MicroPsi 2: The Next Generation of the MicroPsi Framework

Joscha Bach

Berlin School of Mind and Brain, Humboldt University of Berlin
Unter den Linden 6, 10199 Berlin, Germany
joscha.bach@hu-berlin.de

Abstract. The cognitive architecture MicroPsi builds on a framework for simulating agents as neuro-symbolic spreading activation networks. These agents are situated in a simulation environment or fitted with robotic bodies. The current implementation of MicroPsi has been re-implemented from the ground up and is described here.

Keywords: MicroPsi Framework, Cognitive Architecture, Psi Theory.

1 Introduction

MicroPsi (Bach 2003) is a cognitive architecture with a focus on grounded representations, cognitive modulation and motivation. MicroPsi agents are autonomous systems that combine associative learning, reinforcement learning and planning to acquire knowledge about their environment and navigate it in the pursuit of resources. MicroPsi is also being used to model the emergence of affects and higher level emotions (Bach 2012a), and to model human performance and personality properties in the context of problem solving (Bach 2012b). The architecture extends concepts of Dietrich Dörner's Psi theory, and is thus rooted in a psychological theory of motivation and complex problem solving (Dörner 1999, Dörner et al. 2002). The principles and concepts of MicroPsi are described in detail in the book *"Principles of Synthetic Intelligence"* (Bach 2009) and subsequent publications (Bach 2011) and are not discussed here. Instead, we will focus on the *MicroPsi framework*, i.e., the simulation and design framework that allows the construction and execution of our family of cognitive models.

Unlike many other cognitive architecture frameworks that define agents in the form of code (either in a domain specific language, as a set of rules and representational items), MicroPsi uses graphical definitions for its agents, and a graphical editor as the primary interface. In this respect, it is for instance similar to COGENT (Cooper and Fox 1998). While rule-based representations and (hyper-)graphical representations are computationally equivalent, the graphical paradigm highlights weighted associations, allows to visualize conceptual hierarchies, activation spreading, perceptual schemata and parallelism.

The first implementation of the MicroPsi framework spanned the years 2003 to 2009, and was built in Java as a set of plugins for the Eclipse IDE. The graphical editor was built on SWT. It comprised about 60000 lines of code, and although a lot of effort went towards platform independence (with the exception of a DirectX/.Net based 3D viewer component), deployment on the various operating systems and across several versions of Eclipse became support intensive, especially after its adoption by teams outside of our group.

Gradual changes in the formalization of MicroPsi and the emergence of new software development methodologies and tool chains, especially the move from Java design patterns and XML tools towards lightweight and agile Python code, prompted a complete rewrite of the MicroPsi framework, starting in 2011. The following section describes the overall structure of the framework, followed by detailed definitions of the node net formalism and the structure of simulation worlds that enable running MicroPsi agents.

