SIM_AGENT: A toolkit for exploring agent designs*

Aaron Sloman and Riccardo Poli

School of Computer Science
The University of Birmingham
Birmingham, B15 2TT
United Kingdom

{A.Sloman,R.Poli}@cs.bham.ac.uk

Abstract. SIM_AGENT is a toolkit that arose out of a project concerned with designing an architecture for an autonomous agent with human-like capabilities. Analysis of requirements showed a need to combine a wide variety of richly interacting mechanisms, including independent asynchronous sources of motivation and the ability to reflect on which motives to adopt, when to achieve them, how to achieve them, and so on. These internal 'management' (and meta-management) processes involve a certain amount of parallelism, but resource limits imply the need for explicit control of attention. Such control problems can lead to emotional and other characteristically human affective states. In order to explore these ideas, we needed a toolkit to facilitate experiments with various architectures in various environments, including other agents. The paper outlines requirements and summarises the main design features of a Pop-11 toolkit supporting both rule-based and 'sub-symbolic' mechanisms. Some experiments including hybrid architectures and genetic algorithms are summarised.

1 Introduction

This paper describes the design rationale, and some features of, an experimental toolkit, SIM_AGENT, supporting a mixture of symbolic (e.g. rule-based) and sub-symbolic (e.g. neural) mechanisms, which is being used both here in Birmingham and at DRA in Malvern. We end by describing some simple experiments using the toolkit.

SIM_AGENT is intended to support exploration of design options for one or more agents interacting in discrete time. It arose out of a long term research project concerned with architectures for autonomous agents with human-like capabilities including multiple independent asynchronous sources of motivation and the ability to reflect on which motives to adopt, when to achieve them, how to achieve them, how to interleave plans, and so on. Figure 1 depicts approximately the type of architecture we have been exploring (described incrementally in [28, 19, 21, 5, 4, 25, 11]). At present we are primarily concerned with performance in simulated time, not real time.

The figure, based partly on ideas by Luc Beaudoin and Ian Wright, is intended to be suggestive of an architecture in which there are many different coexisting components with complex interactions. Some processes, which we assume evolved early and are to be found in many types of animals, are automatic (pre-attentive) in the sense that all changes are triggered directly. Others, the 'management processes', which evolved later and are probably to be found in fewer types of animals, are 'reflective' or 'attentive' knowledge-based processes in which options are explicitly considered and evaluated before selection. The latter (which need not involve conscious processing) are resource-limited and may sometimes require both attention filters to protect them from disturbance and meta-management processes to regulate and direct them. This requires some sort of self-monitoring. (See [4, 11]). Reflexes (learnt or innate) bypass 'normal' processing. The diagram should not be taken to imply any sharp division between processes concerned with perception or action and other cognitive processes [20].

The internal 'management' (and meta-management) processes involve a certain amount of (asynchronous) parallelism, but we wanted to show how resource limits restricting parallelism in high level processes can lead

* In: *Intelligent Agents Vol II (ATAL-95)*,
Eds. Mike Wooldridge, Joerg Mueller, Milind Tambe, Springer-Verlag pp 392–407
Presented at *ATAL-95, Workshop on Agent Theories, Architectures, and Languages,* IJCAI-95, Montreal, August 1995.
For more information and "movies" see: http://www.cs.bham.ac.uk/~axs/cog_affect/sim_agent.html

