Two Case Studies for Jazzyk BSM

Michael Köster, Peter Novák, David Mainzer and Bernd Fuhrmann

Department of Informatics, Clausthal University of Technology
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
http://cig.in.tu-clausthal.de/
{mko,pno,dm,ifbf}@tu-clausthal.de
The original publication is available at www.springerlink.com
http://dx.doi.org/10.1007/978-3-642-11198-3_3

Abstract. Recently, we introduced *Behavioural State Machines* (*BSM*), a novel programming framework for development of cognitive agents with *Jazzyk*, its associated programming language and interpreter. The *Jazzyk BSM* framework draws a strict distinction between *knowledge representation* and *behavioural* aspects of an agent program. *Jazzyk BSM* thus enables synergistic exploitation of heterogeneous knowledge representation technologies in a single agent, as well as offers a transparent way for embedding cognitive agents in various simulated or physical environments. This makes it a particularly suitable platform for development of simulated, as well as physically embodied cognitive agents, such as virtual agents, or non-player characters for computer games.
In this paper we report on *Jazzbot* and *Urbibot* projects, two case-studies we developed using the *Jazzyk BSM* framework in simulated environments provided by a first person shooter computer game and a physical reality simulator for mobile robotics respectively. We describe the underlying technological infrastructure of the two agent applications and provide a brief account of experiences and lessons we learned during the development.

1 Introduction

One of the long-term aims of Artificial Intelligence is to enable development of intelligent cognitive agents. I.e. such which internally model their environment, their own mental attitudes, reason about them and subsequently base their decisions regarding their future actions upon these models. Even though AI research provides a plethora of approaches for solving partial problems on the way towards this aim, we only rarely encounter approaches enabling integration of the various developed technologies. The field of agent oriented programming, and in consequence multi-agent systems programming, offers a sound theoretical basis allowing synergistic exploitation of heterogeneous AI technologies in a single agent system.

Therefore in our recent work, we introduced the theoretical framework of *Behavioural State Machines* with its associated agent oriented programming language *Jazzyk* [**?**,**?**]. *Jazzyk BSM* provides a simple, theoretically sound language

for modular agent programming based on a generic computational model for reactive systems. It draws a strict distinction between the knowledge representation (KR) and behavioural aspects of an agent program and thus enables exploiting heterogeneous KR technologies in a single agent system.

To provide a proof-of-concept, as well as to further nurture our research towards a methodology of development with *Jazzyk BSM* (cf. [**?**] and [**?**]), we developed two case study applications *Jazzbot* and *Urbibot*. *Jazzbot* is a virtual bot in the simulated 3D environment of an open source first person shooter computer game *Nexuiz*. Its task is to explore a virtual building, search for certain objects in it and subsequently deliver them to the base. At the same time, *Jazzbot* is supposed to differentiate between other players present in the building and seek safety upon being attacked by an enemy player. When the danger disappears, it should return back to the activity interrupted by the attack.

Urbibot, on the other hand, was developed as a step towards programming mobile robots. It is an agent program steering a model of customized *e-Puck*, a small two-wheeled mobile robot in an environment provided by the physical robotic simulator *Webots*. Similarly to *Jazzbot*, *Urbibot* explores its environment in order to find red poles present in it. It tries to bump into each of them, while trying to avoid patrol robots policing the environment. Upon encounter with such a patrol robot, *Urbibot* runs away to finally return to the previously interrupted activity when safe again.

Both agents feature a BDI inspired architecture. While interacting with two different types of virtual bodies, a character in the game and an interface to robot hardware sensors and actuators respectively, both implementations exploit the power of non-monotonic reasoning for representation and reasoning about their beliefs and goals. We employ an interpreted object oriented programming language to enable efficient representation of and reasoning about topological structure of the environment.

After a brief introduction to the framework of *Behavioural State Machines* and its associated programming language *Jazzyk* in Section 2, Sections 3 and 4 describe respectively *Jazzbot* and *Urbibot* agents in a closer detail. Subsequently, Section 5 provides a description of the underlying technological infrastructure used in the implemented agents. Finally, a discussion of our experiences and lessons learned from the development of *Jazzbot* and *Urbibot* agents, together with an outlook to the ongoing and future work wraps up the paper in Section 6.

2 Jazzyk BSM

In [**?**] we introduced the framework of *Behavioural State Machines* (*BSM*). *BSM* framework draws a clear distinction between the *knowledge representation* and *behavioural* layers within an agent. It thus provides a programming system that clearly separates the programming concerns of *how to represent an agent's knowledge* about, for example, its environment and *how to encode its behaviours*. In the core of the framework is a *generic reactive computational model* inspired by Gurevich's *Abstract State Machines* [**?**], enabling for efficient structuring of

the program code. This section briefly introduces the *BSM* framework. For the complete formal description of the *BSM* framework, see [**?**].

