COPLAS: a COnditional PLAnner with Sensing actions

Jorge Lobo*
Department of EECS
University of Illinois at Chicago
851 S. Morgan St Chicago, IL 60607, USA
jorge@eecs.uic.edu

Abstract

This paper describes a partial order planner written in prolog. The planner handles sensing actions, thus, generating conditional plans.

Background

The language $\mathcal{A}$ of Gelfond and Lifschitz (GL93) was designed as the basis of a high level language to represent and reason about actions and their effects. Domain descriptions written in this language define state automata that describe how situations in the world change after actions defined in the domain are executed. $\mathcal{A}$ has served as a platform to study the formalization of action theories. Extensions of $\mathcal{A}$ have been developed to study and reason about the concurrent execution of actions, the non-deterministic effects of some actions and to study many instances of the qualification and ramification problems.

Baral at University of Texas at El Paso, together with his students has implemented a liner planner based on $\mathcal{A}$ (Kah96) that in many instances compares very well to standard planners such as UCPOP (BCF+95), one of the most commonly used (partial order) planners.

In (LMT97) we had proposed a new action description language called $\mathcal{A}_K$. $\mathcal{A}_K$ is a minimal extension of $\mathcal{A}$ to handle sensing actions. A sensing action is an action that does not have any effect in the world. The effect is only in the perception of the reasoning agent about the world. The execution of a sensing action will increase the agent knowledge about the current state of the world. Take for example a deactivated agent placed inside a room. The agent has duties to carry out and will be activated by a timer. Let us assume the agent is always placed facing the door. The agent, once activated, may become damaged if it attempts to leave the room since the door may be closed. Before the agent tries to leave the room it needs to perform some act of sensing in order to determine whether the door is opened or not. The agent has incomplete knowledge with respect to the door. A sensing action such as looking at the door would provide information to the agent concerning the status of the door. The major advantage of adding sensing action is that they give an agent the ability to reason about complex plans that include conditionals and iterations (Lev96). This paper describes a prototype planner systems that handles sensing actions. Similar the planner described in (Kah96), our planner works bottom-up starting from the initial situation searching for plans that reach a situation where the goal is achieve. The planner in (Kah96) is based on the logic programming translations developed for $\mathcal{A}$. Our implementation runs several orders of magnitude faster but we do not have a very clean code, and we do not have proof of correctness of any of the algorithms. We have encoded in prolog the transition functions associated with domain descriptions written in $\mathcal{A}$ or $\mathcal{A}_K$.

Funding for this project has been provided by the Research Institute for Advanced Information Technology (AITEC) of Japan.

The planner

With the support of The Research Institute for Advanced Information Technology (AITEC) of Japan we have developed a domain independent, logic programming-based planner for action domain descriptions written in a language designed under the basis of $\mathcal{A}_K$. The planner, called COPLAS, is able to generate conditional plans from domains with sensing actions. The domain can also describe actions with non-deterministic effects. An action can cause the lost of knowledge if its effect is non-deterministic. Take for example the action of tossing a coin. Assume that in the initial situation we knew that the coin was showing heads. If we toss the coin we know it will land with either heads showing or with tails showing, but exactly which cannot be predicted. Non-deterministic actions and sensing actions have opposite effects on an agent's knowledge. The planer can also handle a limited class of ramification constrains in the form of axioms and actions with universally quantified effects. *This is an experimental systems and there are many areas where it could be improved.* We comment on some of them at the end the paper.

The domain description language

COPLAS requires two pieces of information to run. A description of the domain where the planning is going to take place plus a goal. In this section we describe how domains are specified in the COPLAS language. We will refer to both the planner and the language to specify the domain as COPLAS. COPLAS has five classes of symbols: object symbols, type symbols, fluent symbols, non-sensing action symbols and sensing action symbols plus a countably infinite set of variables. Each fluent and action symbol has associated a positive integer called the arity of the symbol. A fluent is a term of the form $f(t_1, \ldots, t_n)$, where f is a fluent symbol and and each t_i is either a variable or an object symbol. Similarly, a non-sensing action is a term of the form $a(t_1, \ldots, t_n)$, where a is a non-sensing action symbol and each t_i is a variable or an object symbol. A sensing action is a term of the form $a(t_1, \ldots, t_n)$, where a is a sensing action symbol and each t_i is a variable or an object symbol.

