

# BAD, a declarative logic-based language for brain modeling

Alan H. Bond

<sup>1</sup> Semel Institute for Neuroscience, Geffen Medical School,  
University of California at Los Angeles, Los Angeles, California 90095 and

<sup>2</sup> National Institute of Standards and Technology,  
MS 8263, Gaithersburg, Maryland 20899  
[alan.bond@exso.com](mailto:alan.bond@exso.com), <http://www.exso.com>

**Abstract.** We describe a declarative language, called BAD (brain architecture description language), which we have developed for describing and then running brain models. Models are at the system-level of description, so that modules correspond to brain areas. Each module has a process and the set of modules runs in parallel and communicates via channels corresponding to observed brain connectivity. Processes are described using a parallel set of left-to-right first-order logical rules in clause form, but with additional activity in a rule body described by Prolog code. Data items are represented by logical literals. Both data and rules use certainty values. The overall system described by the user consists of more than one agent each controlled by a brain model, and behaving in a 3D virtual environment, which is described by logical literals. Interaction with this environment is described by Prolog code representing sensors and actuators. Brain models have been developed for social interaction, problem-solving, episodic memory, routine memory and spatial working memory.

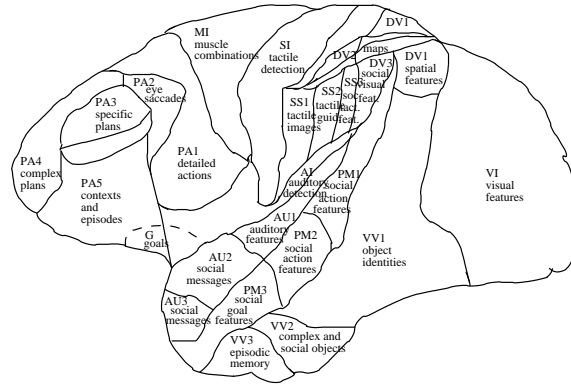
## 1 Introduction

We have developed BAD, Brain Architecture Description language, as a formal description language for specifying the anatomical structure and the information processing functionality of brains at the architectural level. With BAD, the user can specify a particular set of processing modules corresponding to brain areas. For each module a parallel set of rules is given describing processing within that module, and the set of modules is organized as an architecture by defining communication channels among them.

## 2 Motivation from neuroscience

We analyzed the neuroanatomy of the cortex, which is divided into neural areas with genetically determined connections among areas [5]. We reviewed experiments which indicated the functions that each neural area appeared to be involved in, and we construed these results as each neural area's action being to

*construct* data items of data types characteristic of that area. We also defined neural *regions* made up of small numbers of contiguous neural areas, as there are about 50 cortical areas whose functions are sometimes not clearly differentiated. The end result of our analysis is summarized in Figures 1 and 2.

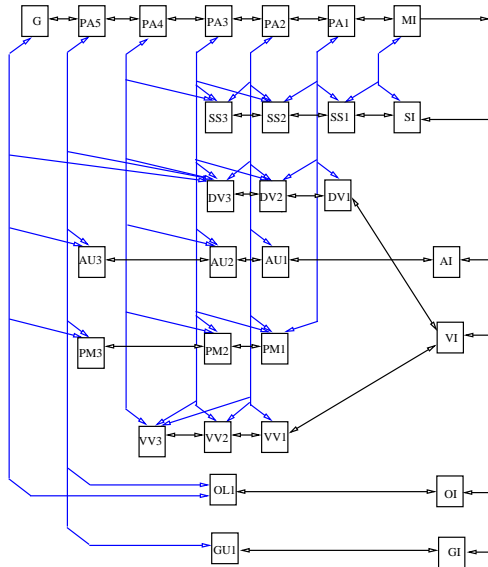


**Fig. 1.** Lateral view of the cortex showing neural regions and functional involvements

In order to design a model of the cortex, we abstracted from our review some biological information-processing principles:

1. Each neural area stores and processes data items of given types characteristic of that neural area; data items are of bounded size.
2. To form systems, neural areas are connected in a fixed network with dedicated point-to-point channels.
3. The set of neural areas is organized as a perception-action hierarchy.
4. Neural areas process data received and/or stored locally by them. There is no central manager or controller.
5. All neural areas have a common execution process, the “uniform process”, which constructs data items.
6. All neural areas do similar amounts of processing and run at about the same speed.
7. There is data parallelism in communication, storage and processing. Processing within a neural area is highly parallel. Parallel coded data is transmitted, stored, and triggers processing. Processing acts on parallel data to produce parallel data.
8. The data items being transmitted, stored and processed can involve a lot of information; they can be complex.
9. The set of neural areas acts continuously and in parallel.