2.1 Syntax

BSM agents are collections of one or more so-called *knowledge representation modules* (KR modules), typically denoted by $\mathcal{M}$, each representing a part of the agent's knowledge base. KR modules may be used to represent and maintain various mental attitudes of an agent, such as knowledge about its environment, or its goals, intentions, obligations, etc. Transitions between states of a *BSM* result from applying so-called *mental state transformers* (*mst*), typically denoted by τ. Various types of mst's determine the behaviour that an agent can generate. A *BSM agent* consists of a set of KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$ and a mental state transformer $\mathcal{P}$, i.e. $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$; the mst $\mathcal{P}$ is also called an *agent program*.

The notion of a KR module is an abstraction of a partial knowledge base of an agent. In turn, its states are to be treated as theories (i.e. sets of sentences) expressed in the KR language of the module. Formally, a KR module $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{L}_i, \mathcal{Q}_i, \mathcal{U}_i)$ is characterized by a knowledge representation language $\mathcal{L}_i$, a set of states $\mathcal{S}_i \subseteq 2^{\mathcal{L}_i}$, a set of query operators $\mathcal{Q}_i$ and a set of update operators $\mathcal{U}_i$. A query operator $\models \in \mathcal{Q}_i$ is a mapping $\models : \mathcal{S}_i \times \mathcal{L}_i \rightarrow \{\top, \bot\}$. Similarly an update operator $\oplus \in \mathcal{U}_i$ is a mapping $\oplus : \mathcal{S}_i \times \mathcal{L}_i \rightarrow \mathcal{S}_i$.

Queries, typically denoted by φ, can be seen as operators of type $\models : \mathcal{S}_i \rightarrow \{\top, \bot\}$. A primitive query $\varphi = (\models \phi)$ consists of a query operator $\models \in \mathcal{Q}_i$ and a formula $\phi \in \mathcal{L}_i$ of the same KR module $\mathcal{M}_i$. Complex queries can be composed by means of conjunction $\wedge$, disjunction $\vee$ and negation $\neg$.

Mental state transformers enable transitions from one state to another. A primitive mst $\oslash\psi$, typically denoted by ρ and constructed from an update operator $\oslash \in \mathcal{U}_i$ and a formula $\psi \in \mathcal{L}_i$, refers to an update on the state of the corresponding KR module. Conditional mst's are of the form $\varphi \longrightarrow \tau$, where φ is a query and τ is a mst. Such a conditional mst makes the application of τ depend on the evaluation of φ. Syntactic constructs for combining mst's are: non-deterministic choice $|$ and sequence $\circ$.

Definition 1 (mental state transformer). *Let $\mathcal{M}_1, \ldots, \mathcal{M}_n$ be KR modules of the form $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{L}_i, \mathcal{Q}_i, \mathcal{U}_i)$. The set of* mental state transformers *is defined as below:*

- **skip** *is a* primitive *mst,*
- *if $\oslash \in \mathcal{U}_i$ and $\psi \in \mathcal{L}_i$, then $\oslash\psi$ is a* primitive *mst,*
- *if φ is a query, and τ is a mst, then $\varphi \longrightarrow \tau$ is a* conditional *mst,*
- *if τ and τ' are mst's, then $\tau|\tau'$ and $\tau \circ \tau'$ are mst's (*choice*, and* sequence *respectively).*

Even though it is a vital feature of the *BSM* theoretical framework, for simplicity we omit the treatment of variables in the definitions of query and update formulae above. For a full fledged description of the BSM framework consult [**?**].

2.2 Semantics

The *yields* calculus, summarised below after [**?**], specifies an update associated with executing a mental state transformer in a single step of the language interpreter. It formally defines the meaning of the state transformation induced by executing an mst in a state, i.e. a mental state transition.

Formally, a *mental state* σ of a *BSM* $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ is a tuple $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ of KR module states $\sigma_1 \in \mathcal{S}_1, \ldots, \sigma_n \in \mathcal{S}_n$, corresponding to $\mathcal{M}_1, \ldots, \mathcal{M}_n$ respectively. $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$ denotes the space of all mental states over $\mathcal{A}$. A mental state can be modified by applying primitive mst's on it and query formulae can be evaluated against it. The semantic notion of truth of a query is defined through the satisfaction relation $\models$. A primitive query $\mathbf{\vDash}\phi$ holds in a mental state $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ (written $\sigma \models (\mathbf{\vDash}\phi)$) iff $\mathbf{\vDash}(\phi, \sigma_i)$, otherwise we have $\sigma \not\models (\mathbf{\vDash}\phi)$. Given the usual meaning of Boolean operators, it is straightforward to extend the query evaluation to compound query formulae. Note that evaluation of a query does not change the mental state σ.

For an mst $\oslash\psi$, we use $(\oslash, \psi)$ to denote its semantic counterpart, i.e., the corresponding *update* (state transformation). Sequential application of updates is denoted by $\bullet$, i.e. $\rho_1 \bullet \rho_2$ is an update resulting from applying ρ_1 first and then applying ρ_2. The application of an update to a mental state is defined formally below.

