# Playing God: A toolkit for building agents

Aaron Sloman[1]

## OBJECTIVES

An experimental toolkit designed to support exploration of design options for one or more objects interacting in discrete time under the control of a “scheduler” which ensures that each object is “run” in each time-slice.

The objects may exist totally within a single simulated “world running in one process, or they may communicate with other software packages or with real machines, via sensors or motor control signals.

## REQUIREMENTS

Each object or agent may have two kinds of behaviour:

- **External behaviour**
  which is detectable by or which affects other objects or agents, e.g. movement and communication.
- **Internal behaviour**
  involving (possibly resource-limited) mechanisms for changing internal state, e.g. beliefs, motives, goals, plans, etc.
- **No ontological commitment**
  i.e. many different kinds of objects should be supported.
- **Rich internal architectures within agents**
  e.g. multiple interacting rule-based systems, neural nets and other trainable sub-mechanisms
- **Use of classes and inheritance**
  e.g. it should be easy to override defaults by defining subclasses
- **Rapid prototyping should be supported.**
- **Control of speed of different objects or agent components.**

The SIM_AGENT library is implemented in Pop-11, using the OBJECTCLASS and POPRULEBASE libraries. Graphics to be added later.

1. With Luc Beaudoin, Ian Wright, Riccardo Poli, Glyn Humphreys, and other members of the Cognition and Affect project at the University of Birmingham

## Types of objects

### Objects may be active or passive, simple or complex

### Compound objects

Are composed of many other objects, e.g. a forest (composed of paths, trees, etc.), a town (composed of roads, houses, parks, people, vehicles, etc.) a house (composed of rooms, doors, etc.) a family (composed of various people), etc.

The sub-objects of a compound object are managed directly by the scheduler. (Non-compound objects may be complex.)

### Passive objects

E.g. walls, ditches, ladders, roads). Most will have no interesting internal behaviour. Exceptions are things like decay of charge in a battery, or interactions of parts of a compound object.

It may be useful in some cases to give a road, or a road-segment information about all of the objects travelling on it, e.g. in a traffic simulation).

### Active objects

Can initiate processes, e.g. volcanoes, people, traffic lights.

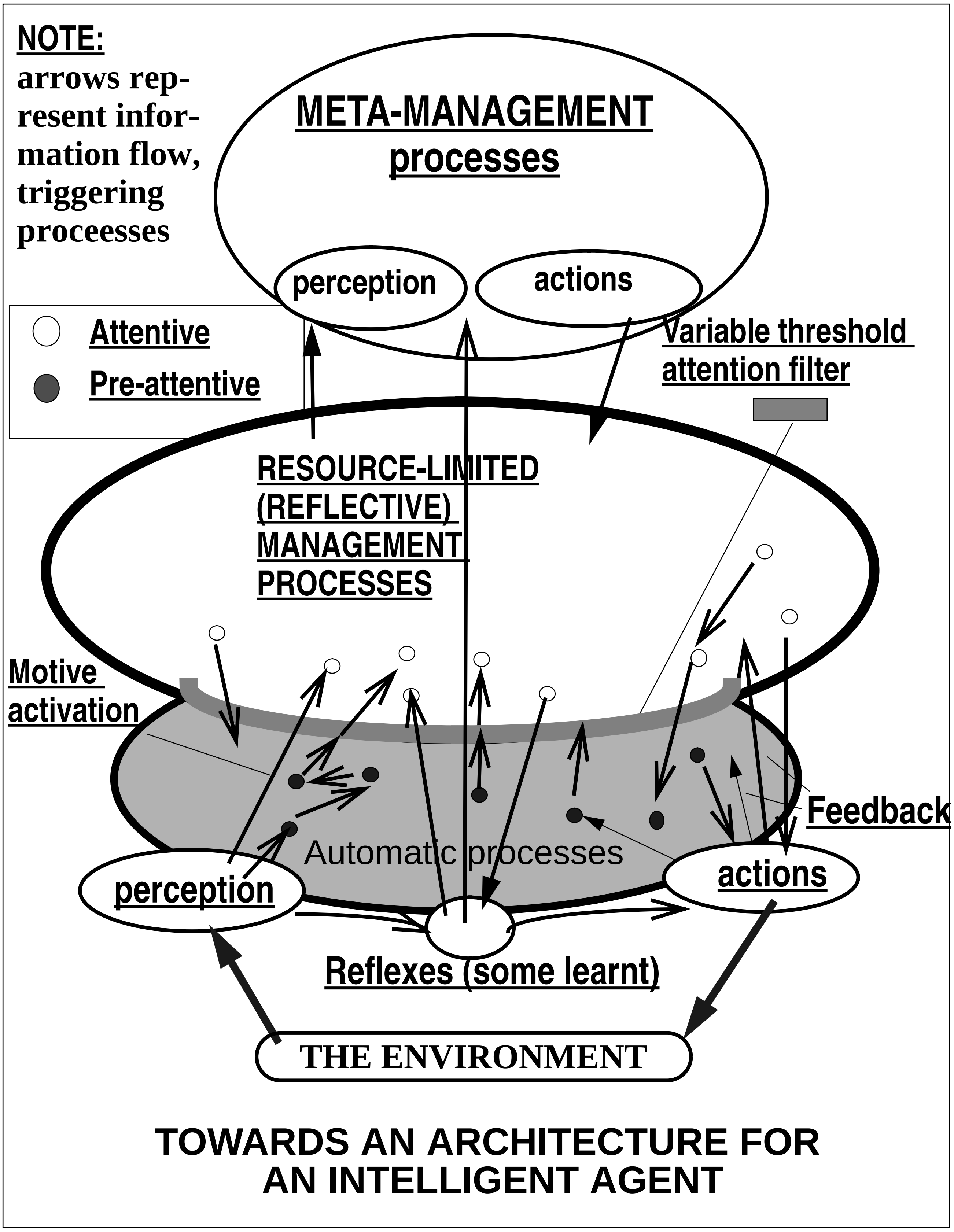
Some objects become active under the control of others, e.g. cars, tanks, spears, drills.