From these and other considerations which we will explain, we designed and implemented a simple abstract model of the cortex [1] [4]. Figure 3 diagrams our initial model, and Figure 4 shows an example scenario which determines the



**Fig. 2.** Connectivity of regions showing perception-action hierarchy, note that the hierarchy is on its side with the top to the left

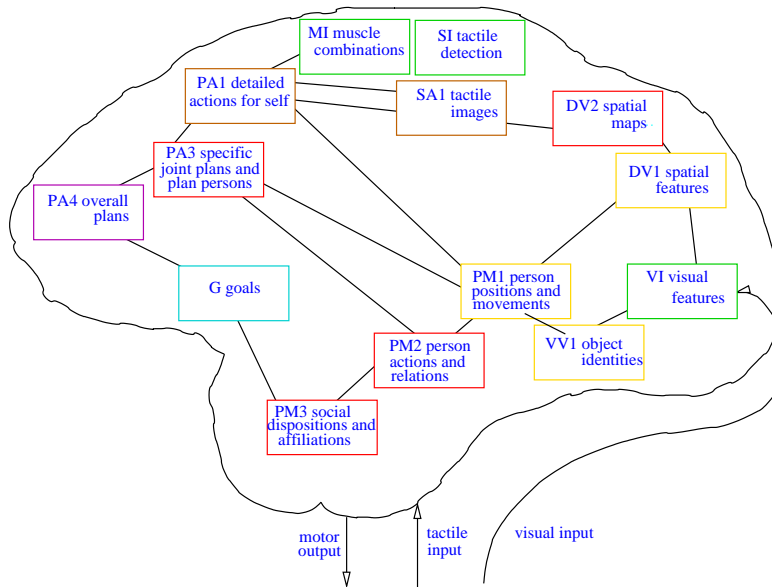
external input and output descriptions used by the model. This example has two modeled brains, one for each primate.

In the last few years, we have conducted a series of studies and models concerning the human brain, including problem solving [3], episodic memory [7], natural language processing [6], routinization [10], spatial working memory [9], and social relationships [2]. We used Sicstus Prolog as a basic implementation language and added another layer which we called BAD, and which represented the parallel execution of modules and communication channels between modules.

### 3 Overview of our modeling approach

A system-level brain model is a set of parallel modules with fixed interconnectivity similar to the cortex, and where each module corresponds to a brain area and processes only certain kinds of data specific to that module.

We represented each neural region by a module. A module has an associative store which is a Prolog store, and a set of left-to-right rules. The store can only store data of given data types, and these are made to correspond to the biological properties of the corresponding brain areas. In general the store is unbounded in size. Modules are connected by channels which again correspond to the connections observed for the brain. Figure 5 shows the components and data flow within a module.



**Fig. 3.** Outline diagram of an initial model

The model is executed in discrete temporal steps, which are fairly fine-grained. The intention is that a time step of one unit corresponds to 20 milliseconds of real time. This is the observed time taken for signals to be passed from one neural area to another in the primate cortex. During one time step, all the rules in all the modules are executed until data stability, without communication among modules, or with the environment. Thus the model is truly parallel and distributed.

Our approach is intended to be complementary to neural network approaches, and it should be possible, for a given abstract model, to construct corresponding neural network models.

## 4 Data types and storage within modules

**Data items.** Data items are represented by ground literals which we call *descriptions*. To indicate what data types can be stored in a given module, for each data type we give a *description pattern*, which defines the set of descriptions consisting of all of its instances. A description pattern is a literal of the form: `predicate(Weights,Context,[List_of_values])` (using identifiers with initial capitals to indicate variables). The first argument of each description is a list of weights  $[Ws_i]$  which can be processed by the user as they wish. For the models we have developed, all our description patterns are of the form:

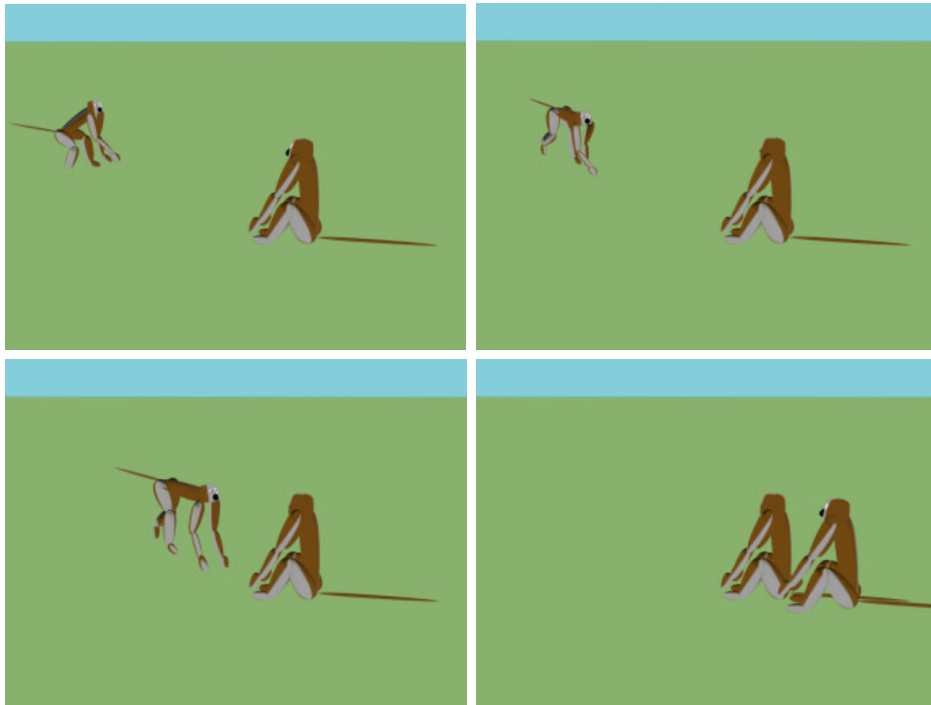


Fig. 4. Grooming sequence

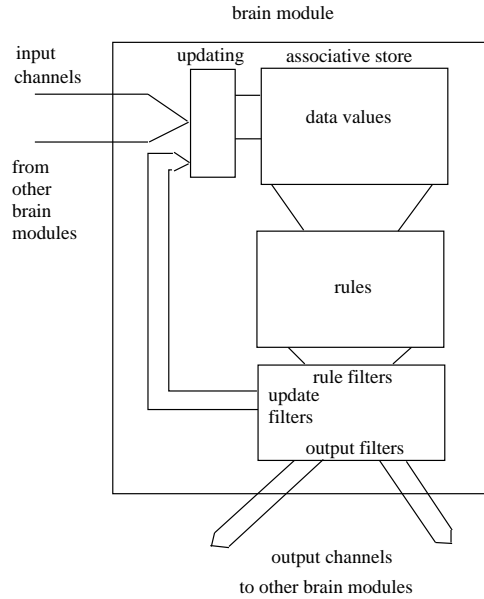
```
predicate([C,W1,W2,Time_stamp,Rstate],Context,[List_of_values])
```

The `Context` term is intended to be used for grouping descriptions into sets, i.e., all those descriptions with the same value of `Context` form a set. `C` is a certainty factor and `W1` and `W2` are weights; all of these are reals. In `List_of_values`, all values are ground terms, and preferably atomic. An example of a description pattern is `position(W,C,[Agent,X,Y,Z])`, which might be a data type whose instances represent the position `X,Y,Z` of a perceived `Agent`, with certain weights. An example of a description, i.e. data item, is a substitution instance of this data type:

```
position([1.0,1.0,1.0,1,[],],any,[adam,300.0,200.0,0.000])
```

At any one time, the module store has a set of stored descriptions of given types. A specification of a data type also includes its updating properties, its competition characteristics and its rates of update and attenuation, which will be explained below.

**Storage of data, uniqueness, similarity and novelty.** When we implemented and ran our model we soon realized that the store was updated from inputs every cycle and these updates were often very similar to those of the previous cycles. In order to update the store, we had to test if the input literal was the same as an existing stored literal. In fact there were four cases: (1) input



**Fig. 5.** The structure of a module, showing data flow and data storage

literal identical to a stored one, including all the weights, (2) the input literal is the “same” as a stored one, i.e. only differing in the values of the weights, (3) “similar”, which is given by equivalence expressions so that a data item may overwrite a “similar” one already in the store, for example if the action is walking and then the next input has it standing, we update the stored one, even though it is not the “same” as defined above, (4) the input literal is “novel” meaning it is not identical, same or similar, in which case it is simply stored. We also have sometimes used “corresponding”, based on X,Y,Z nearness.