Definition 2 (applying an update). *The result of applying an update* $\rho = (\oslash, \psi)$ *to a state* $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ *of a* BSM $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$*, denoted by* $s \bigoplus \rho$*, is a new state* $\sigma' = \langle \sigma_1, \ldots, \sigma'_i, \ldots, \sigma_n \rangle$*, where* $\sigma'_i = \sigma_i \oslash \psi$ *and* σ_i*,* $\oslash$*, and* ψ *correspond to one and the same* $\mathcal{M}_i$ *of* $\mathcal{A}$*. Applying the empty update* **skip** *on the state* σ *does not change the state, i.e.* $\sigma \bigoplus \mathbf{skip} = \sigma$*.*

Inductively, the result of applying a sequence of updates $\rho_1 \bullet \rho_2$ *is a new state* $\sigma'' = \sigma' \bigoplus \rho_2$*, where* $\sigma' = \sigma \bigoplus \rho_1$*.* $\sigma \overset{\rho_1 \bullet \rho_2}{\rightarrow} \sigma'' = \sigma \overset{\rho_1}{\rightarrow} \sigma' \overset{\rho_2}{\rightarrow} \sigma''$ *denotes the corresponding compound transition.*

The meaning of a mental state transformer in state σ, formally defined by the *yields* predicate below, is the update set it yields in that mental state.

Definition 3 (yields calculus). *A mental state transformer* τ *yields an* update ρ *in a state* σ*, iff* $yields(\tau, \sigma, \rho)$ *is derivable in the following calculus:*

$$\frac{\top}{yields(\mathbf{skip}, \sigma, \mathbf{skip})} \quad \frac{\top}{yields(\oslash\psi, \sigma, (\oslash, \psi))} \quad \textit{(primitive)}$$

$$\frac{yields(\tau, \sigma, \rho), \sigma \models \phi}{yields(\phi \longrightarrow \tau, \sigma, \rho)} \quad \frac{yields(\tau, \sigma, \rho), \sigma \not\models \phi}{yields(\phi \longrightarrow \tau, \sigma, \mathbf{skip})} \quad \textit{(conditional)}$$

$$\frac{yields(\tau_1, \sigma, \rho_1),\ yields(\tau_2, \sigma, \rho_2)}{yields(\tau_1 | \tau_2, \sigma, \rho_1),\ yields(\tau_1 | \tau_2, \sigma, \rho_2)} \quad \textit{(choice)}$$

$$\frac{yields(\tau_1, \sigma, \rho_1),\ yields(\tau_2, \sigma \bigoplus \rho_1, \rho_2)}{yields(\tau_1 \circ \tau_2, \sigma, \rho_1 \bullet \rho_2)} \quad \textit{(sequence)}$$

We say that τ *yields an* update set ν *in a state* σ *iff* $\nu = \{\rho | yields(\tau, \sigma, \rho)\}$*.*

The mst **skip** yields the update **skip**. Similarly, a primitive update mst $\oslash\psi$ yields the corresponding update $(\oslash, \psi)$. In the case the condition ϕ of a conditional mst $\phi \longrightarrow \tau$ is satisfied in the current mental state, the calculus yields one of the updates corresponding to the right hand side mst τ, otherwise the no-operation **skip** update is yielded. A non-deterministic choice mst yields an update corresponding to either of its members and finally a sequential mst yields a sequence of updates corresponding to the first mst of the sequence and an update yielded by the second member of the sequence in a state resulting from application of the first update to the current mental state.

The following definition articulates the denotational semantics of the notion of mental state transformer as an encoding of a function mapping mental states of a *BSM* to updates, i.e. transitions between them.

Definition 4 (mst functional semantics). *Let $\mathcal{M}_1, \ldots, \mathcal{M}_n$ be KR modules. A mental state transformer τ encodes a function $\mathfrak{f}_\tau : \sigma \mapsto \{\rho | yields(\tau, \sigma, \rho)\}$ over the space of mental states $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle \in S_1 \times \cdots S_n$.*

Subsequently, the semantics of a *BSM* agent is defined as a set of traces in the induced transition system enabled by the *BSM* agent program.

Definition 5 (BSM semantics). *A* BSM $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ *can make a step from state σ to a state σ', iff $\sigma' = \sigma \bigoplus \rho$, s.t. $\rho \in \mathfrak{f}_\mathcal{P}(\sigma)$. We also say, that $\mathcal{A}$ induces a (possibly compound) transition $\sigma \xrightarrow{\rho} \sigma'$.*

A possibly infinite sequence of states $\sigma_1, \ldots, \sigma_i, \ldots$ is a run of BSM $\mathcal{A}$, iff for each $i \geq 1$, $\mathcal{A}$ induces a transition $\sigma_i \rightarrow \sigma_{i+1}$.

The semantics of an agent system characterized by a BSM $\mathcal{A}$, is a set of all runs of $\mathcal{A}$.

Additionally, we require the non-deterministic choice of a *BSM* interpreter to fulfil the *weak fairness condition*, similar to that in [**?**], for all the induced runs.

Condition 1 (weak fairness condition) *A computation run is weakly fair iff it is not the case that an update is always yielded from some point in time on but is never selected for execution.*

2.3 Jazzyk

Jazzyk is an interpreter of the *Jazzyk* programming language implementing the computational model of the *BSM* framework. The syntax of the *Jazzyk* language is an instantiation of the abstract mathematical syntax of the *BSM* theoretical framework. when ϕ then τ construct encodes a conditional *mst* $\phi \longrightarrow \tau$. Symbols ; and , stand for choice | and sequence $\circ$ operators respectively. To facilitate operator precedence, mental state transformers can be grouped into compound structures, blocks, using curly braces {...}.

