The Artificial Emotion Project Handbook
Rev. 0.5b

Ronnie Vuine Joscha Bach

October 29, 2003

Contents

1. Introduction

Welcome to the AEP project, welcome to the AEP Developer's Handbook.

As we're now done with the formalities, you'll be eager to get to know what this is all about: AEP, the Plugin and MicroPsi. Well, at first glance, this is about an interesting plugin for IBM's excellent Java IDE *Eclipse*.

At second glance, the AEP plugin is the entry point to a theory of mind, life, the universe and all the rest (as some of us, always modest, realistic and away from clichés, like to put it). Using the AEP framework and tools, you may be able to create intelligent software agents of unseen quality and beauty. With the AEP Plugin you can create and control the minds of those agents, watch them as they learn, interrupt and pause everything at any time, edit and watch the results. You will, most important, get to know the theory behind all this: You'll understand our approach to some of the oldest questions in artificial intelligence (and philosophy of mind, indeed), and, better yet, you'll eventually learn about pretty good solutions for those problems. Well, perhaps, at least. We hope to learn about them ourselves.

A word of warning: This handbook is meant to get you involved, to make you like what we are doing and to put as much passion into this work as we do. That does not mean that there is going to be too much hype here. But, as this handbook is directed at potential users of AEP/MicroPsi technology, the handbook is clearly not as modest and calm as it would have to be if it were directed to the scientific community. Shorter: You can't judge the project by this handbook. But you can become engaged by reading it.

What is the AEP ("artificial emotion project")? It is a project at Humboldt University in Berlin, dedicated to problems of artificial intelligence and cognitive science. We are mostly computer scientists, but the project is interdisciplinary in its nature. What are we doing? We, basically, create three kinds of stuff: *Theory*, *agents*, and *tools*. Theory, that is *MicroPsi*, a theory on cognition and the things that need to be done to create truly intelligent software agents. Agents, that is: Implementations of MicroPsi; and tools: That is the plugin and the framework that will help you to create agents (and simulated worlds) with MicroPsi/AEP technology, and that's how it all fits together. You, if a newbie, will learn to use the tools first, then get a notion of how our agents work, and as you understand the agents in more detail, you'll get to know the theory. Of course you can also do it the other way round, but then, unfortunately, you'll have to read this handbook back to front.

Anyway, what can you expect ? First, you will be shortly told how to install the system, what you need to have in order to do so and what change you can make to the configuration. There will also be some information on the configuration files in this second chapter, but in

order to understand fully what you are doing with these files, you'll need to know Chapter 5, the framework overview. Chapter 4 is the "Getting Started" part and mostly independent of the rest of this handbook. If you already have installed your Eclipse AEP plugin and want to get to know the system as fast as possible, read Chapter 4 now!
Chapter 5, as mentioned, explains the whole AEP system and architecture in a more detailed manner. There's much more to AEP than the plugin; the plugin is, in fact, only one of five components. Don't worry, the other components come and install with the plugin. Chapter 5 will also answer your questions regarding scalability and flexibility: Can you use AEP/MicroPsi technology for purpose X, could you build it into a robot and if so, what would have to be done?
Chapter 6 explains how to create artificial minds with the mind editor. That's cool, you might think, and it is of course, but there's no theory in chapter 6, it's merely a description of the user interface, what it is about and what you can do with it.
One of the most important things when writing a software agent with the AEP framework or when working with an existing MicroPsi agent is understanding native modules. In Chapter 7 you'll get all the information you need to find out what native modules are, why and when you need them and how to write and use your own ones.
Chapter 8 tells you about node nets, the stuff all agents in AEP are made of: What node nets are, what's in there, and how, precisely, these nets do work.
At the end of this handbook, you'll find a glossary with all the terminology you'll come across when reading the handbook. Use it a lot. Not only it will help you to understand AEP, it will improve communication with other AEP people a lot if you know the terminology and use it precisely. There are also appendices with web links, recommended reading and entry points for the JavaDoc pages of the AEP source base.

2. Installation and Configuration

2.1. What you need

Here's what you need for a basic setup, and where to get it:

- Eclipse. Eclipse is available, for free and for many platforms including Mac OS X, Linux and Windows from the website of the Eclipse project http://www.eclipse.org. This download is of about 60 megabytes in size (You need the SDK version). **Although the Eclipse Project is making good progress towards Eclipse 3.0, we generally don't recommend the 3.0 Milestone builds. The AEP toolkit is designed to work with the current release Version of Eclipse, 2.1. More recent milestones are likely to work, but our toolkit has not been tested against the 3.0 trunk, and all information in this handbook refers to Eclipse 2.1)**
- **The toolkit, which is downloaded automatically on installation. (see below)**
- **You may want to try our early implementation of a MicroPsi agent, available at our download page: http://www.artificialemotion.de/download**

2.2. How to install

This is how to install the toolkit, including the complete framework and all components:

1. Extract eclipse and put it somewhere where you like to have your programs. If you're on windows and wonder if there is no registry-fiddling to be done: No. No installation needed. Start Eclipse!
2. Go to Help → Software Updates → Update Manager.
3. At the bottom left, create a new Site bookmark by right-clicking and selecting New → Site bookmark
4. Give the site a name, e.g. "AEP Update Site". The URL is `http://www.informatik.hu-berlin.de/~bach/artificial-emotion/update` Leave the site type as it is, for the site is an "Eclipse Update Site".
5. Select the site, open it, open the "Toolkit" entry and double-click the "aeptoolkit" entry.
6. You now already see our logo - click "Install" and the wizard will guide you through the rest of the setup process.

Note that, as you have bookmarked our update site, you can check back there for new versions of the toolkit whenever you feel there should be such - or someone told you that that is the case.

You now have installed the toolkit. The four AEP perspectives ”Admin”, ”World”, ”Mind” and “NetDebug” can be opened in Eclipse’s perspective selection dialog (Window → Open perspective → Other). You should customize these to your needs now.

To install the agent, perform the following steps:

1. Open the Java perspective
2. Select New → Project → MicroPsi agent project, name it ”simpleagent”, Finish.
3. Right-click the project, select ”Import”
4. Import the ZIP-file: simpleagent.zip (get it from our website!).
5. Go to the Mind perspective, select the ”Load agent state” button.
6. Select state initial - done.

2.3. OS-specific issues

2.3.1. Windows

Currently none. We advise to use the Eclipse 2.1 stream.

2.3.2. Linux

Releases are not tested under Linux, however, many users report that it works fine. Reports also say that the Linux version’s speed is much dependent on which VM (IBM or Blackdown) you use, on the Widget system and that it also depends on – ehm, many other things. But if you’re using Linux for development, you like to play around with that stuff anyway, so just go ahead...

2.3.3. MacOS X

Although there is support for OSX, you will encounter some minor problems. By default, even the most recent Eclipse versions use Java 1.3.1 on the Mac. Our plugin won’t start without Java 1.4.1. So, besides that you need to have it installed, you’ll also need to tell Eclipse to use it. If you’re using the 3.0 Milestones (which we recommend on OSX because SWT support is much better in the 3.0 stream), you can change the VM version in Eclipse’s parameter array in the Info.plist file. (Inside Eclipse.app).

There’s more trouble: SWT and Java 1.4.1 have a problem with color and font dialogs. So

DONT open one, Eclipse will hang if you do. You can still switch back to 1.3 to change fonts and switch back afterwards.
The hanging color dialog is more annoying, though. Especially when you're using the ParameterView of the plugin on the Mac – remember **not** to click "Change color". We have added a workaround for changing the color: You can double-click the color field and enter the rgb value numerically. The Eclipse and Apple people are working on this, there's nothing to do but to pay attention and wait.

3. Advanced configuration

TBD.

3.1. Isn’t it a plugin?

No, it isn’t. The AEP system is used (and started) by one of our Eclipse plugins, which itself is only one of the *components* of the system. The fact that the AEP system runs inside Eclipse’s VM (and inside the scope of the plugin’s classloader) is just a consequence of the default all-in-one setup. In principle, you can run each component in it’s own VM on it’s own computer. Some restrictions apply to console components – these are only able to edit an agent’s mind if in the same VM with the agent component.

For a detailed description of what components exist and how many of them may be within one system see the ”AEP Framework” chapter. TBD

3.2. The ComponentRunner

TBD.

3.3. aepconfig.xml

TBD.

4. Getting started

In a later version of this document, you will find here a step-by-step introduction to editing node nets and getting a simple agent up and running.

