MultiAgents Systems

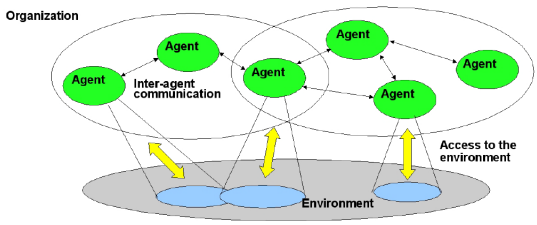
Riccardo Caprile

October 2022

1 Introduction

An **agent** is an hardware or software system :

- Situated
- Autonomous (Act without a continuous input)
- Flexible (reactive , proactive , social)



Each agent has incomplete information , or capabilities for solving the problem , thus each agent has a limited viewpoint.

There is no global system control. Data is decentralized and computation is asynchronous.

Agents are employed as the technology for implementing the software application itself.

Among the oldest areas of research , the activity most closely connected with that of autonomous agents was AI "planning".

A computer system either conceptualized or implemented using concepts that are more usually applied to humans, this is a stronger definition than the one above.

2 Agent Oriented Software Engineering

The features of agent-based systems are well suited to tackle the complexity of developing software in modern scenarios :

1. The autonomy of application components reflects the intrinsically decentralised nature of modern distributed systems and can be considered as the natural extension to the notions of modularity and encapsulation for systems that are owned by different stakeholders

2. The flexible way in which agents operate and interact is suited to the dynamic and unpredictable scenarios where software is expected to operate.

3. The concept of agency provides for a unified view of AI results and achievements , by making agents and MASs act as sound and manageable repositories of intelligent behaviours.

2.1 Autonomy

As far as autonomy is concerned , almost all of today's software systems already integrate autonomous components.

At its weakest, autonomy reduces to the ability of a component to react to and handle events, as in the case of graphical interfaces.

However,in many cases, autonomy implies that a component integrates an autonomous thread of execution, and can execute in a proactive way

2.2 Situatedness

Today's computing systems are also typically situated.

For example, control systems for physical domains and sensor networks, are built to explicitly manage data from the surrounding physical environment and take into account the unpredictable dynamics of the environment.

2.3 Sociality

Sociality in modern distributed systems comes in different flavors :

1. The capability of components of supporting dynamic interactions

2. The somewhat higher interaction level , overcoming the traditional client-server scheme

3. The enforcement of some sorts of societal rules governing interactions

3 Some Applications of AI and MASs

The first application can be autonomous driving , but it has some technological issues. Among the many technologies which make autonomous vechicles possible is a combination of sensors and actuators , sophistaced algorithms and powerful processors to execute software.

Three categories of sensors : 1) Navigation and guidance (where you are , how to get there) , 2) Driving and safety (directing the vehicle with rules of the road) , 3) performance.

3.1 Goal 1 : Know where you are going

The navigation and guidance subsystem must always be active and checking how the vehicle is doing versus the goal.

GPS (Global Positioning System) computes present position based on the analysis of signals received from at least four of the constellation of over 60 low-orbit satellites.

GPS is not sufficient alone because of the interferences.

To supplement the GPS, the autonomous vehicle uses inertial guidance which requires no external signal of any type.

3.2 Goal 2 : See where you are going

The autonomous car must be able to see and interpret what's in front when going forward.

But using cameras alone presents problems. First, there are mechanical issues of setting up multiple cameras correctly and keeping them clean; second, heavy graphic processing is needed to make sense of images; third, there is a need for depth perception as well as basic imaging; and finally, conditions of lighting, shadows, and other factors make it very challenging to accurately decide what the camera is seeing.

To overcome camera limitations , in most self driving cars the primary vision unit is LIDAR system.

A surveying method that measures distance to a target by illuminating that target with a pulsed laser light, and measuring the reflected pulses with a sensor.

The LIDAR output is used to calculate the vehicles position , speed , and direction relative to these external objects to determine the probability of collision , and instruct appropriate action, if needed.

3.3 Goal 3: Get where you are going

While components and subsystems used for navigation and guidance or for image-capture and sensing get the most attention due to their glamour aspects, a large portion of the design of an autonomous vehicle involves mundane issues such as power management.

Other application for MAS for example is FYPA (Find your path , Agent!) which computes paths of trains inside stations. Graph based.

UNMAS has the purpose to rapidly find a sub-optimal scenario able to maintain the whole electricity network configuration in a consistent state wrt an event such as a maintenance operation or a mulfunctioning

4 Agent Architectures

"An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors"

Basic abstract view of an agent

4.1 Reactivity

An agent has to be able to react in an appropriate way to the dynamic changes in its environment. This is one of several properties that an intelligent agent should have.

4.1.1 Kinds of environments

An **accessible** environment is one in which the agent can obtain complete, accurate , up-to-date information about environment's state.

Most moderately complex environments are inacessible.

The more accessible an environment is , the simpler it is to build agents to operate in it.

A **deterministic** environment is one in which any action has a single guaranteed effect, there is no uncertainty about the state that will result from performing an action.

The physical world can to all intents and purposes be regarded as **non-deterministic.**

Non-deterministic environments present greater problems for the agent designer.

In an **episodic** environments, the performance of an agent is dependent on a number of discrete episodes, with no link between the performance of an agent in different scenarios.

Episodic environments are simpler from the agent developer's perspective because the agent can decide what action to perform based only on the current

episode , it need not reason about the interactions between this and future episodes.

A **static** environment is one that can be assumed to remain unchanged except by the performance of actions by the agent.

A **dynamic** environment is one that has other processes operating on it , and which hence changes in ways beyond the agent's control.

The physical world is a highly dynamic environment.

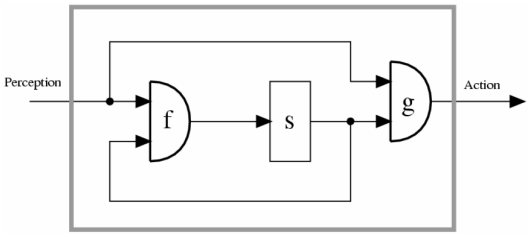
An environment is **discrete** if there are a fixed , finite number of actions and percepts in it.

The real world is a **continuous** environment.