To better support source code modularity and re-usability, *Jazzyk* interpreter integrates GNU M4[1], a state-of-the-art macro preprocessor. Macros are a powerful tool for structuring and modularizing and encapsulating the source code and

[1] `http://www.gnu.org/software/m4/`

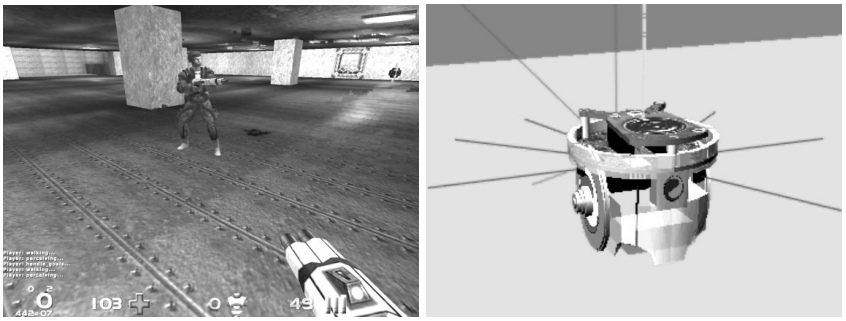

Fig. 1. Screenshots of the *Jazzbot* and *Urbibot* agents.

writing code templates. Before feeding the *Jazzyk* agent program to the language interpreter, first all the macros are expanded. Listing 1 depicts a *Jazzyk* code snippet. For further details on the *Jazzyk* programming language and the macro preprocessor integration with *Jazzyk* interpreter, consult [**?**].

3 Jazzbot

Jazzbot is a virtual agent embodied in a simulated 3D environment of the first-person shooter computer game *Nexuiz*[2]. It is a goal-driven BDI inspired cognitive agent developed with the *Jazzyk* language. The *Nexuiz* death-match game takes place in a virtual building containing various objects (e.g. weapons, flags or armor kits), is capable of simulating diverse terrains like solid floor, or liquid and provides a basic means for inter-player interaction. Because of its accessibility (*Nexuiz* is published under the open source GNU GPL licence), we chose the *Nexuiz* game server as the simulator for *Jazzbot* case-study, the first larger proof-of-concept application for the *Jazzyk BSM* framework. Figure 1 left depicts a screenshot of the *Jazzbot* agent acting in the simulated environment. Demonstration videos and source code can be found on the project website[3].

Jazzbot's behaviour is implemented as a *Jazzyk* program. In the experimental scenario, the bot searches for a particular item in the environment, which it then picks up and delivers to the base point. While during the search phase the agent tries to always move to unexplored segments of the environment, when it tries to deliver the item, it exploits a path planning algorithm to compute the shortest path to the base point. Hence, during the search phase, in every step the bot randomly selects a direction to move to a previously unexplored part of the building and in the case there is none such, it returns to the nearest waypoint from which an unexplored direction exists. The behaviour for environment exploration is interrupted, whenever *Jazzbot* feels under attack, i.e. an enemy

[2] `http://www.alientrap.org/nexuiz/`
[3] `http://jazzyk.sourceforge.net/`

Listing 1 Code snippet from the *Jazzbot* agent.

```
define('ACT','{
  /* The bot searches for an item, only when it does not have it */
  when ⊨goals [{ task(search(X)) }] and not ⊨beliefs [{ hold(X) }] then SEARCH('X');
  /* When a searched item is found, it picks it */
  when ⊨goals [{ task(pick(X)) }] and ⊨beliefs [{ see(X) }] then PICK('X') ;
  /* When the bot finally holds the item, it deliver it */
  when ⊨goals [{ task(deliver(X)) }] and ⊨beliefs [{ hold(X) }] then DELIVER('X') ;
  /* Simple behaviour triggers without guard conditions */
  when ⊨goals [{ task(wander) }] then WALK ;
  when ⊨goals [{ task(safety) }] then RUN_AWAY ;
  when ⊨goals [{ task(communicate) }] then SOCIALIZE
}')
```

player attempts to shoot at it. Upon that it triggers emergency behaviours, such as running away from the danger. After the sense of emergency fades away, it returns back to its previously performed goals of item search, or delivery.

The *Jazzbot*'s control cycle consists of three steps that are executed sequentially. Firstly, the bot reads its sensors (perception), then if necessary it deliberates about its goals, (goal commitment strategies implementation) and finally it selects an action according to its actual goals and beliefs (act). Listing 1 provides an example code implementing selection of goal oriented behaviours, realized as parametrized macros, triggered by *Jazzbot*'s goals. While the bot simply triggers behaviours for walking around, danger aversion and social behaviour, execution of behaviours finally leading to getting an item are guarded by belief conditions.

The Figure 2 provides an overview of the *Jazzbot*'s architecture. The agent features a belief base consisting of two KR modules for representation of agent's actual beliefs and storing the map of the environment, a goal base encoding interrelationships between various agent's declarative, performance and maintenance goals and finally the module interfacing the bot with the simulated environment.