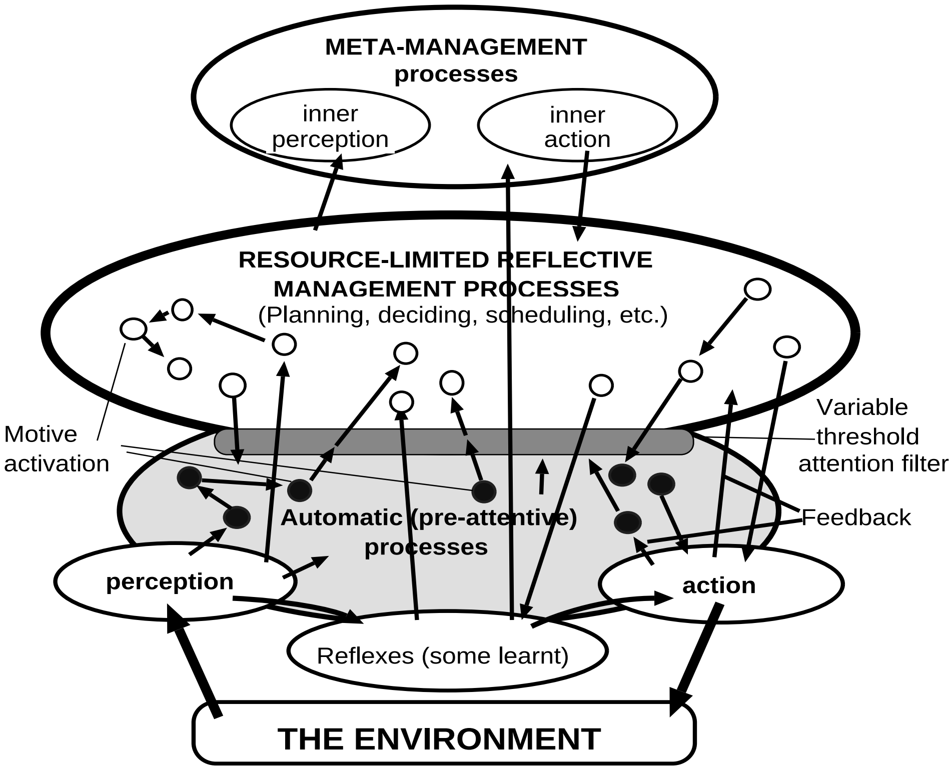

Fig. 1. Towards an Intelligent Agent Architecture

to emotional and other characteristically human states involving partial loss of control of attention and thought processes [18, 21, 22, 4, 11]. This requires an architecture combining a wide variety of types of mechanisms. Like the OZ project [3, 2] we are initially interested in 'broad and shallow' architectures, combining many sorts of capabilities, where each capability is initially implemented in a shallow fashion. Deeper, more complete, implementations will follow.

Whereas many people building agent architectures aim for a single architecture, we wished to explore a variety of architectures, and, moreover, to study interacting agents with different architectures. We also wished to support evolutionary experiments, as described in Section 3.2. A generic toolkit is needed to support the exploration of alternative designs [23, 25]. We needed a toolkit providing the following features:

Minimal ontological commitment: i.e. many kinds of objects with very different architectures should be supported.

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 states that are not directly detectable by others, e.g. beliefs, motives, goals, plans, etc.

Rich internal architectures within agents: e.g. several layers of sensory interpretation, multiple interacting rule-based systems, 'cognitive reflexes', neural nets, and other trainable sub-mechanisms

Use of classes and inheritance: e.g. one or more default classes of objects should be provided, but it should be easy to override defaults by defining subclasses with more specific forms of behaviour, without having to edit the generic code.

Control of relative speed: allowing easy variation of relative speeds of different agents and also relative speeds of different components within an agent (e.g. perception, planning).

Rapid prototyping: Pop-11 supports this via incremental compilation.

The following are among the types of objects to be explored (these distinctions are all provisional and "fuzzy"):

Active objects change their state spontaneously, i.e. without the intervention of other objects. Some of the active objects merely obey physical laws, whereas others, which we call **agents** take in and manipulate information and vary their behaviour as a result of processing information. The division between agents and non-agents is not very sharp [24, 25]. We are specially interested in agents that generate their own motivators. These are "autonomous".

Passive objects are objects which do not change their state spontaneously. Examples might be walls, ditches, ladders, roads. A battery whose charge runs down, a volcano that erupts from time to time or a tree that grows would be examples of *active* objects, though they are not *agents*. Some passive objects become active under the *control* of agents, e.g. cars, spears, drills, screwdrivers, and we can refer to these as **instruments**. Other passive objects may *react* to impact or pressure (e.g. a wall). We call these **reactors**.

Compound objects are composed of (or possibly 'own') two or more 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. A **simple object** has no parts that are objects managed by the scheduler but may nevertheless have a complex *internal* architecture, involving several interacting subsystems. e.g. an intelligent agent.

2 The SIM_AGENT toolkit

The toolkit is implemented in Poplog Pop-11 [1], using our Poprulebase [27] library, and the Objectclass package designed by Steve Knight at Hewlett Packard Laboratories, which extends Pop-11 with CLOS-like object oriented facilities, including classes, inheritance and generic methods that can be redefined for specific user-defined classes of objects or agents. The class hierarchy in the SIM_AGENT toolkit starts with the top level class **sim_object**, and a subclass **sim_agent**. Users can create additional classes.

The toolkit provides a scheduler which 'runs' objects in a virtual time frame composed of a succession of time slices. In each time-slice every object in the world is given a chance to do three things: (a) sense its environment, including creating internal representations derived from sensory data, (b) run processes that interpret sensory data and incoming messages, and manipulate internal states, and (c) produce actions or messages capable of influencing other objects in the next time slice. The 'top level' procedure, sim_scheduler, manages the whole process. 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. 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, detect communications from other agents, and then do as much internal processing as it is entitled to, e.g. changing internal databases and preparing external actions and communications. The Pop-11 pseudocode of this phase is given in Appendix A.
– On the second pass, the scheduler transfers messages from sources to targets, and runs external action routines corresponding to each object with actions pending. Actions that need to be combined into a resultant may need special treatment. (See Appendix A.)

This two-pass strategy makes behaviour (generally) independent of the order of objects in the list given to the scheduler, since nothing detects the state of the environment until everything has completed its actions in the previous time-slice.

The scheduler assumes that objects are all instances of the top level class **sim_object** so that default methods can be used to 'run' them. Users can define more specific methods for their sub-classes, and these will automatically over-ride the generic methods, though objectclass, like CLOS, allows sub-class methods to invoke the generic methods where appropriate.