A domain description for COPLAS consists of four sections: the declaration of types, the action propositions, the axiom set and the initial situation description.

The type declaration section is a collection of fluent declarations, non-sensing action declarations and sensing action declarations. Declarations restrict the kind of fluents and actions allowed in a domain by typing the arguments of the fluent and action symbols.

A fluent declaration is an expression of the form:

$$fluent(f(X_1, \ldots, X_n)) : -t_1(X_1), \ldots, t_n(X_n), Ieq$$

where f is a fluent symbol of arity n, the X_is are variables, the t_is type symbols, and Ieq a set of inequalities involving variables in the expression or object symbols. An inequality of two terms e_1 and e_2 is tested by the predicate $diff(e_1, e_2)$ that is true if e_1 and e_2 do not unify; otherwise it is false. A non-sensing action declaration is an expression of the form:

$$action(a(X_1, \ldots, X_n)) : -t_1(X_1), \ldots, t_n(X_n), Ieq$$

where a is a non-sensing action symbol of arity n, the X_is are variables, the t_is type symbols and Ieq a set of inequalities as above. Similarly, a sensing action declaration is an expression of the form:

$$sensing(a(X_1, \ldots, X_n)) : -t_1(X_1), \ldots, t_n(X_n), Ieq.$$

The action proposition section is a collection of deterministic and non-deterministic effect propositions and action laws plus a set of action executavility conditions.

• A deterministic effect proposition is an expression of the form:

$$causes(A, F, [P_1, \ldots, P_n])$$

where A is a non-sensing action, and F and each of the P_i is a fluent literal or equality or inequality conditions involving variables from the propositions or objects. A fluent literal is either a fluent f or its negation denoted by $neg(f)$. The intuitive reading of this proposition is: the execution of A in a situation where $P_1, \ldots, P_n$ are true causes F to be true.

• A non-deterministic effect proposition is an expression of the form:

$$affects(A, F, [P_1, \ldots, P_n])$$

where A is a non-sensing action, F is a fluent and each of the P_i is either a fluent literal, an equality or an inequality as above. The intuitive reading of this proposition is: the execution of A in a situation where $P_1, \ldots, P_n$ are true causes F to be unknown.

• A knowledge law is an expression of the form:

$$causes\_to\_know(A, F, [])$$

where A is a sensing action and F is a (non-negated) fluent. The intuitive reading of this proposition is: the execution of A causes the value of F to be known.[1].

• Executavility conditions impose restrictions on the situations where an action can be executed. For a non-sensing action A, its executavility conditions are described with propositions of the form:

$$possible(A, [P_1, \ldots, P_n])$$

where each P_i is a fluent literal. This intuitively says that A is executable in a state where $P_1, \ldots, P_n$ are true. For a sensing action A the executavility conditions are expressions of the form:

$$sensing\_possible(A, [P_1, \ldots, P_n])$$

with similar meaning than the executavility conditions of non-sensing actions.

The axiom set is a collection of terms of the form:

$$axiom(F_1, F_2)$$

where F_1 and F_2 are fluent literals. The intuitive meaning of this axiom is that in any state where F_1 is true then F_2 most be true too. The use of axioms is very limited at this point in the planner. It is assumed that the ramification effects of axioms are one step only. That is, there cannot be two ground instances (i.e. replacement of variables with object symbols) of axioms such that the ground axioms are of the form $axiom(f_1, f_2)$ and $axiom(f_2, f_3)$. The reader must note that the meaning of $axiom(f_1, f_2)$ is not equivalent to the material implication $f_1 \Rightarrow f_2$ since $\neg f_2$ does not imply $\neg f_1$.