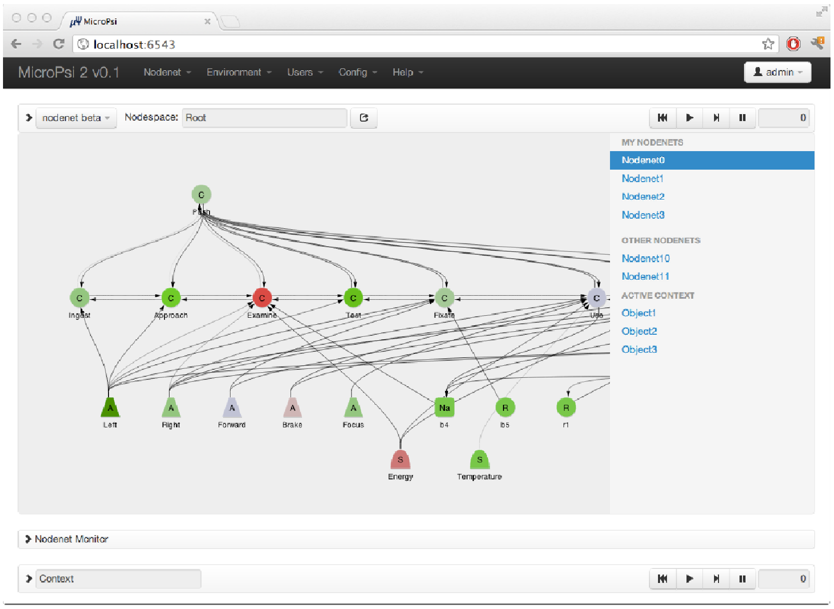

Fig. 1. MicroPsi User Interface, Node Net View

2 The MicroPsi 2 Framework

MicroPsi 2 is being written in Python, with a minimum of external dependencies, to make installation as simple as possible. Instead of a standard GUI, we decided to render the user interface in a web browser, and to deploy the MicroPsi agent simulation as a *web application* (figure 1). The MicroPsi server acts as a (local or remote) Web server that delivers UI components as HTML/Javascript, and facilitates the communication between the browser based renderer and the agent simulator via *JSON* and *JSON remote procedure calls*. Rendering is supported by Twitter's widget library *Bootstrap* (2012) and the Javascript library *PaperJS* (Lehni and Puckey, 2011).

This design choice makes it possible to remote control a MicroPsi simulation server from a different machine, and even to use the MicroPsi runtime without any local installation at all, as long as customization is not desired.

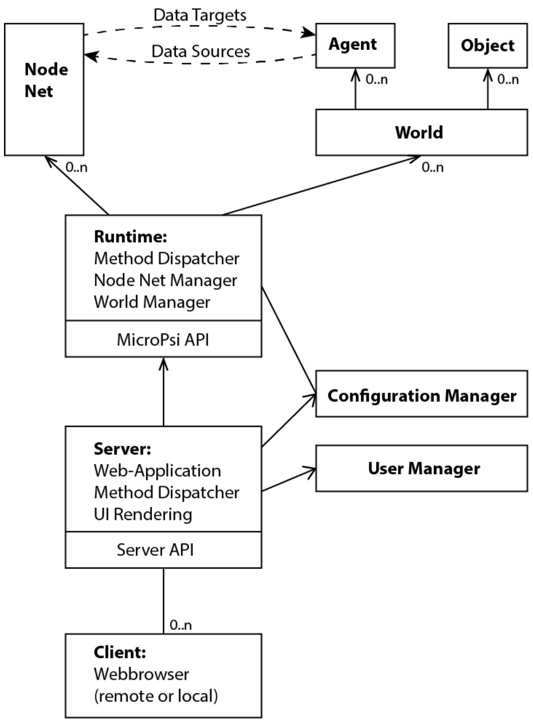

Fig. 2. Components of MicroPsi 2 Framework

MicroPsi consists of a server (the web application), a runtime component, a set of node nets, a set of simulation worlds, a user manager and a configuration manager (figure 2). The server is built on the micro web framework *Bottle* (Hellkamp 2011) and communicates with all current users via their web browsers through the *Server API*. User sessions and access rights are handled by the user manager component.

On startup, the server invokes the runtime component, which interfaces to the server with the *MicroPsi API*. The runtime is designed to work independently of the server and does not need to be deployed as a web application (command line interaction or OS based user interfaces are possible as well).

The runtime supplies a manager for *MicroPsi node nets* (see section 4), and a manager for simulation worlds (or interfaces to outside environments, such as robotic bodies, remote data providers, etc.). Standard simulation worlds (section 6) provide agents (node net embodiments) and objects as situated state machines.

3 MicroPsi Agents

MicroPsi interprets cognitive models as agents, situated in dynamic environments. MicroPsi agents are entirely defined as hierarchical *spreading activation networks* (SAN), which—for lack of a better name—are called *node nets.* Node nets are the brains of these agents—or rather, an abstraction of the information processing provided by brains, and the environment provides a body and stuff to interact with.

The body manifests itself as a set of data sources (which can be thought of as the terminals of sensory neurons) and data targets (the abstracted equivalent of motor neurons). By reading activation values from data sources, and sending activation into data targets, the MicroPsi agent may control its body and interact with its world.

MicroPsi's node nets can be interpreted as neural networks and afford neural learning paradigms. For the purposes of information storage and retrieval, they can be seen as semantic networks with a small set of typed links to express associative, partonomic, taxonomic and causal relationships.

Since the nodes can also encapsulate state machines and arbitrary operations over the node net, they can also be understood as components of a concurrent, modularized architecture, with activation spreading as the primary means of communication between the modules.