The toolkit allows users to explore different forms of internal processing, but provides special support for agents whose internal processing can be implemented using interacting collections of condition-action rules, communicating concurrently via a collection of databases, including rules that invoke non-symbolic 'low level' mechanisms. This uses Poprulebase [27], a forward-chaining production system which provides many of the features associated with conventional rule interpreters as well as the ability to drop into Pop-11 where appropriate. In particular:

- Each agent class has an associated collection of rulesets. Each ruleset is a collection of condition-action rules that interact via one or more 'databases' or working memories internal to the agent. Thus different rulesets might be concerned with: interpreting low-level sensory data, reasoning, generating motivators in response to new beliefs, assessing the importance of motivators, planning, meta-management, etc.
- Rules can switch between databases, push them onto a stack, restore them, etc., as in SOAR [15]. Rules can also transfer control to a new ruleset. Rulesets may be stacked then restored.
- Each ruleset corresponds to a sub-mechanism or 'context' in the agent. 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.
- Within an agent, learning or development may change individual rules, or introduce new rulesets, or introduce new interactions between rulesets, e.g. by adding new communication channels.
- If 'sub-symbolic' mechanisms are required, they can be invoked by appropriate rules, e.g. using FILTER conditions and SELECT actions, described in Appendix B (see also [16, 26]).
- The rules can also invoke Pop-11, and Pop-11 can invoke other Poplog languages or libraries, e.g. Prolog, theorem provers, planners.
- Parallelism between rulesets within an agent can be simulated by limiting the number of cycles allocated to each ruleset in each time-slice, and repeatedly running all the (runnable) rulesets. Internal parallelism can also be achieved by creating sub-objects all handled directly by the scheduler.
- Varying relative speeds of different kinds of internal processing can be achieved by associating different numbers of interpreter cycles with different rulesets. Thus it is possible for one agent to have perceptual processes speeded up relative to planning processes, and another agent to have planning processes speeded up relative to perception.

In order to support experiments with different resource allocations, the toolkit uses a 'virtual' representation of time instead of real time. Real time or cpu time measures are highly arbitrary and implementation dependent, and meaningless relative to the aims of such experiments. Some processes may be relatively slow merely because of features of the implementation, e.g. a simulated neural net where a real one would operate much faster. If the toolkit were required for 'real' time interactions with some external mechanism, it would be possible to use Pop-11's lightweight process mechanisms and timer interrupt facilities to control processing in each time-slice. The present implementation has the following features:

- 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 which perform the internal actions, the sensory detection, and the external actions. They should all assume that the amount of *virtual* time available in a time-slice is fixed.
- The use of virtual time allows simulated speeding up of processing in a particular subset of agents by allowing them to do more in each time-slice, by giving them higher values for their 'sim_speed' slot. This determines the number of times their internal rulesets are all run in each time-slice. Similarly faster physical motion would be represented by larger *physical* changes in each time-slice.
- Specifying relative speeds is entirely up to the user, and can be determined by class of object, by ruleset within an agent, etc. Since only discrete variation in speed is supported, it is up to the user to ensure that programs are broken down into small enough chunks to allow appropriate variations in relative speeds.

The toolkit has been designed to support flexible design and exploratory development through rapid prototyping, rather than optimal speed or space efficiency.

3 Using the toolkit

In order to use the toolkit, the user must be prepared to do the following:

- Define the ontology, i.e. classes of objects and agents required, making all of them subclasses of the classes provided.
- Define the sensor methods and do_action methods for the classes. This includes defining internal formats for sensory information and action specifications.
- Define the send_message method for the classes which need it, and decide on formats for different kinds of messages and the protocols for sending and receiving messages. (E.g. some may require an acknowledgement some not, and there may be requests, orders, questions, answers to questions, advice, permissions, etc.)
- Define the (Poprulebase) rulesets for internal processing of 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, if the world itself is not an object).
- Define any required object-specific tracing methods, e.g. graphical tracing methods.

Through experiments in different "worlds", collections of re-usable libraries corresponding to particular ontologies and applications can be developed and shared between users. For instance, simulation of simple movement with graphical tracing could be implemented in a re-usable library class. The use of new Pop-11 macros or syntax words allows translation from another representation into the format used by Poprulebase, and this could be applied to ontologies created by others, e.g. in the KQML/KIF libraries [12].

In the following sections we provide two examples illustrating the scope of the toolkit, and showing how it allows natural integration of reactive behaviour (e.g. implemented via neural FILTERs) and symbolic reasoning within an agent and how different kinds of processing can be combined freely, (e.g. in parallel, in sequence, hierarchically, etc.).

3.1 The Robot in a Box

In the first experiment we implemented a simulated trainable robot called RIB (Robot In a Box). RIB has a hybrid architecture based on the 'Filter' conditions and 'Select' actions of Poprulebase (see Appendix B). The world is made up of two objects: a 'reactor' and an 'agent' (the RIB). The reactor is a 2-D room (the box) with a complex shape. The agent has very simple visual and motor sub-systems. There are 8 sensors that measure average distance of visible surfaces in each of 8 different 45 degree cones, providing the RIB with an all-round, 8 pixel, low-resolution range image. The motors merely produce acceleration impulses in the X and Y directions. The RIB has only three physical properties: spatial occupancy, velocity and a unitary mass. The motor system of the RIB does not directly control the robot's speed or position, but simply the accelerations (like jets on a space ship). Velocity and the position are determined by numerically integrating motion equations. Such a system is quite hard to control.