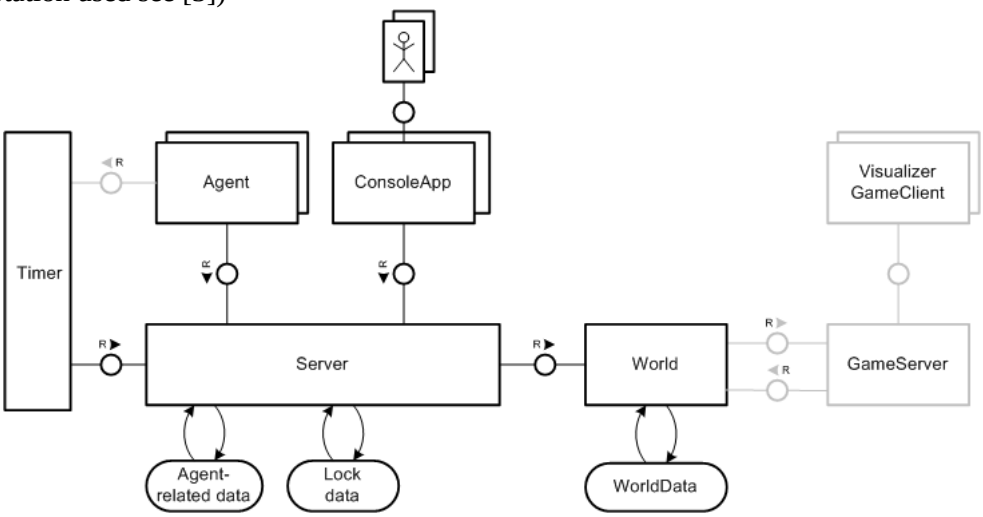
5. The AEP framework

5.1. Overview

What we call ”framework” is divided into two major parts: The **system framework** and the **agent framework**. The system framework is a flexible way of a putting one or more agents into a situation, of controlling and managing the agents and everything that is needed to keep ’em running, even distributed over more than one computer: You can run virtually every component separately, connecting the whole system over HTTP.

- One or more agent components.
- A world component. The world provides perception and executes the agent’s actions. The default implementation of the world component is a simulated a-life environment, but a replacement for this could be any interface to any reality.
- A timer component
- A component called ”server” for routing data between the other components
- Console components, for interacting with the system

The following diagram shows the system framework with all components. The components in light grey are facultative and not necessarily part of the framework. (For explanation of the notation used see [3])



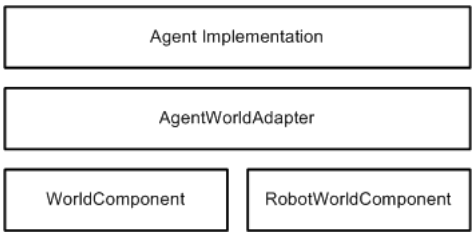
Any of these component can be replaced independently, if the replacements implements the protocols in a proper way - the AEP system protocol is not documented in this handbook, but we can provide that information if you're seriously planning something interesting.

The agent framework allows the embedding of various types of agents into the system framework, MicroPsi being one of them (and currently the only one). The agent framework defines how agents receive and send information and what immediate surroundings there are for agents, both technically and in terms of embodyment.

5.2. The Agent Adaptation API

We tried to make the agent framework as flexible as possible. Our goal is to enable you to use the AEP infrastructure for world/agent communications in virtually any context. Agents defined and implemented within the AEP framework are applicable in many environments. Depending on the type of environment, the agents must be modified and adapted in different ways.

To achieve that, we defined three "levels of flexibility":



- **Agent implementation** Trivially, agents can be adapted to worlds within their implementation. Our only implementation of the AEP AgentIF interface, the MicroPsi agent, can easily be replaced by some other implementation, if you want to use the AEP framework, but are not interested in MicroPsi.

- **AgentWorldAdapter** Any AEP agent implementation, including the MicroPsi agent, can be interfaced to any type of world by using AgentWorldAdapters. World adapters are Java classes that translate the world's "language" into that of the agent. In MicroPsi agents, the adapters provide the DataSources for sensors and DataTargets for the actor nodes - depending on the type of world the agent is supposed to live in.

- **WorldComponent / RobotWorldComponent** By default, our WorldComponent simulates a world for the AEP agents to live in. But it is also possible to replace the WorldComponent with a RobotWorldComponent, an interface that creates AEP perception messages from a "real" environment, such as the web, a text document, camera image, odometry, infrared sensor or what have you. It also can receive action messages from AEP agents and execute it by accessing real actuators.

Of course the distinction between simulated and real environments is somewhat artificial - how much more real is the web than a simulated island? ”Real” environments are simply referred here as ”real” because (and if) there are connections to something outside the AEP system.

Assume you want to use the AEP framework for deploying an agent into some environment - what kind of modification would you have to make to the system? – That, obviously depends on the environment and the type of agent you want to use. The following table defines possible scenarios.

	Our A-Life simulation	Other simulated environment	Real environment
Our MicroPsi agent	default, the MicroPsi development scenario	the MicroPsi flexibility test scenario	the MicroPsi application scenario
Custom (MicroPsi-type) node net agent	the alternative architecture test scenario	the node-net research/transfer scenario	the node-net application scenario
Custom AEP agent	the alternative agent test scenario	the framework-only research/transfer scenario	the framework-only application scenario

Table 5.1.: The AEP scenario matrix

5.2.1. The MicroPsi development scenario

Obviously, you don’t need to adapt or customize anything. That’s the configuration we use for developing our agent. That means that the scenario is - at least currently - very much Dörner-like and surely academic. We make - and change - assumptions on how our simple A-Life environment looks and works just as we think these assumptions are suitable for the development of a general cognitive architecture.

5.2.2. The MicroPsi flexibility test scenario

Changes would have to be made to the simulated environment, and a suiting WorldAdapter would have to be written. We consider it too early to transfer MicroPsi agents to other simulated environments at the current stage of development.

5.2.3. The MicroPsi application scenario

The RobotWorldWomponent would have to be used, along with a suiting, newly developed WorldAdapter. We consider it too early to transfer MicroPsi agents to real-world-scenarios at the current stage of development. Anyway, this is a hot topic for the future.

5.2.4. The alternative architecture test scenario

This would mean to develop an alternative node-net-agent, and that is easy and involves no *Agent Adaptation API* usage. We have no plans to do so ourselves, but surely there's people out there who would like to implement their own architectural ideas and compare their results to ours. This could also involve alternative Dörner-style agents that, possibly, stick more strictly to the theory from the *Mechanik* [5]

5.2.5. The node net research/transfer scenario

This is likely to be a very common one, as node nets are a very flexible formalism. In this scenario, changes to the virtual environment would have to be made, again along with a suiting WorldAdapter, and additionally a node net agent would have to be developed. There are people in our group that are doing exactly that, implementing Braitenberg-vehicles with node nets. [7]. There's also thought about using node nets in a more connectionist manner than in the standard MicroPsi agent and exploring the neural-net possibilities of node nets.

5.2.6. The node net application scenario

This is very similar to the previous one, but would involve the use of the RobotWorldComponent.

5.2.7. The framework-only scenarios

These are, to us, not the most interesting ones, at least at the moment. Although the framework is meant to be usable without all the MicroPsi/node net stuff, we ourselves have no interest in doing so. Still, if someone should be interested in our infrastructure simply for architectural reasons, maybe looking for a framework that allows a distributed multi-agent-a-life-simulation: You're welcome, we look forward to hear from you. (A word of warning: The distribution aspect of the whole thing is, although already implemented, not quite the focus of our work at the moment and hasn't been extensively tested.)

5.3. Changing the world: Perceiving and taking action

Changes to the wold influence what actions are available to the agent and what it perceives if it asks for perception. Depending on what kinds of objects are inside the world and what kinds of agents are supported, the world can be of very different complexity. (Ranging from worlds with only two lightbulbs and the action "setwheelspeed" to complex A-Life simulations with different kinds of agents with different sets of actions). In principle, there are two ways to modify the world: Modify the *contents* of the world or the *means of interaction and perception*.

5.3.1. World content

You can add simple "dumb" objects just by creating them in the UI's world editor. There's also a possibility to add more sophisticated objects that implement some behavior. But as there's still

a lot of ongoing development in that field, we decided to document the procedures for doing so in a later version of this document.

5.3.2. Interaction and perception

The agent's mind is calculated in a *agent component*. But, inside the world component, something must execute the actions that were create by the agent's mind and create the perception that the agent's mind requested. This is done by an "agent object" inside the world component. Note the difference to AgentWorldAdapters: The latter adapt agent mindss to existing agent objects. Agent objects themselves don't adapt or translate actions/percepts, they *define* them.

There is one default implementation of an agent object, called the "SteamVehicleAgentObject". It currently defines the *move*, *eat* and *drink* actions and object-based percepts.

When implementing a new agent object for an agent with different kinds of interaction and perception, you'd have to do the following:

- Extend AbstractAgentObject. (See Appendix A for complete classpath and apidoc link.)
- implement the method getPerception(). The object you have to return from this method must contain the percepts that the agent is to receive after requesting perception data. You can retrieve these percepts in some way or another from the world. You can access the world via the member field "world".
- implement the method handleAction(MAction action, WorldObjectIF targetObject). You should perform the requested action in the implementation, that is: *actually do something to the world*. After all the translating and routing, this is the place where actions are really *executed*. Implementations must return a response object containing the success of the action and, if appropriate, body parameter changes that are direct results of the action.

5.4. Writing a WorldAdapter

5.4.1. Why write a WorldAdapter?

Whenever you want to use an existing agent with a new world, or a new agent with an existing world, or both, you need to write a WorldAdapter. By writing a world adapter, you're providing the agent and the world with exactly the logic they need to interact, nothing more, nothing less, but especially all the technical details are wrapped away into the AEP framework.

5.4.2. Writing a WorldAdapter

How a world adapter looks largely depends on what the world and the agent look like. In any case, you need to do the following:

- Implement AgentWorldAdapterIF. — An instance of your implementation will be used to access the agent controller and the action/percept translators you implemented.

