# aPMS: From theory to implementation

**Marrella Andrea, Valentini Stefano**

*This is a very very very….first draft of a possible future document we are going to realize*

## 1. The Code

```
:- dynamic controller/1.

/* DOMAINS-SORTS AVAILABLE IN THE DOMAIN */

/* Domain representing available services; in the example proposed our team is composed by 7 operators*/
operators([1,2,3,4,5,6,7]).

/* Domain of possible tasks executables by services */
tasks([photo,rescue,evacuation,survey,senddata,evaluatephoto,census]).

/* Domain representing the capabilities */
capabilities([makephoto,compilesurvey,compilecensus,gprs,evaluation,rescues]).

/* Domain representing the necessaries identifiers for the univocal identification of each task */
identifiers([id_1,id_2,id_3,id_4,id_5,id_6,id_7,id_8,id_9,id_10,id_11,id_12,id_13,id_14,id_15,id_16,id_17,id_18,id_19,id_20,id_21,id_22,id_23,id_24,id_25,id_26,id_27,id_28,id_29,id_30,id_31,id_32,id_33,id_34,id_35,id_36,id_37,id_38,id_39,id_40,id_41,id_42,id_42,id_44,id_45,id_46,id_47,id_48,id_49,id_50]).

/* There is nothing to do caching on (required becase cache 1 is static)*/
cache(_):-fail.

/* Some domain-independent predicates to denote the various objects of interest in the framework: */
service(N) :- domain(N,operators).
task(X) :- domain(X,tasks).
capability(B) :- domain(B,capabilities).
id(D) :- domain(D,identifiers).


/*  FLUENTS and CAUSAL LAWS */
%
%  A basic action theory (BAT) is described with:
% -- fun_fluent(fluent)     : for each functional fluent (non-ground)
% -- rel_fluent(fluent)     : for each relational fluent (non-ground)
%
% -- causes_val(action,fluent,value,cond)
% --          when cond holds, doing act causes functional fluent to have value
```

%

*/* Basically, there has to be some definition for predicates causes_true/3 and causes_false/3, at least one for each. We have added the following dummy code: */*

**causes_true(_,_,_) :- false.**

**causes_false(_,_,_) :- false.**

*/* The service N provides the capability B */*

**rel_fluent(provide(N,B)) :- service(N), capability(B).**

*/* The task X requires the capability B to be executed*/*

**rel_fluent(required(X,B)) :- task(X), capability(B).**

*/* This relational fluent indicates that task X with id D has been begun by service N*/*

**rel_fluent(started(X,D,N)) :- task(X), service(N), id(D).**

*/* **started(X,D,N)** becomes **false** if the service N calls the exogenous action **end(X,D,N,O,V)**, indicating the ending of the task. If a service N wants to end a task X, the task X has to be already begun by service N /**

**causes_val(end(X,D,N,O,V),started(X,D,N),false,started(X,D,N)=true).**

*/* **started(X,D,N)** becomes **true** if the service N calls the exogenous action **begin(X,D,N)**, indicating the starting of the task. Obviously, if a service N wants to start a task X, N has to be available (it means that at the moment it must not execute any other task) and the task X has to be already assigned to service N /**

**causes_val(begin(X,D,N),started(X,D,N),true,and(assigned(X,D,N)=true,available(N)=true)).**

*/* This relational fluent indicates that task X with id D has been assigned to service N*/*

**rel_fluent(assigned(X,D,N)) :- task(X),service(N),id(D).**

*/* **assigned(X,D,N)** becomes **true** if the PMS engine calls the action **assign(X,D,N,I,O)** /**

**causes_val(assign(X,D,N,I,O),assigned(X,D,N),true,true).**

*/* **assigned(X,D,N)** becomes **false** if the PMS engine calls the action **release(X,D,N)** /**

**causes_val(release(X,D,N),assigned(X,D,N),false,true).**

*/* This relational fluent **available(N)** indicates that service N is available to execute a task (therefore, at the moment, it isn't executing any other task). It means that, for the service N, **started(X,D,N)** , for each task X and for each id D, has to be **false** */*