In addition, we found we had to specify update characteristics for each data type; we used memory item update characteristic patterns, which we call *item types*. Items matching these patterns replace existing items, thus, for:

`item_type(Person,Module,Type,[Input_pattern,Stored_pattern]).`

an incoming description is matched to the `Input_pattern`, if it matches, then the `Stored_pattern` is matched to the store. If this matches then the incoming description updates the matching stored description. If no `Input_pattern` matches, or if the `Stored_pattern` does not match to the store then the description is simply stored, as it is a novel item.

So, e.g., `item_type(Person,Module,data,[goal(G),goal(G)]).`

any incoming goal description updates only an identical one. Thus there can be an indefinite number of goal descriptions, with different goals G.

And, e.g., `item_type(Person,Module,data,[action(M1,A1),action(M1,A2)]).`

an incoming action description for a given person M1 updates any other action

description for the same person. Thus, only one action description can exist for a given person, however there can be an indefinite number of different action descriptions each for a different person.

The updating of weights of memory items from incoming weights of matching item uses a multiplicative linear increment to the excitation of the item:

```
update_weights_increment(Person,Module,Incw1,Incw2): -
(((W1_input >= W1_old),W1_new is (W1_old*(1.0 + Incw1))) |
((W1_input < W1_old),W1_new is (W1_old*(1.0 - Incw2))))).
```

**Attenuation.** All stored data items are subject to an exponential attenuation process each cycle. Attenuation reduces the weight by a standard fraction each cycle. This fraction can be set to zero so there is no attenuation, and it can also be set to one, which erases the data item after one cycle (not before it contributes to rule matching in the cycle). Thus, attenuation factors are specified:

```
attenuation_factor_input(Person,Module,Type,AF,Description pattern).
AF is a list of real factors, one for each weight in the list of weights. e.g. for data
to attenuate to noise level in about 20 cycles
```

```
attenuation_factors(Person,Module,data,[0.1,0.1,0.1],C)
```

Noise level can be set and is the value at which a description is removed from the store: `noise_level(Person,Module,Type,Noise_level)`.

As a result of continuous linear updating and exponential attenuation, with a constant input stream of data items, the stored value will settle to a steady value. Initially, there will be a build up of the weight taking a few cycles, starting from noise level, and when the input ceases or changes to some other data items, the weight will take a few cycles to attenuate to noise level, where it will then be deleted.

## 5 Processing within modules by rules

**The form of rules.** Computation in modules is represented by a set of *description transformation rules* of the form:

```
rule(agent,rule_name,context,
  if((sequence of description patterns)),
  then ([list of description patterns]),
  provided((clause_body)),
  weights(weights)).
```

It is executed by matching the left hand side to bind variables and then constructing and possibly storing the descriptions given by the right hand side.

For example:

```
rule(M,macroaction_2,context(all),
  if((position(W,C,[M,X,Y,Z]),
      position(W_p,C,[MP,XP,YP,ZP]))),
  then([near(W_1,any,[M,MP])],Wa),
  provided((MP \== M,
```

```

distance(X,Y,Z,XP,YP,ZP,D),D<25.0),
weights(1.0,[1.0,1.0],[1.0])
).

```

The meaning of this rule is that if the primate perceives that another is near to it, then it notes this fact.

We prefer a style in which there is no branching in a rule, just conditionals, so for branching we use several different rules.

The set of rules in a module is executed in parallel, and the set of modules is executed in parallel.

The execution cycle first inputs all incoming data from other modules, then executes all rules and then outputs any output data to other modules.

In general, a rule is executed by matching all the left hand side description patterns to the store, executing all possible rule instances. The right hand side is executed, for a particular instance, by constructing from each description pattern on the right hand side a description obtained by substituting terms for variables. These constructed descriptions may then be stored in the module or output to other modules.

**Processing of weights.** Rules also have lists of weights, `weights(Wo,[Wli],[Wrj])`, one for each of the left hand side patterns `[Wli]`, one for each of the right hand side patterns `[Wrj]` and also an overall rule weight, `Wo`, for the rule as a whole. These weights are usually fixed at the moment, but also can be variables and can be computed within the body of the rule.

We usually use just one weight, `W1`, in calculations, at the moment. When the rule is evoked, the weight for the rule instance is the bilinear combination of these weights:  $W_a = W_o * \sum_i (W_{l_i} * W_{s_i})$ . `Wsi` are of course the weights of the stored data items which matched to the left hand side patterns. It is `Wa` that is used in rule competition, the strongest rule activation usually being taken in preference.

The computed weight `Wa` of the rule instance is multiplied by the `Wrj` to give the weights for the right hand descriptions. These weights are used in update and output competition.

A brief discussion of the several uses of weights and uncertainty in this system can be found in [8].