4 Definition of Node Nets

This section gives an overview over the definition of MicroPsi node nets.

$$NodeNet \equiv$$
$$\langle States: \{s\}, s_0, \mathrm{f}^{\mathrm{net}}, NodeTypes: \{nt\}, DataSources: \{ds\}, DataTargets: \{dt\} \rangle$$

A node net is characterized by a set of states, a starting state s_0, a network function $\mathrm{f}^{\mathrm{net}}: s_t \times DataSources \rightarrow s_{t+1} \times DataTargets$ that determines how to advance to the next state, and set of *node types*. Data sources and data targets provide a connection to the environment; a data source represents a value that is determined by an environmental state, while the values of data targets can be changed to effect something in the environment.

$$s \equiv \langle Nodes: \{u\}, Links: \{l\}, NodeSpaces: \{ns\}, t \rangle$$

The state of a node net is given by a set of *nodes*, or *units*, a set of directed *links*, a set of *node spaces* and the current simulation step t. Each node is part of exactly one node space. The primary role of node spaces is to provide some additional structure to node nets, similar to folders in a file system.

Node spaces form a tree—thus, each node space, with the exception of the root node space, has exactly one parent node space.

$$ns \equiv \left\langle \begin{matrix} parent \in NodeSpaces \cup \emptyset, \\ SlotTypes: \{st\}, GateTypes: \{gt\}, \\ Activators: \{act\}, Associators: \{assoc\}, \\ decay, \theta^{decay} \end{matrix} \right\rangle$$

Node spaces do not only provide some additional structure to node nets, they may also limit the spreading of activation via node space specific $Activators$, control how connections between nodes are strengthened based on $Associators$, or how they are weakened over time using a $decay$ parameter. (More on these things below.)

$$u \equiv \langle id, parent \in NodeSpaces, nt, params^u, Gates, Slots \rangle$$

Each node is characterized by its identifier id, its type nt, an optional set of parameters $params^u$ (which can make the node stateful), a set of gates and a set of slots. Gates are outlets for activation, while slots are inlets.

$$nt \equiv \langle GateTypes: [gt], SlotTypes: [st], \mathrm{f}^{\mathrm{u}} \rangle$$

The types of slots and gates of a node are defined within the node type, next to additional functionality f^{u} performed by the node whenever it becomes active. In most cases, f^{u} is limited to transmitting activation within the node, from the standard slot 'gen' to the gates.

$$\mathrm{f}^{\mathrm{u}}_{\mathrm{default}} \equiv \text{for each } gate \in Gates_u : \alpha_{gate} = \alpha_{slot\ \mathrm{gen}_u}$$

Nodes can store additional parameters and change them in the course of the node function, which makes them state machines:

$$\mathrm{f}^{\mathrm{u}}: \{\alpha_{Slots_{u},t}\} \times params_{u,t} \rightarrow \{\alpha_{Gates_{u},t+1}\} \times params_{u,t+1}$$

More generally, some nodes may contain arbitrary functions, such as the creation of new nodes and links, procedures for neural learning, planning modules etc. These functions take the form of a program in a native programming language (here, *Python*), and hence, such nodes are also called *native modules*.

$$\begin{aligned}\mathrm{f}^{\mathrm{u}}_{\mathrm{native}}: Nodes_t \times Links_t \times NodeSpaces_t \rightarrow \\ Nodes_{t+1} \times Links_{t+1} \times NodeSpaces_{t+1}\end{aligned}$$

The nodes form a directed graph, with links connecting their gates to slots.

$$l \equiv \langle gate \in Gates_{origin \in Nodes,} slot \in Slots_{target \in Nodes}, \omega, c \rangle$$

A link is characterized by the gate of origin, the slot of the target node, a weight ω and a confidence parameter c. Usually, $-1 \leq \omega \leq 1$ and $0 \leq c \leq 1$.

$$gate \equiv \langle gt, \alpha, out, params^{gate}, min, max, \mathrm{f}^{\mathrm{out}} \rangle$$

A gate is determined by its gate type gt, an activation α, an output activation out (which is transmitted via the links originating at that gate), a minium and a maximum value, and the output function $\mathrm{f}^{\mathrm{out}}$.

$$gt \equiv \langle id, \mathrm{f}^{\mathrm{gate}}, nt \rangle$$

$$\mathrm{f}^{\mathrm{gate}}: \alpha_{gate} \times params_{gate} \rightarrow \alpha'_{gate}$$
$$\mathrm{f}^{\mathrm{out}}: \alpha'_{gate} \times \alpha_{act:gt_{gate}, ns_{u:gate}, t} \rightarrow out_{gate}$$