[1] In the current implementation the third argument of the knowledge laws is empty. However, in general, this argument could be a list $P_1, \ldots, P_n$ of fluent literals equalities and inequalities with the meaning that in a situation where $P_1, \ldots, P_n$ are true if A is executed then F will be known

The initial situation description comprises two subsections. The first subsection introduces the objects of the domain and their types. This is defined with terms of the form:

$$t(o)$$

where t is a type symbol and o is an object symbol. The second subsection introduces the state of the initial situation. This is defined with terms of the following two forms:

$intially(f_1)$
$intially\_unknown(f_2)$

where f_1 and f_2 are ground fluents (i.e. fluents with no variables). The intended meaning of these terms are: In the initial situation f_1 is true and f_2 is unknown. If a ground fluent f is not mentioned in any of the initial terms it is assumed false in the initial situation.

Semantics of COPLAS domains

The semantics of COPLAS must describe how an agent knowledge changes according to the effects of actions defined by a domain description. We begin by presenting the structure of an agent's knowledge. We will represent the knowledge of an agent by a pair $\langle T, F\rangle$ called a situation, where T and F are set of legal fluents. A legal fluent is a ground fluent of the form $f(o_1,\ldots,o_n)$ such that there exists a fluent declaration of the form $fluent(f(X_1,\ldots,X_n))$: $-t_1(X_1),\ldots,t_n(X_n)$, and $t_1(o_1),\ldots,t_n(o_n)$ are members of the initial situation description. For the rest of the paper a fluent will be assumed to be a legal fluent. A situation $\langle T, F\rangle$ is consistent if $T \cap F = \emptyset$. A fluent f is true in a situation $\langle T, F\rangle$ if $f \in T$. It is false (and therefore $neg(f)$ is true) if $f \in F$. Otherwise f (and $neg(f)$) is unknown.

Interpretations for COPLAS are transition functions Φ, that map pairs of legal actions and situations into situations. A legal action is ground action of the form $a(o_1,\ldots,o_n)$ such that there exists either a non-sensing declaration of the form $action(a(X_1,\ldots,X_n)) : -t_1(X_1),\ldots,t_n(X_n)$ or sensing action declaration of the form $sensing(a(X_1,\ldots,X_n)) : -t_1(X_1),\ldots,t_n(X_n)$, with $t_1(o_1),\ldots,t_n(o_n)$ members of the initial situation description. For the rest of the paper an action refers to a legal action, and action propositions and axioms refer to ground instances of action propositions and axioms that refer to legal actions and legal fluents only.

An interpretation Φ is a *model* of a domain description D iff for every situation $\sigma = \langle T, F\rangle$, $\Phi(a,\sigma)$ is such that

1. For a fluent literal f of any deterministic effect proposition of the form $causes(a, f, [p_1,\ldots,p_n])$ in D, f is true in $\Phi(a,\sigma)$ if $p_1,\ldots,p_n$ is true in σ,
2. For a fluent literal f of any deterministic effect proposition of the form $causes(a, f, [p_1,\ldots,p_n])$ in D, f is unknown in $\Phi(a,\sigma)$ if $p_1,\ldots,p_n$ is unknown in σ and f is false in σ,
3. For a fluent f of any knowledge law of the form $causes\_to\_know(a, f, [p_1,\ldots,p_n])$ in D, f must be known in $\Phi(a,\sigma)$ (i.e. f is either in the T or the F part of σ) if $p_1,\ldots,p_n$ is true in σ,[2]
4. For any fluent literal f of any axiom of the form $axiom(p, f)$ in D, f must be true in $\Phi(a,\sigma)$ if p is true in σ,
5. For a fluent f such that there are no effect propositions of the above types, f is true/false/unknown in $\Phi(a,\sigma)$ if and only if f is true/false/unknown in σ unless there is a non-deterministic effect proposition of the form $affects(a, f, [p_1,\ldots,p_n])$ for which $p_1,\ldots,p_n$ is true or unknown in σ, in which case, f must be unknown in $\Phi(a,\sigma)$.

and when there exists at least one executability condition either of the form $possible(a, [p_1,\ldots,p_n])$ or of the form $sensing\_possible(a, [p_1,\ldots,p_n])$ in D then at least one of the preconditions $p_1,\ldots,p_n$ in one of the executability conditions must be true in σ. Otherwise, $\Phi(a,\sigma)$ is undefined.