**Computation within rules.** In addition, we can have computations in a rule. This is specified by the Landinesque *provided* part of the rule. Its argument is an arbitrary Prolog code “block”, which can refer to any of the variables in the if and then parts of the rule. Computations are usually simple and used as filtering tests. However, for some purposes, it may be necessary to define complex predicates in Prolog and to use them in rules. The basic idea is that computations should be conceivably done by a neural area, and of course they can only use information present in the module. Ideally, all variable values should be obtained from the left hand side pattern match, but sometimes this ideal may need to be violated and a match made during the course of computation, for example, “not-exists” tests. For example:

```

(position(W,C,[Agent,X,Y,Z]),Z > 100.0,! ,fail) | true), which, if the

```

variable `Agent` does not occur in the left-hand side of the rule, means there does not exist any agent with  $Z > 100.0$ . If `Agent` does occur in the left-hand side of the rule then this is equivalent to having a pattern `position(W,C,[Agent,X,Y,Z])` in the left hand side and a test  $Z > 100.0$  in the body of the rule.

**Competition.** All descriptions and all rule activations have a weight which is a number representing their strength. The simplest form of competition is just to compare the weights of all the rule instances and to select just the strongest one. The products of this one rule are then stored and/or transmitted.

However, we have found it useful, and necessary, to develop other forms of competition, as follows. First rule instances compete and then literals compete. This is necessary or else it would be possible for one literal from one rule and one from another to succeed, but other literals from the same rules to fail.

Rule instance competition is specified by the user in `list_of_rule_number_sets`: `list_of_rule_number_sets(M,Mod,[list of lists of rule numbers])`. Results from all the rule instances from rule numbers in each set compete against each other using the  $W_a$  values. For example, `list_of_rule_number_sets(M,Mod,[[R1,R2],[R3,R5]]))`. which defines a list of rule number sets e.g. `[R1,R2],[R3,R5]`. Then all the results from rules `R1` and `R2` have their  $W_a$  values compared and only the one with the largest  $W_a$  is used. The same for `R3` and `R5`. Results from any other rules all go through and are used.

After rule competition, all the surviving updates and outputs from the rule instances are made into two overall lists. These lists, of updates and outputs respectively, are then, if desired, subjected to individual competition among descriptions in the list. The user specifies *update\_patterns* and *output\_patterns*. For each pattern in a specified list of update patterns, all the generated descriptions which match to that pattern are compared and only the one with the largest  $W_1$  weight is allowed to actually update the store. In addition, the winning  $W_1$  weight must be greater than a specified threshold. Any generated updates which do not match any update pattern are allowed to update the store. The analogous method is used for outputs also.

In general, for rule competition, we are thinking of changing to using rule priority ordering, that is, each rule having a place in a, possibly partial, ordering which determines its dominance in competition, independently of computed strengths. Such an ordering could be dynamic. This would give a more stable method, with better expressive discrimination, than using rule instance weights.

**Execution of rules until data coherence.** The set of rules in a given module is executed repeatedly until data coherence, that is, until no more changes to the updates and outputs occur. The purpose of this is to ensure logical coherence of the store and coherence of the outputs to be sent to other modules. During these iterations, updates are performed but not outputs. Iteration continues until there are no more novel literals created. It does not continue until the weights have settled.

This process is very similar to finding the fixed point of the logic program defined by the set of rules, as in the treatment of logic programming semantics

by VanEmden and Kowalski [11]. If there are no function letters involved, this process will be finite. Also, simple rule sets are similar to Datalog. At the moment, we are still using some recursion in rules and the computations within rule bodies are more general than allowed by Datalog.

## 6 Communication among modules and updating within a module

Communication is specified by the user using *output\_data\_item* and *update\_data\_item* declarations. Connections will be given, for each module, by a set of statements each of which gives a description pattern and the name of a destination module.

```
output_data_item(Person,Mod_name,Descr_pattern, Target_module_name),  
e.g.,  
output_data_item(M,person_motion,action(Name,Action),person_action).
```

What a connection statement means is that all output descriptions matching the given description pattern will be transmitted to the given destination module. They will actually be located in the channel until stored by the module at the beginning of the next cycle.