- Implement AgentControllerIF — Used for non-action-non-percept-specific communication with the agent.
- Implement any number of ActionTranslatorIFs — Every action translator describes what kind of message is to be sent to the world when the action is executed, and all translators together decide which action will be sent to the world.
- Implement any number of PerceptTranslatorIFs — Every percept translator describes how percept data from the world is to be fed into the agent.
- Implement any number of UrgeCreatorIFs — Urge creators can be used to create urges for the agent, especially if these depend on perception or body data. You must not use UrgeCreatorIFs for the agent's urges, but it is good practice to do so, because if urges are part of a world adapter that they depend on logically, they are added and removed with the world adapter.
- Add your world adapter to the "worldadapters" section of the agent's configuration. (In aepconfig.xml) — for more information on aepconfig.xml, see the chapter on advanced configuration.

5.4.3. Pitfalls: WorldAdapters for node net agents

WorldAdapters for node-net-based agents are, of course, of special interest. There are some pitfalls due to the fact that the internal timing of the node net and the timing of the aep system are not in sync. (Node nets can even be in suspend mode while the rest of the simulation goes on). Especially the action translators need to make sure that actions are only sent once when an actor node went on and that their data target (where the actor node writes its activation) is reset *after* the action was sent and *before* resetting would mean dropping activation for a possible next action.

To ensure that these ugly technical details don't bother you too much, the MicroPsi agent comes with a default ActionDataTarget implementation that you should use when implementing your translator. Have a look at the translators for our (default) island world for information on how to use the ActionDataTarget class.

5.5. Using the RobotWorldComponent

To set up a RobotWorldComponent, you need to do the following:

- Implement RobotActionExecutor. — Your implementation will receive actions from the agent that are to be executed in the real world. What actions you can receive here obviously depends on what WorldAdapter the agent uses.
- Implement RobotPerceptionExtractor. — Your implementation will have to provide perception data here. Again, what kind of perception the agent can use depends on the agent's WorldAdapter.

- In aepconfig.xml, change the ”class” entry of the world component to `de.artificialemotion.comp.robot.RobotWorldComponent`
- Change (or add) the ”executor” entry in the world section of aepconfig.xml (as a sibling of the ”class” entry). As value of the executor tag, enter the full qualifying classname of your RobotActionExecutor.
- Change (or add) the ”extractor” entry in the world section of aepconfig.xml (as a sibling of the ”class” entry). As value of the executor tag, enter the full qualifying classname of your RobotActionExtractor.

6. The User interface

6.1. Overview

This chapter covers the user interface of the Plugin: How it works, how to use it and what's behind all the widgets and buttons.

6.1.1. Some basic terminology

In order to understand the following, you need to know what perspectives, views and widgets are in Eclipse. The explanation is very brief, if you want or need more information: read the Eclipse documentation [1], it is, although somewhat sparse on other topics, very good in the basics. Here we go:

Widgets When you look at an open Eclipse window, everything you see is a widget. As that information doesn't help you a lot, here are some examples: Buttons, Lists, Trees, Text, Windows, basically everything you can distinguish. It's that easy.

Views When you look at Eclipse, you'll notice that the whole window is partitioned into smaller areas, each with a title bar, separately scrollable and resizable. These are called views. By the way: You can expand views to full-size by double-clicking the view's title bar.

Perspectives An assembly of views into something that makes sense in some way is called a "perspective". You can add views to perspectives or remove them if you want, but normally the people that made up the perspective knew what they were doing.

6.2. The Mind perspective

The Mind perspective contains six views and the editor area. There's the *MindEdit* view, the *EntityEdit* view, the *LinkageEdit* view, the *IncomingLinks* view, the *Library* and the task list. You might also want to add the *LogView* view by selecting Window → Show View → Other → AEP → LogView.

The MindEdit view is the "main" view of this perspective, as the whole perspective is about manipulating the relations and states of entities. When you select an entity in the MindEdit perspective (you click it or draw a frame and it becomes blue), the EntityEdit view will update it's content and show information on the selected entity. If you select a gate in the EntityEdit view, the LinkageEdit view will be updated and show information on the gate and the links attached to it. You can also select single links directly. By using the "copy" modifier while dragging selected entities around, you can create clones of that entities.

The Library can be used to store fragments of node nets. Just select a group of entities in the MindEdit view and Drag'n'Drop it to the Library by using the "Copy" modifier, on most

systems the Ctrl-Key. Having dropped a group of nodes in the Library, you will be prompted for an name for the new entry. After that, you can copy the nodes back to any nodespace at any time by simply dragging it there.

The task list can be used to memorize things. Okay, you never forget something, but: Don't remove it before you were told that eclipse puts compiler errors, warnings and code marked with a "todo" there. It's useful, really. The editor area contains the editors that appear when you double-click a native module in the MindEdit view.

If you added the LogView, you can watch log messages that come in from different loggers. Each AEP component has its own logger. You'll probably be particularly interested in the logs from the agent - not only because the framework outputs messages there; when you write your own native modules you can add code that outputs data to that logger - this is extremely helpful.

6.2.1. The MindEdit view

When you look at the MindEdit view, you'll see boxes and colored lines connecting them. The boxes are entities, the lines are links. You can drag around the boxes, select or deselect them or open a context menu. Right-clicking is, in general, a good idea in the MindEdit view. Entities have context menus, as well as the background has. (The "background" is also an entity, the root node space of the net, but don't care about that for now.)

To understand what the colors mean, you need to know this rules:

- Black links have positive weights.
- Blue links have negative weights.
- Green links are currently propagating activation. (Which means that they are attached to an active gate and will propagate the gate's activation to the linked node in the next cycle.) The greener the link, the higher the activation.
- Red links are currently propagating negative activation. (Either because of a negative gate activation or a negative weight at the link.)
- Green entities are active. The greener the node, the higher the activation.
- Red entities are active with negative activation.

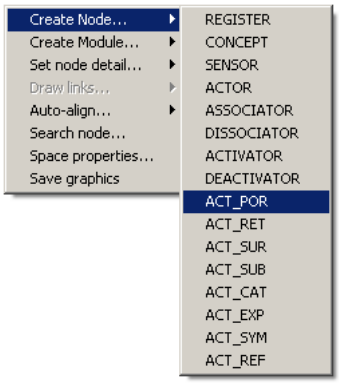
To understand where links originate and end on default node types, you need to know this rules:

- GEN links originate from the node's "title bar"
- Links ending in a node's title bar end in the GEN gate.
- Links originating or ending on the node's horizontal axis are POR/RET links.
- Links originating or ending on the node's vertical axis are SUB/SUR links.
- Links on the lower-left to upper-right diagonal axis are CAT/EXP links.

- Links on the lower-right to upper-left diagonal axis are SYM/REF links.

Don’t be scared, don’t learn this by heart, it’s really quite intuitive when you are familiar with what all this means and have used the editor for a while.

Background Pop-Ups



Create Node Creates a node in the current nodespace. Hovering this will open another popup with the types of nodes you can create.

Create Module Creates a new module. Hovering this will open another popup from which you can choose if you want to create a NodeSpace (this one will be created immediately) or a NativeModule (will open the wizard in which you choose the implementation for the new module).

Set node detail You can choose here between the default node display style and the old looks that used color-coded links.

Draw links Here you choose which links will be displayed. The default, ”all”, draws all links. This can be quite annoying and slow and messy if there are really many entities with lots of links, especially when running the net. If too many drawn links are disturbing you or slowing the agent down, try one of the other options: ”None” draws no links at all, ”Selected Only” draws only links of entities you selected, ”Dragged only” only draws links of entities that you drag around.

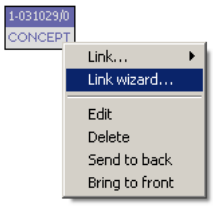
Auto-align Chose an align strategy to be applied to all entities in the current nodespace. This strategy will be applied immediately, and the affected entities will be positioned.

Search node Opens a dialog for searching entities by ID or by name.

Space properties Opens a dialog with the properties of the current node space.

Save graphics Lets you select a location for saving the current nodespace as a bitmap. The file will be quite large and is rendered in memory, so be a bit careful if you’re short on RAM or have a huge nodespace.

Entity Popups, standard nodes



Link Lets you first select the type of link you want to create (depending on the gates you have at the entity), then you can draw the link to a slot of the entity you want to link, if the targeted entity is in the same nodespace. If not, you'll have to use the Create-Link wizard.

Link wizard Opens the Create-Link wizard.

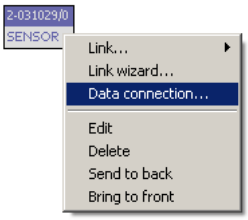
Edit With standard nodes, this does nothing.

Delete Deletes the entity.

Send to back Sends the entity to the back.

Bring to front Brings the entity to the front. z-axis information is not made persistent currently.

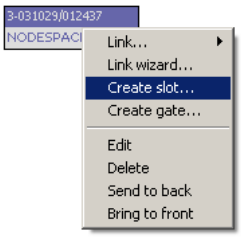
Entity Popups, sensors/actors



All except "Change data connection" See standard nodes section.