**rel_fluent(available(N)) :- service(N).**

*/* **available(N)** becomes **true** when the fluent **started(X,D,N)** becomes **false**. It means that the task x with id d has been already begun by service n (started(x,d,n)=true), and that service n has called the exogenous action end(x,d,n,o,v), indicating the ending of the task x /**

**causes_val(end(X,D,N,O,V),available(N),true,started(X,D,N)=true).**

*/* **available(N)** becomes **false** when the fluent **started(X,D,N)** becomes **true**. It means that the task x with id d has been already assigned to service n (assigned(x,d,n)=true), and that service n has called the exogenous action begin(x,d,n) to start the task/**

**causes_val(begin(X,D,N),available(N),false,assigned(X,D,N)=true).**

*/* This relational fluent is used in order to avoid that two tasks of the same kind are assigned to the same service .* Therefore, two tasks of the same kinds (i.e. two tasks photo) can't be assigned at the same moment to N*/

**rel_fluent(kind_assigned(X,N)) :- task(X),service(N).**
**causes_val(assign(X,D,N,I,O),kind_assigned(X,N),true,true).**
**causes_val(release(X,D,N),kind_assigned(X,N),false,true).**

**/* FLUENTS GENERATED EACH TIME FOR MANAGE DECISION POINT */**

/* These fluents are generated automatically in order to manage the decision points in the process */
**fun_fluent(numphoto(D)) :- id(D).**

/* When service n call the exogenous action end(X,n,O,V), it indicates that output O has to have value V. If in the activity diagram representing the process is present a decision point, in the pms.pl file will be generated a number of fun_fluents equal to the conditions presented in the decision point. In this way (and exploiting the id of each task) the engine can capture the value of the ouput */

**causes_val(end(X,D,N,O,V),numphoto(D),V,O=numphoto).**

**fun_fluent(evaluate(D)) :- id(D).**
**causes_val(end(X,D,N,O,V),evaluate(D),V,O=evaluate).**

**/* ACTIONS and PRECONDITIONS*/**

/* Every task execution is the sequence of four actions:

- *(i)* the assignment of the task to a service.
- *(ii)* The notifcation to the service N to start executing the task X (It happens when the service N calls the exogenous action begin(X,D,N)).
- *(iii)* the PMS stops the service acknowledging the successful termination of its task. (It happens when the service N calls the exogenous action end(X,D,N,O,V)).
- *(iv)* Finally, the PMS releases the service, which becomes available again.

We formalize these four actions as follows (these are the only actions used in our formalization): */

/* Assign task X identified by id D to service N, with input I and requested output O */
**prim_action(assign(X,D,N,I,O)) :- task(X), service(N), id(D).**
/*Is possible to assign a task X to service N if N is available and another task X hasn't been assigned to N. Therefore, two tasks of the same kinds (i.e. two tasks photo) can't be assigned at the same moment to N*/
**poss(assign(X,D,N,I,O), and(isAvailable(N),kind_assigned(X,N)=false)).**

