Oxford Computing Lab, 30 Jan 2001

Birmingham 13 Oct 2003 – York 11 Feb 2004

SimAgent:
TOOLS FOR DESIGNING MINDS

(A toolkit for philosophers and engineers)

Aaron Sloman

http://www.cs.bham.ac.uk/~axs/

School of Computer Science

The University of Birmingham

INCLUDING IDEAS FROM:

Jeremy Baxter (DERA), Richard Hepplewhite (DERA)
Riccardo Poli, Brian Logan, Darryl Davis, Catriona Kennedy
Matthias Scheutz, Nick Hawes and others

THE TOOLKIT IS AVAILABLE WITH SOURCES AT THE BIRMINGHAM FREE POPLOG SITE

http://www.cs.bham.ac.uk/research/poplog/

Further information on the toolkit is here

http://www.cs.bham.ac.uk/ axs/cogaff/simagent.html

This and other related slide presentations are available here

http://www.cs.bham.ac.uk/research/cogaff/talks/

What is an AI toolkit?

There are various levels at which we can build machines, some much harder to start from than others:

- Physical components
- Digital electronic components
- Machine code for an existing computer
- Assembly language for an existing computers
- Source language for a compiler for various computers
- Higher level languages (lots and lots of them, making different things easy ...)
- Operating systems
- Re-usable procedure libraries
- Architecture toolkits: Several for AI
  - SOAR
  - ACT, ACT-R, ACT-RP
  - PRS, JACK
  - COGENT
  - MOZART
  - SIMAGENT

Requirements for our toolkit

Minimal requirements:

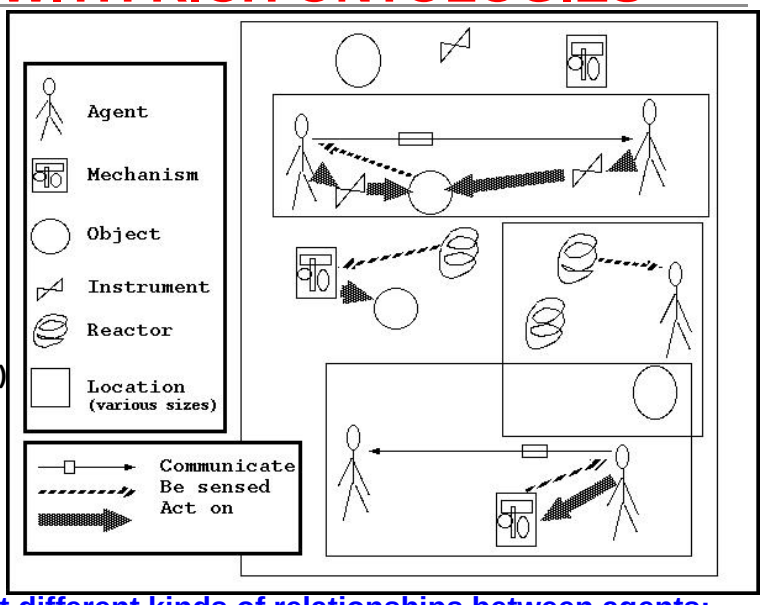
- **The toolkit must support scenarios with rich ontologies.**
  (Many kinds of objects, properties, relationships, internal and external actions, processes....)
- **The toolkit should support different sorts of agents, with different architectures.**
- **Agents can have complex internal architectures.**
  E.g. different components of a complex architecture may include:
  - **different kinds of information,**
  - **different forms of representation,**
  - **different kinds of mechanisms**
- **Concurrency is required at all levels:**
  - **Different entities in the "world" run concurrently.**
  - **Different components of individual agents run concurrently, performing various tasks in parallel, e.g.**
    reacting, deliberating, carrying out plans, generating motives, learning, evaluating, self-monitoring, etc.
- **The toolkit should support research on machines that understand what they are doing and monitor and control some of their own internal processes.**

Other toolkits may have other requirements!

The toolkit must support SCENARIOS WITH RICH ONTOLOGIES

Diverse concurrently active entities, e.g.:

- AGENTS: which can communicate with one another,
- MECHANISMS: which sense and react to other things,
- INSTRUMENTS: which can act if controlled by an agent,
- "REACTORS" which do nothing unless acted on (e.g. a mouse-trap)
- LOCATIONS: of arbitrary extents with various properties, including continuously varying heights, terrain features, etc., etc.

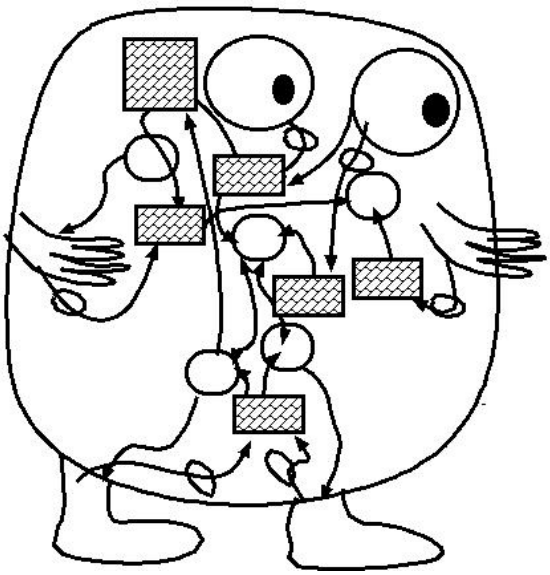


The toolkit should also support different kinds of relationships between agents:

- **X perceives Y**
- **X acts on Y**
- **X communicates with Y**
- **One event or process causes or modifies another**

INSIDE ONE AGENT

Agents can have complex internal architectures



- Rectangles represent short- or long-term databases or communication buffers
- Ovals represent processing units.
- Arrows represent flow of information, including control information

Some components are linked to sensors and motors (physical or simulated)

Some are connected only to other internal components

Some sub-mechanisms change states continuously others only discretely.
(The former can only be approximated on computers.)

NB: we are mostly talking about virtual machine components whose relationships to brain-mechanisms may be very complex (a type of implementation relationship.

For some architectures the diagram is too 'flat': there should be different structures at different levels of abstraction.

See also http://www.cs.bham.ac.uk/research/cogaff/talks/#super

The toolkit should support different sorts of agents, with different architectures

E.g. agents:

- performing different sorts of tasks
- with various kinds of sensors and motors (either simulated or physical)
- connected to different kinds of concurrently active internal processing modules changing at different speeds
- with different kinds of internal short term and long term databases,
- with some components monitoring or controlling others
- using different forms of representation and reasoning
- with different levels of abstraction, and different levels of control

See the other slides on architectures and requirements for agents here:

http://www.cs.bham.ac.uk/research/cogaff/talks/

It should be possible to include different sorts of agents and different sorts of objects in the same simulation, and it should be easy to add and new sorts.

