The AEP Toolkit for Agent Design and Simulation

Joscha Bach, Ronnie Vuine

Institut für Informatik, Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
bach|vuine@informatik.hu-berlin.de

Abstract: The design of artificial agents that are meant to model behavioral, cognitive, economic or social structures asks for tools that aid in layout and implementation of agent architectures. To implement agents based on Dörner's Psi theory of emotion and cognition, our group has introduced a toolkit that assists in designing modular architectures, as well as representational structures, such as semantic networks, control scripts and connectionist structures by means of a graphical editor. At the same time, the framework supports the inclusion of functionality written in a native programming language. This paper gives an overview over the implementation of agents according to Dörner's theory, and while it also aims at giving an insight into the functioning of these agents (which we call "MicroPsi" agents), its main purpose is the explanation of the use of the toolkit.

1. Introduction

The modeling of cognition, emotion, sociality and behavior with multi-agent systems has become an important tool to test and develop hypotheses or to represent a slice of a given reality. Our group is especially concerned with the design of a model of human emotion in the context of an AI agent architecture. The resulting architecture is called MicroPsi and represents a 'broad and shallow approach' (as suggested for instance by Bates, Loyall and Reilly [5]). MicroPsi is based on theoretical work by the psychologist Dietrich Dörner [7,8,9]. Applications of MicroPsi include experiments on learning in complex environments, modeling of emotion in interaction with humans and simple cognitive modeling.

The concepts of MicroPsi have been described in earlier publications by the authors ([1,2,3]). MicroPsi is based on a network formalism ("node nets") that aids as a general method for specifying the architectural components, individual functionality of cognitive modules, representation of actuatoric and sensoric schemata and so on. To support the design of such agents, we have developed a toolkit, which we currently call the "AEP framework" (for *Artificial Emotion Project*), but which lends itself to the design of agents with different, not necessarily emotional architectures as well.

The AEP framework [4,14] consists of several components that support the design of individual agents (the node net editor and simulator), the multi-agent simulation environment (the world editor and simulator), the interaction between agents and human actors (the visualization tools) and the integration of the former parts (the AEP server and console). This paper briefly describes the node net formalism and then sets out to explain the implementation of several modules of MicroPsi agents. The paper concludes with a note on distributed simulation, a short outlook and examples of applications of the framework.

2. Node Nets

2.1. Representation with Nodes

In his work on modeling human emotional and cognitive behavior, Dörner suggests the use of a kind of neural network for the representation of control structures, declarative and protocol memory. [7] Dörner asks for a mechanism that can represent both symbolic knowledge and connectionist configurations with the same data structures. Such representations could for instance be implemented as influence or belief networks (see, for instance [13]) with some additional logic to make them executable and interpretable.

In MicroPsi agents, these representations are directional spreading activation networks called *node nets*. These networks consist of units with inputs ("slots"), outputs ("gates"), propagation and node functions. Units may become active, causing them to execute their node functions. Activation is stored in the gates, which also possess threshold and amplification values. The arcs or links between units connect gates with slots, and activation may spread between units.

The most common node type used to represent semantic relationships is called "concept nodes". These have nine different gates: general activation (*gen*), links for causal relations forwards (*por*) and backwards (*ret*), for *part-of* and *contains* relations (*sur, sub*), for membership (*cat*, *exp*), and for naming (*sym*, *ref*). Of these, *por, ret, sur* and *sub* are part of the original description of the Psi theory and derived from a theory of representation by Klix [11]; the others have been added to simplify the implementation and notation. Usually, concept nodes are symmetrically linked (i.e. for every *ret* link, there is a *por* link in the opposite direction, and so on.)

Concept nodes may be connected to special nodes, so-called "directional activators" which are connected to all individual gates of a certain type. Gates may only transmit an activation, if their corresponding activator is active, which allows for a *spreading activation* mechanism. A set of nodes that is connected to the same set of directional activators is called a *node space*.

By choosing appropriate weights and thresholds, links between nodes can express logical AND and OR terms. It can be demonstrated that this notation is suitable to express first order logic [9]. On the other hand, information retrieval with node spaces is very similar to using hierarchical Case Retrieval Networks with directional activation [12].

The current implementation of MicroPsi comes with a graphical front-end that allows to define, maintain and execute node nets. The resulting networks can be saved in XML format.