**prim_action(stop(X,D,N)) :- task(X), service(N), id(D).**
**poss(stop(X,D,N), true).**

**prim_action(start(X,D,N)) :- task(X), service(N), id(D).**
**poss(start(X,D,N), true).**

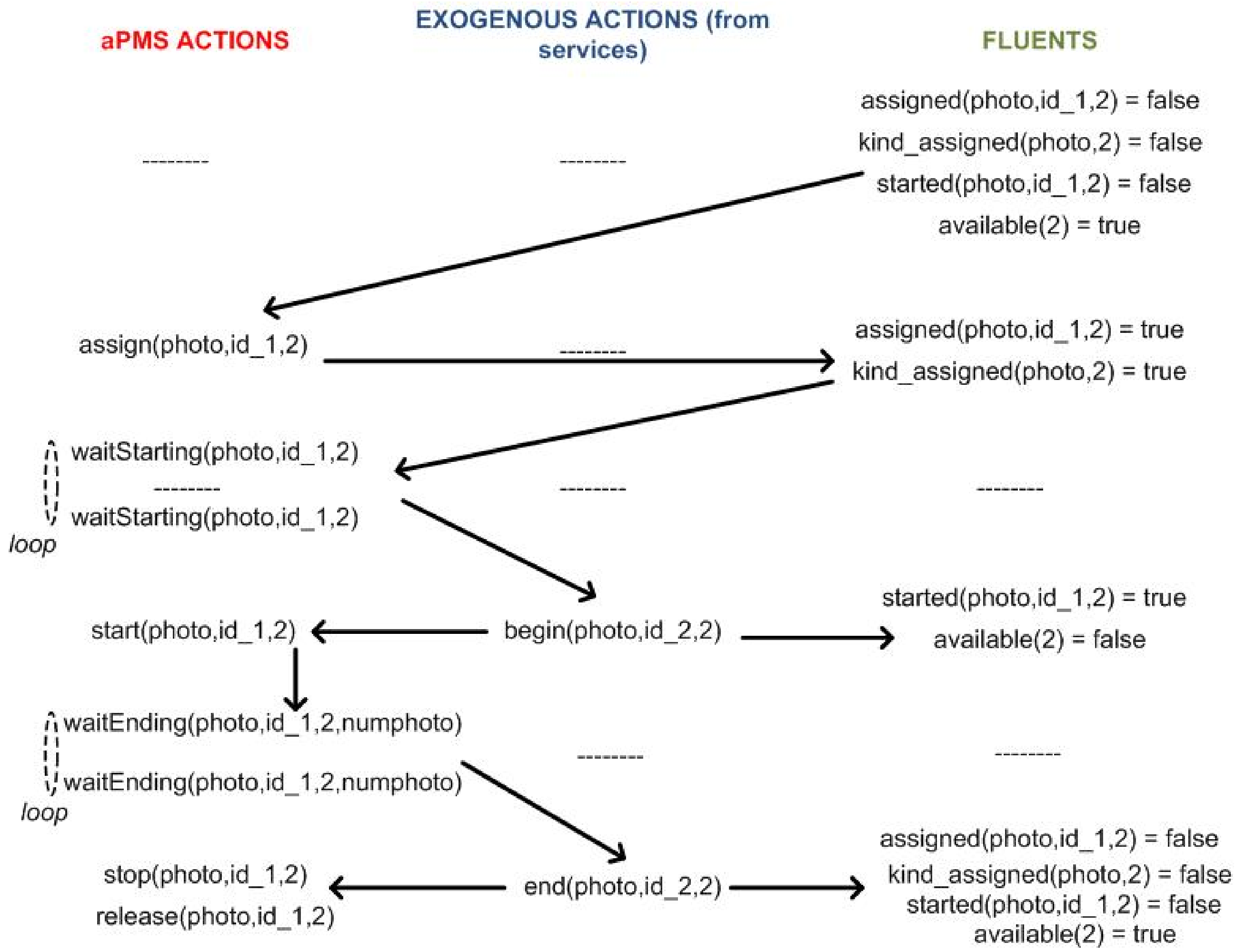
```
prim_action(release(X,D,N)) :- task(X), service(N), id(D).
poss(release(X,D,N), true).

 /* EXOGENOUS ACTIONS*/

/* After an assignment of a task, a service n can indicate to PMS that it has started\finished the execution
of a task using these actions */
exog_action(begin(X,D,N)) :- task(X), service(N), id(D).
exog_action(end(X,D,N,O,V)) :- task(X), service(N), id(D).


/* FICTITIOUS ACTIONS */

/* These are fictitious actions; waitStarting(X,D,N) indicates that the program is waiting that service N calls
the exogenous action begin(X,D,N) to effectively start the task X */
prim_action(waitStarting(X,D,N)) :- task(X), service(N), id(D).
poss(waitStarting(X,D,N), true).
/* waitEnding(X,D,N,O) indicates that the program is waiting that service N calls the exogenous action
end(X,D,N,O,V) to effectively stop the task X with output O and value of the output V */
prim_action(waitEnding(X,D,N,O)) :- task(X), service(N), id(D).
poss(waitEnding(X,D,N,O), true).


/* ABBREVIATIONS */

/* These are simple boolean functions; for example, when in a procedure we call isAvailable(n), the
program verifies if the fluent available(n) is true or not */
proc(isAvailable(N), available(N)=true).
proc(isStarted(X,D,N), started(X,D,N)=true).
proc(isProvided(N,B), provide(N,B)=true).
proc(isRequired(X,B), required(X,B)=true).


/* INITIAL STATE:  */

/* Definition of the initial state */
initially(available(N),true) :- service(N).

initially(assigned(X,D,N),false) :- task(X), service(N), id(D).
initially(kind_assigned(X,N),false) :- task(X), service(N).

initially(started(X,D,N),false) :- task(X), service(N), id(D).

initially(provide(N,B),false) :- service(N), capability(B), N\=1, N\=2, N\=3, N\=4, N\=5, N\=6, N\=7.
```

```
initially(provide(1,gprs),true).
initially(provide(1,evaluation),true).

initially(provide(2,compilecensus),true).
initially(provide(2,rescues),true).

initially(provide(3,compilecensus),true).
initially(provide(3,rescues),true).

initially(provide(4,compilecensus),true).
initially(provide(4,rescues),true).

initially(provide(5,compilesurvey),true).
initially(provide(5,makephoto),true).

initially(provide(6,compilesurvey),true).
initially(provide(6,makephoto),true).

initially(provide(7,compilesurvey),true).
initially(provide(7,makephoto),true).

initially(required(X,B),false) :- task(X), capability(B), X\=photo, X\=rescue, X\=evacuation, X\=survey,
X\=senddata, X\=evaluatephoto, X\=census.

initially(required(photo,makephoto),true).
initially(required(survey,compilesurvey),true).
initially(required(census,compilecensus),true).
initially(required(senddata,gprs),true).
initially(required(evaluatephoto,evaluation),true).
initially(required(rescue,rescues),true).
initially(required(evacuation,rescues),true).

initially(numphoto(D),0) :- id(D).
initially(evaluate(D),false) :- id(D).
```