4.2 Agent architectures

An architecture proposes a particular methodology for building autonomous agent.

How the construction of the agent can be decomposed into the construction of a set of component modules. How these modules should be made to interact. These two aspects define how the sensor data and the current internal state of the agent determine the actions and future internal state of the agent.



f = state update function
s = internal state
g = output function

4.2.1 Main kinds of agent architectures

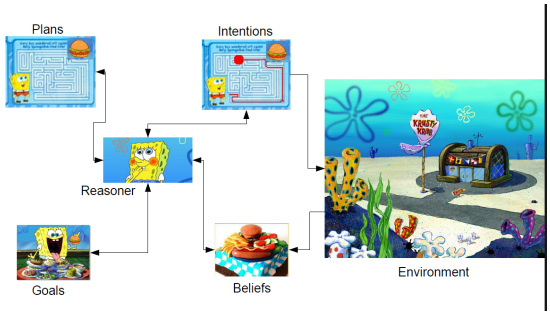
- **Reactive architectures** : Focused on fast reactions/response to changes detected in the environment
- **Deliberative architectures** : Focused on long-term planning of actions , centred on a set of basic goals.
- **Hybrid architectures** : Combining a reactive side and a deliberative side

4.2.2 Classic example : ant colony

A single ant has very little intelligence, computing power or reasoning abilities.

The union of a set of ants and the interaction between them allows the formation of a highly complex , structured and efficient system.

4.3 BDI-Architecture



5 BDI Architecture

The BDI architecture is one of the best known and most studied architectures for cognitive agents.

AgentSpeack(L) is an elegant , logic-based programming language inspired by the BDI architecture.

5.1 AgentSpeak(L) syntax

An AgentSpeak(L) is an agent created by the specification of a set of base beliefs and a set of plans. A belief atom is a simply first order predicate in the usual notation , and belief atoms or their negations are termed belief literals.

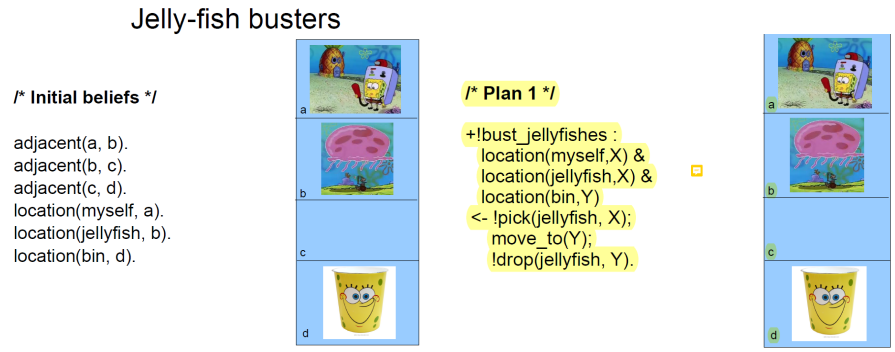
BDI architectures are based on the following constructs :

- A set of beliefs
- A set of goals
- A set of intentions , or better a subset of the goals with an associated stack of plans for achieving them.
- A set of internal events : elicited by a belief change (update,addition,deletion) or by goal events(goal achievement or a new gol adoption)
- A set of external events , perceptive events coming from the interaction with external entities(message arrival ,signals).
- A plan library

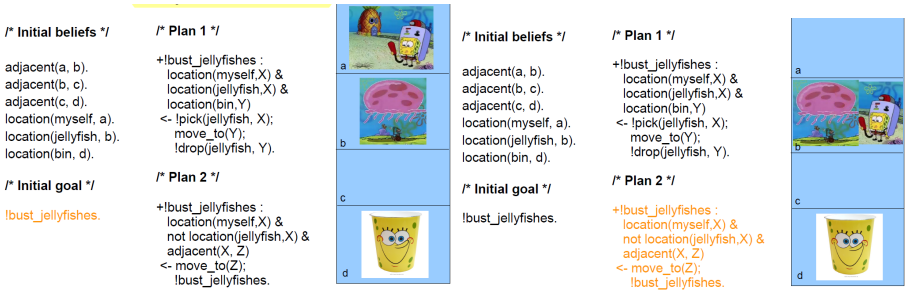
$$
\begin{array}{lll}
ag & ::= bs \quad ps & \\
bs & ::= at_1 \ldots at_n & (n \geq 0) \\
at & ::= P(t_1, \ldots t_n) & (n \geq 0) \\
ps & ::= p_1 \ldots p_n & (n \geq 1) \\
p & ::= te : ct \leftarrow h & \\
te & ::= +at \mid -at \mid +g \mid -g & \\
ct & ::= at \mid \neg at \mid ct \ \wedge \ ct \mid \mathsf{T} & \\
h & ::= a \mid g \mid u \mid h; h & \\
g & ::= !at \mid \ ?at & \\
u & ::= +at \mid -at &
\end{array}
$$

Ag stands for agent , bs = beliefs state , ps = plan set , p = plan , h = body , g = goal , at = atom , u = update , te = triggerin event , ct = context

5.2 Jelly-fish Busters



myself is at location X and jellyfish too, the agent needs to pick the jellyfish in location X , then the agent moves to Y and it drops the jellyfish into the bin We cannot use this plan, at the beginning, because in the beliefs jellyfish and the agent are not in the same location



Then it follows plan 1.

6 Uninformed Search

6.1 Problem-Solving agents

Restricted form of general agent :

```
function SIMPLE-PROBLEM-SOLVING-AGENT (percept) returns an action
    static seq, an action sequence, initially empty
        state, some description of the current world state
        goal, a goal, initially null
        problem, a problem formulation
    state ← UPDATE_STATE(state,percept)
    if seq is empty then
        goal ← FORMULATE_GOAL(state)
        problem ← FORMULATE_PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← RECCOMENDATION(seq,state)
    seq ← REMAINDER(seq,state)
    return action
```

Example : Romania

On holiday in Romania; currently in Arad. Flight leaves tomorrow from bucharest.