2.2 Definition of Node Nets

Node nets consist of sets of net entities U, which are called nodes and modules, and which are connected to each other by links V and to the agent environment by a vector of $DataSources$ and $DataTargets$.

$$NN = \{U, V, DataSources, DataTargets, \mathrm{f}_{\text{net}}\}$$

where f_{net} is a propagation function calculating the transition from one state of the node net to the next.

$$U = \{(id, type, I, O, \alpha, \mathrm{f}_{\text{act}}, \mathrm{f}_{\text{node}})\}$$

Generally speaking, an entity $u \in U$ consists of a vector I of *slots*, a vector O of *gates*, an activation α, an activation function $\mathrm{f}_{\text{act}} : I \to \alpha$ and a node function $\mathrm{f}_{\text{node}} : NN \to NN$ (that is to say, there are no real limits to what the node function can do to the node net). The id makes it possible to uniquely identify a net entity.

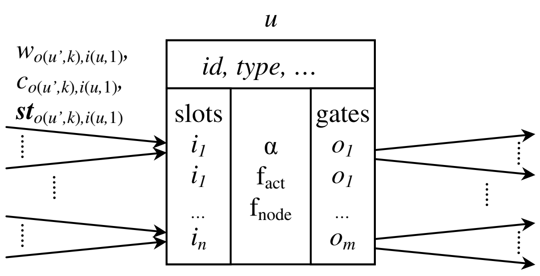

Fig 1: Node net entity.

Entities come about in different *types*, such as register nodes, concept nodes and so on, which will be explained below.

Nodes may be grouped into *node spaces*:

$$S = \{U^S, DataSources^S, DataTargets^S, \mathrm{f}^S_{\text{net}}\}$$

By mapping the $DataSources^S$ of a node space to slots, the $DataTargets^S$ to gates and the local net function $\mathrm{f}^S_{\text{net}}$ to a node function, it is possible to embed a node space into a single node entity, called a *node space module*. Thus, hierarchies of node spaces may be created.

Often, node spaces contain a number of nodes that have special properties, such as $Activators^S \subset U^S; Activators^S = \{u_{gateType_1}, \ldots, u_{gateType_n}\}$. Activators may influence the way activation spreads within a node space and are explained later on.

$$V = \{(o_i^{u_1}, i_j^{u_2}, w, c, st)\}$$

Note that nodes (u_1 and u_2) can be connected by more than one link. Links are defined by the gate $o_i^{u_1}$ and the slot $i_j^{u_2}$, which they connect, and are annotated by a weight $w \in \mathbb{R}_{[-1,1]}$, a certainty value $c \in \mathbb{R}_{[0,1]}$ and a vector $st \in \mathbb{R}^4; st = (x, y, z, t)$ containing spatial-temporal values.

$$O = \left\{ \left(gateType, out, \theta, amp, min, max, \mathrm{f}_{\mathrm{out}} \right) \right\}$$

Gates provide the output of net entities and consist of an output activation $out \in \mathbb{R}$, a threshold θ, an amplification factor amp, upper and lower boundaries on the activation min and max and an output activation function $\mathrm{f}_{\mathrm{out}} : \alpha \times O \times Activators \rightarrow out$ that calculates the values of the gates, usually by:

$$out = \begin{cases} \min\left(\max\left(amp \cdot \alpha \cdot act_{gateType_o}, min\right), max\right), \text{ if } \alpha \cdot act_{gateType_o} > \theta \\ 0, \text{ else} \end{cases}$$

where $act_{gateType_o}$ is the output activation out of the activator node $u_{gateType_o} \in Activators^S$ of the respective node space. (This calculation can be replaced by other functions, using for instance a sigmoid, which is useful for implementing a variety of neural network learning functions.) By triggering an activator, the spreading of activation from gates of the particular gate type is enabled.

Input to the nodes is provided using an array of slots:

$$I = \left\{ \left(slotType, in \right) \right\}$$

The value of each slot i_j^u is calculated using $\mathrm{f}_{\mathrm{net}}$, typically as the weighted sum of its inputs. Let $\left(v_1, ..., v_k \right)$ be the vector of links that connect i_j^u to other nodes, and $\left(out_1, ..., out_k \right)$ be the output activations of the respective connected gates:

$$in_{i_j^u} = \frac{1}{k} \sum_{n=1}^{k} w_{v_n} c_{v_n} out_n$$

2.3. Defining Specific Node Types

Concept Nodes, as mentioned before, are the typical building blocks of MicroPsi node nets. They consist of a single slot of the type *gen* (for "generic") and their node activation is identical with their input activation: $\alpha = in_{gen}$. Dörner's representations make use of the link types *por*, *ret*, *sub* and *sur*, which are represented by gates. Additionally, concept nodes have the gates *cat*, *exp* (for "category", denoting membership, and "exemplar", pointing to members) and *sym*, *ref* (for symbols and referenced concepts). Finally, concept nodes contain a gate *gen*, which makes the input activation available if it is above the threshold θ_{gen} – there is no *gen* activator.