E.g. the sheepdog example includes sheep, trees, the pen and the dog. It is easy to add more of each kind, and to add new kinds, e.g. wolves.

Concurrency is required at all levels

Different entities in the "world" run concurrently.

Different components of individual agents run concurrently, performing various tasks in parallel, e.g.

- Perception, of different kinds
  - Using different senses: vision, hearing, smell, touch, proprioception,
  - Doing different levels of perceptual analysis,
- Acting under control of continuous feedback loops,
- Linguistic processing (understanding, communicating)
- Learning of various kinds,
- Triggering "alarms" (various kinds of emotion)
- Reasoning
- Generating new motives, evaluating motives, comparing motives,
- Planning, executing plans,
- Generating and changing internal control states (emotions, moods, etc.)
- Monitoring internal processes (reflection, meta-management).
- Monitoring and reasoning about other agents.

The running speeds of different components in the simulation may vary.

Compare M.Minsky The Society of Mind.

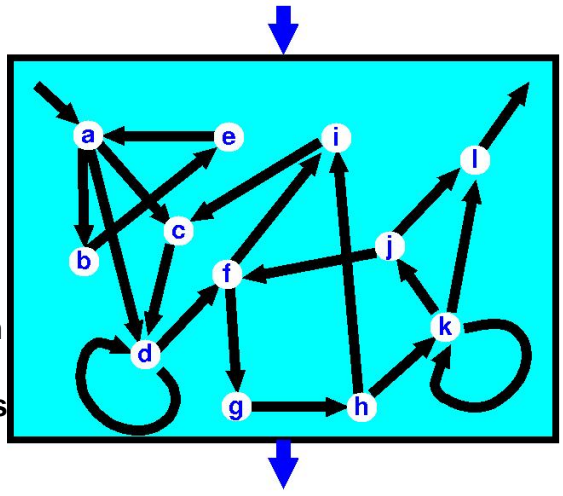
Perhaps we need to think of an "ecosystem of mind".

See Cogaff papers http://www.cs.bham.ac.uk/research/cogaff/

Atomic State Functionalism

Functionalism is one kind of attempt to understand the notion of virtual machine, in terms of states defined by a state-transition table: there's a total state which affects input/output contingencies, and each possible state can be defined by how inputs determine next state and outputs.

E.g. if the system is in state d then depending on the next input it may remain in state d or switch to state f, and the output it produces, if any, will depend on the state it is in and the input.



E.g. see Ned Block's accounts of functionalism.
http://www.nyu.edu/gsas/dept/philo/faculty/block/papers/functionalism.html

Many so called "cognitive architectures" conform to this model, though some of the state transitions may involve very complex processes and some transitions may occur without any input or without any output.

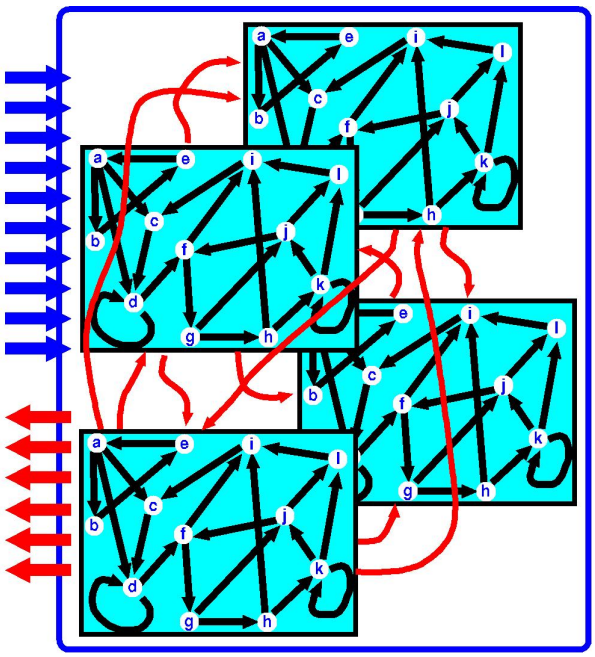
Our toolkit needs to support a richer, deeper, notion of functionalism, which assumes a complex machine with many interacting components

Virtual machine Functionalism

We assume a kind of functionalism that allows many virtual machine components to co-exist and interact, including some that observe others, all within one agent.

So there is not just a single (atomic) state which switches when some input is received.

- **The different states may change on different time scales: some change very rapidly, others very slowly, if at all.**
- **Some sub-states may change in complexity over time, e.g. growing a parse tree, or a plan.**
- **Sub-states can vary in their granularity: some sub-systems may be able to be only in one of a few states, whereas others can switch between vast numbers of possible states (like a computer's virtual memory).**
- **Some may change continuously, others only in discrete steps.**



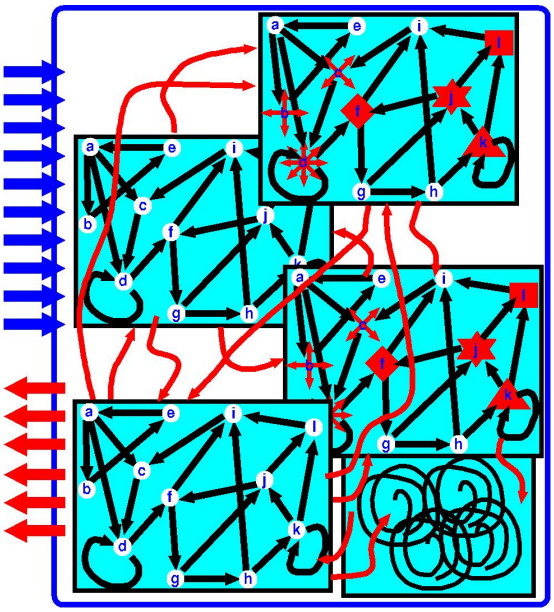
As theorists and designers we wish to be able to explore such systems and understand their implications, their strengths their weaknesses. (In the footsteps of evolution).

More on Virtual Machine Functionalism

The situation is more complex than the previous diagram suggests.

- **There need not be a fixed set of sub-processes.**
- **Within each sub-process there need not be a fixed set of possible states.**
- **The individual states may vary in complexity.**
- **Some sub-systems may vary continuously instead of discretely.**
- **Sub-systems may change their speeds of operation asynchronously, instead of everything always changing in step.**

The figure illustrates this.