Formulate goal : be in Bucharest

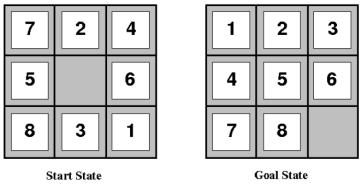
Formulate problem : States (various cities) , actions (drive between cities)

Find solution : sequence of cities

6.2 Problem Types

- **Deterministic , fully observable** : Single state problem , Agent knows exactly which state it will be in; solution is a sequence.
- **Non Observable** : Conformant problem, Agent may have no idea where it is; solution (if any) is a sequence.
- **Nondeterministc** : Contigency problem, Percepts provide new information about current state, solution is a contigent plan or policy
- **Unknown state space** : Exploration problem (Online)

6.2.1 The 8-puzzle



States??: integer locations of tiles (ignore intermediate positions.)
Actions??: move blank left, right, up, down (ignore unjamming etc.)
Goal Test??: goal state (given)
Path Cost??: 1 per move

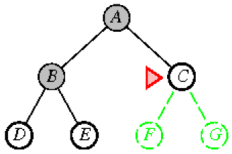
6.3 Uninformed Search Strategies

Uninformed strategies use only the information available in the problem definition.

6.3.1 Breadth-first search

Expand shallowest unexpanded node

Implementation:
fringe is a FIFO queue, i.e., new successors go at end

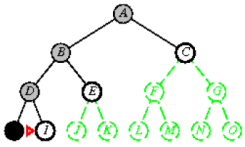


6.3.2 Uniform-cost search

Implementation : fringe = queue ordered by path cost, lowest first.

6.3.3 Depth-first search

Expand deepest unexpanded node
Implementation:
fringe = LIFO queue, i.e., put successors at front



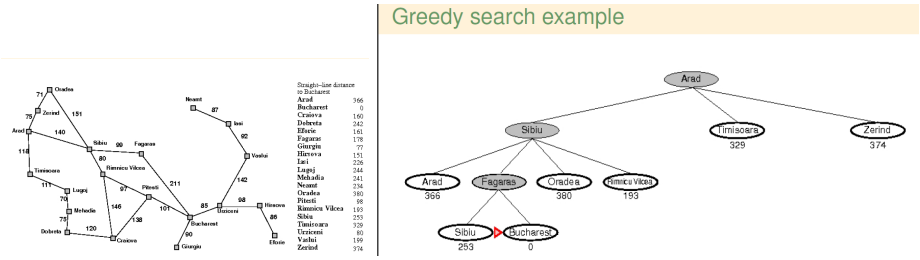
7 Informed Search

7.1 Best-first search

Idea : use an evaluation function for each node - estimate of "desirability". Expand most desirable node

Implementation : fringe is a queue sorted in decreasing order of desirability.

Special cases : greedy search and A* search.



7.1.1 Greedy search

Evaluation function h(n) = estimate of cost from n to the closest goal.

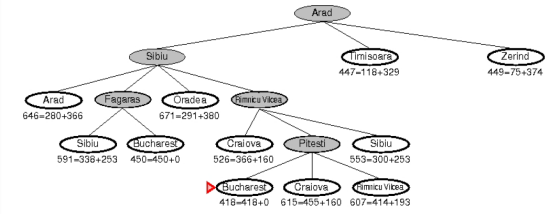
Greedy search expands the node that appears to be closest to goal.

7.2 A* search

Idea : avoid expanding paths that are already expensive.

Evaluation function f(n) = g(n) + h(n)

g(n) is the cost so far to reach n , h(n) estimated cost to goal from n , f(n) estimated total cost of path through n to goal.



8 Game Playing

8.1 Minimax

Perfect play for deterministic , perfect-information games.

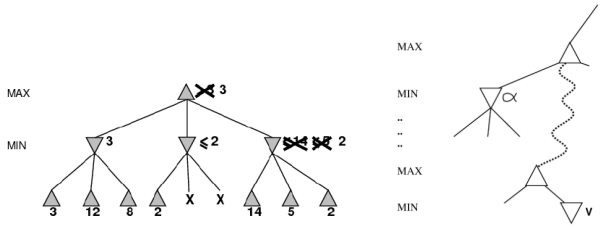
Idea : choose move to position with the highest minimax value (best achievable payoff against best play)

8.1.1 $\alpha - \beta$ pruning example

α is the best value (to MAX) found so far off the current path.

If V is worse than α , MAX will avoid it : prune that branch.

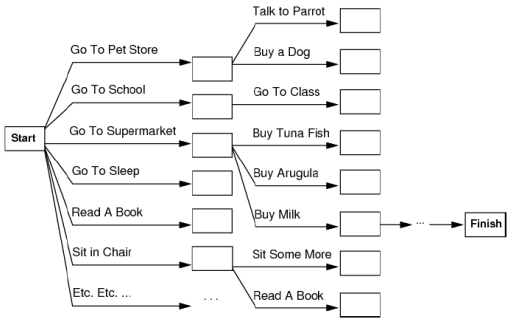
Define β similarly for MIN



9 Planning

9.1 Search vs Planning

Consider the task "get milk , bananas and a cordless drill". Standard search algorithms seem to fail miserably :



9.2 STRIPS

STRIPS (Stanford Research Institute Problem Solver) is an automated planner.

The same name was later used to refer to the formal language of the inputs to this planner

9.2.1 States and Goals

States are a conjuction of positive literals. Literals may be propositional ones , such as Poor $\wedge$ Unknown , or first order ones , like At(Plane1,Melbourne) $\wedge$ At(Plane2,Sydney).

In this latter case , they must be ground and function free. Function freedom ensures that any action schema for a given problem can be turned into a finite collection of purely propositional action representations with no variables.

Goals are partially specified states, represented as conjunctions of positive ground literals , such as Rich $\wedge$ Famous or At(Plane2,Tahiti).

9.2.2 Action Schemas

An action is specified in terms of the preconditions that must hold before it can be executed and the effects that ensue when it is executed.

Action(fly(p,from,to)
PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
EFFECT : ¬ At(p.from) ∧ At(p,to))

9.2.3 Applicable actions and their results

An action is **applicable** in any state that satisfies its precondition. For example , consider the current state described by :

At(P1,JFK) ∧ At(P2,SFO) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)

This state satisfies the PRECOND , so the action described above is applicable to the current state

9.2.4 Action Results

Starting in state s, the **result** of executing an applicable action a is a state s' that is the same as s except that any positive literal P in the effect of a is added to s' and any negative literal ¬P is removed from s' .