The box is an instance of a subclass of class sim_object with a list of straight-line segments representing the walls, including protrusions, intrusions, internal blocks, etc. A box is passive unless the agent gets too close to a wall, in which case it reacts by producing a partly elastic 'bounce', changing the velocity of the approaching agent so as to enforce the constraint of wall solidity.

A RIB instance has a state, comprising: (a) its X and Y coordinates, (b) its velocity, and (c) the previous range sensor image. From the current and previous images, the visual system extracts additional information, including proximity information which makes explicit whether and in which direction(s) the RIB is too close to walls, and 'novelty' information indicating a sudden increase in distance, caused by passing a corner.

The RIB has several groups of rules, each providing the robot with a specific behaviour, which together make up the architecture in Figure 2. The physics of the RIB world is implemented in two *motion rules* which, in each time slice, perform the integration of the motion equations and determine the new position and velocity of the RIB. These provide the basic 'behaviour' of moving in straight trajectories and bouncing into the walls. In addition there are several groups of sensor-based control rules, which can be turned on or off in different combinations, to produce different resultant behaviours:

- An *obstacle avoidance rule* creates accelerations that tend to prevent bouncing into nearby objects. This is a hybrid rule which includes a FILTER condition which feeds information about the current range image into a neural network trained on hand-compiled examples. Its four element output vector goes to a SELECT action which creates up to four accelerations.
- A *wall-following rule,* responsible for the highest level behaviour of the RIB, i.e. approaching and then following the walls of the box. This is also a hybrid rule, using the FILTER/MAP construct. The FILTER condition gives a neural net the velocity vector of the robot and the range sensor 'image'. The net's output is an acceleration vector handed to the MAP action. The net was trained in advance on hand-drawn example trajectories (starting from several different positions and with different velocities, as shown in Fig. 3(e)).
- A *wall-approaching rule* which uses range sensor information to detect whether the robot is too far from the nearest wall, and if so causes an oblique acceleration towards the nearest wall. This also uses a FILTER/SELECT construct, but the predicate invoked by the FILTER condition is purely symbolic.
- Four *corner following rules* improve the wall-following performance of the RIB in the presence of convex corners. These are classical CONDITION/ACTION rules which use a sudden increase in a range sensor value to detect a corner at right angles to the current direction of motion, triggering additional accelerations to make the RIB turn more quickly.

An auxiliary *clean-up rule* (not shown in Figure 2) removes the old sensory data from the short term memory of the RIB and updates its internal state.

This redundant set of rules illustrates the use of a behaviour-based architecture to obtain improved performance and graceful degradation. Different subsets of the rules achieve approximate wall-following behaviour (even without the main wall-following rule), but their combination produces less erratic behaviour (illustrated below).

Figures 3(a)–(d) show four runs of the sim_object toolkit with a RIB starting with null velocity at coordinates (0.65,0.6), where the side of the box is 1 unit and the origin is bottom left. In the first run (Figure 3(a)) only the obstacle-avoidance, the motion and the wall-following rules were used. The RIB is easily distracted by the internal box, so cannot follow all the main walls. In Figure 3(b) the wall-approaching rule was used instead of the wall-following one. Here the RIB repeatedly bounces into the walls, despite the efforts of the obstacle-avoidance rule. Much better results are obtained if all three rules are used together (Figure 3(c)), reducing the attractions of the small box and the tendency to hit walls. A defect remains: the RIB overshoots convex corners. This is reduced by the corner following rules, as shown in Figure 3(d). In fact, even with all the rules working the RIB sometimes (eventually) gets trapped in an "attractor", or diverted from its path by a distant object or an unusually strong bounce. The use of a "subsumption" architecture, with additional rules to detect those conditions and override normal behaviour could provide a cure.

3.2 The Blind/Lazy Scenario

In another set of experiments we used the toolkit to explore architectures required for cooperative behaviour. As in the previous example, we used a "top-up approach", i.e. starting with agents having a high-level symbolic architecture only partially specified and some high-level functions, like perception, communication and motor control, already available ("top") and using optimisation techniques like Genetic Algorithms (GAs) [8] to complete the design ("up").