Plans and plan evaluations

Our next step is to define the structure of our plans. In the simplest case a plan is a sequence of actions. However, with sensing we are able to verify plans that contain branching. At the root of the branch a sensing action will be executed and depending on the outcome the correct plan will be chosen. For example, a robot is planning to visit my office, and its plan may have the following structure. First, it will enter in the elevator; then it will push the button to go to the eleventh floor; it will get off the elevator in the eleventh floor; it will look at the bulletin board (it will sense); *if* my office number is even it will go to the left corridor; *else* it will go to the right corridor.

We recursively define a *plan* as follows,

1. an empty sequence denoted by ε is a plan.
2. If a is an action and α is a plan then the concatenation of a with α denoted by $a;\alpha$ is also a plan.
3. If f is a fluent and α, α_1 and α_2 are plans then **if** f **then** α_1 **else** $\alpha_2;\alpha$ is also a plan.
4. Nothing else is a plan.

The meaning of a plan depends on the interpretation we give to actions. Thus, we will define how a plan will change an initial situation based on an interpretation.

Definition 1 *The plan evaluation function Γ_Φ of an interpretation Φ is a function such that for any situation σ*

1. $\Gamma_\Phi(\varepsilon,\sigma) = \sigma$.
2. $\Gamma_\Phi(a;\alpha,\sigma) = \Gamma_\Phi(\alpha, \Phi(a,\sigma))$ *for any action a.*

[2]Recall that in the current implementation n is always assumed to be 0.

3. $\Gamma_\Phi([\text{ if } f \text{ then } \alpha_1 \text{ else } \alpha_2; \alpha, \sigma) = \Gamma_\Phi(\alpha, \sigma')$, *where* $\sigma' =$

$$\begin{cases} \Gamma_\Phi(\alpha_1, \sigma) & \text{if } f \text{ is true in } \sigma \\ \Gamma_\Phi(\alpha_2, \sigma) & \text{if } f \text{ is false in } \sigma \\ \text{undefined otherwise} \end{cases}$$

Notice that before we test the value of a fluent in an if-condition we must be sure that the value is not unknown, otherwise the evaluation of the plan is not defined.

Goal and plans

A goal will consists of a set of properties that we would like them to hold in any situation we reach after the execution of our selected plan. The plan will be executed starting from an initial situation derived from the domain description. This situation is unique and is defined as follows. The initial situation σ_0 for a domain description D is the situation where every fluent f of a term $intially(f)$ in D is true in σ_0, every fluent f of a term $initially\_unknown(f)$ in D is unknown in σ_0 and any other fluent is false in σ_0. It is also assumed that every axiom holds in σ_0. If there is no consistent situation with such a structure or the domain has no models we will say that the domain D is inconsistent.

We will restrict our goals to be a conjunction of fluent literals denoted by $p_1, \ldots, p_n$. We say that a plan α *achieves* a goal $p_1, \ldots, p_n$ in a domain D if for every model Φ of D, if each fluent literal in the goal is true in $\Gamma_\Phi(\alpha, \sigma_0)$. That is, for any possible interpretation that models the domain, the evaluation of the plan produces a situation where the goal is true.

We will write

$$D \models p_1, \ldots, p_n \textbf{ after } \alpha$$

when the plan α achieves $p_1, \ldots, p_n$ in D. COPLAS takes a consistent domain description D and a goal, and returns a plan α that achieves the goal, if that plan exists.

This achieves relation between domains, goals an plans is equivalent to the consequence relation called 0-approximations of Son and Baral (SB98), and it sounds with respect to the consequence relation introduced in (LMT97).

An example

The following example adapted from Pednault (Ped88) illustrates a domain with sensing actions and universal effects. The domain refers to situations where we would like to move objects from different locations using a briefcase.

There are three non-sensing action symbols that have the following type declarations:

$action(move\_b(L)) : -$
$\quad location(L).$
$action(take\_out(O)) : -$
$\quad object(O), diff(O, briefcase).$
$action(put\_in(O)) : -$
$\quad object(O), diff(O, briefcase).$

The first action will move the briefcase from its current location to location L. The second action will take the object O out of the briefcase. The last action will put the object inside the briefcase. Note that these definitions are encoding many legal actions. The number will depend on the number of objects and locations defined in the domain.