For example , after performing Fly(P1,JFK,SFO), the current state becomes At(P1,SFO) ∧ At(P2,SFO) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO).

Note that if a positive effect is already in s it is not added twice , and if a negative effect is not in s , then that part of the effect is ignored. Also , every literal not mentioned in the effect remains unchanged.

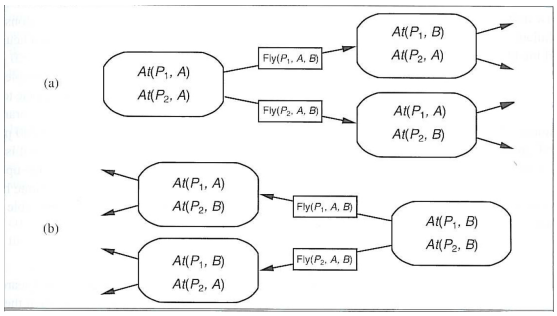
The solution for a planning problem is just an action sequence that , when executed in the initial state , result in a state that satisfies the goal.

9.2.5 Example : Transportation of air cargoes

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
EFFECT: $\neg At(p, from) \land At(p, to))$

Figure 11.2 A STRIPS problem involving transportation of air cargo between airports.

9.3 Searching for a plan



A) Forward progression stat-space search , starting in the initial state and using the problem's actions to search forward for the goal state.

B) Backward state-space search : a belief-state search starting at the goal state and using the inverse of the actions to search backward for the initial state.

9.4 Partial-order Planning

9.4.1 State space vs. plan space

Standard search : node = concrete world state

Planning search: node = partial plan

Defn : Open condition is a precondition of a step not yet fulfilled.

Operators on partial plans : Add a link from an existing action to an open condition, add a step to fulfil an open condition , order one step wrt another.

9.4.2 Partially ordered plans

Each partial plan has the following components :

1. A set of actions that make up the steps of the plan. They are taken from the set of actions in the planning problem. The empty plan just include Start and Finish.

2. A set of ordering constraints of the form A ¡ B , read as "A before B"

3. A set of causal links written $A- > pB$ read as "A achieves p for B". p is an effect of action A , and a precondition of action B , and must remain true from the time of action A to the time of action B.

4. A set of open preconditions , namely preconditions not achieved by some action in the plan.

9.5 Consistent plans, solutions , plan space

A **consistent plan** is a plan that contains no cycles in the ordering constraints and no conflicts with the causal links.

A consistent plan with no open preconditions is a **solutions**

9.5.1 Plan space

The graph space to search for solving the POP problem is defined as follows :

1. Nodes are partial plans
2. The initial plan contains Start and Finish , Start ¡ Finish, no causal links , and all the preconditions of Finish are open preconditions.
3. Given one partial plan , its children in the plan graph are obtained by selecting one open precondition p on an action B and generating all the partial plans for every possible consistent way of choosing an action A that achieves p
4. Since partial plans generated in step 3 are consistent, the goal test just needs to check that there are no open preconditions.

9.5.2 Planning process

Operators on partial plans:

- Add a link from an existing action to an open condition
- Add a step to fulfill an open condition
- Order one step wrt another to remove possible conflicts

Gradually move from incomplete/vague plans to complete.
Backtrack if an open condition is unachievable.

9.5.3 Clobbering and promotion/demotion

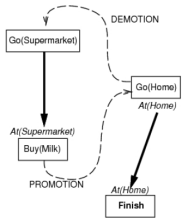
A **clobberer** is a potentially intervening step that destroys the condition achieved by a causal link.

9.6 Properties of POP

Nondeterministic algorithm : backstracks at choice points on failure:

- Choice of S_{add} to achieve S_need
- Choice of demotion or promotion for clobberer
- Selection of S_{need} is irrevocable

POP is sound, complete and systematic



Demotion: put before *Go*(*Supermarket*)

Promotion: put after *Buy*(*Milk*)

POP algorithm sketch

function POP(*initial*, *goal*, *operators*) **returns** *plan*
plan ← MAKE-MINIMAL-PLAN(*initial*, *goal*)

loop do
if SOLUTION?(*plan*) **then return** *plan*
$S_{need}, c \leftarrow$ SELECT-SUBGOAL(*plan*)
CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, c)
RESOLVE-THREATS(*plan*)
end

function SELECT-SUBGOAL(*plan*) **returns** S_{need}, c
pick a plan step S_{need} from STEPS(*plan*)

with a precondition c that has not been achieved
return S_{need}, c

POP algorithm contd.

procedure CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, c)
choose a step S_{add} from *operators* or STEPS(*plan*) that has c as an effect

if there is no such step **then fail**
add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
if S_{add} is a newly added step from *operators* **then**
add S_{add} to STEPS(*plan*)
add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

procedure RESOLVE-THREATS(*plan*)
for each S_{threat} that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS(*plan*) **do**

choose either
Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
Promotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
if not CONSISTENT(*plan*) **then fail**
end

9.6.1 POP Example : putting shoes on

The "shoes dressing" problem description

$Goal(RightShoeOn \wedge LeftShoeOn)$
$Init()$
$Action(RightShoe, \text{PRECOND:}RightSockOn, \text{EFFECT:}RightShoeOn)$
$Action(RightSock, \text{EFFECT:}RightSockOn)$
$Action(LeftShoe, \text{PRECOND:}LeftSockOn, \text{EFFECT:}LeftShoeOn)$
$Action(LeftSock, \text{EFFECT:}LeftSockOn)$.

Partial order plan

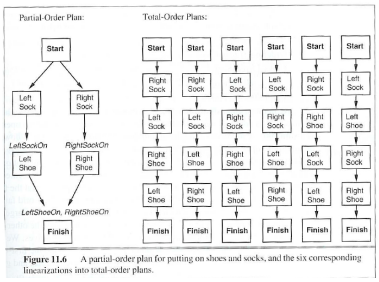


Figure 11.6 A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans.