Register nodes are the most basic node type. They consist of a single slot and gate, both of type *gen*, and like in concept nodes, their output activation amounts to $out_{gen} = [amp \cdot \alpha]_{min}^{max}$, if $\alpha > \theta, 0$ else; $\alpha = in_{gen}$.

Sensor nodes are similar to register nodes, however, their activation out_{gen} is computed from an external variable $dataSource \in DataSources^S$: $out_{gen} = [amp \cdot \alpha]_{min}^{max}$, if $\alpha > \theta, 0$ else; $\alpha = in_{gen} \cdot dataSource$.

Actor nodes are extensions to sensor nodes. Using their node function, they give their input activation in_{gen} to an external variable $dataTarget \in DataTargets^S$. The external value may be available to other node spaces, or, via the technical layer of the agent, to the agent environment (e.g. the world server). In return, an input value is read that typically represents failure (-1) or success (1) of the action returned as a sensor value to out_{gen}.

Concept, register, sensor and actor nodes are the 'bread and butter' of node net representations. To *control* node nets, a number of specific register nodes have been introduced on top of that:

Activators are special registers that exist in correspondence to the gate types (*por, ret, sub, sur, cat, exp, sym* and *ref*) of concept nodes of a node space. Their output is read by the output activation function of the respective gate of their nodespace. By setting activators to zero, no activation can spread through the corresponding gates.

General activation nodes are special nodes with a single slot and gate of type *gen*, and when active, they increase the activation α of all nodes in the same node space.

General deactivation nodes are the counterpart of general activation nodes; they dampen the activation of all nodes within the same node space. They are mainly used to gradually reduce activity in a node space until only the most activated structures remain, or to end activity altogether.

Associator nodes are used to establish links between nodes in a node space. This happens by connecting all nodes with active gates, using a weight

$$w^t_{u_1^i u_2^j} = \sqrt{w^{t-1}_{u_1^i u_2^j}} + \alpha_{\text{associator}} \cdot associationFactor^S \cdot \alpha_{u_1} \cdot \alpha_{u_2}$$

where t is the current time step, and $associationFactor^S \in \mathbb{R}_{[0,1]}$ a node space specific constant.

Disassociator nodes are the counterpart of associator nodes; they decrease or remove links between currently active nodes in the same node space.

Additionally, there is functionality for adding and removing nodes, also encapsulated in node entities.

3. Native Modules for MicroPsi Agents

It is possible to write and execute complete programs with AEP node nets. In theory, they are sufficient to set up all behavior and control scripts of MicroPsi agents. However, the execution of scripts made up of nodes is slow, and they are hard to maintain, even using a graphical editor. This makes it desirable to add more nodes for specific tasks, and to encapsulate long scripts. This is where *native modules* come into play; they are entities with arbitrary numbers of slots and gates. In their node function f_{node}, they hide program code written in a native computer language.
In the current implementation, native modules contain Java code and can perform any kind of manipulation on the node net. By integrating Java IDE, graphical node net

editor and agent runtime environment, the extension of the agents becomes quite comfortable. For the basic functions of MicroPsi agents, a number of native modules have been added, of which three will be explained here:

3.1. Script Execution

Node scripts consist of chains of concept nodes that are connected by *por*/*ret* links. With *sub/sur* links, macros and hierarchies are defined. This may be read as: after 'step 1' follows 'step 2' (*por*), and 'step 1' consists of 'part 1', 'part 2', 'part 3' and so on. 'Part 1' can again be the beginning of a chain or network of *por*-linked nodes.

The linking of the 'parts' determines whether they are alternatives or conjunctions. The lowest level of these hierarchies is always formed by sensor and actor nodes.
Because of these structures, it is possible to execute hierarchical plans with thousand of basic actions at the lowest level and few abstract elements on the highest levels and thus reduce the computational complexity of plan construction. Such hierarchical scripts can be run using the native module "ScriptExecution". ScriptExecution has two slots – Abort and ScriptActivation – and seven gates: Current, ProgramRegister, Macro, Idle, Success, Failure and FailAbort.