There is one sensing action symbol with the following declaration:

$sensing(check\_in(O)) : -object(O), diff(O, briefcase).$

The action checks whether the object O is inside the briefcase.

There are two fluent symbols with following type definitions:

$fluent(in(O)) : -object(O), diff(O, briefcase).$
$fluent(at(O, L)) : -object(O), location(L).$

The first fluent is true when the object O is inside the briefcase. The second fluent is true when the object O is at location L.

executavility conditions are defined as follows:
$possible(move\_b(L), [at(briefcase, L0), diff(L, L0)]).$

It says that the briefcase can be moved to location L if it is currently in a different location.
$possible(take\_out(X), [in(X), at(briefcase, L)]).$

This condition says that an object can be taken out of the briefcase only if it is inside the briefcase.
$possible(put\_in(X),$
$\quad [neg(in(X)), at(briefcase, L), at(X, L)]).$

This condition says that an object can be put inside the briefcase if it is not already inside and it is locate at the same place than the briefcase.

The executavility condition for the sensing actions says that we should check if an object O is inside the briefcase if we do not know about it.
$sensing\_possible(check\_in(O), []).$

Observe that for this example to have meaning if we do know know whether an object is inside the briefcase then either we do not know where the object is located or it must be located at the same place where the briefcase is.

We have four effect propositions associated with the action symbol $move\_b$.
$causes(move\_b(L), neg(at(briefcase, L0)),$
$\quad [at(briefcase, L0)]).$
$causes(move\_b(L), at(briefcase, L), []).$
$causes(move\_b(L), at(Object, L), [in(Object)]).$
$causes(move\_b(L), neg(at(Object, L0)),$
$\quad [in(Object), at(Object, L0)]).$

The first proposition says that when we move the briefcase to location L from location $L0$, it will not be in location $L0$ anymore. The second propositions says that

briefcase most be at location L. The third proposition is an application of universal quantification. The variable *Object* does not appear in the action, this when a legal action *move_b(location)* is applied *all* the objects that are inside the briefcase are also moved.[3] The last effect proposition it also has a universal quantification to make sure that all objects in the briefcase are in only one location, where the briefcase was moved.

The effect propositions for the two other action symbols are as follows:

causes(put_in(Object), in(Object), [at(briefcase, L), at(Object, L)]).
causes(take_out(Object), neg(in(Object)), [in(Object)]).

The knowledge law for *check_in* is:

causes_to_know(check_in(O), in(O), []).

The execution of the action on an object O determines if the object is inside or not of the briefcase.

An initial situation description where there are two location, home and office, and three objects, the briefcase, a dictionary and a paycheck is the following

location(home).
location(office).

object(briefcase).
object(paycheck).
object(dictionary).

initially(at(briefcase, home)).
initially(at(paycheck, home)).
initially(at(dictionary, home)).
initially_unknown(in(paycheck)).

A plan that achieves the goal *at(briefcase, office)*, *at(dictionary, office)*, *at(paycheck, home)*]) in this domain is

put_in(dictionary);
check_in(paycheck);
if *in(paycheck)* **then**
  take_out(paycheck);
  move_b(office);
else
  move_b(office);

Suppose we add to initial situation the location *bank* (i.e. location(bank)). A plan that achieves the goal *at(paycheck, bank)*, *at(dictionary, office)*, *at(briefcase, home)* is

put_in(dictionary);
check_in(paycheck);
if *in(paycheck)* **then**
  move_b(office);
  move_b(bank);
  take_out(paycheck);
  move_b(office);
  take_out(dictionary);
  move_b(home);
else
  put_in(paycheck);
  move_b(office);
  take_out(dictionary);
  move_b(bank);
  take_out(paycheck);
  move_b(home);

How to run the planner

COPLAS is written in Sicstus prolog version 3. It has been tested in a Ultra 1 station running Solaris System 2.5.1. Sources and many examples can be found in the web server of the Research Institute for Advanced Information Technology at the following address:

```
http://www.icot.or.jp/AITEC/FGCS
/funding/itaku-H9-index-E.html
```

Although the planner has not been tested in other platforms it it should work in any system running Sicstus 3.