Components of the plan shown in the previous figure

Actions:$\{RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish\}$
Orderings:$\{RightSock \prec RightShoe, LeftSock \prec LeftShoe\}$
Links:$\{RightSock \xrightarrow{RightSockOn} RightShoe, LeftSock \xrightarrow{LeftSockOn} LeftShoe,$
$RightShoe \xrightarrow{RightShoeOn} Finish, LeftShoe \xrightarrow{LeftShoeOn} Finish\}$
Open Preconditions:$\{\}$.

10 Communication as Action

Communication , in general , is the intentional exchange of information brought about by the production and perception of signs drawn from a shared system of conventional signs

10.1 Speech acts

Speaker → Utterance → Hearer

Speech acts achieve the speaker's goals:

- Inform : "There's a pit in front of you"
- Query : "Can you see the gold?"
- Command: "Pick it up"
- Promise : "I'll share the gold with you"
- Acknowledge : "OK"

10.1.1 Stages in communication

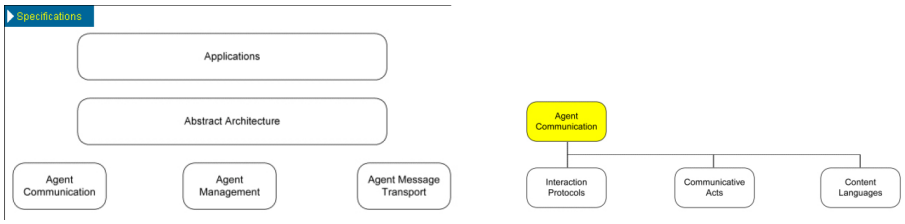
- Intention : S wants to inform H that P
- Generation : S select words W to express P in context C
- S utters word W
- Perception : H perceives W' in context C'
- Analysis :H infers possible meanings $P_1, ..., P_n$
- Disambiguation :H infers intended meaning P_i
- Incorporation : H incorporates P_i into KB

How could this go wrong?

Insincerity (S does not believe P) , speech wreck(noise in the transmission) , ambiguous utterance.

10.2 FIPA specifications : an overview and communication

Accept proposal : The action of accepting a previously submitted proposal to perform an action



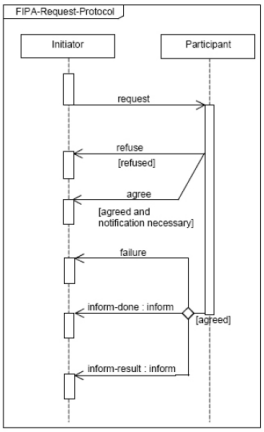
FIPA Message Structure

Parameter	Category of Parameters
performative	Type of communicative acts
sender	Participant in communication
receiver	Participant in communication
reply-to	Participant in communication
content	Content of message
language	Description of Content
encoding	Description of Content
ontology	Description of Content
protocol	Control of conversation
conversation-id	Control of conversation
reply-with	Control of conversation
in-reply-to	Control of conversation
reply-by	Control of conversation

10.2.1 FIPA Ontology Agent

The model of agent communication in FIPA is based on the assumption that two agents , who wish to converse , share a common ontology for the domain of discourse. It ensures that the agents ascribe the same meaning to the symbols used in the message.

10.2.2 FIPA Request Interaction Protocol



10.2.3 Example 1

Problem : We want to represent the following agent interaction protocol involving Alice and Bob. Alice greets Bob with some nice words that may be either "Hello" or "Good Morning" or "Ciao". Bob either answers with some similar greeting , or says "Sgrunt!" if he is angry. The protocol stops.

Stage 1 . performatives and content language to be used : we may use "inform", even if it is not the very best choice, since no more specific performative exists for greeting. We may use simple words or phrases in natural language as content language.

Stage 2 : Grouping messages that can be used interchangeably :

the three messages
$alice \overset{inform(''Hallo'')}{\Longrightarrow} bob$
$alice \overset{inform(''Goodmorning'')}{\Longrightarrow} bob$
$alice \overset{inform(''Ciao'')}{\Longrightarrow} bob$
can all be used for starting the protocol. We may abstract this notion of "having the same message type" (more general, event type) by saying that there exists an event type *agb* (for Alice greets Bob), and that the three messages above have that type.
$agb = \{alice \overset{inform(''Hallo'')}{\Longrightarrow} bob, alice \overset{inform(''Goodmorning'')}{\Longrightarrow} bob, alice \overset{inform(''Ciao'')}{\Longrightarrow} bob\}$.
In the same way we may define
$bga = \{bob \overset{inform(''Hallo'')}{\Longrightarrow} alice, bob \overset{inform(''Goodmorning'')}{\Longrightarrow} alice, bob \overset{inform(''Ciao'')}{\Longrightarrow} alice\}$ and
$bsa = \{bob \overset{inform(''Sgrunt!'')}{\Longrightarrow} alice\}$.

Stage 3 : Expressing protocol AIP1 in terms of event types : AIP1 = agb:(bga:$\epsilon\vee$ bsa: ϵ)

Semicolon means sequence. Alice greets Bob then Bob greets Alice and terminates (epsilon) or bob sgrunt Alice and terminates.

Stage 4 : Understanding the operational semantics :

At the beginning, the state of the protocol is exactly *AIP*1; if anyone of the messages in *agb*, namely $alice \overset{inform(''Hallo'')}{\Longrightarrow} bob, alice \overset{inform(''Goodmorning'')}{\Longrightarrow} bob, alice \overset{inform(''Ciao'')}{\Longrightarrow} bob$ is exchanged, then the protocol is respected (so far...) and can move to its next state, ($bga{:}\epsilon \vee bsa{:}\epsilon$)
From here, the protocol is respected if either one message in *bga* is exchanged or one message in *bsa* is.
If this is the case, the protocol moves to ϵ, meaning that it terminates.