Together with the gate function $\mathrm{f}^{\mathrm{gate}}$, which is supplied by the type gt of the gate, the output function specifies how to calculate the output activation.

$$\mathrm{f}^{\mathrm{gate}}_{\mathrm{default}} \equiv \begin{cases} \alpha_{gate}, \text{if } \alpha_{gate} > \theta_{gate} \\ 0, \text{else} \end{cases}$$

The default gate function assumes a threshold parameter θ and sets the activation to zero, if it is below this threshold. This turns the node into a simple threshold element. The reason that the gate calculations are split in two separate functions is customization: gate functions may be sigmoidal, to enable back-propagation learning, or bell-shaped, to build radial basis function networks, etc.

$$\mathrm{f}^{\mathrm{out}} \equiv \alpha_{gate}\, act_{gt_{gate}, ns_{u:gate}} \Big]_{min_{gate}}^{max_{gate}}$$

After the application of the gate function, the output function may control the spread of activation through a gate by multiplying the gate's activation with the value of the *activator* act that corresponds to the type of the gate (and is defined and adjusted on the level of the node space that contains the node).

Next to gates, nodes feature *slots*.

$$slot \equiv \langle st, \alpha \rangle$$
$$st \equiv \langle id, \mathrm{f}^{\mathrm{slot}}, nt \rangle$$

Slots are characterized by their type st and their activation α.While nodes may have multiple slots to receive activation, most offer just one (of type 'gen'). The activation of a slot is determined by the slot function $\mathrm{f}^{\mathrm{slot}}$, which sums up the incoming activation.

$$\mathrm{f}^{\mathrm{slot}} \equiv \left\{(\omega, c)_{Links_{slot}}\right\} \times \left\{out_{gate_{Links:slot}}\right\} \rightarrow \alpha_{slot}$$
$$\mathrm{f}^{\mathrm{slot}}_{\mathrm{default}} \equiv \sum_{l \in Links_{slot}} \omega_l\, out_{gate_l}$$

Again, alternate slot functions can be defined (for instance, a squared average or a maximum function), and are stored or changed on the level of the node space that contains the respective node.

The slot functions provide the transmission of activation between nodes, along links. The changes in strength of these links are influenced by the associator functions and decay functions, which act on the weights of all links originating in a given node space.

$$\mathrm{f}^{\mathrm{assoc}} \equiv \omega_{gate^i_{u_1}, slot^j_{u_2}, t+1} = \left(\sqrt{\omega_{gate^i_{u_1}, slot^j_{u_2}, t}} + assoc_{ns_{u_1}}\, \alpha_{gate^i_{u_1}}\, \alpha_{slot^j_{u_2}}\right)^2$$

The association between two nodes is strengthened based on the activation of the respective slots and gates the link connects, and the activity of the association factor $assoc$ of the respective node space.

$$\mathrm{f}^{\mathrm{decay}} \equiv$$
$$\omega_{gate^i_{u_1}, slot^j_{u_2}, t+1} = \begin{cases} \sqrt{max\left(0, \omega^2_{gate^i_{u_1}, slot^j_{u_2}, t} - decay_{ns_{u_1}}\right)}, \text{if } \omega < \theta^{decay}_{ns_{u_1}} \\ \omega_{gate^i_{u_1}, slot^j_{u_2}, t}, \text{ else} \end{cases}$$

If the decay factor of the respective node space has a value between 0 and 1, and the weight of the link is below the decay threshold θ^{decay}, the link is weakened in every simulation step. This provides a way of 'forgetting' unused connections. The decay threshold ensures that very strong connections are never forgotten.

In each simulation step, the network function $\mathrm{f^{net}}$ successively calls all slot functions $\mathrm{f^{slot}}$, the node functions $\mathrm{f^{u}}$ and gate functions $\mathrm{f^{gate}};\mathrm{f^{out}}$ of all active nodes, and the associator functions $\mathrm{f^{assoc}}$ and decay functions $\mathrm{f^{decay}}$ for all links.