JzNexuiz KR module (cf. Subsection 5.4), the *Jazzbot*'s interface to the environment, the body, provides the bot with capabilities for sensing and acting in the virtual world. The bot can move forward, backward, it can turn, or shoot. Additionally, the *Jazzbot* is equiped with several sensors: GPS, sonar, 3D compass and an object recognition sensor. The module communicates over the network with the *Nexuiz* game server and thus provides an interface of a pure client side *Nexuiz* bot, i.e. the bot can access only a subset of the perceptual information a human player would have available.

The *Jazzbot*'s *belief base* is composed of two modules: *JzASP* (cf. Subsection 5.1) and *JzRuby* (cf. Subsection 5.2). While the first one integrates an *Answer Set Programming* [**?**] (*ASP*) solver *Smodels* [**?**] and contains a logic program reflecting agent's beliefs about itself, the environment, objects in it and other players, the second, based on an interpreted object oriented programming language *Ruby*, stores the map of the agent's environment.

The *Jazzbot*'s *goal base* is again an *ASP* logic program representing agent's current goals and their interdependencies. Goals can be either of a declarative (*goals-to-be*), or performative nature (*goals-to-do*, or *tasks*). In *Jazzbot* agent

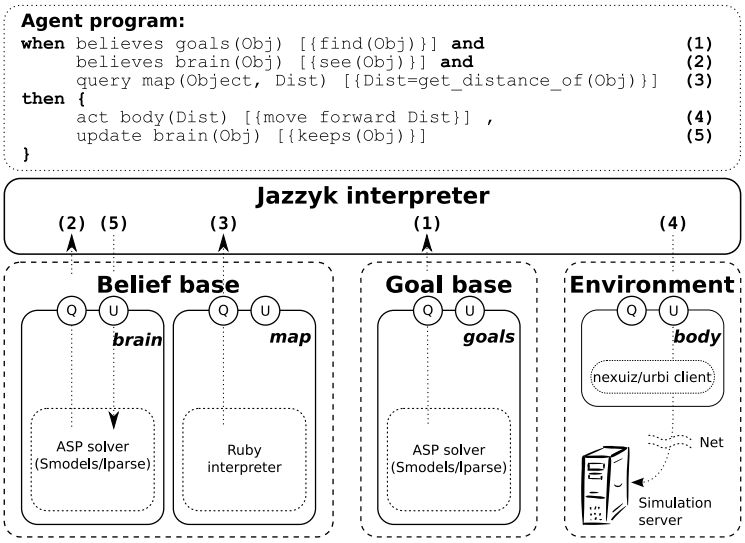

Fig. 2. Internal architecture of *Jazzbot* and *Urbibot* agents.

implementation, each *goal-to-do* activates one, or more *tasks*, which in turn trigger, one or more corresponding behaviours the agent is supposed to execute. On the ground of holding certain beliefs, the agent is also allowed to adopt new, or drop goals which are either satisfied, irrelevant, or subjectively recognized as impossible to achieve. The agent thus implements goal *commitment strategies*. We explore the details of the programming methodology employed in the *Jazzbot* project in [**?**]. The Section 5 below provides details on the implementation of the *Jazzbot*'s KR modules.

4 Urbibot

Urbibot [**?**] is the second case-study developed as a step towards applications of the *Jazzyk BSM* framework in the mobile robotics domain. It is a robot exploring a maze where it searches for red poles and then tries to kick them down while at the same time avoiding patrols policing the space. *Urbibot* is embodied as an *e-Puck*[4], a small educational mobile robot simulated in *Webots*[5] [**?**], a robotics oriented physical world simulator. The robot is steered using *URBI*[6], a highly flexible and modular robotic programming platform based on event-based programming model. The main motivation for using *URBI* is the direct transferability of the developed agent program from simulator to the real robot.

Similarly to the *Jazzbot*, the overall agent design is inspired by the BDI architecture and reuses parts of the code developed for *Jazzbot*. In turn, except

[4] `http://www.e-puck.org/`
[5] `http://www.cyberbotics.com/`
[6] `http://www.gostai.com/`

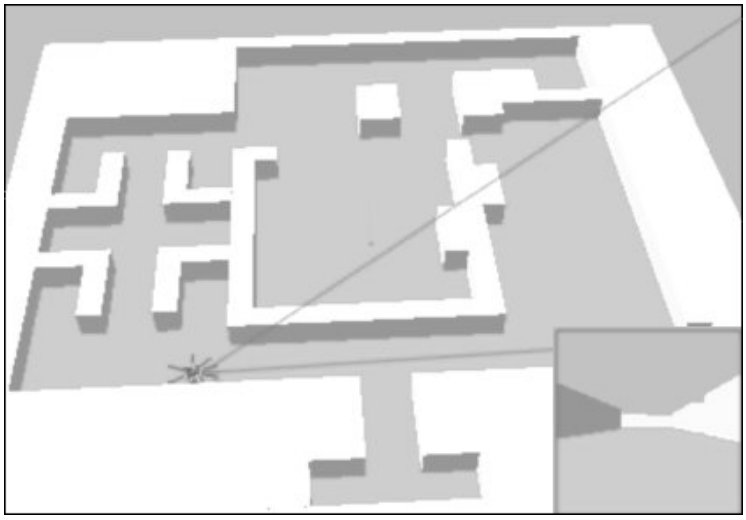

Fig. 3. *Urbibot* exploring the simulated environment. The lower right corner provides the current snapshot of the *Urbibot*'s camera perception.

for using *JzASP* KR module to represent agent's beliefs about itself, *Urbibot* features similar agent architecture as the one depicted in the Figure 2 for the *Jazzbot* agent. *Urbibot*'s beliefs comprise exclusively information about the map. The interface with the simulator environment is provided by the *JzUrbi* KR module (see Subsection 5.3).