### Agents

Are active objects that generate their own motives and act on them, after suitable planning etc. The package uses the mechanisms of Poprulebase for internal processes. This is a forward-chaining production system interpreter, with special features to allow:

- interacting sub-systems,
- invocation of other mechanisms (e.g. neural or procedural),
- resource control.

# An example agent architecture



## Some processes involving motivators

The following are among the internal behaviours to be modelled:

- **Motivator generation and (re-)activation**.
- **Mechanisms to suppress or ‘filter’ motivators**:
  to protect resource-limited management processes.
- **Management of motivators**
  - **Assessing motivators.**
    e.g. importance, benefits, side-effects, likelihood of satisfaction, cost of satisfaction, urgency.
  - **Deciding**:
    whether to adopt the motivator, i.e. form an intention.
  - **Scheduling**:
    when or under which conditions to execute a motivator.
  - **Expansion**:
    deciding how to execute a motivator (planning).
  - **Predicting effects.** (Helps with planning and assessment)
  - **Assigning an “intensity” measure.**
  - **Detecting conflicts between motivators.**
  - **Detecting mutual support between motivators.**
  - **Setting thresholds for the management interrupt filter.**
  - **Termination of motivators.**
    E.g. explicitly terminate on satisfaction, decay, abort.
- **Meta-management:**
  I.e. processes that (recursively) control management or meta-management processes (e.g. deciding which to do when).
- **Execution of plans**,
  with or without high level management.
- **Learning**:
  improving or extending performance
- **Extending the architecture:**
  developing new abilities, or new “cognitive reflexes”.
- **Global switches or modulators:**
  e.g. mood changes, arousal changes, speed or style of processing.

# The scheduler

The “top level” procedure **sim_scheduler**, manages the whole process in a succession of time-slices. It is run by a command of the form:

**sim_scheduler( <objects>, <lim> )**

where <objects> is a list of all the objects, and <lim> is an integer specifying the number of time-slices for which the process should run, or **false** if the process should run forever.

The objects should all be instances of the top level class **sim_agent** so that methods appropriate to that class can be applied to them to make them run, or to perform their actions.

Users can define sub-classes for which more specific versions of the methods are defined, and these will automatically over-ride the generic methods.

In each time-slice the scheduler has two runs through the list of objects.