Stage 5 : Understanding the denotational semantics :

Which are the traces of messages that respect the protocol?
$\{ alice \overset{inform(''Hallo'')}{\Longrightarrow} bob \quad bob \overset{inform(''Hallo'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Hallo'')}{\Longrightarrow} bob \quad bob \overset{inform(''Goodmorning'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Hallo'')}{\Longrightarrow} bob \quad bob \overset{inform(''Ciao'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Hallo'')}{\Longrightarrow} bob \quad bob \overset{inform(''Sgrunt!'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Goodmorning'')}{\Longrightarrow} bob \quad bob \overset{inform(''Hallo'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Goodmorning'')}{\Longrightarrow} bob \quad bob \overset{inform(''Goodmorning'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Goodmorning'')}{\Longrightarrow} bob \quad bob \overset{inform(''Ciao'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Goodmorning'')}{\Longrightarrow} bob \quad bob \overset{inform(''Sgrunt!'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Ciao'')}{\Longrightarrow} bob \quad bob \overset{inform(''Hallo'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Ciao'')}{\Longrightarrow} bob \quad bob \overset{inform(''Goodmorning'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Ciao'')}{\Longrightarrow} bob \quad bob \overset{inform(''Ciao'')}{\Longrightarrow} alice,$
$alice \overset{inform(''Ciao'')}{\Longrightarrow} bob \quad bob \overset{inform(''Sgrunt!'')}{\Longrightarrow} alice \}$

11 Agent Speak Language

Beliefs represent the information available to an agent. (publisher(wiley))

Goals represent states of affairs the agent wants to realize. Achievement goals : (**!write(book)**)

Or attempts to retrieve from the belief base : Test goals (?publisher(P))

An agent reacts to **events** by executing plans. Events happen as a consequence to changes in the agent's beliefs or goals.

Plans are recipes for action , representing the agent's know-how. An AgentSpeak plan has the following general structure :

triggering_event : context $<-$ body

Where : **triggering_event** denotes the events that the plan is meant to handle. **context** represent the circumstances in which the plan can be used. **body** is the course of action to be used to handle the event if the context is believed true at the time a plan is being chosen to handle the event

AgentsSpeak triggering events:

- +b (belief addition)
- -b (belief deletion)
- +!g (achievement-goal addition)
- -!g (achievement-goal deletion)
- +?g (test-goal addition)
- -?g (test-goal deletion)

The context is logical expression , typically a conjunction of literals to be checked whether they follow from the current state of the belief base. The body is a sequence of action and (sub)goals to achieve.

12 Jade

During the years , various tools have been proposed to transform standard MAS specifications into actual agent code , and a variety of middleware have been deployed to provide proper services supporting the execution of distributed MASs

In order to be interoperable , such infrastructures should be compliant with the FIPA abstract architecture.

JADE is a software framework fully implemented in Java. It simplifies the implementation of MASs through a middleware that complies with the FIPA specifications and through a set of tools that supports the debugging and deployment phases.

12.1 More details on JADE : the Halloworld agent

A **type of agent** is created by extending the jade.core.Agent class and redefining the setup() method.

Each **Agent instance** is identified by an AID (jade.core.AID).

An AID is composed of a unique name plus some addresses. An agent can retrieve its AID through the getAID() method of the Agent Class

```
import jade.core.Agent;

public class HalloWorldAgent extends Agent {

  protected void setup() {
    System.out.println("Hallo World! my name is "+getAID().getName());
  }
}
```

12.2 Local names , GUID and addresses

Agent names are of the form $< local - name > @ < platform - name >$.

The complete name of an agent must be globally unique.

The default platform name is main-host : main-port/JADE

The platform name can be set using -name option.

Within a single JADE platform agents are referred through their names only.

Given the name of an agent its AID can be created as : AID id = new AID(localname, AID.ISLOCALNAME); or AID id = new AID(name,AID.ISGUID);

The addresses included in an AID are those of the platform MTPs and are only used in communication between agents living on different FIPA platforms.

12.3 Passing arguments to an agent

It is possible to pass arguments to an agent.

java jade.Boot .... a:mypackage.MyAgent(arg1 arg2)

The agent can retrieve its arguments through the getArguments() method of the Agent class.

```
protected void setup() {
  System.out.println("Hallo World! my name is "+getAID().getName());
  Object[] args = getArguments();
  if (args != null) {
    System.out.println("My arguments are:");
    for (int i = 0; i < args.length; ++i) {
      System.out.println("- "+args[i]);
    }
  }
}
```

12.4 Agent Termination

An agent terminates when its **doDelete()** method is called.

On termination the agent's **takeDown()** method is invoked (intended to include clean-up operations).

```
protected void setup() {
  System.out.println("Hallo World! my name is "+getAID().getName());
  Object[] args = getArguments();
  if (args != null) {
    System.out.println("My arguments are:");
    for (int i = 0; i < args.length; ++i) {
      System.out.println("- "+args[i]);
    }
  }
  doDelete();
}

protected void takeDown() {
  System.out.println("Bye...");
}
```

12.5 The Behaviour class

The actual job that an agent does is typically carried out within "**behaviours**"

Behaviours are created by extending the jade.core.behaviours.Behaviour class.

To make an agent execute a task it is sufficient to create an instance of the corresponding Behaviour subclass and call the addBehaviour() method of the Agent class.

Each Behaviour subclass must implement :

- public void action() : what the behaviour actually does
- public boolean done(): whether the behaviour is finished

12.6 Behaviour scheduling and execution

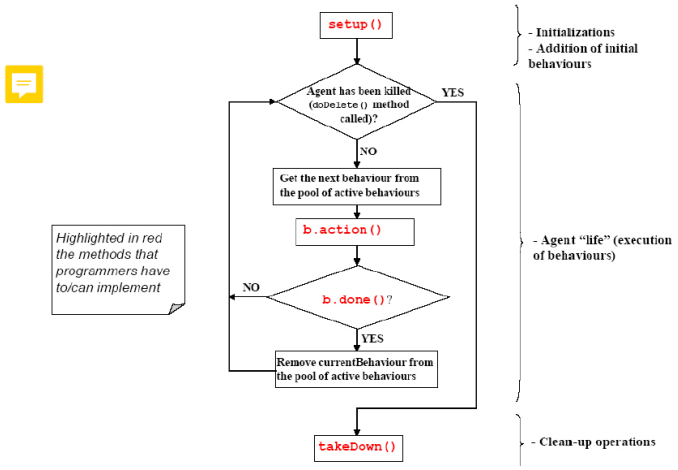
An agent can execute several behaviours in parallel , however , behaviour scheduling is not preemptive , but cooperative and everything occurs within a single Jave Thread.