Our toolkit should, in principle, cope with all of this, though some aspects may be only approximated.

Misleading traditions in AI

- 'Virtual Machine Functionalism' (VMF) does not take a stand on the forms of representation used: some sub-systems may use logic, others executable procedures, others forward chaining rule-systems, others neural nets, others image-structures, others trees or graphs, others physical feedback loops, ...
- Most work in AI does not fit the sort of described here, as most AI systems do not have such diverse, concurrently acting sub-systems.
- VMF is not consistent with the view often attributed to, or even espoused by, AI researchers that intelligent systems engage in a sense-plan-act, or more generally, perceive-think-act cycle – for that implies a sequence of well-defined state-changes, whereas VMF allows components to change on different time-scales

  For instance, there could be rapidly changing percepts and motor control processes, less rapidly changing plans and states of plan execution, less rapidly changing goals, and many enduring attitudes, personality traits, beliefs, collections of skills, etc.
- Despite inconsistencies with many actual AI systems and conflicts with what AI theorists may write, VMF comes close to implicit requirements for ambitious AI systems, including robots with multiple sensors and multiple effectors.
- VMF is implicit in the switch from concern with *algorithms* to concern with *architectures* over the last two decades, even if many who discuss architectures consider only the special cases that are not adequate for a human-like agent: So we need appropriate tools.

Principled exploration of design space

How can we investigate designs for complex virtual machines?

Very often researchers take a problem and try to produce a design that solves it, usually by modifying or combining previously constructed designs and mechanisms used to implement them.

This may suffice for engineers having to deliver results in a hurry, but a more principled approach may be useful for science and engineering in the long term.

- In particular we need to understand the space of possible designs and the space of possible sets of requirements and the relationships between the two spaces.

  http://www.cs.bham.ac.uk/research/cogaff/talks/#talk4
- However doing this for the full space of designs is too ambitious – and intractable.
- So it is useful to explore relatively small neighbourhoods in design space.
- One such neighbourhood is specified (at least partially) by the CogAff schema – a conceptual framework that has developed over a decade or more.
- The toolkit has evolved in that time to help with exploration of designs inspired by the schema, which covers a wide variety of architectures for integrated multi-functional agents.

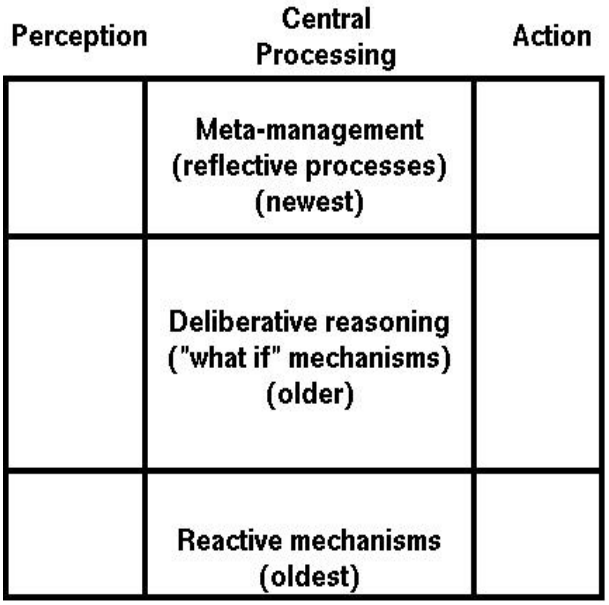
The Birmingham 'CogAff' Architecture Schema

This is a conceptual framework for describing architectures with multi-level concurrently active components within perceptual, central and motor sub-systems.

The different levels correspond to different stages in evolution (not all levels found in all animals, in or new-born infants!).

They also involve different types of abstraction, different forms of representation, etc.

A good toolkit should support diverse instances of this schema, varying as regards which boxes contain what sorts of mechanisms and how they are connected, what forms of representation they use, etc.



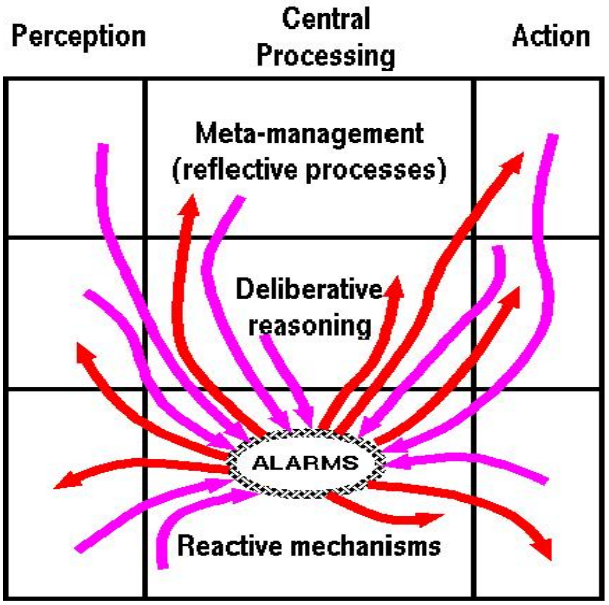
It should also allow some parts of the architecture to be run at different speeds relative to others, to reflect individual differences, or different states of arousal, etc.

One elaboration of the 'CogAff' Schema

An example elaboration of the schema:

One or more reactive "alarm" mechanisms could get input from many components of the architecture and take rapid decisions about what to do then send signals to many other components, E.g. flee, fight, feed, freeze, mate (the five Fs)

Is that what the amygdala does?
Brain stem?
Other protective reflexes?

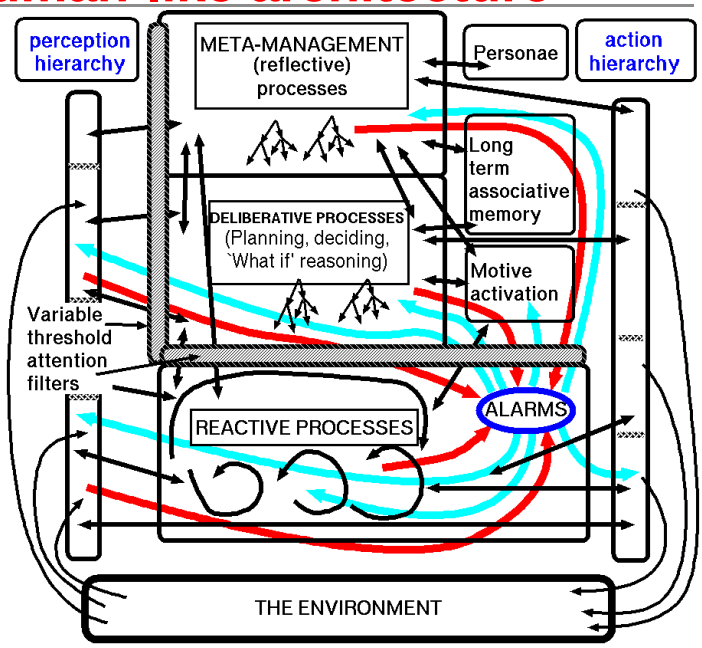


This could account for several different sorts of emotions, e.g. primary, secondary, tertiary emotions, defined in relation to the levels involved.