**% THIS IS THE MAIN PROCEDURE FOR INDIGOLOG**

*/*This is the core procedure of the engine. First of all, it finds the necessary capability b to execute the task X. Then it finds a service n that **(i)** provides the capability b, **(ii)** isAvailable (no other tasks have been begun from the service) **(iii)** hasn't already assigned a task of the same kind of X (i.e. two tasks photo can't be assigned at the same time to n). At this point (suppose that pi() doesn't fail) it calls **assign(X,D,n,I,O)**; it means that task X with id D, input I and requested output O is assigned to service n. Now,while the service n doesn't call the exogenous action **begin(X,D,n)**, the program enters in a loop, calling the fictitious action **waitStarting(X,D,n).** When service n calls the exogenous action **begin(X,D,n),** finally the PMS can call the action **start(X,D,n)**, indicating the starting of the task X. Now,while the service n doesn't call the exogenous*

*action **end(X,D,n,O,V)**, the program enters in a loop, calling the fictitious action **waitEnding(X,D,n,O).** When service n calls the exogenous action **end(X,D,n,O,V),** finally the PMS can call the actions **end(X,D,n**) and **release(X,D,n)**, indicating the ending and the releasing of the task X by the service n*
**/*
***In the picture below it is shown an example of execution of the task "photo" using the procedure manageTaskAssignation. We suppose that the pi() function has individuated the service 2 as executor of this task.***