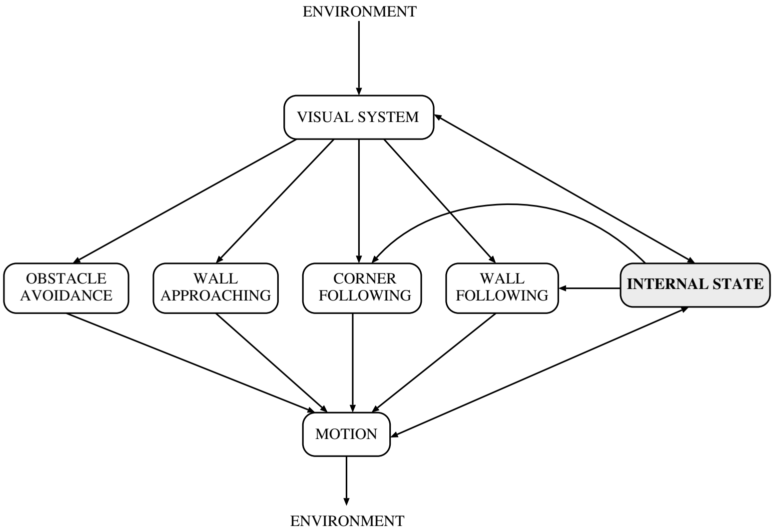

Fig. 2. Architecture of the RIB.

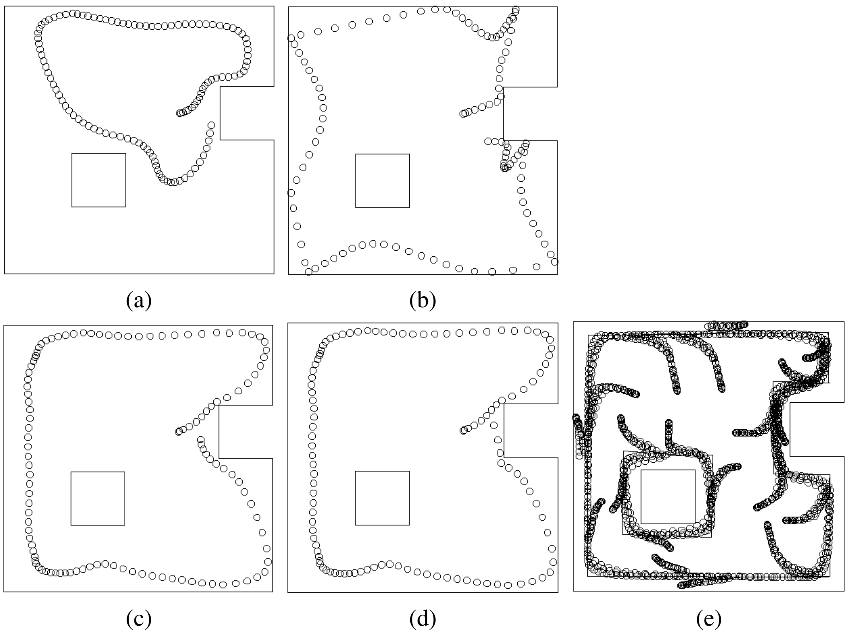

Fig. 3. Experimental results obtained with a RIB having the following subsets of rules: a) obstacle-avoidance, motion and wall-following rules; b) obstacle-avoidance, motion and wall-approaching rules, c) obstacle-avoidance, motion, wall-approaching and wall-following rules, d) as in c) plus corner following rule, e) the training set for the wall-following neural net. (S=start)

The scenario is very simple: there is a flat 2-D world inhabited by two agents, a blind agent and a lazy one. The blind agent can move in the world, can send messages and can receive messages sent by the lazy agent, but cannot perceive where the other agent is. The lazy agent can perceive the (roughly quantised) relative position of the blind agent and can send messages, but cannot move. Both have very simple internal states and a very simple communication syntax. Initially there is no semantics. The objective of the experiment is to see whether a GA can evolve rules to use the set of mental states and the communication syntax to allow the agents to cooperate in performing the task of getting together, thereby giving a semantics both to mental states and the communication language.

The agents' internal states are represented by entries in their database of the form `[internal_state X]`, where `X` is a label. As we allow only four possible labels, each agent can only be in one of 16 different states. The communication language allows the transmission of statements of the form `[message_out X Y]` where `X` is either `lazy` or `blind` and `Y` is a label. Each label can be interpreted as a word, and the set of these statements as a multi-word sentence. As only four labels are allowed and their meaning is considered to be independent of the order with which they are collected to form sentences, each agent can only send/receive 16 different sentences.

Apart from motion rules, cleanup rules and a stop condition, all the processing is performed inside the agents using a small number of hybrid rules. Each rule includes a FILTER condition which invokes a GA-tunable procedure and a SELECT action which converts the outputs of the procedure. The GA-tunable procedure is a vector Boolean function with as many input variables as the conditions in the FILTER and as many outputs as the actions in the SELECTor. The Boolean function is implemented as a vector truth table, which can easily be encoded as a binary string and optimised by a GA. There are three types of rules:

- Rules mapping sensory data (messages and/or "visual" data) to internal states
- Rules mapping internal states into internal states
- Rules mapping internal states into actions (messages and/or motion)

These suffice to implement a kind of finite-state automaton, i.e. a computer program. As they are tunable, the GA can "program" the agents to act in the best way according to a given fitness function, applied to the behaviour of the agents in each of eight different relative starting positions.