A special case
H-Cogaff: A human-like architecture

The H-Cogaff architecture is a special case of CogAff still under development.

In H-Cogaff, all the boxes of the CogAff schema have occupants and there are many kinds of links. So far no working model of the whole system has been implemented.
The ideas are still under development and too complex to discuss here.



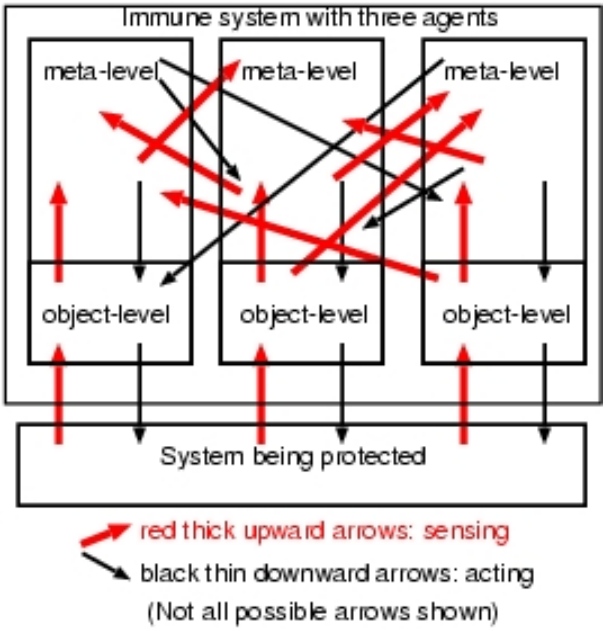
Many of the ideas are discussed in the Cognition and Affect project papers and talks:

http://www.cs.bham.ac.uk/research/cogaff/
http://www.cs.bham.ac.uk/research/cogaff/talks/

Catriona Kennedy's Mutual meta-management

This was an unexpected development of the CogAff schema.

Towards robust immune-system-like intruder-monitoring: instead of having a single monitor or a hierarchy of monitors have many of them all monitoring the system being protected and also monitoring each other, including monitoring each others' monitoring.
See http://www.cs.bham.ac.uk/~cmk



This work used the SimAgent toolkit, and imposed requirements for extended facilities to support mutual monitoring.

These facilities allow processes in one agent to inspect databases in other agents – if they have pointers to them.

A related requirement arose out of a project using the toolkit at Hull University.

APPROACHES TO DIVERSITY

Tools to support this variety of architectures cannot be expected to anticipate all types of entities, causal and non-causal relationships, states, processes, etc. which can occur.

So users should be able to extend the ontology, and kinds of interactions, as new requirements arise.

Toolkit designers have adopted various approaches:

- **User provides *logical axioms* defining new classes and sub-classes and new behaviours, from which a toolkit builds a system.**
- **User assembles architectures diagrammatically, e.g. moving boxes and arrows around on the screen, selected from various menus.**
- **User writes lots of low level code, in an AI language, or worse, in C or C++ or Java.**
- **User somehow guides an evolutionary process that creates the desired system – e.g. the user might provide a fitness function for evaluating proposed architectures.**
- **User defines new classes and sub-classes using an object oriented programming language (e.g. with multiple-inheritance), and specifies behaviours for each class in a special behaviour-description language (which can invoke other languages, providing different mechanisms, as needed)**

THE LAST IS THE APPROACH SUPPORTED BY SIMAGENT.

- **Most of the behaviours are internal: perceiving, interpreting, remembering, reasoning, learning, planning, deciding, generating motives, detecting conflicts, resolving conflicts, triggering alarms, etc., and the behaviour description language (poprulebase) caters for this.**
- **It is impossible to get a design right in one go: so an iterative, incremental, exploratory process is required.**

Examples of uses of the tools

There are a few online movies showing simple examples of the toolkit running:

http://www.cs.bham.ac.uk/research/poplog/figs/simagent/

Other examples available for demonstration during the talk:

1. **The marching platoons demo (simple swarming).**
2. **Sim_feelings: what is perceived changes a global state, which affects behaviour.**
3. **Using RCLIB to build a control panel.**
4. **The sheepdog scenario (Peter Waudby, Tom Carter)**
5. **The hybrid (reactive/deliberative) sheepdog (Marek Kopicki).**
6. **The gblocks demo (not strictly SimAgent, but illustrates the graphics and some grammar utilities available)**
7. **Dean Petters' simulation exploring development of attachment in infants.**
8. **Steve Allen's Abbot system developed for his PhD**
   http://www.cs.bham.ac.uk/research/poplog/abbott
9. **Several student projects, e.g. casualty ward, space wars, bugs, a Braitenberg toolkit.**
10. **Earliest demo: Riccardo Poli's RIB (Robot in a Box - IJCAI/ATAL 1995) (included neural and symbolic mechanisms sharing subtasks).**

Possible future work includes using the toolkit to implement the "mind" of a robot, e.g. with either an on-board PC running linux, or a radio link etc. (With Jeremy Wyatt: See http://www.cs.bham.ac.uk/research/projects/cosy).

Other projects
(Undergraduate, MSc, PhD, Researchers)

- Predator-prey (Nick Hawes)
- Bee-scenarios (various)
- Ian Wright (Minder 1)
- Catriona Kennedy (mutual and self monitoring of monitoring)
- Anytime planning in computer games (Nick Hawes)
- Anytime planning in a hierarchy of abstraction spaces with multiple soft constraints (Brian Logan).
- Tile-world and other things at Nottingham.
- Various projects by Darryl Davis at Hull university.
- Matthias Scheutz – Simworld at University of Notre Dame.
  http://www.nd.edu/%7Eairolab/simworld/index.html

There are many trade-offs in the design of tools

E.g. there are trade-offs

- between flexibility/generality of the toolkit and ease of use.
- between flexibility and efficiency.
- between run-time efficiency and support for interactive development.
- Between simplicity of the syntax or development interface and the variety of things the toolkit can do.

