPTION OF GATES
% ================================

h(val(OUTPUT_WIRE, 1), T) :-
    output_from(GATE, OUTPUT_WIRE),
    and(GATE),
    input_to(GATE, INPUT1),
```

```
    input_to(GATE, INPUT2),
    INPUT1 != INPUT2,
    h(val(INPUT1, 1), T),
    h(val(INPUT2, 1), T),
    time(T).

h(val(OUTPUT_WIRE, 0), T) :-
    output_from(GATE, OUTPUT_WIRE),
    and(GATE),
    input_to(GATE, INPUT),
    h(val(INPUT, 0), T),
    time(T).

h(val(OUTPUT_WIRE, 0), T) :-
    output_from(GATE, OUTPUT_WIRE),
    or(GATE),
    input_to(GATE, INPUT1),
    input_to(GATE, INPUT2),
    INPUT1 != INPUT2,
    h(val(INPUT1, 0), T),
    h(val(INPUT2, 0), T),
    time(T).

h(val(OUTPUT_WIRE, 1), T) :-
    output_from(GATE, OUTPUT_WIRE),
    or(GATE),
    input_to(GATE, INPUT),
    h(val(INPUT, 1), T),
    time(T).

h(val(OUTPUT_WIRE, 1), T) :-
    output_from(GATE, OUTPUT_WIRE),
    inverter(GATE),
    input_to(GATE, INPUT),
    h(val(INPUT, 0), T),
    time(T).

h(val(OUTPUT_WIRE, 0), T) :-
    output_from(GATE, OUTPUT_WIRE),
    inverter(GATE),
    input_to(GATE, INPUT),
    h(val(INPUT, 1), T),
    time(T).

% ================================
% INITIAL STATE
% ================================

h(val(1, 1), 0).
h(val(2, 0), 0).
h(val(3, 1), 0).

% ================================
% SOLVER DIRECTIVES
% ================================

#hide.
#show h(_,_).
```

2.3.3 Towers of Hanoi

```
% ================================
% OBJECTS
% ================================

#const n = 7.
```

```
time(0..n).

disc(1).
disc(2).
disc(3).

peg(p1).
peg(p2).
peg(p3).

location(X) :- peg(X).
location(X) :- disc(X).

% ============================
% FLUENTS
% ============================

fluent(on(Disc,Location)) :-
    disc(Disc),
    location(Location),
    Disc != Location.

fluent(clear(Location)) :-
    location(Location).

% ============================
% ACTIONS
% ============================

action(move(Disc,Source,Destination)) :-
    disc(Disc),
    location(Source),
    location(Destination),
    Source != Destination.

% ============================
% CAUSAL LAWS
% ============================

holds(on(Disc,Destination), T + 1) :-
    occurs(move(Disc,Source,Destination), T),
    disc(Disc),
    location(Source),
    location(Destination),
    time(T).

-holds(on(Disc,Source), T + 1) :-
    occurs(move(Disc,Source,Destination), T),
    disc(Disc),
    location(Source),
    location(Destination),
    time(T).

:- occurs(move(Disc,Source,Destination), T),
   not holds(on(Disc,Source), T),
   disc(Disc),
   location(Source),
   location(Destination),
   time(T).

:- occurs(move(Disc,Source,Destination), T),
   not holds(clear(Disc), T),
   disc(Disc),
   location(Source),
   location(Destination),
   time(T).
```

```
:- occurs(move(Disc,Source,Destination), T),
   not holds(clear(Destination), T),
   disc(Disc),
   location(Source),
   location(Destination),
   time(T).

:- occurs(move(Disc,Source,Destination), T),
   disc(Destination),
   Destination < Disc,
   disc(Disc),
   location(Source),
   location(Destination),
   time(T).

-holds(clear(Location), T) :-
    holds(on(Disc,Location), T),
    location(Location),
    disc(Disc),
    time(T).

holds(clear(Location), T) :-
    not -holds(clear(Location), T),
    location(Location),
    time(T).

% ==============================
% INERTIA AXIOMS
% ==============================

holds(Fluent, T + 1) :-
    holds(Fluent, T),
    not -holds(Fluent, T + 1),
    fluent(Fluent),
    time(T).

-holds(Fluent, T + 1) :-
    -holds(Fluent, T),
    not holds(Fluent, T + 1),
    fluent(Fluent),
    time(T).

% ==============================
% INITIAL STATE
% ==============================

holds(on(1,2), 0).
holds(on(2,3), 0).
holds(on(3,p1), 0).

% ==============================
% GOAL SPECIFICATION
% ==============================

goal(T) :-
    holds(on(1,2), T),
    holds(on(2,3), T),
    holds(on(3,p3), T),
    time(T).

done :-
    goal(T).

:- not done.

% ==============================
% PLANNING MODULE
% ==============================
```

```
1{occurs(Action,T) : action(Action)}1 :-
    not goal(T),
    time(T).

% ================================
% SOLVER DIRECTIVES
% ================================

#hide.
#show occurs(_,_).
```

References

[1] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth Bowen, editors, Proceedings of the Fifth International Conference on Logic Programming, pages 10701080, Cambridge, Massachusetts, 1988. The MIT Press.

[2] Chitta Baral. Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, January 2003.

[3] Marcello Balduccini. Answer Set Based Design of Highly Autonomous, Rational Agents. PhD thesis, Texas Tech University, December 2005.

[4] Murray Shanahan. Solving the frame problem: A mathematical investigation of the commonsense law of inertia. MIT Press, 1997.

[5] Victor Marek and Mirek Truszczynski. Stable models and an alternative logic programming paradigm. The Logic Programming Paradigm, pp. 375–398, Springer, 1999.

[6] Ilkka Niemela. Logic programming with stable model semantics as a constraint programming paradigm. AMAI, 25(3,4), 1999.

[7] Martin Gebser, Roland Kaminski, et. al. A User's Guide to gringo, clasp, clingo, and iclingo. 2010.