2.1 An abstract model of reactive agents with sensing

Following Wooldridge & Lomuscio, a (simplified) environment ***Env*** is a tuple $\langle \boldsymbol{E}, \boldsymbol{\tau_e}, \boldsymbol{e_0} \rangle$, where

- $\boldsymbol{E}=\{e_1,e_2,\ldots\}$ is a set of *states* for the environment
- $\boldsymbol{\tau_e} : E \times Act \rightarrow E$ is a *state transformer* function for the environment, with *Act* a set of *actions*
- $\boldsymbol{e_0} \in E$ is the *initial state* of the environment

2.1 An abstract model of reactive agents with sensing

and an agent ***Ag*** is a tuple $\langle \boldsymbol{L}, \boldsymbol{Act}, \boldsymbol{see}, \tau_a, \boldsymbol{do}, \boldsymbol{l_0} \rangle$, where

- $\boldsymbol{L}=\{l_1, l_2,\ldots\}$ is a set of *local states* for the agent
- $\boldsymbol{Act}=\{a_1,a_2, \ldots\}$ is a set of *actions*
- $\boldsymbol{see}: E \rightarrow P$ is the *perception* function
- $\boldsymbol{\tau_a} : L \times P \rightarrow L$ is the *state transformer* function
- $\boldsymbol{do}: L \rightarrow Act$ is the *action selection* function,
- $\boldsymbol{l_0} \in L$ is the *initial state* for the agent

2.1 An abstract model of reactive agents with sensing

An *agent system* is a pair ***{Ag,Env}***, its set of *global states* ***G*** is any subset of $L \times E$ i.e., $g_i = \langle l_i , e_i \rangle$

A *run* of a agent system is a (possibly infinite) sequence of global states $(g_1, g_2, \ldots)$ over ***G*** such that

$$\forall i,\ g_i = \langle \tau_a(l_{i-1}, see(e_{i-1})), \tau_e(e_{i-1}, do(l_i)) \rangle$$

2.2 A concrete model of reactive agents with sensing

Let ***S*** be the set of sentences of first order logic with arithmetic whose set of predicates includes the predicate ***do/1***, and let $P=S$ and $L=\wp(S)$. If we incorporate the perception function and selection of actions within the functions τ_a and τ_e, then we get two new functions

- $\tau_{a,see} : L \times E \rightarrow L$
- $\tau_{e,do} : E \times L \rightarrow E$

2.2 A concrete model of reactive agents with sensing

Equivalently, these new functions can be seen as procedures with side effects i.e.,

- $\tau_{a,see} \ : \ L \times E \rightarrow L \ \Rightarrow$ ***procedure*** *sense(l,e)* with side effects on *l.*
- $\tau_{e,do} \ : \ E \times L \rightarrow E \ \Rightarrow$ ***procedure*** *react(e,l)* with side effects on *e*

We can define these procedures as follows:

2.2 A concrete model of reactive agents with sensing

- ***procedure*** *sense(l,e)*
  if *"the agent receives the percept p"*
  then $l \leftarrow \tau_a(l,p)$

- ***procedure*** *react(e,l)*
  if $l \vdash do(a)$
  then $e \leftarrow \tau_e(e,a)$

2.2 A concrete model of reactive agents with sensing

We write $l \vdash do(a)$ to mean that the formula *do(a)* can be proved from the formula *l*, meaning in turn that *a* is an applicable action: we thus define a logical agent model

An agent's run is then defined as follows

procedure *run(e,l)*

loop *sense(l,e);*

react(e,l)

2.2 A concrete model of reactive agents with sensing

Although environments are supposed to evolve deterministically, the choice to be made among applicable actions is left unspecified. Consequently, the *run* procedure can be seen as a *non-deterministic* abstract machine generating runs for logical agents (=a concrete model of non-deterministic agents)

2.3.1 A concrete model of a reactive agents with sensing and plans

Intuitively, an agent's plan can be described as an ordered set of actions that may be taken, in a given state, in order to meet a certain objective. As the choice among applicable plans will be left unspecified, agent will remain non-deterministic

2.3.1 A concrete model of a reactive agents with sensing and plans

We assume a set $P = \{p_1, p_2, \dots\}$ of non-deterministic plan names (*nd-plan* in short) and three predicates *plan/1*, *do/2* and *switch/2.*

For any agent, its current nd-plan $p \in P$ refers to a set of implications "***conditions***" $\Rightarrow$ ***do(p, a)*** or "***conditions***" $\Rightarrow$ ***switch(p, p')***, where *a* is an action.