See also:
http://www.cs.bham.ac.uk/research/cogaff/talks/#talk11
(On AI vs Software Engineering development environments.)

The design of SimAgent aims to optimise

- generality (diversity of architectures)
- flexibility: easy modification or re-design of architectures
- support for multiple types of mechanisms, including self-awareness
- support for flexible run-time de-bugging by modifying rules or procedures in a running system.

As a result of the generality and flexibility, it takes longer to learn to use than some more restricted toolkits.

This could be alleviated by developing higher level tools and libraries aimed at specific classes of architectures, especially tutorial examples.

Support for multiple paradigms

SimAgent permits use of several different programming paradigms:

- Conventional procedural and functional programming.
- List processing and pattern matching.
- Rule-based programming in POPRULEBASE
- Use of meta-rules (illustrated later)
- Support for (simple) 'reason-maintenance' mechanisms
  Automatically retract items whose justifications no longer hold.
- Object oriented programming OBJECTCLASS
  Including generic functions and multiple inheritance.
- Event-driven programming
  The X window system and RCLIB, and agent class methods.
- Other computational paradigms needed for particular applications,
  E.g. neural nets or evolutionary mechanisms.
- Extendable syntax and semantics (macros and beyond)
- Invocation of other languages as needed
  PROLOG, ML, LISP, C, others via C ...
- Automatic store management and FAST garbage collection (from Poplog).
  (Essential for programs that frequently create temporary structures of many kinds.)

It should be possible to re-implement SimAgent in Common Lisp.
It would be probably be difficult in most other languages.

The architecture of the toolkit:

SimAgent is built on Pop-11 subsystem of Poplog[*] extended with:

- POPRULEBASE
  an unusually flexible, forward-chaining pattern-driven rule interpreter, able to invoke arbitrary procedures in its conditions and actions, with meta-rules, and support for hybrid architectures (e.g. a rule's conditions can run a neural net).
- SimAgent SCHEDULER
  Providing concurrently running, interacting agents, each with concurrently running, interacting internal components.
- RCLIB
  Object-oriented "relative coordinates" graphical tools, supporting graphical interfaces linked to a simulation, including declaratively specified control panels.
  Multiple-inheritance used to achieve modularity.
- OBJECTCLASS
  (designed by Steve Leach). Like CLOS, it supports object oriented programming with multiple inheritance and generic functions (multi-methods).
  Important for re-usability and event driven programming – allows new agent and object classes to be added, and default behaviours to be overridden for sub-classes
- LIBRARIES: re-usable code and documentation (including tutorials)

[*] For information on Pop-11 see http://www.cs.bham.ac.uk/research/poplog/primer/
For information on Poplog see http://www.cs.bham.ac.uk/research/poplog/freepoplog.html

Specifying an agent's architecture

An agent's architecture consists of a rulesystem which is made up of a collection of rulesets and rulefamilies all run in concurrent threads, in a time-sliced (non-preemptive) scheduler.

- Ruleset:
  A collection of condition-action rules with various processing strategies.
  NOTE: a rule can act as a 'rich' communication channel if its conditions check some subsystems and its actions alter other subsystems: a powerful feature noticed by using the kit.
- Rulefamily:
  A collection of rulesets, with control switching – only one is active at a time.
- Ruleset-specific and/or dynamic setting:
  of operation modes, e.g. conflict resolution, tracing modes.
- Interactive debugging and development:
  Edit and recompile a ruleset in a running system.

E.g. a rulesystem can be made up of a collection of different rulesets and rulefamilies concerned with processes like sensing, interpreting sensory data, generating emotional reactions, interpreting sentences, planning, learning, forming intentions, continuing execution of ongoing behaviours, etc.

Poprulebase

Poprulebase is the default language for specifying rulesets and rulefamilies.

It is a very rich behaviour-specification language, built on Pop-11.

It supports many varieties of conditions and actions, including conditions that run arbitrary code and actions that run arbitrary code.

Arbitrary code includes running or interrogating neural nets for example.

Execution is multi-threaded, with facilities for varying relative 'speed' of threads.

It is also extendable: users can define new kinds of conditions and actions.

The ability to use rules with conditions checking one sub-system and actions altering another, permits very rich communication within a system.

(Does something like that happen in brains? They are very highly inter-connected.)

Rulesets in Poprulebase

Each rule in a ruleset has conditions and actions of various sorts

The format for a ruleset definition

```
define :ruleset rulesetname;
    RULE rulename1
        conditions
            ==>
        actions
    RULE rulename2
        conditions
            ==>
        actions
    RULE rulename3
        ......
enddefine;
```

The conditions and actions can be simple or complex

- **Simple conditions and actions:**
  Merely check or alter a database.
- **Complex conditions and actions:**
  can invoke 'lower level' mechanisms in another language or subsystem
  e.g. (Pop-11, C, Prolog, neural nets...)

Ruleset definitions can start with declarations of processing strategy, tracing mode, etc.

Further details on Poprulebase can be found here
http://www.cs.bham.ac.uk/research/poplog/newkit/prb/help/poprulebase

Rulefamilies

A rulefamily is composed of a collection of rulesets, only one of which is active at a time.

This is the default format for specifying a rulefamily

```
define :rulefamily <name> ;

    ruleset: <ruleset name>
    ruleset: <ruleset name>
    ruleset: <ruleset name>
    ......

enddefine;
```

- **The first named ruleset is the one that is initially active.**
- **Control can be transferred between the rulesets in a rulefamily using actions with these keywords,**

  `SAVERULESET RESTORERULESET SWITCHRULESET PUSHRULESET POPRULESET`

  E.g.

  `[RESTORERULESET <rulesetname>]`

- **A ruleset remains the current one unless it explicitly transfers control to another ruleset in the rulefamily.**

Further details on rulefamilies can be found in HELP RULESYSTEMS:
http://www.cs.bham.ac.uk/research/poplog/newkit/prb/help/rulesystems

Rulesystems (for use in SimAgent)

A rulesystem, defining the internal processing architecture of an object or agent, is composed of a collection of rulesets and rulefamilies.

- Rulesystems provide for simulated concurrency within and between agents.
- On each scheduler 'time-slice' every item in every rulesystem gets a chance to run.