The interaction with the planner is using the predicate `query`. A query is a conjunction of ground fluent literals representing a goal for the planner. In your domain description you must include the goal for which you would like to find a plan. The format for a goal of the form `fl1,...,fln` is:

```
query([fl1,...,fln]).
```

For example, in the first example above the query will be:

```
query([at(paycheck,home),
at(dictionary,office), at(briefcase,home)]).
```

Order in the list is irrelevant.

Before running the planner is necessary to create a file with a domain description. You must also add the following heading to the domain file:

```
%==========HEADINGS==============
:- multifile causes/3.
:- multifile affects/3.
:- multifile sensing/1.
:- multifile initially/1.
:- multifile initially_unknown/1.
:- multifile axiom/2.
%================================
```

You must also define for each action symbol a heuristic function and rewrite the executavility conditions in a special format. How these functions can be defined and the re-writing rules will be explained in the next section.

[3]The first proposition is also using a universal quantified variable but we are assuming in the example that an object cannot be in two places at the same time. Thus the universal quantification does not have any effect in that proposition.

After you create your domain file do the following steps. Let us assume the file where the domain is stored is called `domain.pl`

1. Start your prolog interpreter.
2. Consult the planner: `consult(planner).` The file `planner.pl` must be in your local directory in order for this command to execute successfully.
3. Consult your domain description: `consult('domain.pl').` The file `domain.pl` must be in your local directory in order for this command to execute successfully.
4. Run the query: `query([fl1,...,fln]).`

If you would like to run a new example with a new domain, let say, `domain1.pl`, after you get the answer for the previous query do the following.

1. Clean the environment by typing the query: `clean_all.`
2. Consult your new domain description: `consult('domain1.pl').` The file `domain1.pl` must be in your local directory in order for this command to execute successfully.
3. Answer "y" to all the renamings, if any.
4. Run the query: `query([fl1,...,fln]).`

If the planner is compiled before loading it the system will have much better per performance. To compile using Sicstus execute the following steps.

1. Start your prolog interpreter.
2. Compile the planner: `fcompile(planner).` The file `planner.pl` must be in your local directory in order for this command to execute successfully.
3. Exit the the interpreter.

This sequence of steps will generate a new file `planner.ql` the compiled version of the planner. To run the example using the compiled planner do the following steps.

1. Start your prolog interpreter.
2. Load the planner: `load(planner).` The file `planner.ql` must be in your local directory in order for this command to execute successfully.
3. Consult your domain description: `consult('domain.pl').` The file `domain.pl` must be in your local directory in order for this command to execute successfully.
4. Run the query: `query([fl1,...,fln]).`

You can switch domains by following the same steps as above. Notice also that making modifications to your current domain is very simple. You can go edit the domain description file changing any part of the domain (i.e., effect propositions, heuristics, executability conditions, etc) and re-consult the domain file to try new goals.

There is an alternative to run queries to the one described above. The user can include in his/her domain description file one query using the following format:

```
goal([fl1,...,fln]).
```

This goal corresponds to the query: `query([fl1,...,fln]).` Now after loading the domain description file following the steps above, the user can run the query by merely typing `plan` as the query. This option is useful when queries are long and the user may want to type them once.

Limitations and features of heuristic definitions in the current version

Planning is a computational complex process. There will be very few examples that will run without guiding the planner on how to search for a plan in the search space. There several domain independent heuristics that can be implemented. The most obvious is to avoid plans that take you back to a previous visited situation. In the current implementation there is only a simple check done. When an actions is executed the planner verify that the actions does not take you back to the previous state.

In addition to domain independent heuristics is also important to have domain dependent heuristics. In the current version the user can add heuristics by defining prolog predicates of the form:

```
heuristic(Action,Situation,Plan)
```

where `Situation` is the situation where the `Action` is about to be applied and `Plan` is the plan where the `Action` will be added. In its current version the parameter `Plan` has very little for non-sequential plans (i.e. plans that includes if) because it does not identify to which branch the `Action` will be added.