Change data connection This opens a wizard that lets you select and activate the data connection (source for sensors, target for actors) for the node.

Entity Popups, nodespaces

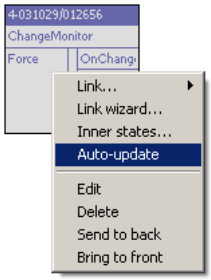


All except the "Create... " entries See standard nodes section.

Create Slot Opens a wizard that lets you create a slot for the nodespace. When creating a slot at a nodespace, a new sensor data source is generated. (So you can read the slot's value from inside the nodespace by a sensor).

Create Gate Opens a wizard that lets you create a gate for the nodespace. When creating a gate at a nodespace, a new actor data target is generated, so you can write to that gate from inside the nodespace).

Entity Popups, native modules



Default entries except "Edit" See standard nodes section.

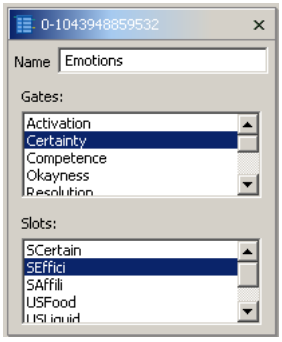
Inner states Opens a dialog that lets you edit the inner states of the module - if the module defines inner states. Not that these are untyped – you can enter strings where the module expects numbers, causing the module to reset the inner stat to zero without warning.

Edit Opens the java editor for the implementation. (same as "Edit")

Auto-update If you enable this, the implementation of the module will be replaced by the newest version every time a new version of the implementation class is compiled. The replacement will try to preserve linkage and inner states of the module (only if the "innerstate" object is used, of course.). Normally, a new class will be compiled by Eclipse every time you save the corresponding source file. However this is only the case if Eclipse's autobuild-preference is enabled.

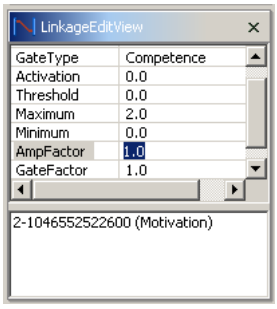
Note that there is an Eclipse bug that turns off this preference sometimes. So if you edit your source file, save, run the net and find that mysteriously nothing has changed – check if autobuild is still on.

6.2.2. The EntityEdit view



At the EntityEditView, you can change standard properties of entities. Currently, this means you can alter the name of the node. You can also select a gate or slot in order to access it's properties and links in the LinkageEditView/IncomingLinksView.

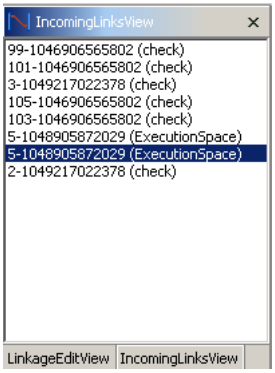
6.2.3. The LinkageEdit view



At the LinkageEditView, you can change gate parameters: Select the row of the parameter you want to alter, then click the value itself. If you enter unacceptable values, the displayed value will simply be the same as before when you leave the row.

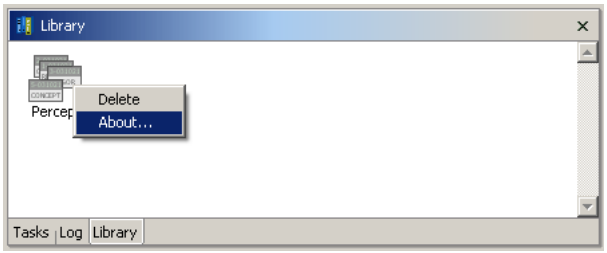
You can also select and right-click one of the links in the list, that gives you the option to delete the link, change the link's parameters, bring the linked node to front or look up the node, if it is in a different nodespace or otherwise different to locate.

6.2.4. The IncomingLinks view



The IncomingLinks view is very similar to the LinkageEditList, besides that, as the links listed there are incoming links and hence attached to slots, there are no parameters to edit here. (Slots don't have any parameters.) So what you see is simply a list of links attached to the selected slot.

6.2.5. The Library view



The Library contains node net fragments that can be dragged into the mind edit view. You can create new library entries by dragging groups of entities from the mind edit view to the library. (In both directions, you need to use the “copy” modifier, on most systems the Ctrl key). You can delete entries or add a description by right-clicking it.

Note that the the Library view can be useful not only for storing “building blocks”, but also as temporary storage when moving structures between nodespaces.

6.3. The Admin perspective

6.3.1. Overview

In brief, the admin perspective lets you take a look into the technical details of a running AEP system (system framework). You can here perform most of the actions that the UI performs in a more low-level manner. The most important view in the admin perspective is the RawCom view, where you can send what we call ”questions” and receive answers.

6.3.2. The RawCom view

The RawCom view allows you to interact directly with running AEP components: You can access all functionality that the components expose to the outside. This is done in a question/answer metaphor: You ask questions to the component, the component provides answers. When posing a question, you give details on how often and when you want your answer and what additional information you want to send along with your question. The question is then routed to the addressee, answered and the answer is returned to you (the user of the console component) and finally the answer(s) get(s) displayed.

Needless to say that you should be knowing what you are doing when posing questions. There is currently no detailed documentation of the questions exposed by the default components, and no plans were made to create such documentation. Again, if you really want to replace one of the components and expose functionality via questions yourself, contact us directly.

For the rest of you: You will hardly ever need the RawCom view, as there is good UI for most of the functionality. One question, though, has proven extremely useful: The `getrunnerlog` question displays the logfile with the stack traces of all exceptions.

6.3.3. The Log view

The log view contains the same information as the log file, but updates immediately when something is logged and displays it neatly sorted by loggers (There is one logger for each running component.)

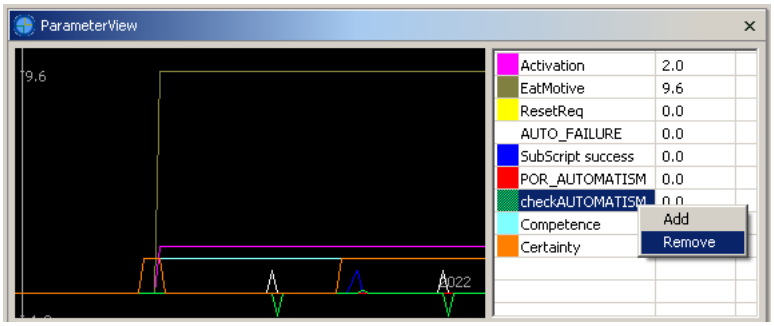
6.3.4. The LocalSystemView

The LocalSystem view is currently of no use at all. In the future, this is the place where you will be able to set up AEP systems in a point-and-click-manner, without having to edit xml files and the like. But there’s a long way to go.

6.4. The NetDebugPerspective

Debugging Node net agents isn't at all easy. While the implementation of native modules is normally somewhat straightforward and does not involve complex call structures, threading issues or any other of the hellish things that make debugging a tough task normally, the net itself is a beast in itself: Once there is activation on some entity that you simply don't know where it comes from and why it is there at that very moment you don't want it - you are in trouble. And this is a common situation.
There is hope, though. There are some debug mechanisms. As the node net run inside the same VM as eclipse itself, we clearly can't offer a real code debugger - but mostly, that's not what you would want anyway, messing around with net internals. Typically, debugging a MicroPsi-style agent poses two questions: *What does my native module do?* and *What about that activation at node XYZ?*

6.4.1. The Parameter view



At the parameter view, you can watch the output of any gate in the net as it develops through time. To add a "monitor" to some gate, simply right-click it at the EntityEdit view, choose a color and give it a name. The real output of that gate will now be shown in the parameter view. You can monitor as many gates as you like. – This has shown to be extremely helpful: At a glance, you see the values of those variables that matter.
You can use this view to track down "ghost" activation (activation that shouldn't show up where it does) by following the links backwards, add monitors to the linking gates and finally find out where the activation comes from.

The parameter view can do more than that. Currently, there are only monitors for gate activation, but we're thinking about providing monitors for *counting* links at some gate, monitoring inner states of native modules and monitors for various other parameters.

6.4.2. The Log view

As said in the admin perspective section, every component has it's own logger. As each agent is a component, it has a logger. And this logger can be accessed from inside native module implementations: logger.debug("Hello world");

Use this for your debug output in native modules. The log view shows that debug output. The color of the text indicates the log level. Black is ”debug”, green is ”info”, blue are warnings, and red tones are errors.

6.5. The World perspective

The world perspective is self-explanatory. They say. Still.

7. A closer look at native modules

7.1. Overview

Native modules are entities just like the other types, with one major difference: When calculating the values of their gates from those of the slots (when the net is calculating the state of its next step), native modules can set the gates to whatever values they want and even manipulate the rest of the net in every possible way. To put it different: Native modules can perform any action every step, any action implemented natively, that is: in Java.

7.2. Creating a native module

7.2.1. MicroPsi agent projects

Native modules are normal java classes. But to be able to create and use them, some conditions must be met: Some libraries must be included, and the native module must be located at a particular location within the package tree.
There is a wizard that will set up an Eclipse project for you so you don't have to do the setup yourself. Select New → (Other →) Project → Java → MicroPsi agent project. You then only need to give the name of your project, then press finish. The resulting project will contain the appropriate libraries (with source attachments, so you have JavaDoc and parameter names) and a package where you can create your native modules.