This is the default format for specifying a rulesystem

```
define :rulesystem <name> ;
    <optional global specifications>
    include: <rulefamily name>
    include: <ruleset name>
    include: <ruleset name> with_limit = <integer>
    include: <rulefamily name> with_limit = <integer>
    include: <ruleset name> with_interval = <interval>
    include: <rulefamily name> with_interval = <interval>
enddefine;
```

- The limit specification specifies how much 'time' the item gets on each time-slice
- The interval specification can specify gaps between runs, or the probability of running.

Further details on rulesystems can be found in HELP RULESYSTEMS:

http://www.cs.bham.ac.uk/research/poplog/newkit/prb/help/rulesystems

Ruleset: Code example

Here are some examples of rules, using conditions and actions that access the database (plus some [SAY ...] actions, for tracing).

Assume that a user has typed a sentence which was stored in the database in the format:

[sentence blah blah blah blah]

```
RULE hello
  ;;; A disjunctive condition matched against the sentence
  [OR
      [sentence == hi ==]
      [sentence == hello ==] ]
       ==>
  [SAY Greetings and welcome to the consultation]
  [NOT sentence ==] ;;; delete sentence -- after dealt with it

RULE I_feel
  [sentence I feel ??x]
       ==>
  [SAY Do you often feel ??x ?]
  [NOT sentence ==]

;;; Now a rule with two pattern variables
RULE someone_said
  [OR
      [sentence ??x1 said ??x2]
      [sentence ??x1 thinks ??x2] ]
      ==>
  ;;; Remember a variant of what was said
  [old_sentence ??x1 thought ??x2]
  [SAY Does anyone besides ??x1 think ??x2 ?]
  [NOT sentence ==]
```

A ruleset for an Eliza-like program could have many more rules than this.

Some rules are communication channels

Rules with complex conditions and actions, can function as communication channels between sub-mechanisms.

- Complex conditions and actions can run arbitrary code
- Conditions of a rule can get information from one part of a complex architecture while the actions of that rule transform the information and deliver it to another part.

  This allows

  - A rule that simply transfers intermediate results of one process (e.g. a perceptual process) to another process (e.g. a motive-generation process)
  - A rule in one part of the system that monitors behaviour in another part, by checking its intermediate data-structures.
- Alternatively, rules that get information from different sub-systems can put it in a location where various rules can find it and transfer it to other sub-systems.

Example communication channel

Rules with complex conditions and actions, can function as communication channels between sub-mechanisms.

E.g. if different rulesets deal with a visual-data subsystem and a visual-memory sub-system, then a bridging ruleset might include this rule:

```
RULE find_squares
    [visual_data ?obj1 ?obj2 ?obj3 ?obj4][->> Data]
    [WHERE square_corners(obj1, obj2, obj3, obj4)]
        ==>
    [DEL ?Data]                                        ;;; delete item
    [LVARS [loc = centroid(obj1, obj2, obj3, obj4)]] ;;; compute location
    [POP11 train_net(obj1, obj2, obj3, obj4, loc)]   ;;; train a network
    [LVARS [time = sim_cycle_number]]
    [visual_memory square ?loc ?time]                  ;;; symbolic record
```

Another ruleset might ‘clean up’ old memories, or implement a memory-decay mechanism.

Yet another might compare the newest memories with older ones and notice that a square has been seen where none was previously.

(The variable `sim_cycle_number` is provided in the SimAgent extension to Poprulebase.)

Example non-standard conditions and actions

Poprulebase supports both simple and compound conditions and actions. Many of the conditions and actions simply interrogate or update a database. However there are also non-standard compound conditions which can do more complex things.

- A [WHERE ...] condition runs arbitrary code that produces a boolean result.
  E.g. it could run a neural net to recognize patterns.

- An [LVARS ....] action can run arbitrary code and introduce new variables
  These variables are lexically scoped, and can be accessed both in patterns in the remainder of the rule and in Pop11 code in conditions and actions in the rule.

- A [POP11 ...] action runs arbitrary code.
  It could train a neural net, or use a neural net, for instance.

See http://www.cs.bham.ac.uk/research/poplog/newkit/prb/help/

Illustrating flexibility: Meta-rules

A meta-rule using an [ALL ...] condition can get its conditions and actions from the database instead of having them 'hard-wired'.

E.g. in a ruleset concerned with planning we might have this rule.

```
RULE check_constraints
  ;;; see if there is any constraint information
  [constraint ?name ?checks ?message]
  ;;; if so, run all the conditions in ?checks
  [ALL ?checks]
      ==>
  ;;; the conditions are simultaneously satisfied, so
  [constraint_violated ?name]     ;;; record violation
  [SAY ??message]                 ;;; print out message
  [RESTORERULESET fix_problem]    ;;; switch control to fixing mode ruleset
```

Constraints can be added and removed dynamically, without changing the above rule.

Illustrating flexibility: Meta-rules

A meta-rule using an [ALL ...] condition can get its conditions and actions from the database instead of having them 'hard-wired'.

E.g. in a ruleset concerned with planning we might have this rule.

```
RULE check_constraints
  ;;; see if there is any constraint information
  [constraint ?name ?checks ?message]
  ;;; if so, run all the conditions in ?checks
  [ALL ?checks]
      ==>
  ;;; the conditions are simultaneiously satisfied, so
  [constraint_violated ?name]     ;;; record violation
  [SAY ??message]                 ;;; print out message
  [RESTORERULESET fix_problem]    ;;; switch control to fixing mode ruleset
```

That rule might pick up these constraints in a database for example:

```
  ;;; Constraint: robot should not leave things with what they like to eat.
  [constraint Eat
      [[isat ?thing1 ?location]
          [NOT isat robot ?location]
          [?thing1 can eat ?thing2]
          [isat ?thing2 ?location]]
      [?thing1 can eat ?thing2 GO BACK]]

  ;;; Constraint: prevent new states that are already in the history.
  [constraint Loop
      [[state ?state] [history == [= ?state] == ]]
      [LOOPING previously_in_state  ?state]]
```


SimAgent extends poprulebase, and provides

- **Some default classes (object, agent, ...)**
- **Default methods for sensing, acting and communicating.**

  The default sensing method for agents supports different sensors with different capabilities; it runs them all and moves their outputs to internal sensory-buffers, for processing.