An example of heuristic functions in the briefcase domain is the following:

```
heuristic(move_b(_),_,_).

heuristic( take_out( X), Situation, _) :-
  goal( Goal),
  location( L),
  true_in_state( at( X, L), Situation),
  member( at( X, L), Goal).

heuristic( put_in( X), Situation, _) :-
  object(X),
  location( L),
  true_in_state( at( X, L), Situation),
  goal( Goal),
  \+ member( at( X, L), Goal).
```

The first rule defines no heuristic for the action `move_b`.[4] In general, for an action symbol a of arity n, if no heuristic is defined the domain description must include:

```
heuristic(a(_,...,_),_,_).
```

[4]The "_" symbol represents a don't care variable in prolog.

The second rule defines a heuristic for the action `take_out`. An object will be taken out from the briefcase only if that object must state at its current location according to the goal. The last rule says that if an object is not located in the place where the goal specifies the object should be, the planner should put the object in the briefcase (to be moved). Both heuristic functions use the predicate `true_in_state( Fluent, Situation)`. As the name suggests this predicate checks if the `Fluent` is true in the `Situation`. The predicates `false_in_state` and `unknown_in_state` are also available to the user for heuristic definitions.

Executavility conditions

To improve performance you must re-write the executavility conditions of your domain into new prolog predicates. If you have a executavility condition of the form:

$$possible(A, [P_1, \ldots, P_n])$$

write in your domain:

```
possible(A,Situation) :-
  fluentliteral(L1), test(L1,Situation),
  ...,
  fluentliteral(Ln),test(Ln,Situation).
```

where `Li` is equal to P_i and test is `true_in_state` if P_i is positive fluent literal. Otherwise, if P_i is of the form `neg(F)`, `Li` is `F` and `test` is `false_in_state`. For example, the condition $possible(put\_in(X), [neg(in(X)), at(briefcase, L), at(X, L)])$ is translated into:

```
possible( put_in( X), Situation) :-
  fluentliteral(in(X)),
  false_in_state(in(X), Situation),
  fluentliteral(at(briefcase,L),
  true_in_state( at(briefcase,L), Situation),
  fluentliteral(at(X,L)),
  true_in_state( at(X,L), Situation).
```

For sensing action symbols we will restrict the domains to only one knowledge law per action and no preconditions.[5] Thus, for a executavility condition of the form:

$$sensing\_possible(A, [P_1, \ldots, P_n])$$

write in your domain:

```
sensing_possible(A,Situation) :-
  unknown_in_state(F,Situation),
  fluentliteral(L1), test(L1,Situation),
  ...,
  fluentliteral(Ln),test(Ln,Situation).
```

[5]The semantics of COPLAS can deal with both extensions, but they will be developed in future versions of the system.

where `F` is the fluent being sensed by the action `A` and the rest is defined as above. For example, the condition $sensing\_possible(check\_in(O), [])$ is translated into:

```
sensing_possible(check_in(O),Situation) :-
   unknown_in_state(in(O),Situation).
```

If you are using the planner in classroom situation where you would like to strictly separate the domain description from the planner implementation you can make the following modifications to the planner.

Find in the planner the two lines with the comment `% independent exec. conditions` and remove the `%` symbol from the front of the line.

Find in the planner the two lines with the comment `% dependent exec. conditions` and add the `%` symbol to the front of the line.

This modification to the coding of executavility conditions will slow down the performance of the planner about 10%.

Notes about the plans generated by the planner

Similar to the partial order planner described in Rusell and Norvig (RN95) that handles sensing actions, once an if-then-else has been introduced in a plan no other actions will appear after the if-then-else. If a sequence of actions is needed after the if-then-else, that sequence will be repeated in both the "then" part and the "else" part. In other words the structure of the program is a tree where each branch corresponds to a different set of models of the domain. We can also associate a situation with each branch. All these situations represent all the possible final situations that could result after the execution of the plan (module sensing actions). Detecting when situations can be combined is not trivial and can be expensive (since it might be necessary to check that a collection of sets are equal). On the other hand, the current implementation adds a new situation each time a sensing action is executed and the planner must find a plan for each situation. If most of the plan is repeated among the branches this results in a large amount of redundant computation.

There is also a property that holds for every plan. A sensing action will appear in a plan if and only if the action will be followed by a a conditional. Furthermore, the condition tested is always the fluent that is sensed by the action.