Initially, the script is retrieved and linked to a register on its highest level of hierarchy. This register is connected to the ScriptExection module. The execution starts by connecting *Idle* (which is active by default and thus susceptible to the *associator*) to the first concept node of the script. ScriptExecution first deactivates and unlinks *Idle*, and the first element is linked to *Current*.

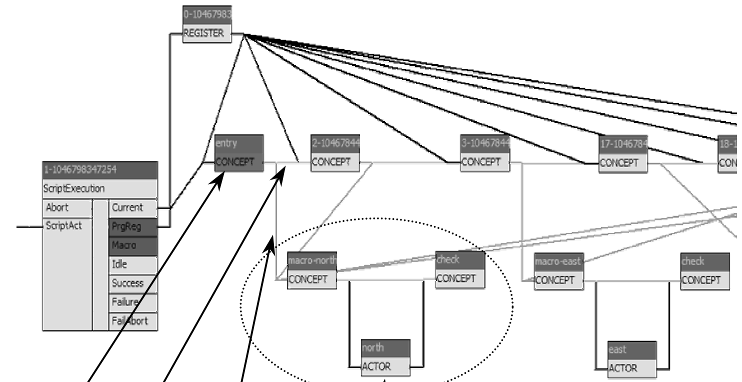

Fig 2: Script execution.

Now, in every step, the connected concept node is activated with the value of *ScriptActivation*. If this node is the parent of a macro, that is, if it has sub links, then the sub-linked concept node with the highest activation is chosen as new current node. (Using this pre-activation mechanism, scripts can be configured to follow certain paths before execution or even during execution.) If one of the sub-linked macros was successfully executed or there are no macros at all, the *por*-linked node with the highest activation becomes the new current node – if none of the *por*-linked nodes is active, ScriptExecution waits until one of the following happens:

- a *por*-linked node becomes active, causing it to become the new current node,
- a *por*-linked node becomes active with negative activation, causing failure,
- a timeout occurs, also causing the macro to fail.

When a macro just failed or was executed successfully, the entry point to this macro will again become the current node; ScriptExecution then decides how to go on in the manner given above. Macro success and failure are signaled at the *Success* and *Failure* gates. Fig. 2 shows a very simple script with just two levels of hierarchy.[1]

3.2. Emotional Regulation

This module calculates the emotional parameters from urges, relevant signals and values from the previous step. The module maintains the following internal states: *competence*, *arousal*, *certainty*, resolution level (*resLevel*) and selection threshold (*selThreshold*). These values are directly visible at the module's gates. Any subsystem of the agent that is subject to emotional regulation is linked to these gates, receiving the current emotional parameters via f_{net}.

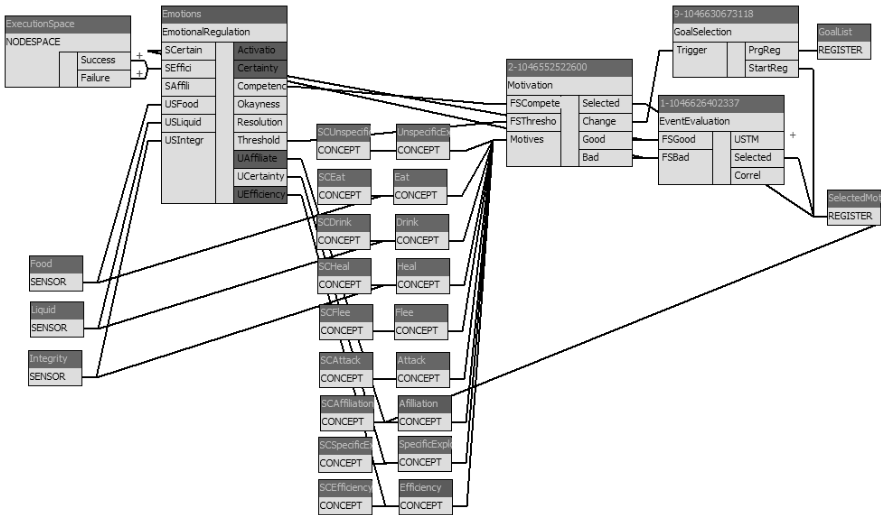

Fig 3: Emotional regulation and motivation (editor view).