  The default classes and methods can be modified or extended for particular applications.
- **A default communication protocol.**
- **A default scheduler supporting flexible multi-threading (discrete event simulator):**

  A collection of rulesets and rule-families, executed "in parallel", forms a rulesystem defining an agent's internal processing architecture, as explained on a previous slide.
- **Support for self-monitoring and meta-management (explained later)**
- **SIM_PICAGENT**
  - **Supports linkage between simulation events and screen events (e.g. mouse events)**
  - **Allows individual agents to be in multiple windows**
- **SIM_HARNESS**

  Provides default control panel and startup mechanisms
- **A growing library of utilities, demonstrations and tutorial examples, using an extension of the poplog library and documentation mechanisms.**

Self-monitoring and meta-management

SimAgent makes it possible for an agent to inspect and alter its own architecture.

- Each agent's rulesystem is represented as a collection of items in its database. I.e. the architecture consists of mechanisms for operating on a database which contains the architecture, in the following formats:

```
[RULE_SYSTEM <name> <rulecluster> <rulecluster> <rulecluster> ... ]
[RULE_CLUSTER <name> <ruleset or rulefamily> <limit or interval> ]
[RULE_SYSTEM_STARTUP <4-element vector>] (optional control information)
```

  This means that rules in the current rulesystem can fairly easily access and change components of the rulesystem, allowing learning, development, and 'self-repair'.

  For further information about how a rulesystem and its components are defined, and how they are stored in the agent's database, see
  http://www.cs.bham.ac.uk/research/poplog/newkit/prb/help/rulesystems
  http://www.cs.bham.ac.uk/research/poplog/newkit/sim/help/newkit

- Various kinds of tracing can optionally store records internally for subsequent meta-management processing, instead of printing out trace information.
- Poprulebase allows meta-rules which get their conditions and actions from the database, as explained above.

Note: Some of the self-monitoring mechanisms were partly specified by Catriona Kennedy.

More on Self-monitoring and meta-management

A meta-management ruleset RM may get information from a 'lower-level' sub-system S in various ways.

- MM may continuously observe the contents of the data-bases created and modified by S, because conditions in MM check the data-bases (and other things) acted on by S.
- S may include mechanisms which produce special items of information recording its actions, specifically for MM to observe.
- Some parts of the low-level mechanisms of SimAgent (e.g. the condition-checking mechanisms) have been 'instrumented' so as to allow them in certain contexts to produce internal records (traces) of their operation, which allow other sub-systems to monitor the processes – e.g. which conditions were tested, which succeeded and failed, etc.

See http://www.cs.bham.ac.uk/research/poplog/newkit/prb/help/poprulebase

Look for section headed:
User-definable procedures for self-monitoring
and following sections.

Users can define new sub-classes, and extend or replace the methods.

THERE IS NO FIXED ARCHITECTURE: ONLY A FLEXIBLE FRAMEWORK FOR EXPLORING A VARIETY OF ARCHITECTURES.

Each agent's architecture can include:

- **condition-action rules**
  In a flexible, user-extendable formalism, supporting many varieties of conditions and actions including some that invoke sub-symbolic mechanisms.
- **rulesets**
  composed of a collection of rules which work together to perform some task
- **rule-families**
  Consisting of a group of rulesets, only one of which is active at any time – allowing context-sensitive processing modes
- **a rulesystem**
  Made up of a collection of rulesets and rulefamilies which run in parallel. Each agent has one rulesystem at any time, though it can change over time.
- **Various default methods are provided for sensing, acting, communicating, tracing**
  But these can be over-ridden for new classes, or for individuals.
- **A database (or set of databases) which function as:**
  long term knowledge stores, temporary workspaces,
  communication channels between sub-systems

By default agents do not access the internals of other agents, but for some simulations (e.g. some inspired by Minsky's 'Society of Mind' idea), access across agent boundaries is possible.

Rapid prototyping and self-modifying software

Use of Pop-11's incremental compiler, along with heavy use of indirection makes it easy to experiment with changes and extensions to a running system without having to re-start every time.

This depends on support for dynamic replacement of modules (at run time), which is essential for:

- Debugging complex systems
- Self-modifying systems
- Rapid prototyping
  i.e. rapid exploration, testing, evaluation, etc. — all required when you don't start off with a precisely defined, well understood problem.

  E.g. by default the toolkit allows a recompiled ruleset to simply replace the previous version for all agents that use it.
  However, there can be a slight efficiency cost, and this cannot work for agents that modify their rulesets while running.
  For simulations where many agents use the same rulesets it can save a lot of space.

As far as I know, ONLY AI programming languages (e.g. Common Lisp and Pop-11) provide full support for all of these features.
Users of other languages can replicate some of the features by writing interpreters for subsets of the AI languages.

SIMAGENT: IS BASED ON POPLOG ESPECIALLY POP-11

POPLOG: a multi-language AI development environment with incremental compilers for several languages:

- **Pop-11**
  a Lisp-like, incrementally compiled language, with a Pascal-like syntax, even more extendable than Lisp – used to implement incremental compilers for all the other languages, which it can invoke if necessary.
- **Prolog**
- **Common Lisp**
- **Standard ML**

It was the original development environment for Clementine, a data-mining system used world wide, developed by ISL, and now sold by SPSS.

Pop-11 provides

- **facilities for adding new incremental compilers**
- **a rich interface to the X window system**
- **a very fast general garbage collector**
- **access to operating system facilities**
- **light-weight processes**
- **a built in pattern matcher**
- **large and easily extended collection of code and documentation libraries and AI/Cognitive Science teaching materials**

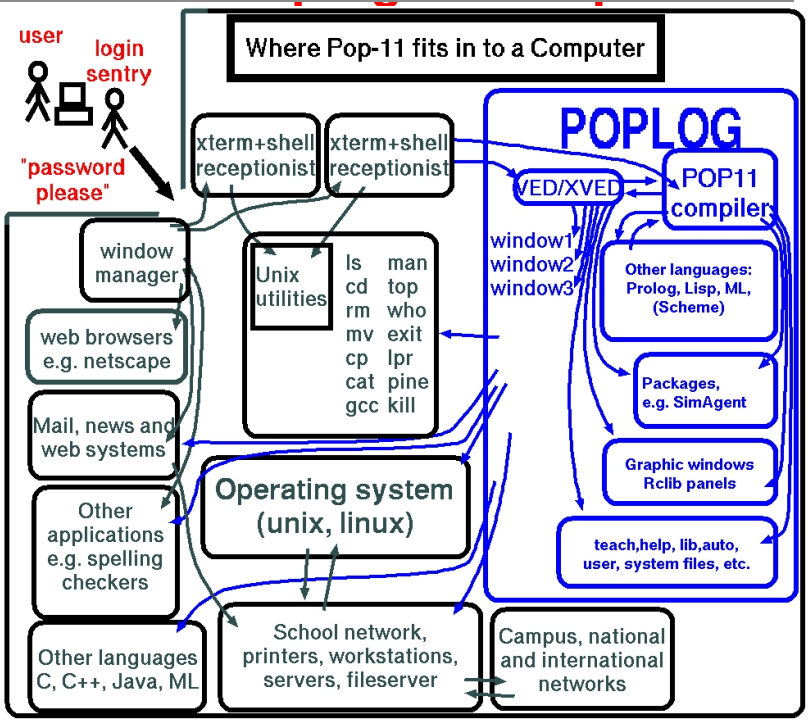
"Poplog" is a trade mark of the University of Sussex where it was originally developed.