We further assume that an agent's initial nd-plan p_0 can be deduced from l i.e., that $l \vdash plan(p_0)$.

2.3.1 A concrete model of a reactive agents with sensing and plans

Example: a vacuum cleaner robot

To illustrate these concepts, let us consider a vacuum cleaner robot that can choose either to *work* i.e., *move* and *suck* any dirt on sight, or to go *home* and wait. Let us further assume that the robot must *stop* whenever an *alarm* condition is raised. These three behaviors correspond to three possible *nd-plans*, i.e. *work*, *home* and *pause*.

2.3.1 A concrete model of a reactive agents with sensing and plans

Example: a vacuum cleaner robot

The robot behavior can be represented by a decision tree rooted at a single *initial* plan

		plan(initial)
alarm	$\Rightarrow$	**switch**(initial,pause)
$\neg$alarm	$\Rightarrow$	**switch**(initial,start)
dirt(_,_)	$\Rightarrow$	**switch**(start,work)
$\neg$dirt(_,_)	$\Rightarrow$	**switch**(start,home)
		do(pause,stop)
in(X,Y) $\wedge$ dirt(X,Y)	$\Rightarrow$	**do**(work,suck(X,Y))
in(X,Y) $\wedge$ $\neg$dirt(X,Y)	$\Rightarrow$	**do**(work,move(X,Y))
in(X,Y)	$\Rightarrow$	**do**(home,back(X,Y))

2.3.1 A concrete model of a reactive agents with sensing and plans

Let us further extend the definition of an agent's global state to include its current active nd-plan *p*. We finally have the following new procedures:

procedure *react(e,l,p)*
if $l \vdash do(p, a)$
then $e \leftarrow \tau_e(e,a)$
else if $l \vdash switch(p, p')$
  then *react(e,l,p′)*

procedure *run(e,l)*
loop *sense(l,e);*
  if $l \vdash plan(p_0)$
  then $react(e,l,p_0)$

2.3.1 A concrete model of a reactive agents with sensing and plans

At each run cycle, procedure *react* will be called with the (possibly variable) initial plan p_0 deduced for the agent. In each recursive *react* call, the agent's first priority is to deduce and carry out an action *a* from its current plan *p*. Otherwise, it may switch from *p* to *p′*.

2.3.1 A concrete model of a reactive agents with sensing and plans

If the *switch* predicate defines decision trees rooted at each p_0, then *react* will go down this decision tree. As a result, actions will be chosen one at a time. The mechanism just described allows an agent to adopt a new plan whenever a certain condition occurs, and then to react with an appropriate action.

2.3.1 A concrete model of a reactive agents with sensing and plans

This extended virtual machine constitutes a model of *reactive* and *proactive* agents, this latter capability deriving from the deduction of initial plans p_0

2.3.2 A concrete model of a reactive agents with priority processes

Similarly to plans, *processes* of explicit priority *n* are defined by implications "***conditions***" $\Rightarrow$ ***do(n, a)***.

Consider then the following procedure

procedure *process(e,l,n)*
if $l \vdash do(n, a)$
then $(e,l) \leftarrow \tau(e,l,a);$
    process(e,l,n)
else if $n > 0$
    then *process(e,l,n-1)*

2.3.2 A concrete model of a reactive agents with priority processes

The procedure *process*, when called with an agent's highest priority n_0, will execute, in descending order of priorities, all processes whose conditions are satisfied.

We shall further assume that n_0 can be deduced from l i.e., that $l \vdash priority(n_0)$.

2.3.3 A Prolog implementation

We need to represent

- the deduction of plans and actions i.e.,
  $l \vdash plan(p_0)$ and $l \vdash do(p, a)$
- the state transformer functions i.e. ,
  $\tau_e(e,a)$ and $\tau_a(l,p)$
- the capture of perceptions

2.3.3 A Prolog implementation

- Agents will be represented as simple objects encapsulating the formulas that hold in their local state *l* .

- These formulas will include the agent's representation of the environment i.e., both transforming functions will affect the agent's local state.