5 Basic Node Types

The most primitive node type is a *Register*. It provides a single slot and a single gate of type gen and acts as a threshold element.

$$Register \equiv \langle GateTypes = [\mathrm{gen}], SlotTypes = [\mathrm{gen}], \mathrm{f^u} = \mathrm{f^u_{default}} \rangle$$

The basic conceptual element, analogous to Dietrich Dörner's Psi theory, is the *Quad*. It makes use of a single 'gen' slot and the four directional gates 'por', 'ret', 'sub', 'sur'. 'Por' encodes succession, 'ret' predecession, 'sub' a *part-of* relationship, and 'sur' stands for *has-part*. With the 'gen' gate, associative relationships can be expressed.

$$Quad \equiv \left\langle \begin{matrix} GateTypes = [\mathrm{gen,por,ret,sub,sur}], \\ SlotTypes = [\mathrm{gen}], \mathrm{f^u} = \mathrm{f^u_{default}} \end{matrix} \right\rangle$$

Concept nodes extend quads by the gates 'cat' (for *is-a* relations), and 'exp' (for their inverse), as well as 'sym' and its inverse 'ref' for symbolic labeling. Concept nodes may be used to express taxonomies.

$$Concept \equiv \left\langle \begin{matrix} GateTypes = [\mathrm{gen,por,ret,sub,sur,cat,exp,sym,ref}], \\ SlotTypes = [\mathrm{gen}], \mathrm{f^u} = \mathrm{f^u_{default}} \end{matrix} \right\rangle$$

The connection to the environment is provided by sensor nodes, which have no slots and only a single gate, which receives its activation from the associated data source. The sensor type is given as a node parameter.

$$Sensor \equiv \langle GateTypes = [\mathrm{gen}], SlotTypes = \emptyset, \mathrm{f^{node}}\colon \alpha_{gate\,\mathrm{gen}} = ds_{sensorType} \rangle$$

Likewise, actor nodes influence the environment by writing the activation received through their single 'gen' slot into a data target. The actor type is given as a node parameter.

$$Actor \equiv \langle GateTypes = \emptyset, SlotTypes = [\mathrm{gen}], \mathrm{f^{node}}\colon dt_{actorType} = \alpha_{slot\,\mathrm{gen}} \rangle$$

Activator nodes are special actors. Instead of a data source, they target the activator act corresponding to the activator type $actType$ (given as a node parameter) of their node space. Thus, activator nodes may be used to restrict the spreading of activation to certain link types.

$$Activator \equiv \left\langle \begin{matrix} GateTypes = \emptyset, SlotTypes = [\mathrm{gen}], \\ \mathrm{f^u}\colon \alpha_{act_{actType}, ns_{activator}} = \alpha_{slot\,\mathrm{gen}} \end{matrix} \right\rangle$$

Associator nodes work just like activators, but target the association factor $assoc$ of their node space.

$$Associator \equiv \left\langle \begin{matrix} GateTypes = \emptyset, SlotTypes = [\mathrm{gen}], \\ \mathrm{f}^{\mathrm{u}}: \alpha_{assoc_{assocType}, ns_{associator}} = \alpha_{slot\ \mathrm{gen}} \end{matrix} \right\rangle$$

6 Environment

Within the MicroPsi framework, agents may be embedded into an environment ($world$). The environment must provide a *world adapter* wa for each MicroPsi agent. The world adapter offers *data sources*, from which the agent's node net may read environmental information, and *data targets*, which allow the agent to effect changes in the world. Since the environment only has write access to data sources, and read access to data targets, node net and environment may be updated asynchronously.

The world adapter may interface a local multi-agent simulation, a robotic body, a computer game client or simulation server, dynamically updated stock data, etc. Here, we give a simple simulation world as an example.

$$World \equiv \langle States: \{ws\}, ws_0, terrain, WorldAdapters: \{wa\}, \mathrm{f}^{\mathrm{world}} \rangle$$

$$wa \equiv \langle DataSources: \{ds\}, DataTargets: \{dt\} \rangle$$

The simulation is determined by its state, a set of fixed properties ($terrain$), a set of world adapters (which provide connections to agents and additional environments) and a function $\mathrm{f}^{\mathrm{world}}: ws_t \times terrain \times \{DataTargets\} \rightarrow ws_{t+1} \times \{DataSources\}$ that determines how to advance to the next state.

$$ws \equiv \langle Objects: \{obj\}, t_w \rangle$$

The state of the world consists of a set of objects and the time step of the simulation.

$$obj \equiv \langle pos, ObjectStates: \{os\}, \mathrm{f}^{\mathrm{obj}} \rangle$$

