2.1 An abstract model of reactive agents with sensing

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

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

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

- $L = \{l_1, l_2, \ldots\}$ is a set of local states for the agent
- $Act = \{a_1, a_2, \ldots\}$ is a set of actions
- $see : E \to P$ is the perception function
- $\tau_a : L \times P \to L$ is the state transformer function
- $do : L \to Act$ is the action selection function,
- $l_0 \in L$ 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., \( g_i = (l_i, e_i) \)

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

\[
\forall i, g_i = (\tau_a(l_{i-1}, see(e_{i-1})), \tau_e(e_{i-1}, do(l_i)))
\]

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 \( \tau_a \) and \( \tau_e \), 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.,

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

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

We can define these procedures as follows:

- **procedure sense**(l,e)
  if “the agent receives the percept p”
  then $l \leftarrow \tau_{a}(l,p)$

- **procedure react**(e,l)
  if $l \leftarrow \text{do}(a)$
  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:

\[
\text{procedure } \text{run}(e,l) \\
\text{loop } \text{sense}(l,e); \\
\text{react}(e,l)
\]

Although environments are supposed to evolve deterministically, the choice to be made among applicable actions is left unspecified. Consequently, the \( \text{run} \) 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.

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

For any agent, its current nd-plan $p \in P$ refers to a set of implications “$\text{conditions} \Rightarrow \text{do}(p, a)$” or “$\text{conditions} \Rightarrow \text{switch}(p, p')$”, 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 $l \vdash \text{plan}(p_0)$.
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.

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

\[
\text{plan}(\text{initial})
\]

\[
\text{alarm} \Rightarrow \text{switch}(\text{initial}, \text{pause})
\]

\[
\neg \text{alarm} \Rightarrow \text{switch}(\text{initial}, \text{start}) \Rightarrow \text{switch}(\text{start}, \text{work}) \Rightarrow \text{switch}(\text{start}, \text{home}) \Rightarrow \text{do}(\text{pause}, \text{stop})
\]

\[
\text{in}(X,Y) \land \text{dirt}(X,Y) \Rightarrow \text{do}(\text{work}, \text{suck}(X,Y))
\]

\[
\text{in}(X,Y) \land \neg \text{dirt}(X,Y) \Rightarrow \text{do}(\text{work}, \text{move}(X,Y))
\]

\[
\text{in}(X,Y) \Rightarrow \text{do}(\text{home}, \text{back}(X,Y))
\]
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:

```plaintext
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')
```

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

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.

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

```plaintext
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)
```

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.,
  \[ l \vdash plan(p_0) \text{ and } l \vdash do(p, a) \]
- the state transformer functions i.e.,
  \[ \tau_e(e,a) \text{ and } \tau_a(l,p) \]
- 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 \( L \)

Predicate

\[ \text{instance} : O \times L \rightarrow boolean \] true if the object contains an instance of the formula

Operations

\[ \text{new} : \rightarrow O \] creates an empty object
\[ \text{insert} : O \times L \rightarrow O \] inserts a formula into the object
\[ \text{remove} : O \times L \rightarrow O \] removes all instances of a formula
\[ \text{insertList} : O \times List_L \rightarrow O \] inserts a list of formulas
2.3.3 A Prolog implementation

An ADT for objects holding logical formulas

Any formula $P$ of agent $A$ is asserted as $\text{instance}(A,P)$ (where $A$ is the actual name of the agent)

\begin{verbatim}
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)).
\end{verbatim}

Example plans:

\[
\begin{aligned}
\text{plan}\left(\text{initial}\right),
\text{alarm} & \Rightarrow \text{switch}\left(\text{initial},\text{pause}\right),
\text{not} \text{ alarm} & \Rightarrow \text{switch}\left(\text{initial},\text{start}\right),
\text{dirt}(\_,\_)& \Rightarrow \text{switch}\left(\text{start},\text{work}\right),
\text{not} \text{ dirt}(\_,\_)& \Rightarrow \text{switch}\left(\text{start},\text{home}\right),
\ldots
\end{aligned}
\]

\[
\text{insertList}\left(\text{robot},\text{plans}\right).
\]
2.3.3 A Prolog implementation

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

\begin{verbatim}
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).
\end{verbatim}

\begin{verbatim}
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.
\end{verbatim}
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

**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:* \( \text{Agent.Call} \)

*Example:* \( \text{robot.suck}(1,1) \)

Messages are interpreted as

\[
\text{Agent.Call} \leftarrow \text{instance(Agent,}
\text{method(Agent.Call,Body),}
\text{call(Body)}.\right)
\]

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

\[
\text{machine:}
\text{[method(Agent.sense,}
\text{(interrupt(Call)
\to (instance(Agent,}
\text{method(Agent.Call,Body))
\to call(Body);}
\text{insert(Agent,Call));}
\text{true))],}
\]

2.3.3 A Prolog implementation

Implementing the virtual machine itself

\[
\text{method} (\text{Agent}.\text{react}(\text{Plan}),
\begin{array}{l}
\text{ist}(\text{Agent}, \text{do}(\text{Plan}, \text{Action}))
\rightarrow \text{Agent}.\text{Action};
\text{ist}(\text{Agent}, \text{switch}(\text{Plan}, \text{NewPlan}))
\rightarrow \text{Agent}.\text{react}(\text{NewPlan});
\text{Agent}.\text{noOp}(\text{noAction}))
\end{array},
\]
2.3.3 A Prolog implementation

Implementing the virtual machine itself

Extra-logical simulations:

\[
\text{interrupt}(P) :- \ \text{getb}(C), \\
\quad (C = 13 \\
\qquad \rightarrow \ \text{read}(P); \\
\qquad \text{false}).
\]

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

\[
\text{loop}(P) :- \ \text{repeat}, \ \text{call}((P,!) ), \text{fail}.
\]

\textit{Don’t ask how it works !!!}
2.3.3 A Prolog implementation

Implementing the virtual machine itself

*Bootstrap:*

\[
\text{Agent.newAgent:- new(Agent),} \\
\quad \text{insertList(Agent,machine),} \\
\quad \text{insertList(Agent,actions),} \\
\quad \text{insertList(Agent,plans).}
\]

---

2.3.3 A Prolog implementation

Implementing the virtual machine itself

*Example:*

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