The fitness function was a weighted sum of the distance between the agents at the end of each run of the simulation and the total number of words exchanged by the agents during the simulations. The first part of the fitness function measures how well the problem is solved, and the second part the resources used. A string of 512 bits suffices to represent the truth tables of the aforementioned rules. Each such string represents a possible set of rules for the blind and lazy agents.

The behaviour of the agents on the training set is shown in Figure 4 where a small amount of noise was added to the actual positions of the blind agent to separate overlapping paths. Though fairly good, the results are not perfect, as shown by detours taken by the blind agent when starting from certain positions. This is because of the considerable computation load involved in optimising such long binary strings and, more importantly, in running the simulation needed to evaluate the fitness of a set of rules.

Figure 5 shows a run of the simulation with relative starting positions not in the training set, while Figure 6 represents the "conversation" between the agents and their internal states during the run. Analysis of these figures (a bit of "Artificial Psychology") reveals several interesting facts:

- The language developed is really very concise as messages include only single words and the agents use a "ask when necessary" protocol with a default behaviour in the absence of reply;
- Instead of using some of the internal states as "mental states", i.e. roughly speaking states corresponding to a single behaviour (for example, S1 could stand for "go north", S1S2 for "go north-west", etc.), the agents tend to use loops of internal states (e.g. S1S2S4→ S3S4→ S1S3S4→S1S2S4) as if the task required the use of higher level concepts or internal dynamics capable of grasping the concept of time (for example to remember what was in the last message and how long ago it was received).
- Such higher-order mental states tend to share sub-paths (e.g S1S2S4→ S3S4→ S1S3S4→S1S2S4 and S1S2S4→ S3S4→ S1S3S4→ S1S2S3S4→S1S2S4) which correspond to common sub-behaviours.

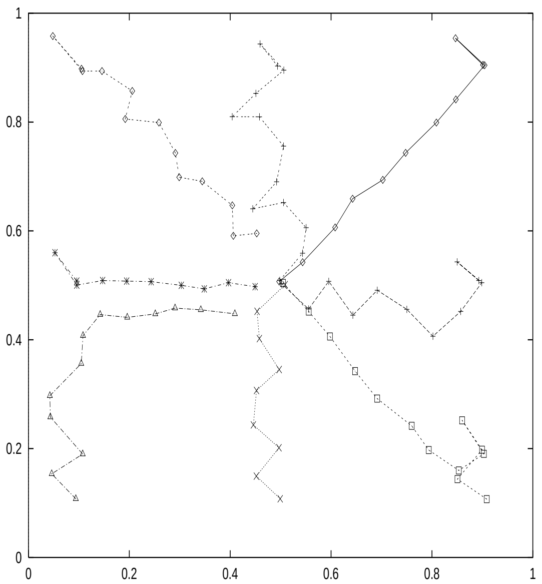

Fig. 4. Behaviour of the blind/lazy agents when starting from the positions in the training set.

4 Relations to other work

Until recently the majority of work in AI was not concerned with complete agent architectures. Rather, the main focus was (and still often is) on some sub-mechanism, or mechanisms, e.g. vision, language processing, planning, learning, etc. There is now growing interest in agent architectures, and increasingly ambitious attempts to design complete agents, whether physical robots or software agents (e.g. [2, 7, 10]. However, the motivational systems are generally very primitive. There are some attempts to describe complete human-like architectures (e.g. [14]) but implementations are usually lacking, partly because of the enormous complexity of the task (as Minsky comments in [17]). It is particularly difficult if 'real' robot equipment is used, and as a result such work often oversimplifies some aspects of the task because of the difficulty of other aspects. (It is possible to hide 'toy' problems under a pile of expensive equipment.)

There are some attempts to design and implement more complete architectures (e.g. [9, 6, 17]) though these projects are still at a relatively early stage, and those involving complete physical robots include only a small subset of the motive management processes that interest us. Bates *et al.* [3, 2] attempt to implement 'broad and shallow' systems based on rules of a type that might emerge from the architecture (Fig. 1) that we are considering. This can be useful, e.g. for the purpose of new forms of computer-based art or entertainment involving simulated actors, though deeper models are needed if human-like behaviours are to be explained rather than simply simulated.

Most work on agent architectures assumes a fixed architecture and then attempts to demonstrate its properties and advantages over others, whereas we do not wish to restrict ourselves to any particular architecture, for we do not believe enough is known about what sorts of architectures are possible nor what their implications are. We therefore provide a framework within which alternative architectures can easily be constructed and their capabilities explored and evaluated. Part of the task is to devise good evaluation criteria [23, 25], e.g. performance criteria and cognitive modelling criteria.