Behaviour switch occurs only when the action() method of the currently scheduled behaviour returns

12.7 The agent execution model

12.8 Behaviour types

- One shot behaviours : Complete immediately and their action() method is executed only once. Their done() method simply returns true. jade.core.behaviours.OneShotBehaviour class.
- Cyclic behaviours : Never complete and their action() method executes the same operation each time is invoked. Their done() method simply returns false. jade.core.behaviours.CyclicBehavior class
- Complex behaviours : Embed a state and execute in their action() method different operation depending on their state. Complete when a given condition is met.



12.9 Scheduling operations at given points in time

JADE provides two ready-made classes by means of which it is possible to easily implement behaviours that execute certain operations at given points in time.

- WakerBehaviour: the action() and done() method are already implemented so that the onWake() method (to be implemented) is executed after a given timeout. After that execution the behaviour completes.
- TicketBehaviour: The action() and done() method are already implemented so that onTick()(to be implemented by subclasses) method is executed periodically with a given period. The behaviour runs forever unless its stop() method is executed.

The onStart() method of the Behaviour class is invoked only once before the first execution of the action() method. Suited for operations that must occur at the beginning of the behaviour.

The onEnd() method of the Behaviour class is invoked only once after the done() method returns true. Suited for operations that must occur at the end of the behaviour.

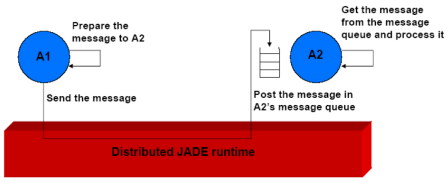
Each behaviour has a pointer to the agent executing it : the protected member variable myAgent.

The removeBehaviour() method of the Agent class can be used to remove a behaviour from the agent pool of behaviours. The onEnd() method is not called.

When the pool of active behaviours of an agent is empty the agent enters the IDLE state and its thread goes to sleep.

12.10 The communication model

Based on asynchronous message passing. Message format defined by the ACL language(FIPA)



12.11 The ACLMessage class

Messages exchanged by agents are instances of the jade.lang.acl.ACLMessage class.

12.11.1 Sending and receiving messages

Sending a message is as simple as creating an ACLMessage object and calling the send() method of the Agent class.

Reading messages from the private message queue is accomplished through the receive() method of the Agent class

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Peter", AID.ISLOCALNAME));
msg.setLanguage("English");
msg.setOntology("Weather-Forecast-Ontology");
msg.setContent("Today it's raining");
send(msg);
```

```
ACLMessage msg = receive();
if (msg != null) {
  // Process the message
}
```

12.12 Blocking a behaviour waiting for a message

A behaviour that processes incoming messages does not know exactly when a message will arrive : it should poll the message queue by continuously calling myAgent.receive().

This of course would completely waste the CPU time.

The block() method of the Behaviour class removes a behaviour from the agent pool and puts it in a blocked state.

Each time a message is received all blocked behaviours are inserted back in the agent pool and have a chance to read and process the message.

```
public void action() {
  ACLMessage msg = myAgent.receive();
  if (msg != null) {
    // Process the message
  }
  else {
    block();
  }
}
```

This is the strongly recommended pattern to receive messages within a behaviour

12.13 Selective reading from the message queue

The receive() method returns the first message in the message queue and removes it.

If there are two (or more) behaviours receiving messages , one may steal a message that the other one was interested in.

To avoid this it is possible to read only messages with certain characteristics specifying a jade.lang.acl.MessageTemplate parameter in the receive() method.

```
MessageTemplate tpl = MessageTemplate.MatchOntology("Test-Ontology");

public void action() {
  ACLMessage msg = myAgent.receive(tpl);
  if (msg != null) {
    // Process the message
  }
  else {
    block();
  }
}
```

12.14 Receiving messages in blocking mode

The Agent class also provides the blockingReceive() method that returns only when there is message in the message queue.

There are overloaded versions that accept a MessageTemplate (the method returns only when there is a message matching the template) and or timeout(if it expires the method returns null).

Since it is blocking call it is dangerous to use blockingReceive() within a behaviour. In fact no other behaviour can run until blockingReceive() returns.

Use receive() +Behaviour.block() to receive messages within behaviours. Use blockingReceive() to receive messages within the agent setup() and takeDowun() methods.

12.15 Interacting with the DF Agent

The DF is an agent and as such it communicates using ACL. The ontology and language that the DF understands are specified by FIPA : it is possible to search/register to a DF agent of a remote platform.

The jade.domain.DFService class provides static utility methods that facilitate the interactions with the DF.

When an agent registers with the DF it must provide a description basically composed of : the agent ID , a collection of service descriptions.

13 NetLogo

NetLogo is a programmable modelling environment for simulating complex systems.

Modelers can give instructions to hundreds of independent agents all operating in parallel.

This makes it possible to explore the connection between : the micro-level behaviour o

13.1 Agents

The NetLogo world is made up of **agents**. Agents are beings that can follow instructions. Each agent can carry out its own activity , all simultaneously.

In NetLogo there are three type of agents :

- Turtles : are agents that move around the world. The world is two dimensional and is divided up into a grid of patches
- Patches : Each patch is a square piece of ground over which turtles can move
- Observer : It does not have a location , you can imagine it as looking out over the world of turtles and patches

13.1.1 Patches

Patches have coordinates. The patch in the center of the world has coordinates (0,0). We call the patch's coordinates pxcor and pycor (integers)

The total number of patches is determined by the setting screen-edge-x and screen-edge-y.

13.1.2 Turtles

Turtles have coordinates too : xcor and ycor.

Each turtle has an identifier **who**

NetLogo always draws a turtle on-screen as if were standing in the center of its patch , but in fact , the turtle can be positioned at any point within the patch

13.2 Primitives

Commands and reporters tell agents what to do :

- **Commands** : are actions for the agents to carry out
- **Reporters** : carry out some operation and report a result

Commands and reporters built in NetLogo are called **Primitives**

13.3 Procedures

Commands and reporters you define yoursel are called **procedures**

Each procedure has a name , preceded by the keyword to . The keyword marks the end of the commands in the procedure.

Once you define a procedure , you can use it elsewhere in your program.

Many commands and reporters take inputs values that the command or reporters uses in carrying out its actions.