Updates into the module's own store are specified and treated similarly.

```
update_data_item(Agent_name,Mod_name,Literal,Mod_name).  
e.g., update_data_item(M,plan,working_goal(W,C,[WG])).
```

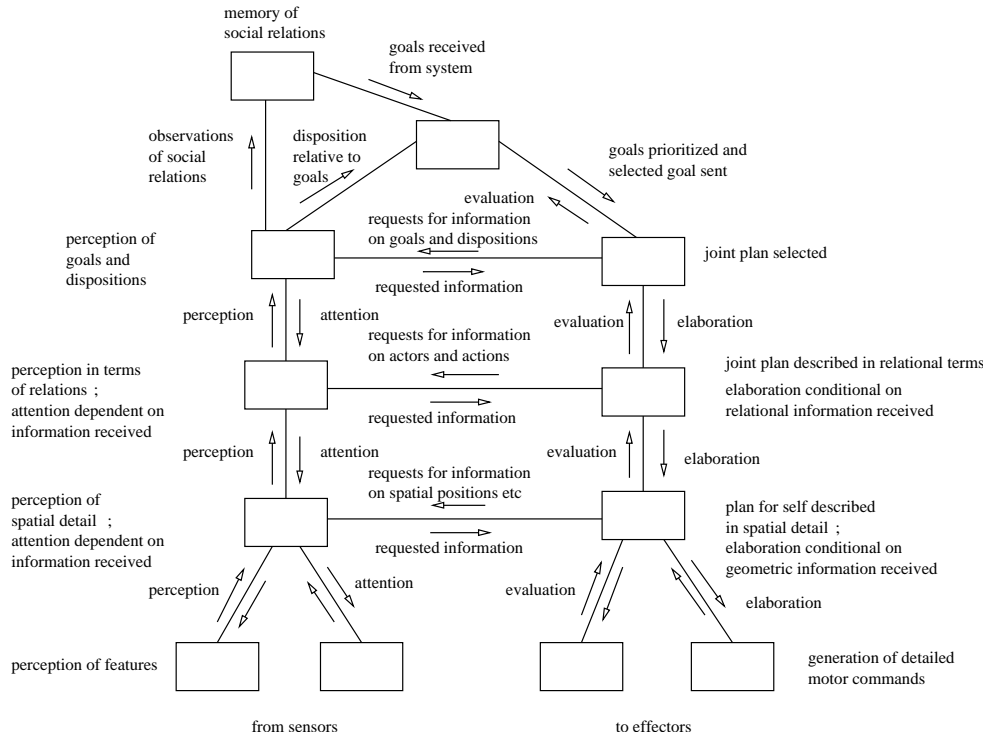
## 7 The environment

Brain models operate in a 3D spatial environment which is defined by a set of logical literals which describe all the objects and agents and their spatial properties. A brain has a set of sensors for perceiving the environment and these create data items which are input to certain modules. A brain has a set of effectors, and certain modules send motor commands, represented as ground literals, to the effectors, which change the environment.

A complete model consists of a set of agents each controlled by a brain model. Agents interact via the environment, so their positions and movements are perceived, and gestures can also be explicitly used. Language communication uses a physical channel corresponding to acoustics. The environment resolves any conflicts among the set of motor commands it receives from the set of agents. The environment includes the agent's body, including muscles, blood, glands and so on.

## 8 Agent architecture

As we indicated above, we have in our own work usually arranged the modules in a perception-action hierarchy. A schematic diagram of our concept of perception-action hierarchy is given in Figure 6.



**Fig. 6.** The mechanisms of a perception-action hierarchy

Modules are arranged in a hierarchy of abstraction. The system elaborates a selected goal into a plan and then into a succession of more detailed plans until finally a concrete action for the next cycle is sent to effectors which act on the environment. We made the effector actually output a low-level motor goal which was renewed every cycle. We made the top level of plan description correspond to social plans, which specified, in each plan step, not only the action of the agent but also the expected perceived action(s) of the other interacting agent(s). In executing a social plan, the perceived behavior of the other interacting agents was first checked to ensure it was in agreement with the currently executed social plan, and also to extract matched variable values for use in constructing the agent's own actions. An example of a social plan is approaching and shaking the hand of another person.

At each level the currently selected plan element is computed, so all levels continuously compute in parallel every cycle. The interaction of the perception and action hierarchies results in conditional elaboration and attention mechanisms. By conditional elaboration, we mean that the current percept can modify the elaborated plan step, so for example the agent can track and act upon a changing environment. By attention, we mean that information from plan elabo-

ration is passed to the perception hierarchy and can modify the use of perceptual resources to process in more detail the objects or other agents that the agent is interacting with.