5 Conclusion

Our toolkit is very general, but not so general as to be completely useless: a frequent consequence of generality. It can be used for a variety of experiments involving the explicit design of architectures, including architectures

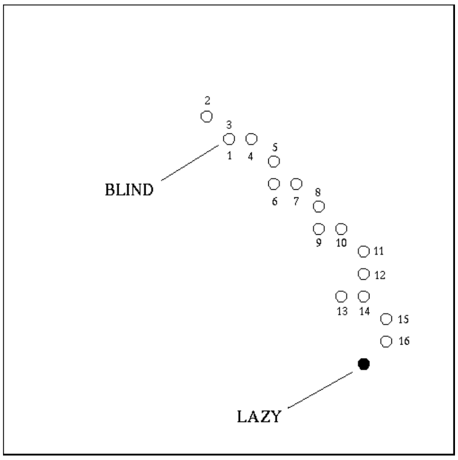

Fig. 5. A run (not in the training set) of the best blind and lazy agents discovered by the GA. The hollow circles with their numeric labels represent the positions of the blind agent at different time steps.

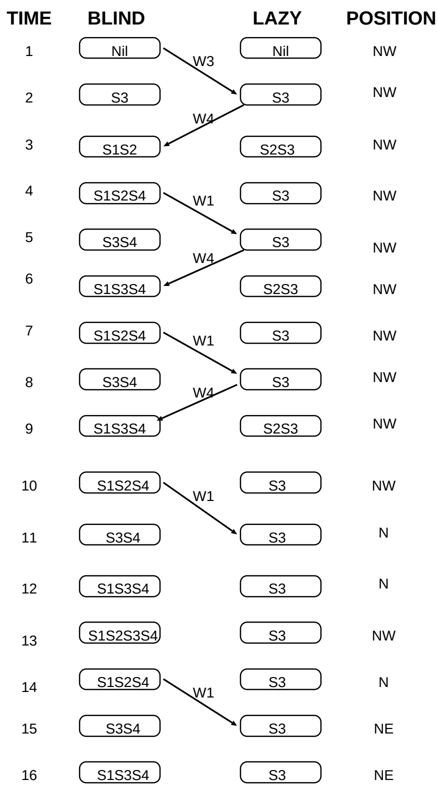

Fig. 6. Graphical representation of the messages exchanged between the agents and their mental states during the simulation in Figure 5.

that change themselves or use trainable sub-mechanisms. Colleagues at DRA are using it to develop simulation software for training army commanders. It can be used for evolutionary experiments, using genetic algorithms or similar techniques to evolve designs instead of creating them explicitly. It should be easy to copy the ideas in any Lisp-like language rich enough to support flexible rule-based processing. AI languages may run more slowly than C or C++, but compensate by simplifying the programming tasks and facilitating rapid prototyping. Increases in processor power and the use of multi-processor hosts will compensate further.

6 Acknowledgements

The work reported in this paper arises out of collaborative research involving Luc Beaudoin, Darryl Davis, Glyn Humphreys, Ian Wright, and other members of the Cognition and Affect project at the University of Birmingham [29]. Jeremy Baxter and Richard Hepplewhite (at DRA Malvern) are users of the toolkit and made several suggestions for improvement. This research is funded by the Renaissance Trust, the UK Joint Council initiative in HCI and Cognitive Science, and DRA Malvern.

Appendix A

Running an agent's internal processes

The default method used by sim_scheduler to run each object on the first pass in each time-slice can be summarised as follows (omitting the tracing and debugging mechanisms):

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

Control of relative speeds is supported by sim_speed(object), and ruleset_limit(ruleset).

Running the 'external' actions

On the second pass through the objects, the following is done by sim_scheduler.

```
    for object in objects do
        <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 generic ones but can be overridden by methods for sub-classes of the top level class.

Appendix B: Support for 'hybrid' mechanisms in Poprulebase

We have implemented and extended ideas from [16], allowing a rule to contain a complex 'filter' which operates simultaneously on a set of conditions, not all of which need be satisfied. The simplest format is:

```
[FILTER BFP C1 C2 ... Cn]
```

where **BFP** (boolean filter procedure) is a procedure that will be applied to a vector of **n** items derived by matching the **n** conditions **C1**,...,**Cn** against the current database. The whole condition will succeed only if the result returned by **BFP** is non-false.

A more complex version can determine which of a set of actions should be selected:

```
[FILTER VFP -> var C1 C2 ... Cn]
```

Here **VFP** (vector filter procedure) is applied to the vector of **n** items derived from the **n** conditions which returns either FALSE, in which case the condition fails, or a vector stored in the variable **var** for later use by a corresponding 'select' or 'map' action. Conditions may include variables whose values (e.g. sensor values — contrast [13]) are passed to the filter. This allows a single rule to perform a task that would otherwise require a complex set of 'AND' and 'OR' conditions. In some cases, the **VFP** or **BFP** in a filter condition may invoke a trainable neural net, programmed by being given examples ([16]).

The 'select' action has the following format:

```
[SELECT ?var A1 A2 ... Am]
```

where **var** is a variable containing a vector of length **m**, derived from a **VFP** in one of the 'filter' conditions of the rule. The non-false elements of the list will be used to select the corresponding actions to be performed from the set **A1 ... Am**.

The 'map' action has a similar format, but includes a mapping procedure **MP**:

```
[MAP ?var MP A1 A2 ... Am]
```

MP is applied to the value of **var**, and the list of actions **A1 ... Am**. The result of **MP** is interpreted as the list of actions to be performed.

References

1. J.A.D.W. Anderson, editor. *POP-11 Comes of Age: The Advancement of an AI Programming Language*. Ellis Horwood, Chichester, 1989.
2. J. Bates. The role of emotion in believable agents. *Communications of the ACM*, 37(7):122–125, 1994.
3. J. Bates, A. B. Loyall, and W. S. Reilly. Broad agents. In *Paper presented at AAAI spring symposium on integrated intelligent architectures*, 1991. (Available in SIGART BULLETIN, 2(4), Aug. 1991, pp. 38–40).
4. L.P. Beaudoin. *Goal processing in autonomous agents*. PhD thesis, School of Computer Science, The University of Birmingham, 1994.
5. L.P. Beaudoin and A. Sloman. A study of motive processing and attention. In A.Sloman, D.Hogg, G.Humphreys, D. Partridge, and A. Ramsay, editors, *Prospects for Artificial Intelligence*, pages 229–238. IOS Press, Amsterdam, 1993.
6. R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.
7. M.P. Georgeff. Agents and their plans, 1995. Invited lecture *Proc 14th Int. Joint Conf. on AI*.
8. David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Massachusetts, 1989.
9. B. Hayes-Roth. Intelligent control. *Artificial Intelligence*, 59:213–220, 1993.
10. B. Hayes-Roth. Agents on stage: Advancing the state of the art of ai. In *Proc 14th Int. Joint Conf. on AI*, pages 967–971, Montreal, 1995.
11. A. Sloman I.P. Wright and L.P. Beaudoin. Towards a design-based analysis of emotional episodes, 1996(to appear). Available at URL ftp://ftp.cs.bham.ac.uk/pub/groups/cog_affect in the file Wright_Sloman_Beaudoin_grief.ps.Z.
12. The KQML project is described in URL http://www.cs.umbc.edu/kqml. This WEB page also contains pointers to information on other languages, protocols, and agent frameworks, including AKL, Agent0, toolTalk, CORBA, etc.
13. P. Maes. Modeling adaptive autonomous agents. *Artificial Life*, 1(1):135–162, 1994.

14. M. L. Minsky. *The Society of Mind*. William Heinemann Ltd., London, 1987.
15. A. Newell. *Unified Theories of Cognition*. Harvard University Press, 1990.
16. R. Poli and M. Brayshaw. A hybrid trainable rule-based system. Technical Report CSRP-95-3, School of Computer Science, The University of Birmingham, March 1995.
17. D. Riecken(ed). Intelligent agents, 1994. Special Issue of *Communications of the ACM*, 37,7 July 1994.
18. H. A. Simon. Motivational and emotional controls of cognition'. Reprinted in *Models of Thought,* Yale University Press, 29–38, 1979.
19. A. Sloman. Motives mechanisms and emotions'. *Emotion and Cognition*, 1(3):217–234, 1987. Reprinted in M.A. Boden (ed), *The Philosophy of Artificial Intelligence*, 'Oxford Readings in Philosophy' Series, Oxford University Press, 231–247, 1990.
20. A. Sloman. On designing a visual system (towards a gibsonian computational model of vision). *Journal of Experimental and Theoretical AI*, 1(4):289–337, 1989.
21. A. Sloman. Prolegomena to a theory of communication and affect. In A. Ortony, J. Slack, and O. Stock, editors, *Communication from an Artificial Intelligence Perspective: Theoretical and Applied Issues*, pages 229–260. Springer, Heidelberg, Germany, 1992.
22. A. Sloman. The mind as a control system. In C. Hookway and D. Peterson, editors, *Philosophy and the Cognitive Sciences*, pages 69–110. Cambridge University Press, 1993.
23. A. Sloman. Prospects for ai as the general science of intelligence. In A.Sloman, D.Hogg, G.Humphreys, D. Partridge, and A. Ramsay, editors, *Prospects for Artificial Intelligence*, pages 1–10. IOS Press, Amsterdam, 1993.
24. A. Sloman. Explorations in design space. In *Proceedings 11th European Conference on AI*, Amsterdam, 1994.
25. A. Sloman. Exploring design space and niche space. In *Proceedings 5th Scandinavian Conference on AI*, Trondheim, 1995. IOS Press.
26. A. Sloman. Filtering of rules in lib poprulebase, 1995. Available at URL ftp://ftp.cs.bham.ac.uk/pub/dist/poplog/prb/help/pop_filter.
27. A. Sloman. Poprulebase help file, 1995. Available at URL ftp://ftp.cs.bham.ac.uk/pub/dist/poplog/prb/help/poprulebase.
28. A. Sloman and M. Croucher. Why robots will have emotions. In *Proc 7th Int. Joint Conf. on AI*, Vancouver, 1981.
29. I.P. Wright. A summary of the attention and affect project, 1994. Available at URL ftp://ftp.cs.bham.ac.uk/pub/groups/cog_affect in the file Ian.Wright_Project_Summary.ps.Z.

This article was processed using the LaTeX macro package with LLNCS style