7.2.2. Creating the module

To create a native module, open the standard Eclipse Java Perspective, open your MicroPsi agent project and create a new class. You must place the new class somewhere below `de.artificialemotion.micropsi.modules` so the system finds it afterwards. As supertype, the class must have `de.artificialemotion.micropsi.net.AbstractNativeModuleImpl`. Having created the class, fill out the auto-generated method stubs until all compiler errors are gone. You have now created the implementation of the module, go back to mind perspective now and create a new native module (by right-clicking the background or directly by choosing the wizard). The wizard will ask you for an implementation for the new native module, and you can now select your class from the project you placed it in. After the wizard has done its work, your new native module is in the net. By double-clicking it, you can open the implementation code and modify it. If you enable "implementation auto-replacement" in the context menu of your native module, the module will be updated with your latest code every time you save the implementation's java file. By the way: If you ever encounter an ugly red cross on your native modules, that means

that your implementation had uncatched Exceptions. Stacktraces and error messages will be in the agent's logfile.

7.2.3. Possibilities you have in native modules

In a native modules' `calculate(...)` method, you can do whatever you want to. You probably will either want to calculate new gate values from the modules' slots or change something in the net's structure. Well, you can do both quite easily. For such tasks, the implementation superclass already has instances of classes that provide APIs for manipulation of activation, structure, persistent values and logging. Here's more details on these four objects.

Changing gate parameters: The GateManipulator

An instance of GateManipulator is passed as second argument to the calculate-Method every time it is called. With the gate manipulator, you can:

- Set the activation of one of the native module's gates for the next step
- Change any of the other gate values permanently: maximum, minimum, threshold, gate factor, amp factor.
- Create links from one of the native module's gates to a slot at any entity or unlink gates.

For a more detailed description of what the GateManipulator does, see the JavaDoc for GateManipulator: GateManipulator.html

Changing net structure: The StructureModificator

Then, there is the structure modificator object. It is a protected instance in AbstractNativeModuleImpl, so you can access it anywhere in your module. It is reachable via `this.structure`. Using the structure object, you can:

- Create or delete Concept nodes
- Get GateManipulators for other entities
- Create or drop links between entities
- Activate entities
- Get the sibling entities of the native module and the parent space
- Find an entity instance by the entity's ID

For a more detailed description of what the StructureModificator does, see the JavaDoc for StructureModificator at AbstractNativeModuleImpl.StructureModificator.html

Maintainig inner states: The InnerStateContainer

As the StructureModificator, the StructureModificator object is a protected instance in AbstractNativeModuleImpl, so it's accessible via `this.innerstate`. With the InnerStateContainer object, you can publish values from the implementation to the Eclipse UI, so users can see and change them as the net is used. Published values inside the InnerStateContainer are also persistent. This means that after saving and loading the net, the inner state of the module will be the same as before - if you put all your important variables in there. To do so, store the values at the end of the `calculate(...)`-Method in the innerstate object (via one of the `setState(...)` methods) and read them at the beginning of the method. (Via the `getState(...)` methods).

For a more detailed description of what the InnerStateContainer does, see the JavaDoc for InnerStateContainer at InnerStateContainer.html

Logging

And finally, there's a logger, accessible via `this.logger`. Log messages will appear in the agent's log file and on the LogView. You can use `logger.debug(Your message here);` at any time in your native module.

The logger object is an instance of an Apache Log4J logger. JavaDocs can be found at the Log4J website. [2]

Naming slots and gates

There's one more thing you will want to know: When you define your slots and gates in the `getSlotTypes()` and `getGateTypes()` methods that are called on module initialization, you do this by returning ints. This is fine for the net, but when looking at the module in the Mind view, it would be much nicer to have names at the slots and gates than presenting just numbers. There's a way to add translations to your slot and gate type numbers: The TypeStrings registry. By calling the static method `de.artificialemotion.micropsi.main.TypeStrings.activateExtension(...)` and passing an instance of TypeStringsExtensionIF (from the same package), you can give the UI the information it needs.

If you want to know more about the TypeStrings class and how to use it and how *not* to use it, look at the JavaDoc page for TypeStrings: TypeStrings.html and the TypeStringsExtensionIF.

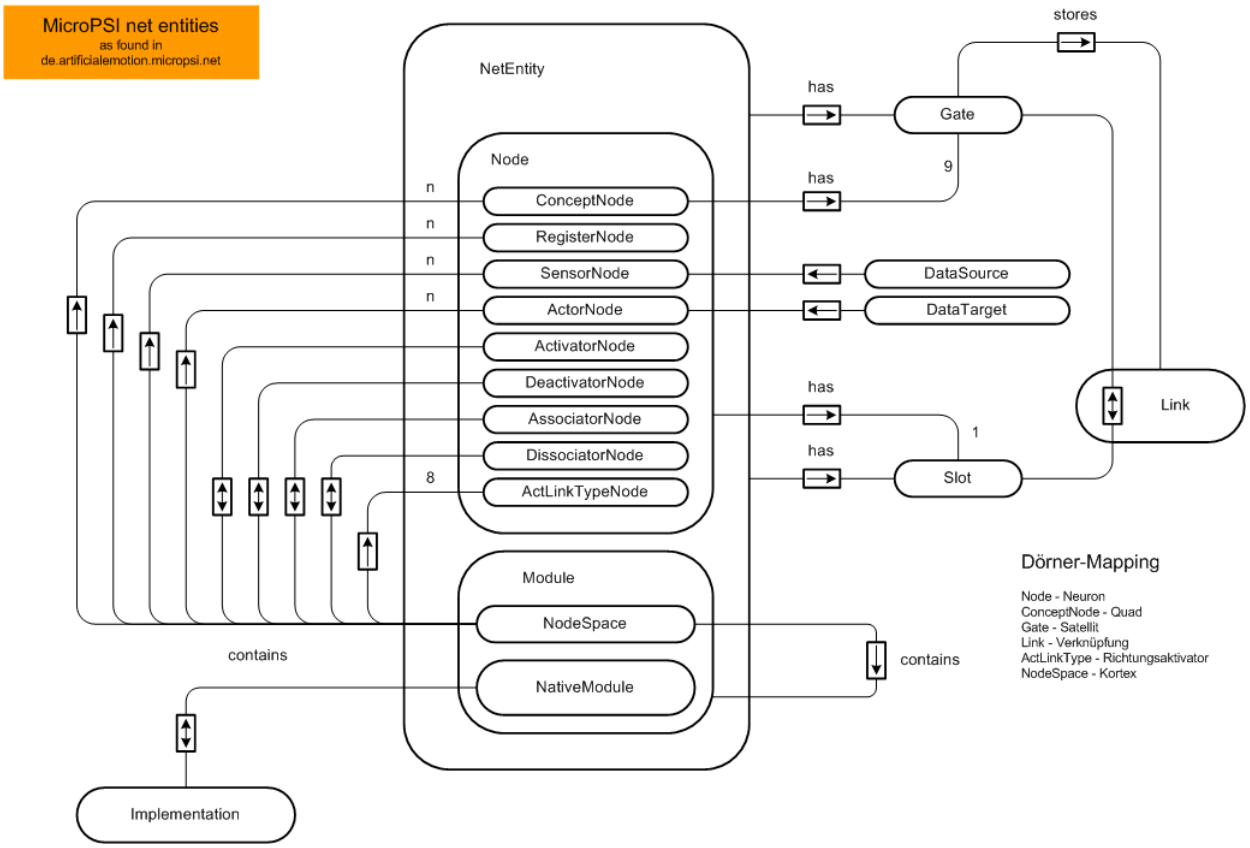
8. Node net theory

8.1. Overview

This chapter provides more information on node nets, the inner structure of MicroPsi agents, the theoretical stuff they are made of. There are detailed verbal descriptions of everything related to node nets in the following sections, and, at the end, there will be a more formal definition of all this.

8.2. What node nets are

As a starting point: Here's the entity relationship diagram of node nets:



8.2.1. NetEntities

As you read from the diagram, nearly everything in a node net is a *NetEntity*. NetEntities are connected by *Links*, which originate at *Gates* and end in *Slots* (Slots and Gates being part of the NetEntities). Each cycle, NetEntities calculate their gates from the values of their slots and then propagate the activation along the links into the slots of linked entities.

Links

Links can be established between two NetEntities, attached to their origin at a gate and at their end at a slot. Links have, in their standard form, two attributes: a *weight* and a *confidence* value. Both values are used multiplicative in the activation propagation process, but they are treated slightly different: Links with weight $w = 0$ will be removed by the net, while the value of the confidence attribute has no such effect.

We speak of NetEntities a and b as being *connected* iff there is a Link from one of the gates of a to one of the slots of b. a and b can be identical.

Slots

Slots are the points where incoming links are attached to NetEntities. During activation propagation, the values of all connected gates will be added to one value at the connected slot. (The value of the slot is simply a sum of all incoming activation). During gate calculation, when native modules are calculated, you can access the slot values and calculate your new gate values.