- First it allows each object to sense the state of the world and, if appropriate, receive communications from other agents, and to do as much internal processing as it is entitled to. This may include preparing external actions and communications
- Next the scheduler transfers the messages from sources to targets, and runs the external action routines corresponding to each object that has actions pending.

This makes behaviour (generally) independent of the order of objects in the list. (Note the counter-examples).

When “running” individual objects the scheduler uses methods defined for the top level object class (sim_agent). If users define more specific methods for sub-classes they are automatically invoked, without altering the source for sim_scheduler. (Thanks to Objectclass)

(This is one of the main benefits of object oriented programming.)

## Running an agent's internal processes

**This is the default method**

```
define :method sim_run_agent(object:sim_agent, objects);
    ;;; More specialised versions of this method may be defined for
    ;;; sub-classes of sim_agent

    < setup sensory input buffers by running sensors>
    < add information from sensory input buffers and message
      input buffers to the internal database>

    repeat sim_speed(object) times

      < get the rulesets associated with the object. Each ruleset
      is a set of rules that can be used by prb_run.>

      for ruleset in rulesets do
        ;;; use POPRULEBASE mechanism on each ruleset
        prb_run(ruleset,sim_data(object), sim_ruleset_limit(ruleset))
      endfor;

    endrepeat;

    < clear input message buffer and clear sensory input buffers >

    < prepare output actions and messages to go out, and
      remove them from the internal database>

    < run user-definable tracing procedures >
enddefine;
```

**Note the mechanisms for controlling relative speeds.**

I.e. sim_speed(object), and sim_ruleset_limit(ruleset)

## Agent mechanisms based on POPRULEBASE

It’s up to the user to decide on agent architectures and mechanisms. But there is special support for the following:

- Each agent type has a collection of rulesets. The rules operate on one or more databases internal to the agents.
- Rules can switch between databases, push them onto a stack, restore them, etc.
- The rulesets may change over time, as may the individual rules within a ruleset.
- If more sophisticated reasoning or logical deduction procedures are required it is possible to invoke prolog, or some sort of theorem prover.
- If other ‘sub-symbolic’ mechanisms are required, they can be invoked by appropriate rules, e.g. using FILTER conditions (described later).
- Each ruleset corresponds to a “context” in the processing of an agent that uses the ruleset. E.g. a context may be analysing incoming messages, or analysing sensory data, or deciding which goals to adopt, or planning, or executing goals, or constructing messages to transmit. The facility in Poprulebase to switch between rulesets or between databases permits rapid switching between these contexts.
- Parallelism between rulesets within an agent can be implemented by limiting the number of cycles allocated to each ruleset, and repeatedly running all the rulesets. This can be achieved by associating different cycle limits with different rulesets via the property sim_ruleset_limit. (Later it may be desirable to make this agent and ruleset specific rather than simply ruleset specific.)
- Internal parallelism can also be achieved by breaking an object into a collection of sub-objects all handled directly by the scheduler.
- Agents can be given different relative speeds of execution by giving them different values for their “sim_speed” slot. This determines the number of times their internal rulesets are all run in each time-slice. Individual rulesets can also be given different relative “speeds”.

**<u>All the above work during the first pass of the scheduler</u>**

## Running the ‘external’ actions

On the second pass through the list of objects, the following is done by the scheduler:

```
for object in sim_objects do
  <run user-definable procedure sim_agent_action_trace>

  <Transmit messages from the object to the message input
    buffers of their intended targets>

  <Perform pending actions in the object’s action output buffer>

  <clear the object’s output message lists and action lists>
endfor;
```

As with the internal actions these processes use methods, which default to those defined for the top-level class but can be overridden by methods for sub-classes

**NOTE 1:**
It might be preferable to subsume message sending under the general category of external actions and message interpretation under the general category of interpretation of sensory input.

But handling messages separately simplifies tracing and debugging of communications between agents.

**NOTE 2:**
It is up to the user to handle things like relative external speeds of objects. (A changeable global multiplier might be useful.)

**NOTE 3:**
Instead of the two-pass mechanism random reordering of the list of agents might be used to achieve “fairness”.