As already noted above, *Urbibot*'s behaviour is similar to that of *Jazzbot* agent. However instead of controlling the agent's body with rather discrete commands, such as `move forward`, or `turn left`, *Urbibot*'s *URBI* allows a more sophisticated control by directly accessing the robot's actuators, which in the case of the *e-Puck* robot, are only it's two wheels. The robot features a mounted camera, a directional distance sensor and an additional GPS sensor (our customization of the original *e-Puck* robot). In the *JzRuby* module, the robot analyzes the camera image stream and by joining it with the output of the distance and GPS sensors it constructs a 2D map of the environment. Upon encountering a patrol robot, *Urbibot* calculates an approximation of the space the patrol robot can see, and subsequently tries to navigate out of this area as quickly as possible. Again, the details on the implementation of the *Urbibot*'s KR modules can be found later in the Section 5. Figures 1 (right) and 3 depicts a screenshot of the *Urbibot* agent and the maze environment with the *Urbibot* acting in it. Furthermore, demonstration videos and source code are provided in the corresponding section of the *Jazzyk* project website[3].

5 Modules

The *Jazzyk Software Development Kit (Jazzyk SDK)* provides a C++ interface from which each KR module plug-in has to be derived. Basically, the class defines five methods: `initialize`, `finalize`, `cycle`, `query` and `update`. It is possible to define multiple `query` and `update` methods corresponding to KR module's query and update operators. These methods then define the plug-in's interface to the *Jazzyk* interpreter. While initialize, finalize and cycle are mainly used for initialization, shutdown and maintenance of the module the query and update provide means for modification of the stored knowledge base.

Below we describe KR module's we employed in development of the *Jazzbot* and *Urbibot* case studies. We introduce two modules facilitating agent's knowledge representation *JzASP* and *JzRuby* followed by description of two modules, *JzUrbi* and *JzNexuiz*, interfacing the agents with their respective environments.

5.1 JzASP

In [**?**] we presented the *JzASP* module. It integrates *Lparse* [**?**] and *Smodels* [**?**], an *Answer Set Programming* grounder and solver respectively. The stored knowledge base thus consists of a logic program in the syntax of *AnsProlog** [**?**] (*Prolog* style syntax) and can be accessed by two query methods sure_believes and poss_believes and two update methods add and del allowing for retrieval and modification of the stored knowledge base. Internally, the *JzASP* module processes the program by passing it to the *Lparse* library and subsequently let's *Smodels* solver to compute the program's answer sets.

The two query methods sure_believes and poss_believes check whether the query formula, an *AnsProlog** term, is contained in all the computed answer sets, or there exists at least a single answer set containing it respectively. Before the query formula is processed by the module, all the free variables occurring in it are substituted by their valuations and subsequently, the query method attempts matching the remaining free variables with a term from a computed answer set.

The update interface methods add and del provide a means to assert, or retract a clause (a fact, or a rule) to/from the stored knowledge base. The variable substitution treatment is similar to that in processing query formulae.

While the *Jazzbot* agent employs the *JzASP* for both, reasoning about its beliefs regarding its environment, other agents and its own body state, as well as to represent and reason about its goals, the *Urbibot* agent employs the module only to treat its goal base. Using the power of non-monotonic reasoning, in particular the default negation, to reason about agent's goals turned out to be advantageous and led to an elegant encoding of interrelations between various goals. We elaborate more on the technique used in [**?**].

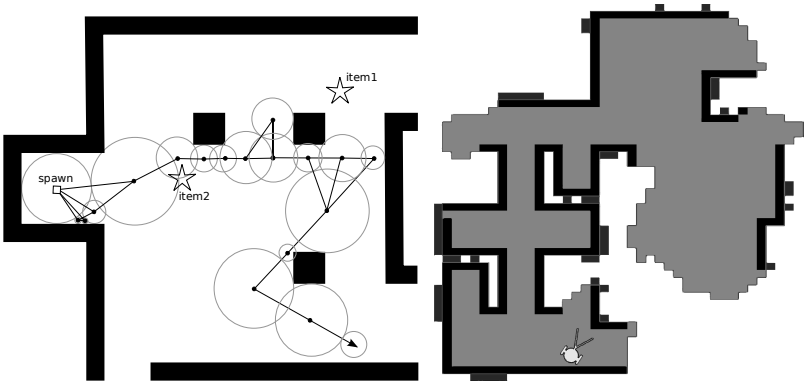

Fig. 4. Environment maps representation in *Jazzbot* and *Urbibot*.