Gates

Links originate from gates. Gates can have more than one link attached to them. There are various parameters at gates that control how activation will be propagated:

- Activation - the gate's activation from the last activation propagation/gate calculation cycle
- Threshold - activation will only be propagated along the gate's links if it exceeds the threshold. The threshold value will be subtracted from activation before propagation.
- Maximum - the maximum activation that may be propagated from this gate. This does not mean this is the maximum activation reaching a linked node's slot: the gate parameter amp is multiplied to the activation *after* the check for the maximum, and of course the weight and confidence values of the link itself are multiplied after all that.
- Minimum - the minimum activation that may be propagated from this gate. Note that, after the creation of a NetEntity, is 0 by default, and that means that the gate will not output negative values.
- Amp factor - this factor is multiplied to the gate's value after all other checks, directly before activation propagation.

- Gate factor - this factor is multiplied to the gate's value before all other checks. It is mainly meant to turn the gate on or off and is used by the directional activators to do so. You normally should have no reason to interfere with what they are doing, so it's good practice to leave the gate factor alone.

- Decay type - A code for the kind of decay that the links at this gate are subject to. -1, the default, means that there will be no decay. We will provide more information on decay functions as we add them. Currently, there is only one more decay function, (code 1), a simple linear decay: Every step the link weight is reduced by 0.05. **Note that decays are computed only when the Entity is used. You won't see the weights changed until the links have been used in activation propagation**. We do this to avoid iterating over a large number of entities every cycle, and of course it's mathematically ok to do so. So don't be confused.

8.2.2. Nodes

Nodes are standard NetEntities that have exactly one slot (ST_GEN), at least one gate (GT_GEN) and a fixed gate calculation function that simply puts the slot activation into all of the gates. Nodes are the standard building blocks of MicoPsi agents. You could build your agents completely with nodes, without caring about native modules. As you will already have noticed from what you know about gates, nodes can be used as neurons in neuronal nets, and that already makes them interesting. But with the different kinds of nodes introduced now, you'll find you can not only compute simple logical functions, but write real programs entirely with nodes. You normally won't want to do so, but you could.

Register nodes

Register nodes are the most basic node types. One slot, one gate. As the name indicates, register nodes are used for passing "parameters" between native modules or to keep links to concept nodes. There are special methods for quick link manipulation at register nodes at the APIs for native modules.

Concept nodes

Concept nodes are nodes that have, besides their GT_GEN gate, a number of standard other gates. These standard gates add meaning to the attached links: The type of the gate that is used for the link denotes the relationship between two concept nodes. Of course, the meaning of these types is simply given by convention, meaning is, as always in MicroPsi agents, only created by use.

- GT_SUB - sublinked nodes are somehow part of the linking node.

- GT_SUR - surlinked nodes are somehow the whole to the linking node. sub/sur links between Concept Nodes are always symmetrical: A sub B implies B sur A.

- GT_POR - porlinked nodes are somehow causal successors of the linking node.

- GT_RET - retlinked nodes are somehow causal predecessors of the linking node. por/ret links between Concept Nodes are always symmetrical A por B implues B ret A. por/ret links normally have spacio-temporal attributes with details on the causal relation between the nodes.
- GT_CAT catlinked nodes are the category for the linking node.
- GT_EXP explinked nodes are exemplars of the linking node. cat/exp links are also always symmetrical.

Two further gate types for symbolic relations are planned, but have not yet been implemented as our own agents still don't need them.

Directional activator nodes

For each standard gate type (see above), there is a directional activator (ACT_SUB, ACT_SUR, etc). When activated, the directional activator will activate all gates of it's type within it's nodespace and all nodespaces contained in that nodespace. "Activate" the gates means setting the gate factors of that gates to 1. As the gate factors for all standard gate types default to 0, no activation will be propagated from standard gates if no directional activator is active.

Sensors

Sensors are nodes that propagate from their GEN gate the value of some external data source. Sensors can be *connected* or *unconnected*. Sensors are connected iff a data source was assigned to them. Unconnected sensors' gates values will always be 0.

Actors

Actors are nodes that output the value of their GEN slots to an external data target (indicating that the agent decided to interact with the world). Actors propagate from their GEN gate the result of that action. (Between -1, failure, and 1, success). Actors can be *connected* or *unconnected*. Actors are connected iff a data target was assigned to them. Unconnected actors don't do anything if they got activation at the slot. Their gate's value will always be 0.

General activators

There will be information on general activators in a later version of this document.

General deactivators

There will be information on general deactivators in a later version of this document.

Associators

There will be information on associators in a later version of this document.

Dissociators

There will be information on dissociators in a later version of this document.

8.2.3. NodeSpace modules

NodeSpace modules are *modules* Modules are NetEntities with a custom gate calculation function. (In contrary to nodes, which have a fixed one).

The particular thing about node spaces is that they *are* entities and *contain* entities. All entities are contained in node spaces (with one exception, the root node space). Node spaces can contain node spaces.

A NetEntitiy is called a *level-one-member* of a node space if the entity is directly contained by the node space. A NetEntity is called a *member* of a node space A if it is a level-one-member of A or of some node space B that is a member of A.

(As you see and as you would have expected: Every entity but the root node space is a member of the root node space)

Node spaces, as they are entites, can have slots and gates. These can be accessed from within the space by actor and sensor nodes. Thus you can wrap functionality into nodespaces, with the spaces' slots and gates as interface to the rest of the system. The contents of the space *is* the gate calculation function of the node space (**module!**).

8.2.4. Native modules

You can program your agents entirely with Nodes. The question then, apparently, is: Why introduce that complicated native module stuff? You'll know when you try to write programs entirely made of Nodes: It's a time-consuming, difficult and annoying task, as complexity increases massively with every node you add. (The special nodes that control the program always effect all nodes within a node space, and that of course creates side effects. It's kind of tricky to get the system to stay in stable states, and every time you add new subsystems, you have to consider side effects on all the other parts.) Programs written entirely with nodes are also slow, and so the native module mechanism provides a convenient way of integrating connectionist/spreading activation principles with good old declarative programming, the latter to be used for tasks that are either too complex to do them with nodes or simply of no use if you have them done with nodes - or both, in most cases.

You'll normally want to use native modules for all low-level tasks: Programs that the agent will never need to be aware of don't need to be written in the "mentalese" language of the agents (nodes!). You need node programs only for things that the agent shall be able to reflect upon.

And although it is possible to write programs that execute themselves, even high-level agent programs will be very likely to be programs that can only be executed by a native module, as it is clearly better to have one execution mechanism and program data than a lot of programs that are also the execution engines for themselves.

Understanding native modules is, at any rate, crucial.

Native modules are modules: Entities with any number of slots and gates and a custom gate calculation function. The difference to node space modules is that this function is not calculated by entities, but by a Java class.

Modules are not restricted to the calculation of gates. And just as inside node space modules may be performed anything that is possible with nodes, inside a native module may be performed anything that is possible with Java, although not everything is encouraged.

To see what *is* encouraged, see *A closer look at native modules*.

8.3. The mathematics of node nets

There is a ready-to-use Java implementation of node nets. When trying to understand what it does and how it works, some people prefer to know the maths behind it all. Besides that, it's of course a good thing to have a precise formal definition. So here it is. (This section is a slightly revised version of material from [9])

8.3.1. Entities, Nodes, Node Spaces, Links

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

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

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

$$U = \{(id, type, I, O, \alpha, f_{act}, f_{node})\}$$

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

Entities come about in different *types*, such as register nodes, concept nodes and so on (see above).
Nodes may be grouped into *node spaces*:

$$S = \left\{U^S, DataSources^S, DataTargets^S, f_{\mathrm{net}}^S\right\}$$

By mapping the $DataSources^S$ of a node space to slots, the $DataTargets^S$ to gates and the local net function f_{net}^S to an entity function, it is possible to embed a node space into a single net entity, called a *node space module*. Thus, hierarchies of node spaces may be created. Often, node spaced contain a number of nodes that have special properties, such as $Activators^S \subset U^S; Activators^S = \{u_{gateType_1}, ..., u_{gateType_n}\}$. Activators influence the way activation spreads within a node space. In a nodespace, there can be one Activator for every gate type (por, ret, sub, sur, cat, exp, sym, ref). The output of the activator is read by the output activation function of all nodes within the nodespace. By setting activators to zero, activation is prevented from spreading through the corresponding gates.

The vector of links between entities is defined as:

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

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 $w \in \mathbb{R}_{[-1,1]}$ and a vector $st \in \mathbb{R}^4; st = (x, y, z, t)$ containing spatial-temporal values.

$$O = \{(gateType, out, \theta, amp, min, max, f_{out})\}$$

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 $f_{out} : \alpha \times O \times Activators \rightarrow out$ 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. By triggering an activator, the spreading of activation from gates of the particular gate type is enabled.

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

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

The value of each slot i_j^u is calculated using f_{net} as the sum of its inputs. Let $(v_1, ..., v_k)$ be the vector of links that end in a slot i_j^u to other nodes, and $(out_1, ..., out_k)$ be the output activations of the respective connected gates:

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

8.3.2. Specific node types

Concept Nodes have a single slot of the type *gen* (for "generic") and their node activation is identical with their input activation: $\alpha = in_{gen}$. Concept nodes have gates of all the standard types: por, ret, sub, sur, cat, exp, sym, ref and gen. The gen gate, as with all nodes, makes the input activation directly available of it is above the threshold θ_{gen} – there is no gen activator.