2.3.3 A Prolog implementation

An ADT for objects holding logical formulas

Basic types

O : the set of objects

L : the language of formulas

$List_L$: the set of lists of formulas of L

2.3.3 A Prolog implementation

An ADT for objects holding logical formulas

predicate

instance : $O \times L \rightarrow$ *boolean* true if the object contains an instance of the formula

operations

new : $\rightarrow O$ creates an empty object

insert : $O \times L \rightarrow O$ inserts a formula into the object

remove : $O \times L \rightarrow O$ removes all instances of a formula

insertList : $O \times List_L \rightarrow O$ inserts a list of formulas

2.3.3 A Prolog implementation

An ADT for objects holding logical formulas

implementation

any formula *P* of agent *A* is asserted as **`instance(A,P)`**
(where *A* is the actual name of the agent)

```
new(A)             :-  retractall(instance(A,_)).
insert(A,P)        :-  assert(instance(A,P)).
remove(A,P)        :-  retractall(instance(A,P)).
insertList(A,Name):-  forall((Name:List,
                             member(P,List)),
                            insert(A,P)).
```


2.3.3 A Prolog implementation

An ADT for objects holding logical formulas

example

```
plans:
[                    plan(initial),
  alarm          => switch(initial,pause),
  not alarm      => switch(initial,start),
  dirt(_,_)      => switch(start,work),
  not dirt(_,_) => switch(start,home),
   …                                    ].

insertList(robot,plans).
```

2.3.3 A Prolog implementation

A meta-interpreter for simple deductions in objects (i.e., implementing a restricted form of $l \vdash P$)

```
ist(A,P)     :-  instance(A,P).
ist(A,Q)     :-  instance(A,P=>Q),
                 ist(A,P).

ist(A,(P,Q)) :-  ist(A,P),
                 ist(A,Q).
ist(A,not P) :-  \+ ist(A,P).
```


2.3.3 A Prolog implementation

A meta-interpreter for simple deductions in objects (i.e., implementing a restricted form of
$l \vdash p$)

```
ist(A, P is Q) :-   P is Q.
ist(A, P = Q)  :-   P = Q.
ist(A, P < Q)  :-   P < Q.
ist(A, P > Q)  :-   P > Q.
ist(A, P \= Q) :-   P \= Q.
```

2.3.3 A Prolog implementation

Representing state transformer functions

An agent's actions will be represented by methods to be encapsulated in the object representing the agent

Format: `method(Agent.Call,Body)`

where `Agent` = the agent's name

`Call` = the method's name with its parameters

`Body` = Prolog code for the action

2.3.3 A Prolog implementation

Representing state transformer functions

example

```
actions:
[method(Agent.suck(X,Y),
        (remove(Agent,dirt(X,Y)))),
   …                                    ].

insertList(robot,actions).
```

2.3.3 A Prolog implementation

Representing state transformer functions

Methods can be called using messages

Format: `Agent.Call`

Example: `robot.suck(1,1)`

Messages are interpreted as

```
Agent.Call  :- instance(Agent,
                        method(Agent.Call,Body)),
               call(Body).
```


2.3.3 A Prolog implementation

Implementing the virtual machine itself

The virtual machine itself is implemented as a list of agent methods plus a bootstrap procedure

```
machine:
[method(Agent.sense,
        (interrupt(Call)
         -> (instance(Agent,
                      method(Agent.Call,Body))
             -> call(Body);
             insert(Agent,Call));
         true)),
```

2.3.3 A Prolog implementation

Implementing the virtual machine itself

```
method(Agent.react(Plan),
        (ist(Agent,do(Plan,Action))
         -> Agent.Action;
         (ist(Agent,switch(Plan,NewPlan))
          -> Agent.react(NewPlan);
          Agent.noOp(noAction)))),
```


2.3.3 A Prolog implementation

Implementing the virtual machine itself

```
method(Agent.run,
        (loop((Agent.sense,
               (ist(Agent,plan(Plan))
                -> Agent.react(Plan);
                Agent.noOp(noPlan))))))].
```

2.3.3 A Prolog implementation

Implementing the virtual machine itself

Extra-logical simulations:

```
interrupt(P) :- getb(C),
                (C =13
                 -> read(P);
                false).
```

This will allow to simulate an external interrupt by hitting the *enter* key, and the passing of time by hitting any other key.

2.3.3 A Prolog implementation

Implementing the virtual machine itself

Extra-logical simulations:

```
loop(P) :- repeat, call((P,!)),fail.
```

Don't ask how it works !!!

2.3.3 A Prolog implementation

Implementing the virtual machine itself

Bootstrap:

```
Agent.newAgent:- new(Agent),
                 insertList(Agent,machine),
                 insertList(Agent,actions),
                 insertList(Agent,plans).
```


2.3.3 A Prolog implementation

Implementing the virtual machine itself

Example:

```
| ?- robot.newAgent.
yes
| ?- robot.run.
|:in(0,0).
|:facing(north).
robot . pause
robot . pause
|: dirt(1,1).
robot . forward
robot . forward
```

```
robot . turndown
robot . forward
robot . suck
robot . forward
robot . turn
robot . forward
robot . turn
robot . pause
robot . pause
…
```