## The representation of time

- The scheduler gives each object a chance to “run” in each time-slice.
- What it means for an object to run is defined via methods. Some of the methods perform the internal actions for the object, some perform the sensory detection, and some perform the external actions.
- The package does not use real time or even cpu time as a basis for interrupting processing, so user software must ensure that no agent or object takes control and keeps it forever.
- This allows users the facility to simulate the speeding up of processing in a particular subset of agents by allowing them to do more in each time-slice. Similarly faster physical motion would be represented by larger physical changes in each time-slice (possibly controlled by a global multiplier, to simplify changing relative internal and exteral speeds.).
- Specifying relative speeds is entirely up to the user, and can be determined by class of object, by ruleset, etc.
- The package will not be suitable for simulations involving continuous change, unless this can be represented to an adequate degree of approximation by a succession of small discrete changes.
- It has been designed to support flexible design and exploratory development through rapid prototyping, rather than optimal speed or space efficiency.

NOTE: we do not base time-slicing on cpu-time because the cpu time measure is meaningless relative to the aims of typical simulations. E.g. some processes may be relatively slow because of features of the implementation. (E.g. simulating a neurla net.)

Thus users will have to ensure that on each run the rules actually terminate. Up to a point this is handled by the last argument to prb_run.

# Some features of Poprulebase

Some unusual features have been added to the condition-action rules, in consultation with Riccardo Poli, implementing and extending ideas from Brayshaw and Poli 1994.

## SPECIAL CONDITIONS:

### Boolean filter conditions: [FILTER BFP C1 C2 ... Cn]

The BFP (boolean filter procedure) will be applied to a veclist (vector or list) of n items derived from the n conditions and the whole condition will succeed or fail depending on whether the result returned by BFP is non-false.

### Vector filter conditions: [FILTER VFP -> var C1 C2 ... Cn]

The VFP (vector filter procedure) is applied to a veclist VL of n items derived from the n conditions and should output either FALSE, in which case the condition fails, or another veclist, the of length m, where m need not be the same as n. The list will be transferred via the variable **var** to the corresponding action.

The VFP or BFP used in filter conditions may be trainable.

## SPECIAL ACTIONS:

These are controlled by veclists produced by FILTER conditions.

### The SELECT action type: [SELECT ?var A1 A2 ... An]

**var** should have a veclist of length n as value, derived from a vector filter procedure in one of the conditions of the rule. The non-false elements of the list will be used to select the corresponding actions to be performed.

### The MAP action type: [MAP ?var MP A1 A2 ... An]

**var** should have as value a veclist derived from a vector filter procedure in one of the FILTER conditions of the rule. The mapping procedure **MP** will be applied to the value of **var** and the list of actions A1 ... An, and the rule_instance containing them.

See the Brayshaw and Poli paper for more detailed discussion.

## Using the SIM_AGENT package

Users must be prepared to do the following.

- Define the ontology, i.e. classes of objects and agents required, making all of them subclasses of sim_agent.
- Define the sensor methods and sim_do_action methods for the classes. This includes defining internal formats for sensory information and action specifications.
- Define the sim_send_message method for the classes. This includes deciding on the formats for different kinds of messages and the protocols for sending and receiving messages. (E.g. some may require an acknowledgement some not, some may be requests, some orders, some answers to questions, some questions, and so on.)
- Define the (Poprulebase) rulesets for internal processing by the different classes of agents, and the rules for each ruleset. This involves defining the formats for the different kinds of information to be used in the internal databases, e.g. sensor information, beliefs about the environment, motivator structures, plan structures, management information, etc.
- Specify the initial databases for each type of agent.
- Specify which collection of rulesets should be used by each type of agent, and in what order, and the relative processing speeds associated with each ruleset.
- Create all the required initial instances of the agent classes and put them into a list to be given to sim_scheduler.
- Create any other data-structures required, and the procedures to access and update them (e.g. a map of the world).
- Define any required object-specific tracing methods, e.g. graphical tracing methods. (Some default tracing methods may be provided)

### Libraries:

Collections of re-usable libraries corresponding to particular ontologies and applications will be developed over time, and shared between users. Simulation of physical movement and graphical projection might be done by a re-usable library class.

## Example: representing motivator structure

Designers of autonomous agents need to think about the permitted forms of motivators and motivator mechanisms.

Motivators (desires, inclinations, goals, etc.) produced by pre-attentive or attentive processes will often include the following components, though they may have other specific features also. Some of these will vary over time.