## 9 Confirmation and viable dynamic states of agents

In order to select rules which were successful in producing useful behavior, we developed a mechanism which we call *confirmation*. If a rule fires in one “source” module and sends a description to another “target” module, then if this causes some rule in the target module to fire then a confirmation message is sent back to the source module. This message is specific to the exact description originally sent. For this, there is no need for global evaluation, and the test is purely local to the target module. The rule instance in the source module then has its value boosted by the confirmatory message. This will tend to keep it in control and keep things stable, avoiding “jitter”. If on the other hand no rule in the target module is caused to fire, then the rule instance is disconfirmed which results in it being placed in a “refractory” state for a certain period of time. This allows other competing rule instances to be tried. The general idea is that the system will then try all the different rule instances until it finds one that makes something happen, so this is a little like backtracking but with a simpler mechanism. This is also our attempt to provide a simple form of logical completeness, i.e., that all rules will be tried at all levels, so a solution will be found if one exists.

The way this is actually implemented is to not send any message in the disconfirmation case, so the source module simply waits a certain number of cycles before timing out and disconfirming the currently active rule instance.

Thus, we keep time stamps stored in each literal which give its waiting state and its refractory state, and the attenuation mechanism updates these each cycle. Thus data items are actually time-dependent literals.

Confirm factors specify the impact of confirmation values on weights:  
`confirm_factors(Person,Module,Type,[CFNEG,CFPOS],`  
`[Confirm_threshold,CSNEG,CSPOS])`

where: (i) `Confirm_threshold` determines where the computed confirmation value is a positive or negative confirmation (ii) `CFNEG` is subtracted from `W1` for negative, i.e. dis-, confirmation and (iii) `CFPOS` is used to multiply the confirmation value that is added to `W1` for positive confirmation. Typical values are e.g. `confirm_factors(M,goal,data,[0.4,0.05],[0.2,1.0,1.0])`.

**Viable states.** A system will tend to transition into what we call a *viable state*, in which the perceived environment tends to support the selected plan and the plan is selected by and supports the currently selected goal. Thus, in a viable state, the agent carries out a planned action which is relevant to its current goals, and which can be successfully carried out in the current external environment. The idea of a viable state is depicted in Figure 7. In each module, there is a dominant rule, depicted as a solid line, which fires and wins the competition with other rule instances. The other rule instances, depicted as dashed lines, continue

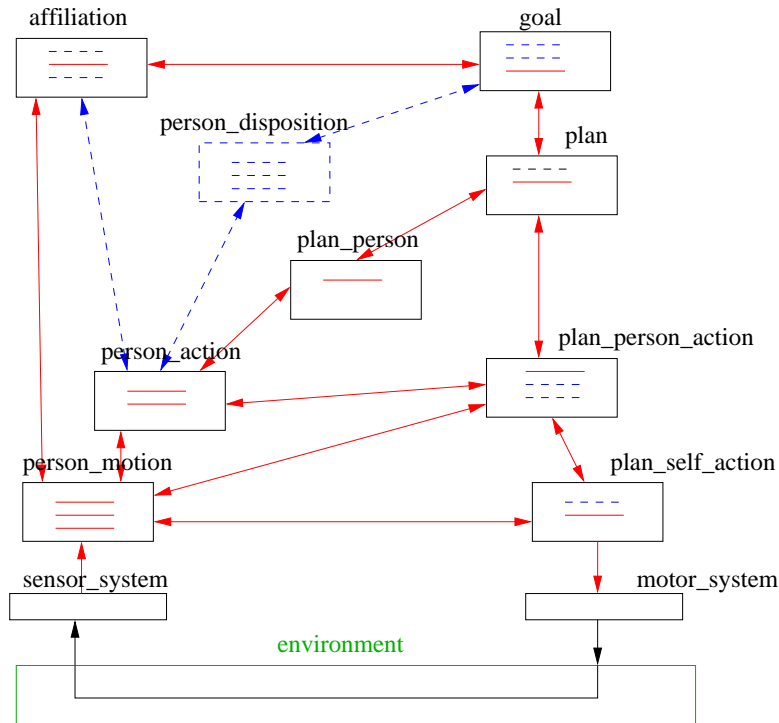


Fig. 7. The idea of a viable state

to be computed each cycle. This latter is necessary to provide for wellformed changes in the dynamic state.

Once in a viable state the system will tend to stay in it for several cycles, a few tens of cycles, up to thousands of cycles, before it has to select a new plan step at some level and to transition into another viable state. During a transition period, modules try different rules until they find one that is confirmed, and then until all the modules are selecting rules which are confirmed. It usually takes about 10 to 15 cycles to establish the next viable state, which corresponds to the experimentally observed time of 300 milliseconds to establish a conscious state.

Multiple interacting agents will also find mutually viable states, where the perceived behaviors of the other agents are compatible with the social plans being executed by each agent.