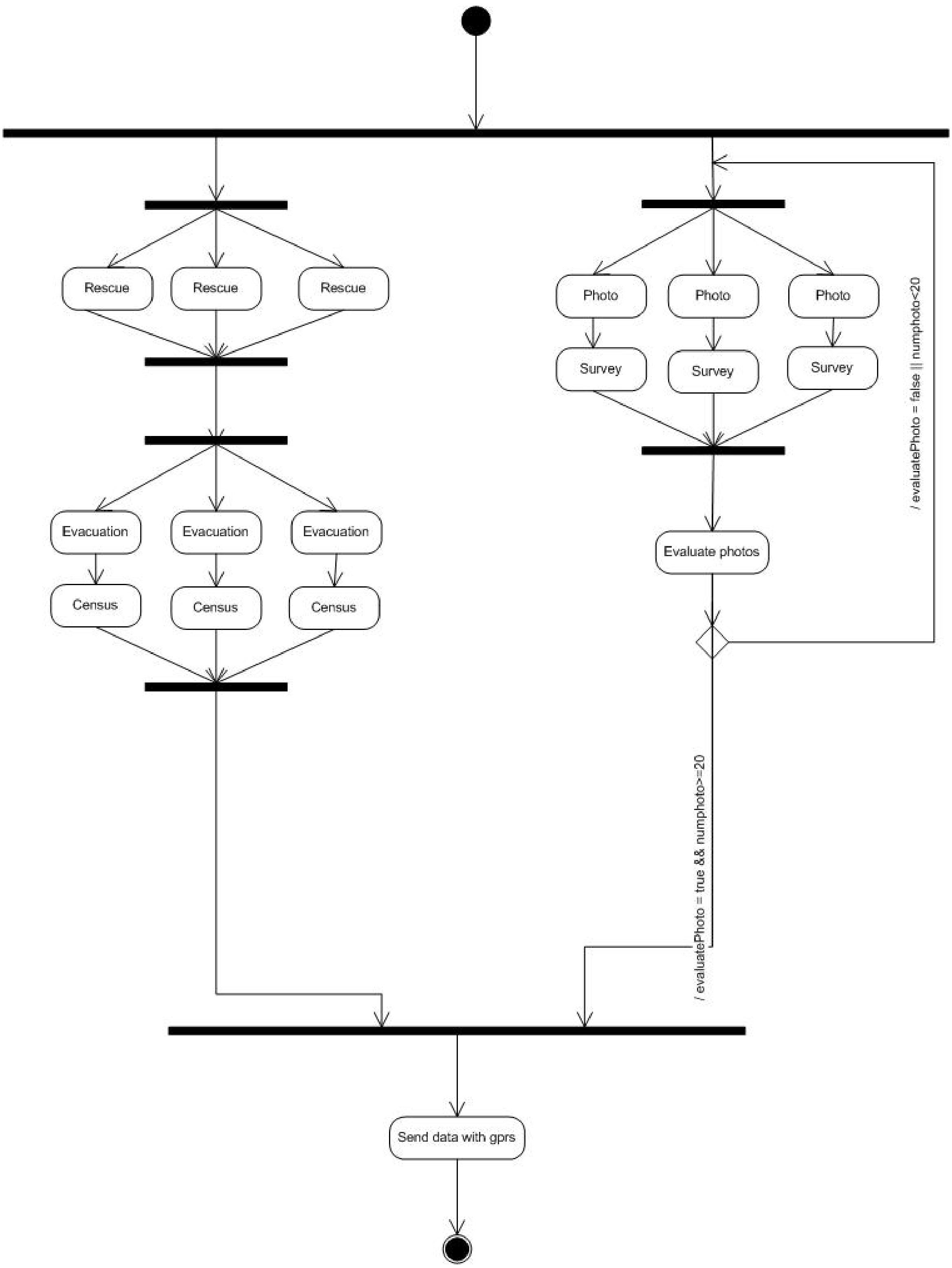

```
proc(manageTaskAssignation(X,D,I,O),
pi(b,[?(isRequired(X,b)),pi(n,[?(and(kind_assigned(X,n)=false,and(isProvided(n,b),isAvailable(n)))),
assign(X,D,n,I,O),
while(neg(isStarted(X,D,n)),waitStarting(X,D,n)), start(X,D,n),
while(isStarted(X,D,n),wait),stop(X,D,n),release(X,D,n)])])).

proc(main,  mainControl(N)) :- controller(N), !.
proc(main,  mainControl(5)). % default one
```

```
/*This is the process representing in the activity diagram*/

proc(mainControl(5), [
itconc([
 [itconc([
       [manageTaskAssignation(rescue,id_8,location,res)],
       [manageTaskAssignation(rescue,id_9,location,res)],
       [manageTaskAssignation(rescue,id_10,location,res)]
      ]),
  itconc([
[manageTaskAssignation(evacuation,id_11,location,evac),manageTaskAssignation(census,id_12,location,
cens)],
[manageTaskAssignation(evacuation,id_13,location,evac),manageTaskAssignation(census,id_14,location,
cens)],
[manageTaskAssignation(evacuation,id_15,location,evac),manageTaskAssignation(census,id_16,location,
cens)]
      ])
 ],
 [
  while(or(numphoto(id_2)+numphoto(id_3) +numphoto(id_4)<20,evaluate(id_7)=false),
  [itconc([

[manageTaskAssignation(photo,id_2,location,numphoto),manageTaskAssignation(survey,id_5,location,q
uestionnaire)],
[manageTaskAssignation(photo,id_4,location,numphoto),manageTaskAssignation(survey,id_18,location,
questionnaire)],
[manageTaskAssignation(photo,id_3,location,numphoto),manageTaskAssignation(survey,id_6,location,q
uestionnaire)]
       ]),
        manageTaskAssignation(evaluatephoto,id_7,location,evaluate)]])
 ]
]),
manageTaskAssignation(senddata,id_17,information,sendingok)
]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  INFORMATION FOR THE EXECUTOR
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Translations of domain actions to real actions (one-to-one)
actionNum(X,X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EOF: PMS/pms.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

## 2. *A Running Example*

We want to show a real example of how our system works.