[1] It is also possible to perform script execution solely by means of a spreading activation mechanism with appropriately adjusted link weights and thresholds. Here, individual nodes propagate their activation into the *sub*-direction, until actor nodes or sensor nodes are reached. In turn the *sub*-linked nodes propagate a small *sur*-activation to keep their parents active. If a sensor or actor succeeds, its activation is *sur*-propagated and excites their parents strong enough to allow for a overcoming the *por*-threshold. The *por*-gate in turn has to have an inhibitory link to stop the activation of its originator.

Additional gates signal the 'cognitive urges': *certaintyU* and *efficiencyU*, which are calculated every step simply as difference between a target value and the actual value. At the slots, the module receives the values of the 'physiological urges' ($extU_{1..3}$) and the amount of change that is to be made to certainty and competence, if some event occurs that influences the system's emotional state (slots *certaintyS* and *efficiencyS*). The way we use these values is very similar to Dörner's 'EmoRegul' mechanism [8,10].

At every time step *t* the module performs the following calculations:

$$competence_t = \max\left(\min\left(competence_{t-1} + in_t^{efficiencyS}, 0\right), l^{competence}\right)$$

$$certainty_t = \max\left(\min\left(certainty_{t-1} + in_t^{certaintyS}, 0\right), l^{certainty}\right)$$

($l^{competence}$ and $l^{certainty}$ are constants to keep the values in range)

$$efficiencyU_t = target^{competence} - competence_t$$

$$certaintyU_t = target^{certainty} - certainty_t$$

($target^{certainty}$ and $target^{certainty}$ are target values representing the optimum levels of competence and certainty for the agent.)

$$arousal_t = \max\left(certaintyU_t, efficiencyU_t, in_t^{extU}\right) - competence_t$$

$$resLevel_t = 1 - \sqrt{arousal_t}$$

$$selThreshold_t = selThreshold_{t-1} arousal_t$$

3.3. Perception

Perceptions of MicroPsi agents are organized as trees, where the root represents a situation, and the leaves are basic sensor nodes. A situation is typically represented by a chain of *por/ret* links that are annotated by spatial-temporal attributes. These attributes define how the focus of attention has to move from each element to sense the next; thus, the memory representation of an object acts as an instruction for the perception module on how to recognize this situation.

Situations may contain other situations or objects; these are connected with *sub/sur* links (that is, they are '*part of*' the parent situation). We refer to situations that consist of other situations as 'complex situations', in contrast to 'simple situations' that contain only single or chained sensor nodes *sur/sub*-linked with a single concept node.

The virtual environment of the agent contains objects representing plants and fruit. Currently, the agent is equipped with a set of elementary sensors on the level of objects (like sensors for bananas or hazel-trees). In Dörner's original design, elementary sensors are on the level of groups of pixels and colors; we have simplified this, but there is no real difference in the concept. Using more basic sensors just adds one or two levels of hierarchy in the tree of the object representation, but the algorithm for perception remains the same. All the agent learns about a virtual banana, for instance, stems from the interaction with this class of objects, i.e. after exploration, a banana is represented as a situation element that leads to a reduction in the feeding urge when used with the eat-operator, might be rendered inedible when subjected to the burn-operator, and which does not particularly respond to other operations (such as shaking, sifting, drinking and so on). The drawback of the current implementation that abstains from modeling visual properties is that it does not allow the agent to generalize about colors etc., and perhaps the situation representation will be extended for other simulation experiments.

3.3.1. Recognition of Simple Situations

Here, we look at the case of situations that have been seen before by the agent, so it already possesses a schema of the situation and uses the module "SimpleHyPercept" for recognition. (In the case of unknown situations, a different module, "Accommodation", is used to acquire a new schema.)

If the agent peeks into an external situation, elementary sensors may become active if a matching object appears. If, for instance, the agent stands in front of a banana object and happens to focus its sensors on it, the corresponding sensor node becomes active and the perception algorithm carries this activation to the concept node that is *sur*-connected with the sensor (i.e. the banana concept). If the object can only be recognized by checking several sensors, the agent retrieves all object representations containing the active sensor as part of a *por/ret* chain from memory. These chains represent for which sensors the agent has to check in which spatial relationship to the first one, to establish which of the object candidates can be assumed to lie in front of the agent (see fig. 4).