Register Nodes, the most basic node type, have a single slot of the type *gen* and a single gate of type *gen*. The gen gates behaves like in concept nodes, that is: $out_{gen} = [amp \cdot \alpha]_{min}^{max}, \text{if} \alpha > \theta, 0\text{else}; \alpha = in_{gen}$.

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

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, the agent environment (e.g. the world server). In return, an input value is read that represents failure (-1) or success (1) of the action – this value is returned as a sensor value to out_{gen}. (Not immediately indeed, as the action typically takes some time to execute.)

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

Activators are special registers that exist in correspondence to the standard gate types of concept nodes in a node space. Their output is read by the output activation function of the concept nodes within their nodespace. (see above)

General activation nodes are registers that, when active, increase the activation α of all nodes in the same node space.

General deactivation nodes, as the counterparts to general activator nodes, dampen the activation of all nodes within the same node space.

Associator nodes are used to establish links between nodes in a node space. This happens by connecting all nodes with active gates. The weight of the new link calculates as

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

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

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

A. The AEP Java API

The classes and interfaces that are documented at the following URLs are published API classes. API classes are relatively stable and unlikely to change very much in future releases. At least there will be special care for backward compatibility when evolving these classes.
This is also true for classes and interfaces directly used by the classes below. (The classes below are the API "entry points")

- `http://www.artificial-emotion.de/javadoc/de/artificialemotion/micropsi/net/AbstractNativeModuleImpl.html`
- `http://www.artificial-emotion.de/javadoc/de/artificialemotion/micropsi/net/GateManipulator.html`
- `http://www.artificial-emotion.de/javadoc/de/artificialemotion/micropsi/net/AbstractNativeModuleImpl.StructureModificator.html`
- `http://www.artificial-emotion.de/javadoc/de/artificialemotion/micropsi/net/InnerStateContainer.html`
- `http://www.artificial-emotion.de/javadoc/de/artificialemotion/micropsi/main/TypeStrings.html`
- `http://www.artificial-emotion.de/javadoc/de/artificialemotion/comp/agent/aaa/AgentWorldAdapterIF.html`
- `http://www.artificial-emotion.de/javadoc/de/artificialemotion/comp/robot/RobotActionExecutor.html`
- `http://www.artificial-emotion.de/javadoc/de/artificialemotion/comp/robot/RobotPerceptionExtractor.html`
- `http://www.artificial-emotion.de/javadoc/de/artificialemotion/comp/world/objects/AbstractAgentObject.html` (Warning: This is still a somewhat moving target and likely to change in the next versions of the toolkit.)

Beware: There are hundreds of other classes that are not API, but some are. Generally, a good way of telling API from non-API is asking yourself the following questions:

- Is the class a member of the de.artificialemotion.micropsi package? If it is, it is definitively API. All classes there are published and can be relied on - all that is public inside these packages is API. Still, you must **not** try to access any protected members by java reflection or cheating with packages, and there is no use in doing so - if you do, you'll cause problems and have a terrible time. You have been warned.

- Is the class mentioned in this document? If it is, it is definitively API.
- Is there good JavaDoc on the class? If there is, it's likely that the class is API or at least will become API in the future. But don't rely.

B. Glossary

- **Alarms** see Meta-Management
- **Action** MicroPsi agents may have an effect on their environment or their internal representations using a basic set of actor nodes. These may be arranged in sequences or alternatives and organized in hierarchies, thus creating more abstract actions macros and scripts. The execution of these scripts may be controlled with sensor nodes, which serve as pre-conditions, post-conditions or suitability measures. Sensors may be grouped into hierarchies to, thereby representing more abstract situations and objects. Often, such a sense macro will contain actions and vice versa.
  Because active and sensoric schemata are represented as node nets, agents might organize and rewrite their own behavior scripts.
- **Activators** are special register nodes that exist in correspondence to the gate types (POR, RET, SUB, SUR, CAT, EXP, SYM, REF) of concept nodes of a node space. Their output is read by the output activation function of the respective gate of their node space. By setting activators to zero, no activation can spread through the corresponding gates.
- **Actor** nodes are extensions to sensor nodes. Using their node function, they give their input activation to an external data target. In return, an input value is read that typically represents failure (-1) or success (1) of the action.
- **AEP** Artificial Emotion Project. An effort to build cognitive agents based on principles of the Psi theory, which has lead to a rather generic toolkit for the construction of architecture for cognitive multi agent systems (not necessarily emotional).
- **Affiliation** See Social Urges
- **Associator** nodes are used to establish links between nodes in a node space. This happens by connecting all nodes with gates having an activation different from zero.
- **Calculate/Propagate Cycle** In every NetStep, the spread of activation in node nets is propagated, and the internal functionality of active nodes (activation functions, or code of native modules) is executed.
- **CAT** Link type in node nets extending to Psi theory, corresponding to gate in concept node, denotes is a/member of relationship.
- **Certainty** Part of annotation of a link. Is not always required, but aids in matching of representations and reasoning. By default, certainty is 1.

- **Cognitive Urges** Direct the cognitive and explorative behavior of an agent according to Psi theory. Currently consist of urge for uncertainty reduction and urge for acquisition of competency.
- **Concept nodes** are the most widespread node type in MicroPsi agent representations. They have a single slot (GEN) and gates for building hierarchical representations (GEN, POR, RET, SUR, SUB, CAT, EXP, SYM, REF). Concept nodes replace Drners Quads.
- **Data source** Input interface of node space to agent environment or other node spaces. Used for sensor nodes (as sensors).
- **Data target** Output interface of node space to agent environment or other node spaces. Used for actor nodes (as actuators).
- **Decay** As a means of forgetting, the links between nodes may decay over time, using node space specific decay functions. Nodes with links below a certain threshold may be removed (garbage collection). Usually, links above a certain strength do not decay.
- **Disassociator** nodes are the counterpart of associator nodes; they decrease or remove links between currently active nodes in the same node space.
- **Drives** see Urges
- **Emotion** In Drners framework, emotion is a set of configurations of the cognitive system of an individual. Cognitive processes are embedded into the emotional sub-system, and so emotional configurations influence how an agent perceives, plans, memorizes, selects intentions, acts etc. Main components of the emotional system are parameters that represent cognitive urges, situation evaluation and the modulators, like arousal, resolution level and selection threshold. They control the usage of semantic schemata during perception, retrieval etc., for instance by limiting search depth and width. The emotional modulation is designed to allocate mental resources in way that is suitable to a given situation and reduce the computational complexity of the current task. (Drner/Schaub 1998)
- **EXP** Link type in node nets extending to Psi theory, corresponding to gate in concept node, denotes has element relationship.
- **Gate** Output of a node. Contains value for output activation, threshold, amplification factor, upper and lower boundaries. The spread of activation through a gate may be restricted by directional activators (gate type specific).
- **Gate Manipulator** Programming interface for gates (to change activation, thresholds, amplification values etc.).
- **GEN** Link type in node nets loosely according to Psi theory, corresponding to gate in concept node, used for generic links without directional activation.
- **General activation nodes** are special nodes with a single slot and gate of type GEN. When active, they increase the activation 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.

- **Hierarchical categories** the similarity of node schemas can be established by a complete or a partial match. By constraining the depth of the comparison, it is possible to discover structural similarity, for instance between a human face and a cartoon face. However, the key to structural similarity is the organization of node schemas into hierarchies (where an abstract face schema may consist of eye, nose and mouth schemas in a certain arrangement, and can thus be similar to a smiley). Furthermore, many objects can only be classified using abstract hierarchies. Such hierarchies can be derived mainly in three ways: by identifying prominent elements of objects (that is, structures that are easy to recognize by interaction or perception and also good predictors for the object category), by guessing, and by communication. (Bach 2002)

- **HyPercept** Hypothesis based perception. According to Drners Psi theory, a mechanism of perception that works by forming a hypothesis (usually based on former experiences of the agent, and inspired by context and features of the environment), and then verifying this hypothesis against the primitive patterns available from the sensors of the agent.

- **Links** Connect gates of nodes with slots of (usually different) nodes, and may be annotated with weights, certainty values, spatial/temporal information. Activation spreads through links. Current link types include GEN, POR, RET, SUR, SUB, CAT, EXP. Of these, POR and RET, SUR and SUB, CAT and EXP are reciprocal.

- **Macro** Part of an active schema that is nested within a hierarchy.

- **Memory** Node nets act as universal data structures for perception, memory and planning. Even though the Psi theory does not distinguish between different types of memory, we have found that differentiating special areas (node spaces) helps to clarify the different stages of cognitive processing.
  The links between nodes decay over time (as long as the strength of the links does not exceed a certain level that guarantees not to forget vital information). The decay is much stronger in short term memory, and is counterbalanced by two mechanisms:

  - usage strengthens the links, and
  - events that are strongly connected to a positive or negative influence on the urges of the agent (such as the discovery of an energy source or the suffering of an accident) lead to a retro gradient connection increase of the preceding situations.

  If a link deteriorates completely, individual isolated nodes become obsolete and are removed. If gaps are the result of such an incision, an attempt is made to bridge it by extending the links of its neighbors. This process is meant to lead to the exclusion of unimportant elements from object descriptions and protocol chains.