```
to setup
   clear-all         ;; clear the screen
   create-turtles 10 ;; create 10 turtles
   reset-ticks
end

to go
  ask turtles [
    fd 1             ;; forward 1 step
    rt random 10     ;; turn right
    lt random 10     ;; turn left
  ]
  tick
end
```

13.3.1 Procedures with input

Procedures can take inputs, just like many primitives do.

To create a procedure that accepts inputs, put their names in square brackets after the procedure name. For example :

```
to draw-polygon [num-sides len]  ;; turtle procedure
  pen-down                       ;; draw when moving
  repeat num-sides [             ;; num_sides interations
    fd len                       ;; move forward len steps
    rt 360 / num-sides           ;; turn right of some degree
  ]
end
```

You might use the procedure by asking the turtles to each draw an octagon with a side length equal to its **who** number

(ask turtles [draw-polygon 8 who])

13.3.2 Reporter Procedures

Just like you can define your own commands, you can define your own reporters. You must do two special things.

First , use **to-report** instead of to to begin your procedure. Then in the body of the procedure, use **report** the value you want to report.

```
to-report absolute-value [number]
  ifelse number >= 0
    [ report number ]
    [ report (- number) ]
end
```

13.4 Variables

A variable can be :

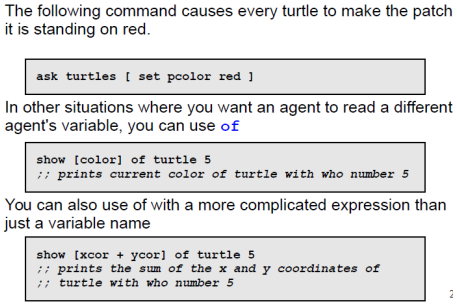
- A global variable : there is only one value for the variable, and every agent can access it
- A local variable : each turtle has its own value for every turtle variable, and each patch has its own value for every patch variable
- Some variables are built in NetLogo : like color

You can make a global variable by adding a switch or a slider to your model , or by using **globals** keyword at the beginning of your code : globals [score]

You can also define new turtle and patch variables using the **turtles-own** and **patches-own** (turtles own [friction]

Use the **set** command to set them.

A turtle can read and set patch variables of the patch it is standing on.



13.4.1 Local Variables

A local variable is defined and used only in the context of a particular procedure or part of a procedure.

To create a local variable , use the **let** command.

If you use let at the top of a procedure , the variable will exist throughout the procedure.

If you use it inside a set of square brackets , for example inside an "ask", then it will exists only inside those brackets.

```
to swap-colors [turtle1 turtle2]
  let temp [color] of turtle1
  ask turtle1 [ set color [color] of turtle2 ]
  ask turtle2 [ set color temp ]
end
```

13.5 Ask

NetLogo uses the **ask** command to give commands to turtles, patches and links.

```
to setup
  clear-all
  create-turtles 100   ;; create 100 turtles with random headings
  ask turtles
    [ set color red    ;; turn them red
      fd 50 ]          ;; spread them around
  ask patches
    [ if pxcor > 0         ;; patches on the right side
        [ set pcolor green ] ]  ;; of the view turn green
  reset-ticks
end
```

```
to setup
  clear-all
  crt 3                          ;; make 3 turtles
  ask turtle 0                   ;; tell the first one...
    [ fd 1 ]                     ;; ...to go forward
  ask turtle 1                   ;; tell the second one...
    [ set color green ]          ;; ...to become green
  ask turtle 2                   ;; tell the third one...
    [ rt 90 ]                    ;; ...to turn right
  ask patch 2 -2                 ;; ask the patch at (2,-2)
    [ set pcolor blue ]          ;; ...to become blue
  ask turtle 0                   ;; ask the first turtle
    [ ask patch-at 1 0           ;; ...to ask patch to the east
      [ set pcolor red ] ]       ;; ...to become red
  ask turtle 0                   ;; tell the first turtle...
    [ create-link-with turtle 1 ] ;; ...make a link with the second
  ask link 0 1                   ;; tell the link between turtle 0 and 1
    [ set color blue ]           ;; ...to become blue
  reset-ticks                 ;; tick counter = 0, sets up all plots
end
```

13.6 Synchronization

When you ask a set of agents to run more than one command, each agent must finish before the next agent starts. For example : ask turtles [fd 1 set color red] , first one turtle moves and turns red , then another turtle moves and turns red , and so on.

If you write : ask turtles [fd 1] ask turtles [set color red], first all the turtles move , then they all turn red.

13.7 Agentsets

An **agentset** is a set of agents.

It can contain either turtles , patches or links. It is not in any particular order.

13.8 Breeds

You can define different **breeds** of turtles.

When you define a breed such as sheep , an agentset for that breed is automatically.

breed [plural_name singular_name]. It defines a breed , first input is the name of the agentset and the second is the name of a single member of the breed. breed [wolves wolf].

```
;; all other turtles:
other turtles
;; all other turtles on this patch:
other turtles-here
;; all red turtles:
turtles with [color = red]
;; all red turtles on my patch
turtles-here with [color = red]
;; patches on right side of view
patches with [pxcor > 0]
;; all turtles less than 3 patches away
turtles in-radius 3
;; the four patches to the east, north, west, and south
patches at-points [[1 0] [0 1] [-1 0] [0 -1]]
;; shorthand for those four patches
neighbors4
;; turtles in the first quadrant that are on a green patch
turtles with [(xcor > 0) and (ycor > 0)
              and (pcolor = green)]
;; turtles standing on my neighboring four patches
turtles-on neighbors4
;; all the links connected to turtle 0
[my-links] of turtle 0
```

- To make a randomly chosen turtle turn green:
  `ask one-of turtles [ set color green ]`
- To tell a randomly chosen patch to sprout a new turtle:
  `ask one-of patches [ sprout 1 ]`
- To remove the "richest" turtle:
  ```
  ask max-one-of turtles
      [sum assets] [ die ]
  ```
- To find out how rich turtles are on the average:
  `show mean [sum assets] of turtles`

A turtle's breed agentset is stored in the breed turtle variable. So you can test a turtle's breed like this : if breed = wolves [...]