After an individual object has been found, it is checked whether it represents a part of a 'bigger picture', like a particular arrangement of bananas that has been seen in the past. This again can be determined by looking at the *sur* links of the banana concept node, which lead to known, potentially fitting situations containing bananas. The perception module builds a list of those nodes; these are the hypotheses that have to be checked.

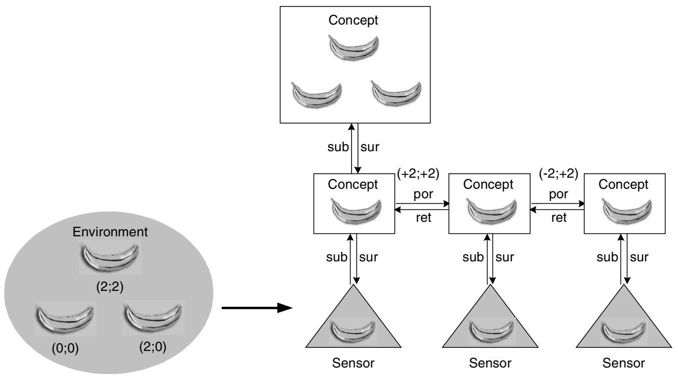

Fig. 4: A triangular arrangement of 'banana objects' is represented as a situation.

Some of the situations in the list might be biased to be checked first, because they are considered to be more likely true. This applies especially to situations that have been recently sensed (i.e. the agent will keep his hypothesis about the environment stable, if nothing happens to disprove it).

Given that list of hypotheses, the perception module now checks one after the other. To check a hypothesis, the *por/ret*-path of the hypothesis' elements is read from memory. The sensors of the agent are then moved to the element at the beginning of the *por/ret* chain, then along the *por* path to the next position, and so on until all elements have been "looked at". After checking each element, the sensor must verify its existence in order not to disprove the hypothesis. If all elements of the situation have been successfully checked, the hypothesis is considered to be consistent with the reality of the agent environment.

If one of the elements does *not* become active, the current hypothesis is deleted from the list, and the next one is checked. If a hypothesis is checked to the end of the *por/ret* chain, it is considered to "be the case" and linked as the new current situation.

As most modules in MicroPsi agents, perception can undergo emotional modulation. Especially the resolution level matters: if it is low, fewer elements of a hypothesis need to be checked for the hypothesis to be considered true. As a result, perception is faster but inaccurate when resolution is low, but slower and precise if resolution is high.

3.3.2. Occlusion

This algorithm obviously has a problem with occlusion (which happens frequently in the real world): If one of the elements of a situation is not visible due to occlusion, or because it is outside the field of view, it won't become active, the testing of the *por/ret* chain does not succeed and the hypothesis, although possibly correct, will be discarded. The most straightforward approach to deal with that problem would be to allow a number of elements not to become active before discarding the hypothesis. Additionally, it has to be maintained that the missing elements are indeed hidden by another object at the same position, or are invisible because of blurriness, distance etc. This comes at the cost of more erroneous recognition, but in the perception of complex situations (where the occlusion problem becomes relevant) it will be a necessity.

3.3.3. Recognition of Complex Situations

Just as simple objects consist of an arrangement of sensor patterns, complex objects and situations may contain other objects. This can be represented by *sub/sur* linking them. By choosing appropriate weights on these links, alternatives and conjunctions may be expressed. The hierarchical definition of objects makes it possible to represent a face, for instance, by two eye schemas, a nose schema and a mouth schema in the correct spatial arrangement. The eye schemas may in turn consist of lid schema, brow schema, iris schema etc.

When attempting to recognize a cartoon face (like a smiley), the agent may correctly recognize the spatial arrangement of a face, but because the eyes might lack detail, discount eyelids and so on as 'occluded' and still maintain the face hypothesis. However, the more complex mechanism of conditional HyPercept has not been implemented yet by our group.

In a similar way to "ScriptExecution", "EmotionalRegulation" and "SimpleHy-Percept", native modules for protocol generation, motivation, goal selection, event evaluation, simple planning, focus control, and so on have been defined. This toolbox already enables the agent to explore its environment, memorize its experiences, attempt try-and-error strategies to satisfy its urges, learn from the results and to follow little plans.

4. Summary and Outlook

A main focus of experiments with AEP based agents will be on multi agent interaction in a complex simulated environment (although our group has also implemented an AEP based control for four-legged Sony robots).