Using the planner for projections

There is a second kind of queries that can be solved with COPLAS. Given an initial situation COPLAS can verify if a conjunctions of fluent literals is true after the execution of a given plan. This process is much simpler than planning since there is no plan to generate, the plan is part of the query. This mode is useful to debug domain descriptions. Moreover, the plans in the queries

can be more complex than the plans generated during planning mode. In addition to conditionals, plan can also contain while loops. Queries in this mode are of the form:

```
entails(Goal,Plan).
```

where the `Goal` is a list of fluent literals as indicated in the previous queries and a plan is recursively defined as follows:

1. `[]` (the empty list in prolog) is a plan.
2. If `A` is an action and `P` a plan then `[A|P]` is also a plan. Arguments in `A` could be variables or ground terms.
3. If `F` is a fluent literal and `P1` and `P2` are plans, then `[if(F,P1)|P2]` is also a plan. `F` most be a ground literal, i.e. all its arguments most be object symbols.
4. If `F` is a fluent literal and `P1`, `P2` and `P3` are plans, then `[if(F,P1,P2)|P3]` is also a plan. `F` most be a ground literal, i.e. all its arguments must be object symbols.
5. If `F` is a fluent literal and `P1` and `P2` are plans, then `[while(F,P1)|P2]` is also a plan. `F` most be a ground literal, i.e. all its arguments must be object symbols.
6. Nothing else is a plan.

We can describe the meaning of a plan with loops formally using standard fix-point techniques used in conventional programming languages. For lack of space will not present the formal semantics here but it is not difficult to see how the semantics of $\mathcal{A}_K$ described in (LMT97) can be adapted to COPLAS domains.

Similar to the planning mode, After you create your domain file do the following steps. Let us assume the file where the domain is stored is called `domain.pl`

1. Start your prolog interpreter.
2. Consult the planner: `consult(planner).` The file `planner.pl` must be in your local directory in order for this command to execute successfully.
3. Consult your domain description: `consult('domain.pl').` The file `domain.pl` must be in your local directory in order for this command to execute successfully.
4. Run the query: `entails([fl1,...,fln], Plan).`

The answer for this query will be `yes` if all the fluent literals in the query are true in the states that result after the execution of the `Plan`. The answer is `no` if the `Plan` does not have an infinite loop and terminates in a set of states where where at least one of the literals in the query is not true. There might be no answer if the `Plan` has an infinite loop.

Final remarks

There are two fundamental ways this system can be used:

1. As a planner: User can write domain descriptions in the language, pose goals to generate plans. Ideally, the system can be used as a sub-module for a robot or a software intelligent agent. For a realistic application the planner needs to be modified to prune search space (specially redundant computation due to revisits to the same situation).
2. As a platform for $\mathcal{A}$ based action description languages: Recently, there has been a tremendous amount of language extensions $\mathcal{A}$ to include, concurrency, narratives, ramification constraints, temporal queries, defeasible actions, etc. Our hopes is that our system will encourage researchers in the field to test their technical ideas by modifying our planner to include such extensions.

To report bugs or for comments or suggestion to improve the code you can write directly to me at `jorge@eecs.uic.edu`.

References

A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, J. Penberthy, Y. Sun, and D. Weld. UCPOP User's manual, version 4.0. Technical Report 93-09-06d, Department of Computer Science and Engineering, University of Washington, 1995.

M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2,3,4):301–323, 1993.

Y. G. Kahl. Planning in complex domains: Expressiveness and efficiency. Master's thesis, Dept of Computer Science, University of Texas at El Paso, 1996.

Hector Levesque. What is planning in the presence of sensing? In *Proc. of AAAI-96*, pages 1139–1146, 1996.

J. Lobo, G. Mendez, and S. Taylor. Adding knowledge to the action description language $\mathcal{A}$. In *Proc. of AAAI97*. AAAI Press, July 1997.

E. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4(4):356–372, 1988.

S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, 1995.

T. C. Son and C. Baral. *Formalizing Sensing Actions - a transition function based approach*. Technical Report, University of Texas at El Paso. Short version of this paper appeared in *Proc. of ILPS97*. MIT Press, 1997.