## 10 Using BAD

There is a BAD manual and also a document giving a complete BAD example of a working BAD program for a simple agent moving in a 3D world. A BAD system

is actually set up as a set of directories, namely, (i) a directory for each agent type, (ii) world, which specifies the environment and visualization to be used, and (iii) exp, which has details of the experiment to be run. These are accessed by directory declarations in Sicstus Prolog. An agent directory has a set of module specifications written in BAD, and in addition files giving the connectivity of modules. Sensors and effectors are specified in BAD and a Prolog file provided for each. An arrangement we have found most convenient and efficient is to have one agent per machine on a TCP/IP network and the world on another machine. The distribution of agents over machines is specified in Prolog. Socket communication between agents and the world is provided by BAD. The world receives sensing literals and motor commands from each agent in turn and replies to them in turn. There is at the moment no other support for managing the running of programs, so the user usually sits at the world machine and opens a window on each remote machine. Prolog sessions are started and the socket linkages started up, then calling a `run(M)` predicate in each session will run each agent and the world for `M` cycles. One can also use a file with a sequence of Prolog predicate calls to represent an experimental protocol. Consulting this file runs the experiment, it also helps clear garbage by taking the Prolog system back to the prompt each time. Typical cpu times for one cycle with an agent with 20 modules with 20 rules in each module are about 100 milliseconds on a 1 GHz Linux machine. The system provides support for VRML 2 output for the world and for agents' system states, as well as the terminal traces of the Prolog sessions. Geoffrey Irving, a Caltech undergraduate, managed to use BAD to program a predator-prey system with multiple predators and multiple prey, inspired by the arctic wolf and musk ox relationship. His project report is available.

## 11 Summary and conclusion

We have developed BAD, a declarative language in which to specify and execute agents inspired by the brain. The design is based on logic programming; data are literals, processes are parallel sets of rules and unification is the basic operation. Agents are modular and distributed based on distribution of storage and processing according to data type. The execution regime uses a discrete time step during which all rules are executed and all intermodule communication takes place before moving to the next time step. The time step is intended to be small relative to the rate of change of the external environment. Real-valued weights on data and rules can be used for many different purposes. BAD provides for updating and attenuation properties of data, and for defining modules by giving their data types and rules. Different connectivities among modules can be specified. A common agent architecture uses perception-action architecture, which has real time control properties. Rule execution proceeds until data coherence. BAD also provides a confirmation mechanism by which distributed processes over the architecture may be stabilized.

**Acknowledgement.** This work was carried out and funded by the author at Expert Software Inc., Santa Monica, California.

## References

1. Alan H. Bond. Describing Behavioral States using a System Model of the Primate Brain. *American Journal of Primatology*, 49:315–388, 1999.
2. Alan H. Bond. Modeling social relationship: An agent architecture for voluntary mutual control. In Kerstin Dautenhahn, Alan H. Bond, Dolores Canamero, and Bruce Edmonds, editors, *Socially Intelligent Agents: Creating relationships with computers and robots*, pages 29–36. Kluwer Academic Publishers, Norwell, Massachusetts, 2002.
3. Alan H. Bond. Problem-solving behavior in a system model of the primate neocortex. *Neurocomputing*, 44-46C:735–742, 2002.
4. Alan H. Bond. A Computational Model for the Primate Neocortex based on its Functional Architecture. *Journal of Theoretical Biology*, 227:81–102, 2004.
5. Alan H. Bond. An Information-processing Analysis of the Functional Architecture of the Primate Neocortex. *Journal of Theoretical Biology*, 227:51–79, 2004.
6. Alan H. Bond. A psycholinguistically and neurolinguistically plausible system-level model of natural-language syntax processing. *Neurocomputing*, 65-66:833–841, 2005.
7. Alan H. Bond. Representing episodic memory in a system-level model of the brain. *Neurocomputing*, 65-66:261–273, 2005.
8. Alan H. Bond. A distributed modular logic programming model based on the cortex. In *Proceedings of the Workshop on Multivalued Programming Languages, Federated Logic Conference FloC-06, Seattle, August, 2006*.
9. Alan H. Bond. A System-level Brain Model of Spatial working Memory. In *Proceedings of the 28th Annual Conference of the Cognitive Science Society, Vancouver, August*, pages 1026–1031, 2006.
10. Alan H. Bond. Brain mechanisms for interleaving routine and creative action. *Neurocomputing*, 69:1348–1353, 2006.
11. Maarten H. VanEmden and Robert A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the Association for Computing Machinery*, 23:733–742, 1976.