(1) Semantic content: a proposition, **P**, denoting a possible state of affairs, which may be true or false

(2) A motivational attitude to **P,** e.g. “make true”, “keep true”, “make false”, etc.

(3) A rationale, if the motivator arose from explicit reasoning.

(4) An indication of the current belief about **P**’s status, e.g. true, false, nearly true, probable, unlikely etc.

(5) An “importance value” (e.g. “neutral”, “low”, “medium”, “high”, “unknown”), importance may be *intrinsic*, or based on assessment of *consequences* of (doing and not doing).

(6) An “urgency descriptor” (possibly a time/cost function)

(7) A heuristically computed “insistence value”, determining interrupt capabilities. Should correspond loosely to estimated importance and urgency.

(8) Intensity -- which influences likelihood of (continuing) being acted on, as against other motivators.

(9) Possibly a plan or set of plans for achieving the motivator

(10)A commitment status (e.g. “adopted”, “rejected”, “undecided”)

(11)A dynamic state (e.g. “being considered”, “consideration deferred till...”, “nearing completion”, etc.)

(12)Management information, e.g. the state of current relevant management and meta-management processes.

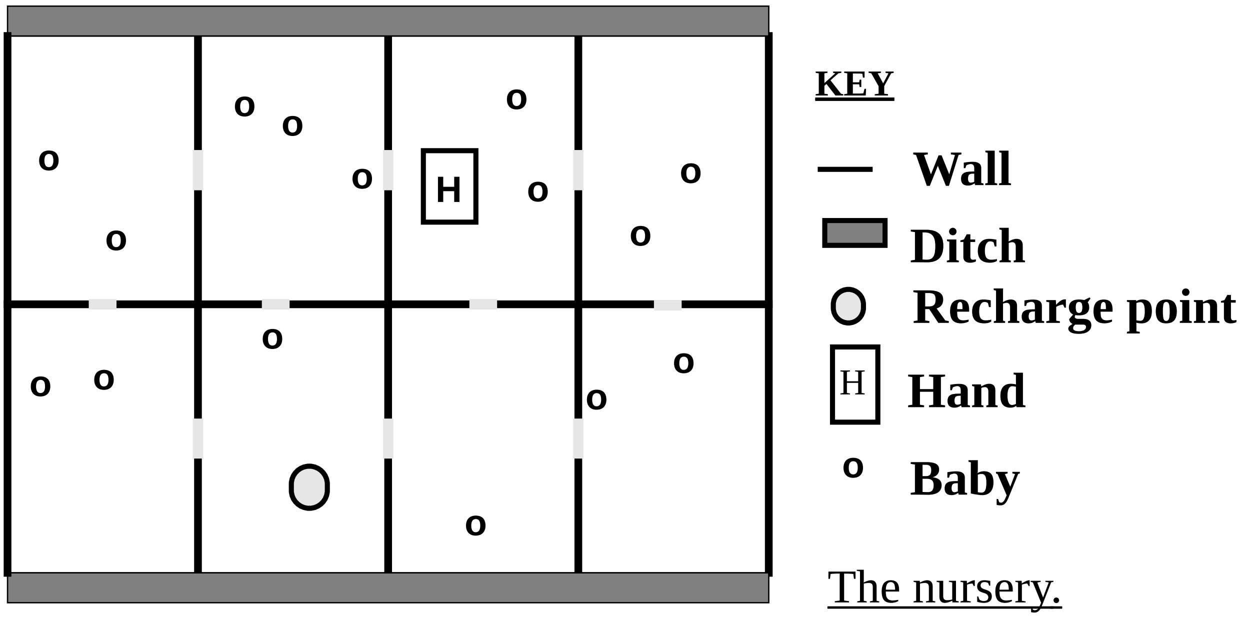
In most animals motivators are probably simpler (and in current robots).

There may be individual differences among humans too.

Exploring “design space” will show what is possible.

# The nursemaid (minder) scenario

This is one of the domains to which these ideas are being applied: the task of the nursemaid is to keep babies alive.



The nursery.

**(The nursemaid's current visual field could be any room)**

This domain is discussed at length in Luc Beaudoin's PhD thesis, 1994.