1) The first step is the drawing of the activity diagram (with an external program) representing the process to be executed. Picture 1 shows the activity diagram we have drawn for the example:



The diagram shows that the two big branches **should be executed in parallel**, and **only when both ones terminate**, it will be executed the last task “Send data with gprs”.

***For the first branch:***
The 3 tasks *rescue* has to be executed in parallel. Only when all these three tasks terminate, is possible to execute the next tasks (the three couples *evacuation-census*).
The 3 tasks *evacuation* has to be executed in parallel. When one of the task evacuation terminate, is possible to execute the task *census* situated below. Therefore is possible a situation in which, for example, an operator is executing a task *evacuation* and other two operators are executing a task *census* (it means that the corresponding task *evacuation* that anticipate it is already terminated). When the three couple of tasks *evacuation-census* terminate, the **LEFT BRANCH** terminate its execution.

***For the second branch:***
The 3 couple of tasks *photo-survey* has to be executed in parallel in the same way of the three couples *evacuation-census*. When the three couple of tasks *photo-survey* terminate, it will be executed the task *evaluate photo.* At this point, if the decision point succeds, the **RIGHT BRANCH** terminate its execution; otherwise its execution start again.

2) The second step is the translation of the activity diagram in XML format (with an external program).
3) The third step is the generation (with an external program), starting from the XML file representing the process to be executed, of the file pms.pl.
   Some parts of the pms.pl (fluent,actions,manageTaskAssignation procedure) will remain the same every time. Other parts (the main_control procedure, the initial state, the values of domain, the fluent that manage decision points) will be re-created every time depending from the diagram describing the process.
   For the diagram in the example i imagine a main_control procedure similar to the one proposed in the *main_control(5)* in the code located on the top.

4) Finally we can execute (using indigolog) the file pms.pl