Objects have a position pos (for instance $\in \mathbb{R}^3$), a set of *object states* os and an object function, $\mathrm{f}^{\mathrm{obj}}$ that determines how the position and states of the object change from one state to the next, based on the previous state, the states and positions of other objects and the terrain.

$$\begin{gathered} Agents: \{agent\} \subseteq Objects, \\ \mathrm{f}^{\mathrm{obj}}_{agent}: pos_{agent,t} \times ObjectStates_{agent,t} \times DataTargets_{wa:agent} \times \\ Objects_t \times terrain \rightarrow \\ pos_{agent,t+} \times ObjectStates_{agent,t+1} \times DataSources_{wa:agent} \end{gathered}$$

Agents are objects in the world like any other, but each agent object corresponds to a world adapter, which links it to a node net. Think of the agent object as the body of the MicroPsi agent, and the object states as its physiological states. The object function of the agent $\mathrm{f}^{\mathrm{obj}}_{agent}$ advances these physiological states, the position of the agent and the inputs to the node net.

In each simulation step, the world function calls all object functions, and takes care of the creation of new objects and the removal of obsolete ones.

7 Applications

Compared with the original implementation of MicroPsi, the current iteration of the framework is still fragmentary; at the time of writing, it supports only a simple generic simulation world for multi agent experiments (instead of the various simulation environments provided in MicroPsi 1). Also, 3D viewing components for environments and facial expressions are completely absent.

The current priority of MicroPsi 2 lies on affective simulation for problem solving experiments (see Bach 2012b), and its application as a general framework for knowledge representation in a hierarchical semantic network.

Acknowledgements. The development of this version of MicroPsi has been supported by a research stipend of the Berlin School of Mind and Brain, and development efforts funded by the Hotheaven AG, Berlin.

References

1. Bach, J.: The MicroPsi Agent Architecture. In: Proceedings of ICCM-5, International Conference on Cognitive Modeling, Bamberg, Germany, pp. 15–20 (2003)
2. Bach, J., Vuine, R.: Designing Agents with MicroPsi Node Nets. In: Günter, A., Kruse, R., Neumann, B. (eds.) KI 2003. LNCS (LNAI), vol. 2821, pp. 164–178. Springer, Heidelberg (2003)
3. Bach, J.: MicroPsi: A cognitive modeling toolkit coming of age. In: Proc. of 7th International Conference on Cognitive Modeling, pp. 20–25 (2006)
4. Bach, J.: Motivated, Emotional Agents in the MicroPsi Framework. In: Proceedings of 8th European Conference on Cognitive Science, Delphi, Greece, pp. 458–461 (2007)
5. Bach, J.: Motivated, Emotional Agents in the MicroPsi Framework. In: Proceedings of 8th European Conference on Cognitive Science, Delphi, Greece (2007)
6. Bach, J.: Principles of Synthetic Intelligence. Psi, an architecture of motivated cognition. Oxford University Press (2009)
7. Bach, J.: A Motivational System for Cognitive AI. In: Schmidhuber, J., Thórisson, K.R., Looks, M. (eds.) AGI 2011. LNCS, vol. 6830, pp. 232–242. Springer, Heidelberg (2011)
8. Bach, J.: A Framework for Emergent Emotions, Based on Motivation and Cognitive Modulators. International Journal of Synthetic Emotions (IJSE) 3(1), 43–63 (2012a)
9. Bach, J.: Functional Modeling of Personality Properties Based on Motivational Traits. In: Proceedings of ICCM-7, International Conference on Cognitive Modeling, Berlin, Germany, pp. 271–272 (2012b)
10. Cooper, R., Fox, J.: COGENT: A visual design environment for cognitive modelling. Behavior Research Methods, Instruments and Computers 30, 553–564 (1998)
11. Dörner, D.: Bauplan für eine Seele. Rowohlt, Reinbeck (1999)

12. Dörner, D., Bartl, C., Detje, F., Gerdes, J., Halcour, D.: Die Mechanik des Seelenwagens. Handlungsregulation. Verlag Hans Huber, Bern (2002)
13. Hellkamp, L.: The Bottle Web Framework (2011), `http://bottlepy.org` (last retrieved August 2012)
14. Lehni, J., Puckey, J.: The PaperJS Vector Graphics Library (2011), `http://paperjs.org` (last retrieved August 2012)
15. Twitter Inc. Twitter Bootstrap Library (2012), `http://twitter.github.com/bootstrap` (last retrieved August 2012)