- **Meta-Management** It is quite possible that the allocation of processing resources of the agent does not meet the demands of the changing environment. This is the task of the meta-management. Because this module is not called very frequently, the agent may fail to adapt quickly to dramatical events. Drner has proposed a securing behavior that should be executed by the agent in regular intervals, while Sloman describes a system which he terms alarms, with the same purpose: to quickly disrupt current cognitive processes if the need arises. (Sloman 1994) There is no alarm system in MicroPsi yet.

- **MicroPsi** An agent architecture that attempts to capture the vital aspects of Drners Psi theory, extending them where deemed necessary. MicroPsi agents are based on the AEP agent toolkit.

- **Modulators** Constrain spread of activation in node nets and direct behavior of agents according to Psi theory.

- **Motivation** According to the Psi theory, the agent possesses a number of innate desires (urges) that are the source of its motives. Events that raise these desires are interpreted as negative reinforcement signals, whereas a satisfaction of a desire creates a positive signal. Currently, there are urges for intactness, energy (food and water), affiliation, competence and reduction of uncertainty. The levels of energy and social satisfaction (affiliation) are self-depleting and need to be raised through interaction with the environment. The cognitive urges (competence and reduction of uncertainty) lead the agent into exploration strategies, but limit these into directions, where the interaction with the environment proves to be successful.
  The execution of internal behaviors and the evaluation of the uncertainty of externally perceivable events create a feedback on the modulators and the cognitive urges of the agent.

- **Native Module** In the current implementation, native modules contain Java code and can perform any kind of manipulation on the node net. Input and output to native modules is provided through slots and gates, so that they may act just like any other kind of node.

- **NetEntity** Part of a node net; a node, node space or native module.

- **NetStep** Simulation cyle of node nets.

- **Neurons** Drner suggests using neural networks (with threshold elements) as means of representation within Psi agents. Since these neurons typically represent objects, features, situations and often have link weights of just 1 or 0, it may be appropriate to look at them as semantic networks, Bayesian nets or influence networks. In Drners implementation, mechanisms for spreading activation have been omitted, and most control structures supposedly implemented with connectionist means are replaced by procedures in the programming language Delphi. In MicroPsi agents, the notion of neurons is replaced by a more general node net formalism.

- **Node Nets** are the general means of representation in MicroPsi agents. They consist of net entities and links and contain control structures, sensoric and active schemata, representations of plans, goals etc. and interfaces to the environment.
- **Node Space** A collection of net entities (nodes, native modules, node space modules) including activators, deactivators etc. which are limited to this space. Node spaces may contain other node spaces (node space modules), so it is possible to build hierarchies and structured arrangements.
- **Node Space Module** By linking the data sources of a node space to a set of slots and the data targets to a set of gates, a node space may be contained in a module that acts just like a node and may be contained in another node space.
- **Perception** The agent represents external situations in the same way as hypotheses or acquired knowledge; this is done in the local perceptual space. To this end, the agent retrieves hypotheses from previous content in the local perceptual space or from its long term memory and tests the immediate external percepts against them. This is called hypothesis based perception, or hypercept. If the expectations of the agent fail, and no theory about the perceived external phenomena can be found, a new object schema is acquired by a scanning process (accommodation) that leaves the agent with a hierarchical node net. Abstract concepts that may not be directly observed (for instance classes of transactions or object categories) are defined by referencing multiple schemas in such a way that their commonalities, differences or process structure become the focus of attention.
- **Planning** The planning algorithms given in the current MicroPsi are very simple: given a goal (derived from a motivational process), the agent tries to find a chain of actions that has lead in the past from the given situation to the goal situation. If no such automatism is remembered, its construction is attempted by combining actions. Here, depth and width of the search are controlled by the modulators.
- **Psi theory** Theory of emotion, motivation and representation by Dietrich Drner (1999, 2002). Has been partially implemented by Drner in simulated agents (Emos). Psi is not an acronym of sorts, but stands for the Greek letter Ψ, which for some reason is a favorite with psychologists.
- **POR** Link type in node nets according to Psi theory, corresponding to gate in concept node, denotes leads to/causes relationship.
- **Quad** A memory unit consisting of threshold elements according to Drners Psi theory. Quads contain a central neuron and usually four auxiliary neurons allowing for spreading activation (POR, RET, SUR, SUB). In MicroPsi agents, Quads are replaced by concept nodes.
- **REF** Link type in node nets loosely according to Psi theory, corresponding to gate in concept node, denotes refers to relationship (used for linguistic labeling).
- **Register** nodes are the most basic node type. They consist of a single slot and gate, both of type GEN, and may act as threshold elements.

- **Representation** Objects, situations, categories, actions, episodes and plans are all represented as hierarchical networks of nodes. Hierarchies are achieved with different link types (like SUR, SUB, POR, RET) and are derived from a theory of representation by Klix (1984). Nodes may be expanded into weighted conjunctions or disjunctions of subordinated node nets, and ultimately bottom out in references to sensors and actuators. Thus, the semantics of all acquired representations result from interaction with the environment or from somatic responses of the agent to external or internal situations. (For communicating agents, they may potentially be derived from explanations, where the interaction partner another software agent or a human teacher refers to such experiences or previously acquired concepts.)
- **RET** Link type in node nets according to Psi theory, corresponding to gate in concept node, denotes comes from/is caused by relationship.
- **Scope** Nodes controlling the spread of activation or the linking and unlinking of active nodes are usually limited in scope on the current node space.
- **Script** Arrangement of POR/RET-linked concept nodes that may be executed sequentially. Macros and hierarchies can be achieved by adding SUB/SUR linked nodes. With appropriate weights and thresholds, conjunctions and disjunctions may be encoded. Scripts bottom out in actor and sensor nodes. They can represent plans or control structures of the agent and are executed using a script execution mechanism.
- **Script execution** Native module for the execution of hierarchical scripts (which are made up from concept nodes and bottom out in sensor and actor nodes).
- **Sensor nodes** are similar to register nodes, however, their activation is computed from an external data source.
- **Slot** Input of a node. Contains a value that is usually the weighted sum of the activation of connected gates.
- **Somatic Urges** Direct agents to ensure their survival according to the environment. Currently consist of urges for food, water and intactness.
- **Social Urges** Direct agents in their social interactions. Currently restricted to an urge for affiliation.
- **ST Link** A link with a spatio-temporal annotation (a four-D vector to denote spatial and/or temporal relationships between features).
- **Structure Modificator** Programming interface for manipulating structures in node nets (such as creating or changing nodes and links).
- **SUB** Link type in node nets according to Psi theory, corresponding to gate in concept node, denotes consists of relationship.
- **SUR** Link type in node nets according to Psi theory, corresponding to gate in concept node, denotes part of relationship.

- **SYM** Link type in node nets loosely according to Psi theory, corresponding to gate in concept node, denotes is represented by relationship (used for linguistic labeling).
- **Urges** Basic drives of an agent according to Psi theory. Lead to motivation and direct actions of an agent. Urges may be somatic (like hunger, thirst) or cognitive (reduction of uncertainty, competency).
- **Weight** Part of annotation of a link. Relevant to spreading activation and summing of activations in slots.

Bibliography

[1] The Eclipse Project, http://www.eclipse.org

[2] Log4j documentation, http://jakarta.apache.org/log4j/docs/index.html

[3] Fundamental modeling concepts, http://fmc.hpi.uni-potsdam.de/

[4] Klix, F. (1984). Über Wissensrepräsentation im Gedächtnis. In F. Klix (Ed.): Gedächtnis, Wissen, Wissensnutzung. Berlin: Deutscher Verlag der Wissenschaften.

[5] Dörner, D., Bartl, C., Detje, F., Gerdes, J., Halcour, D., Schaub, H., & Starker, U. (2002). Die Mechanik des Seelenwagens. Eine neuronale Theorie der Handlungsregulation. Bern, Göttingen, Toronto, Seattle: Verlag Hans Huber.

[6] Dörner, D., & Schaub, H. (1998). Das Leben von PSI. Über das Zusammenspiel von Kognition, Emotion und Motivation. http://www.uni-bamberg.de/ ba2dp1/psi.htm Dörner, D. (1999). Bauplan für eine Seele. Reinbeck: Rowohlt

[7] Braitenberg, V. (1984) Vehicles. Experiments in Synthetic Psychology. MIT Press.

[8] 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, (pp. 3-12)

[9] Bach, J., Vuine, R. (2003) The AEP Toolkit for Agent Design and Simulation. M. Schillo et al. (eds.): MATES 2003, LNAI 2831, Springer Berlin, Heidelberg. (pp. 38-49)

[10] Bach, J., Vuine, R. (2003). The MicroPsi Architecture for Cognitive Agents. Poster, Proceedings of EuroCogSci03, Osnabrck, Germany. (p. 370)

[11] Bach, J. (2003). Connecting MicroPsi Agents to Virtual and Physical Environments. Workshops and Tutorials, 7th European Conference on Artificial Life, Dortmund, Germany. (pp. 128-132)

[12] Bach, J., Vuine, R. (2003). Designing Agents with MicroPsi Node Nets. Proceedings of KI 2003, Annual German Conference on AI. LNAI 2821, Springer, Berlin, Heidelberg. (pp. 164-178)