The current AEP architecture provides a simulation server that communicates with an arbitrary number of agents, the simulation world and – via console applications – with human experimenters. All components can be distributed over a network, allowing for simulations with computationally expensive agents. Communication between AEP components is facilitated via a general protocol that exchanges objects through a communication layer and includes mechanisms for remote maintenance.

The speed of the simulation is controlled with a timer component. Although the world simulation currently only takes place in two dimensions, which are represented in a 2D viewer, a three dimensional interface improves the interaction of humans with our virtual agents. The 3D viewer simply converts the 2D environment into a three dimensional representation that is read and displayed by a graphics engine. However, this engine provides only functionality for observing the agent world (i.e. interaction is not yet possible).

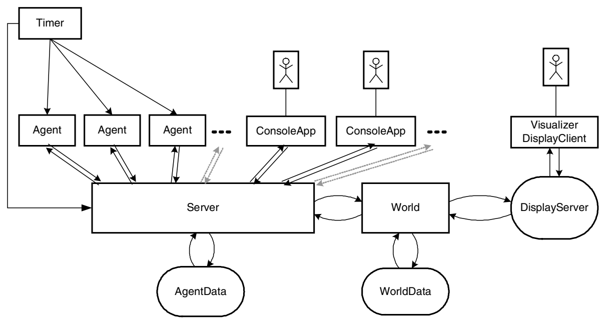

Fig. 5: MicroPsi agent and server.

MicroPsi is part of a larger effort of our group which is centered around the AEP framework. Currently, the toolkit is being used to implement classification algorithms, virtual Braitenberg vehicles [6], an artificial life simulation and to model human emotional expression.

References

1. Bach, J. (2002). *Enhancing Perception and Planning of Software Agents with Emotion and Acquired Hierarchical Categories.* In Proceedings of MASHO 02, German Conference on Artificial Intelligence KI2002: 3-12
2. Bach, J. (2003). *Emotionale Virtuelle Agenten auf der Basis der Dörnerschen Psi-Theorie.* In Burkhard, H.-D., Uthmann, T., Lindemann, G. (Eds.): ASIM 03, Workshop Modellierung und Simulation menschlichen Verhaltens, Berlin, Germany: 1-10
3. Bach, J. (2003). *The MicroPsi Agent Architecture.* Proceedings of ICCM-5, International Conference on Cognitive Modeling, Bamberg, Germany: 15-20
4. Bach, J. (2003). *Artificial Emotion Project/MicroPsi Home Page*: http://www.artificial-emotion.de
5. Bates, J., Loyall, A. B., & Reilly, W. S. (1991). *Broad agents*. AAAI spring symposium on integrated intelligent architectures. Stanford, CA: Sigart Bulletin, 2(4), Aug. 1991: 38-40
6. Braitenberg, V. (1984) *Vehicles*. Experiments in Synthetic Psychology. MIT Press.
7. Dörner, D. (1999). *Bauplan für eine Seele*. Reinbeck: Rowohlt
8. Dörner, D. (2003). *The Mathematics of Emotion.* Proceedings of ICCM-5, International Conference on Cognitive Modeling, Bamberg, Germany
9. Dörner, D., Bartl, C., Detje, F., Gerdes, J., Halcour, D., Schaub, H., & Starker, U. (2002). *Die Mechanik des Seelenwagens. Eine neuronale Theorie der Handlungsregulation*. Verlag Hans Huber, Bern
10. Dörner, D., Hamm, A., Hille, K. (1996): *EmoRegul – die Beschreibung eines Programms zur Simulation der Interaktion von Motivation, Emotion und Kognition bei der Handlungsregulation*, Memorandum des Lehrstuhls Theoretische Psychologie, Universität Bamberg
11. Klix, F. (1984). *Über Wissensrepräsentation im Gedächtnis.* In F. Klix (Ed.): Gedächtnis, Wissen, Wissensnutzung. Berlin: Deutscher Verlag der Wissenschaften.
12. Lenz, M., & Burkhard, H.-D. (1998). *Case retrieval nets: Basic ideas and extensions.* Technical report, Humboldt University, Berlin
13. Shachter, R. D. (1986). *Evaluating influence diagrams*. Operations Research, 34: 871-882
14. Vuine, R., & Bach, J. (2003). *The AEP Handbook.* http://www.artificial-emotion.de/pub/aephandbook.pdf (April 2003)