5.2 JzRuby

The *JzRuby* module, detailed description in [**?**], integrates the interpreted object oriented scripting language *Ruby*[7]. The KR module interface methods for initialization and finalization as well as the query and update routines are able to process plain *Ruby* programs as argument formulae (query/update). *Jazzyk* variables are treated as global variables in the *Ruby* interpreter's memory space. Query invocations of the single query method return the truth value of the code invocation within the *Ruby* interpreter, i.e. provided the code execution yields a value other than 0, the KR module returns $\top$ and $\bot$ otherwise. The single update method of the KR module simply executes the provided update formula, a plain *Ruby* code chunk.

To represent the *Jazzbot*'s information about the topology of its environment, the agent uses a *circle-based waypoint graph* (*CWG*) [**?**] to generate the map of its environment. *CWG*'s are an improved version of *waypoint graphs*, extended with a radius for each waypoint. The radius is determined by the distance between the avatar and the nearest obstacle. This technique ensures, especially in big rooms or open spaces, a smaller number of nodes and connections within the graph what in turn speeds up the path search algorithm. Figure 4 (left) shows a graphical representation of the *CWG* for a sample walk of the *Jazzbot* agent from the spawn point to the point marked by the arrow.

Additionally, each waypoint stores a list of objects present within its range as well as about walls touching it and information about unexplored directions, i.e. such in which there's no connection to another waypoint, nor a wall. By employing a breadth-first graph search algorithm, the agent can compute the shortest path to a particular object, or a position.

The *CWG* graph is constructed by the agent so that in each step it determines whether its current absolute position corresponds to some known waypoint, and

[7] `http://www.ruby-lang.org/`

if not, it turns around in 60° steps and by checking its distance sensor, it determines the nearest obstacle around. Subsequently, the newly added waypoint is incorporated into the *CWG* by connecting it to all the other waypoints with which it overlaps and all the perceived objects together with all the directions in which the agent can see a wall are stored with it.

While similarly to the *Jazzbot*, the *Urbibot* uses the *JzRuby* KR module for representing its environment too, the map representation approach differs. *Urbibot* represents the map of its environment as a 2D grid, where in each cell it stores an information about its content, such as `unknown`, `free`, `wall`, `patrol`, `avoid` or `pole`. A new row or column is added to the grid when the robot reaches the edge of the known region. While the agent continuously adds new information to cells, the map becomes more and more precise. Furthermore, the *Urbibot* employs the A^* path planning algorithm to compute the shortest path between the current position and a position to go, be it an unexplored cell, the nearest safe cell a patrol robot cannot see, or a cell containing a pole which it tries to kick down. Also, the *Urbibot* uses the *JzRuby* module to algorithmically process the sensory input from its mounted camera and the distance sensors.

5.3 JzUrbi

In order to interface a *Jazzyk* program with the robot's body, the *JzUrbi* KR module [**?**] integrates the *URBI* programming language interpreter. It connects over TCP/IP to an *URBI* server on the simulator's side, or with the *URBI* robot controller that controls the robot's body.

The single query method `query` provide the agent program with the sensor information from the body. *Jazzyk* variables are treated the same way as in *JzRuby*, i.e. as global variables of the underlying *URBI* interpreter. Similarly, the single update method `update` simply sends the provided update formula, an *URBI* program, to the *URBI* server.

The *Urbibot* connects over the *JzUrbi* module with the *URBI* server running in the *Webots* simulator and thereby steers the *e-Puck* robot, extended with a GPS sensor. In the particular case of the *Urbibot*, the sensory input accesses the following sensors: camera, distance sensor, GPS, touch-sensor and a light-sensor. Together with the update interface steering the *Urbibot*'s two wheels, these two methods provide the basic interface for *e-Puck*'s control.

5.4 JzNexuiz

Finally, the *JzNexuiz* KR module, invented in [**?**], facilitates *Jazzbot*'s interaction with the *Nexuiz* game environment. By the means of a single query interface method `sense` and a single update method `act`, it enables control of the *Jazzbot*'s avatar body in the virtual building. The query method provides access to several sensors of GPS, sonar, 3D compass and object recognition. The update method allows issuing commands to the avatar's body, such as move, turn, jump, use, attack or say.

```
nex_query  ::= sensor (constant | variable)+
nex_update ::= action (constant | variable)+
sensor     ::= sen_const | variable
sen_const  ::= 'body' | 'liquid' | 'ground' | 'gps' | 'compass' |
               'sonar' | 'map' | 'eye' | 'listen'
action     ::= act_const | variable
act_const  ::= 'move' | 'turn' | 'jump' | 'use' | 'attack' | 'say'
```

Fig. 5. *JzNexuiz* EBNF.

Technically, the module connects over TCP/IP with a *Nexuiz* server and thus provides an interface of a pure client side *Nexuiz* bot. The consequence of this setup is that the *Jazzbot* agent can access only a strict subset of the perceptual information a human player would have. The *Jazzbot* plug-in integrates a stripped down and customized *Nexuiz* client. In turn, the bot's actions are implemented as the corresponding key strokes of a virtual player.