A (partial) picture of Poplog and Pop-11

This is meant to give a feel for the fact that Poplog provides a sort of sub-operating system within an operating sytem (e.g. unix or linux) and provides access to many operating system facilities, including sockets for communicating with remote machines.

The interactive editor Ved (or XVed, the multi-window version) allows editor text to be submitted, as programming commands, to the compiler and compiler output to be spliced into an editor buffer.

It provides a powerful tool for software development, but users can instead use Emacs or any other editor, with slightly more hassle than using Ved.



FUTURE WORK

- Adding more libraries, including libraries supporting particular kinds of architectures
- Extending the “harness”, e.g. with tools to make it easier to assemble and run scenarios (including architecture-specific graphical tools).
- Making it easier for agents to inspect and modify their own architectures (e.g. to model various kinds of cognitive development or self-awareness).
- Adding a more “neural like” database mechanism, with “sloppy” matching and spreading activation (as in ACT-R)

Suggestions from users have led to many improvements and extensions, e.g. including support for self-monitoring.

It is expected that the process of designing extensions guided by user requirements will continue.

Some extensions may be built deep into the system, while others will be optional libraries.

LIBRARIES

It is intended that, with collaborators, we’ll develop a set of libraries for different sorts of classes of agents and environments.

A library can define

- Environmental object classes and mixins
- Agent classes and mixins
- Re-usable sensor methods and action methods
- Re-usable rulesets, and behaviours
- Graphical appearances
- Low level utilities

And can use Poplog’s documentation mechanisms including hypertext links.

Distributed agents

The basic system supports multiple agents in one process on a single CPU. However, some users have used Pop-11 facilities such as the socket library to implement distributed systems, with agents or parts of agents communicating via a local network.

- Work packaging this for less expert users is being done in an EPSRC-funded project by Brian Logan & Mike Lees (Nottingham University) and Georgios Theodoropoulos (University of Birmingham)
- Doing this requires addressing a number of complex trade-offs between convenience, generality, and efficiency.
- For large simulations with many agents all able to perceive only entities in a relatively small neighbourhood quite large gains may be achieved by separating the simulation into regions, though boundary effects can be very messy.

Matthias Scheutz and colleagues have developed a system called SWAGES which can be used to generate, distribute, integrate and monitor a collection of processes running on the same or different CPUs. In particular, some of the processes can be SimAgent processes. His tools include specific support for Java and SimAgent (or more generally Poplog) processes. See his web site at the University of Notre Dame

http://www.nd.edu/~mscheutz/

http://www.nd.edu/%7Eairolab/

Challenges for theorists

- It seems likely that the sort of complexity outlined above will be required even in some safety critical systems. Can we possibly hope to understand such complex systems well enough to trust them?
- Will we ever be able to automate the checking of important features of such designs?
- The design of systems of such complexity poses a formidable challenge. Can it be automated to any useful extent?
- Do we yet have good languages for expressing the REQUIREMENTS for such systems (e.g. what does “coherent integration” mean? What does “adaptive learning” mean in connection with a multi-functional system?)
- Do we have languages adequate for describing DESIGNS for such systems at a high enough level of abstraction for us to be able understand them (as opposed to millions of lines of low level detail)?
- Will we ever understand the workings of systems of such complexity?
- How should we teach our students to think about such things?

FURTHER INFORMATION

For more on SimAgent and its sub-systems see

http://www.cs.bham.ac.uk/research/poplog/packages/simagent.html
Overview

http://www.cs.bham.ac.uk/research/poplog/sim/help/sim_agent
Main integrating library

http://www.cs.bham.ac.uk/research/poplog/prb/help/rulesystems
How to express agent internals

http://www.cs.bham.ac.uk/research/poplog/prb/help/poprulebase
More details

http://www.cs.bham.ac.uk/research/poplog/rclib/help/rclib
The graphical tools.

http://www.cs.bham.ac.uk/research/poplog/figs/simagent
See some movies showing SimAgent at work

Poplog and the toolkit libraries can be fetched from the Free Poplog web site

(with full sources):

http://www.cs.bham.ac.uk/research/poplog/freepoplog.html

Mirror site with some extras: http://www.poplog.org

There is a version of Poplog for Windows, but at present (March 2007) it includes no graphical facilities.

The non-graphical parts of the toolkit can be run in windows Poplog (e.g. Poprulebase).

There is a sourceforge project to port Poplog including graphics to Windows:
http://www.cs.bham.ac.uk/research/poplog/openpoplog.html

Acknowledgements

- I am particularly grateful to the developers of linux and all its add-ons, including the X window system, for making a superb operating system widely available free of charge.
- Poplog is result of many years of work between about 1976 and 1998 by the development team at Sussex University, building on the Pop2 language first developed at Edinburgh university, around 1970. Key contributors to Pop-11 and Poplog were John Gibson (the chief architect), Steve Hardy, Chris Mellish, Jonathan Cunningham, John Williams, Robert Duncan, Simon Nichols, Jonathan Laventhol, Mark Rubinstein, Ben Rubinstein, Jonathan Meyer, Roger Evans, Tom Khabaza, and possibly others I have forgotten. Additional contributions came from staff at ISL before they were bought by SPSS in 1998.
- Steve Leach (while at HP Research Labs) designed and implemented Objectclass. He also made a number of suggestions adopted by the Poplog developers.
- Many of the ideas and some of the code in the SimAgent toolkit came from colleagues, students and collaborators who were using it, including (in approximate chronological order): Luc Beaudoin, Tim Read, Ian Wright, Riccardo Poli, Jeremy Baxter, Richard Hepplewhite, Darryl Davis, Brian Logan, Catriona Kennedy, Matthias Scheutz, Nick Hawes and others.

Some work done by Matthias Scheutz was funded by the Leverhulme Trust. See his web site http://www.nd.edu/~mscheutz/