Figure 5 depicts the syntax accepted by the plug-in's query and update interface methods sense and act. Each query formula starts with the name of the accessed virtual sensor device followed by the corresponding arguments being either constants, or variables facilitating retrieval of information from the environment. Similarly, the update formulas consist of the action to be executed by the avatar followed by a list of arguments specifying the command parameters.

The truth values of query formula evaluation depends on the sensory input retrieved from the environment. In the case the query evaluates to true ($\top$), the additional information about e.g. the distance of an obstacle, or the reading of the body health sensor, is stored in provided free variables.

6 Experiences and Conclusion

The two case-studies described in this paper served us most importantly as a vehicle to nurture and pragmatically drive our research towards a methodology for using an agent oriented programming language exploiting strengths of heterogeneous KR technologies in a single cognitive agent system. For further details concerning the methodology consult [**?**]. As an important side effect, we collected experiences with programming BDI inspired virtual cognitive agents for computer games and simulated environments, as well.

As in the long run we aim at development of autonomous robots, in both cases the virtual agents had to be running autonomously and independently from the simulator of the environment. This choice had a strong impact on the design of the agents w.r.t. the action execution model and the model of perception. In both described applications, the agents are remotely connecting to the simulated environment in which they execute actions in an asynchronous manner, i.e. they can only indirectly observe the effects (success/failure) of their actions through later perceptions. As far as the model of perception is concerned, unlike other

game bots, *Jazzbot* is a pure client side bot, i.e. the amount of information it can perceive is a strict subset of the information provided to the game client used by human players. Hence, the *Jazzbot* agent cannot take advantage of additional information, such as the global topology of the environment, or information about objects in distant parts of the environment, which are accessible to the majority of other bots available for first-person shooter games. In the case of *Urbibot*, the simulator provides only perceptions accessible to the models of robot's sensors. In our case these are most importantly a camera, a directional distance sensor and global positioning, hence the available information is, similarly to *Jazzbot*, only local, incomplete and noisy.

As both implemented agents are running independently from the simulation engine and execute their actions in an asynchronous manner, their efficiency is only loosely coupled to the simulation platform speed. In our experiments, the speed of agent's reactions was reasonable w.r.t. task the bots were supposed to execute. However, especially in the case of *Jazzbot*, due to deficiencies on the side of sensors, such as missing camera rendering the complete scene the bot can see, *Jazzbot* in its present incarnation cannot match the reaction speed of advanced human players in a peer-2-peer match.

Since, the agents store their internal state in the application domain specific KR modules, the control model of *Jazzyk BSM* framework results in agents which can instantly change the focus of their attention w.r.t. an observed change of the context in the environment. The goal orientedness of agent's behaviours emerges from the coupling between behaviour triggers and agent's attitudes modeled in its components [**?**]. This turned out to be of a particular advantage when a quick reaction to interruptions, such as an encounter of an enemy agent, or a patrol, was needed. On the other hand, because of the open plug-in architecture of the *Jazzyk BSM* framework, we were able to quickly prototype and experiment with various approaches to knowledge representation and reasoning, as well as various models of interaction with the environment.

Our research project follows the spirit of [**?**], where Laird and van Lent argue that approaches for programming intelligent agents should be tested in realistic and sophisticated environments of modern computer games. *Jazzbot* project thus follows in footsteps of their *SOAR QuakeBot* [**?**].

Another relevant project, *Gamebots* [**?**], provides a general purpose interface to a first-person shooter game *Unreal Tournament*[8]. *Gamebots*' approach is however server side, i.e. the virtual agent is provided with much more information than a human player has, what was not in the spirit of our aim to emulate mobile robots in a virtual environment. Why we did not pick the *Gamebots* framework for our project was also the fact, that it is specific to the commercially available game *Unreal Tournament* and since 2002, the project does not seem to be further maintained.

To our knowledge, our work on the *Jazzbot* and *Urbibot* case-studies is novel in the sense that it seems to be the first efficient application of non-monotonic reasoning framework of *ASP* in a highly dynamic domain of simulated robotics,

[8] `http://www.unrealtournament3.com/`

or a first-person shooter computer game. Even though, to our knowledge the first attempt by Provetti et al. [**?**] uses *ASP* for planning and action selection in the context of the *Quake 3 Arena*[9] game, authors note that their bot could not recalculate its plans rapidly enough since each computation required up to 7 seconds in a standard setup [**?**]. Thus, in comparison to *Jazzbot* or *Urbibot*, their agent was capable to react to events occurring in the environment only to a lesser extend, because both their action selection and planning was in *ASP*.

Similarly, to our knowledge, the transparent integration of various KR technologies such as a declarative, logic based technology for representing agent's beliefs and goals, an object oriented language for storing the topological information about the environment together with a generic reactive control model of the agent program in the *Jazzyk BSM* framework is unique. In consequence, the *Jazzyk BSM* framework shows a lot of potential for further experimentation with synergies of exploiting various AI technologies in cognitive agent systems, especially in the attractive domain of virtual agents and autonomous non-player characters, for computer games. Yet, we believe, more experimentation is needed to explore the limits and deficiencies of our approach in the domain of simulated, as well as physical reality embodied robotics.

[9] `http://www.idsoftware.com/`