Designing Goal-Oriented Reactive Behaviours

Peter Novák and **Michael Köster**[1]

Abstract. State-of-the-art rule-based agent programming languages, similar to *AgentSpeak(L)*, provide theoretically solid tools for implementing cognitive agents. However, not too much attention was devoted to low-level design issues for development of non-trivial agents in them.

In this paper we discuss some design considerations we faced while implementing *Jazzbot*, a softbot embodied in a simulated 3D environment, implemented in a rule-based framework of *Behavioural State Machines*. Finally, we also make an attempt to lift our experiences to a set of informal *design guidelines* useful for design and implementation of agents with heterogeneous knowledge bases in rule-based agent oriented programming languages.

1 INTRODUCTION

While the landscape of programming languages for cognitive agents is thriving (for state-of-the-art surveys see e.g. [5], or [6]), a little has been published on their application beyond small-scale example agents. In our research project we are interested in implementing embodied non-trivial agents exploiting power of heterogeneous knowledge representation (KR) techniques. To this end we recently proposed the framework of *Behavioural State Machines* (*BSM*) [16] with *Jazzyk* [17], an associated programming language. The *BSM* framework takes a rather liberal software engineering stance to programming cognitive agents. It provides a concise and flexible theoretical computational model allowing integration of heterogeneous knowledge bases (KB) into an agent system, while not prescribing a fixed scheme of interactions between them.

Most of the state-of-the-art logic based programming languages for cognitive agents, such as [7, 9, 10], tackle the problem of programming cognitive agents by introducing first class concepts with an underlying semantics of a chosen set of relations between mental attitudes of an agent. Unlike these, our approach is radically different, more based on a liberal software engineering stance. Instead of choosing a set of supported relationships among agent's knowledge bases storing its mental attitudes, we allow an agent to have an arbitrary number of KBs and provide a *modular and flexible generic programming language* facilitating interactions between them. It is the task of the programmer to maintain a discipline in applying various types of interactions between KBs specific to cognitive agents, such as goal adoption/dropping, triggering reactive behaviours etc., in agent programs. To support this, we try to propose a set of higher level syntactical constructs, *agent programming design patterns*, which can differ between various application domains. These should provide at least a partial, semi-formal semantics so that a programmer can rely on their specification.

[1] Department of Informatics, Clausthal University of Technology, Germany, {peter.novak,michael.koester}@tu-clausthal.de

The liberal nature of the *BSM* framework allows us to experiment with integration of various KR technologies in a single agent system. In [17], we introduce the *Jazzbot* project: our specific aim is to develop a BDI inspired softbot roaming in a simulated 3D environment of a first-person shooter computer game. *Jazzbot* uses non-monotonic reasoning, in particular *Answer Set Programming* [3], as the core KR technology for representing its beliefs about its environment, itself and peer bots.

In this paper, we discuss our considerations and experiences on the way towards proposing a consistent set of agent programming design patterns for development of BDI inspired cognitive agents. Our approach is application driven, i.e. we try to propose such constructs on the ground of real implementation experience with non-trivial applications. As a side effect, during the development of *Jazzbot* demo application, we have got an opportunity to rethink design of BDI inspired agents in rule-based programming languages. On the background of the *Jazzbot* project we present here a collection of design considerations (Section 3) we faced, together with an attempt to lift our solutions and implementation techniques to a set of more general methodological design guidelines (Section 4). However, before coming to the main discussion of the paper, we first briefly introduce the framework of *Behavioural State Machines* (2). A brief summary in Section 5 concludes the paper.

2 BEHAVIOURAL STATE MACHINES

In [16] we recently introduced the programming framework of *Behavioural State Machines* (*BSM*), with an associated implemented programming language *Jazzyk* [17]. The *BSM* framework defines a new and unique agent-oriented programming language due to the clear distinction it makes between the *knowledge representation* and *behavioural* layers within an agent. It thus provides a programming framework 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 following we briefly introduce the framework of *Behavioural State Machines*. A more rigorous description can be found in the original publications [16, 17].

Mental states of *BSM* agents are collections of one or more so-called *knowledge representation modules*, typically denoted by $\mathcal{M}$, each of which represents a part of the agent's knowledge base. Transitions between such states result from applying *mental state transformers* (*mst*), typically denoted by τ. The various types of *mst* determine the behaviour that an agent can generate. A *BSM agent* $\mathcal{A}$ consists of a set of KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$ and a mental state transformer τ, i.e. $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$. The mst τ is also called an *agent program*.

A KR module of a *BSM* agent can be seen as a database of statements drawn from a specific KR language. KR modules may be used to represent and maintain various attitudes of an agent such as its

knowledge about its environment, or its goals, intentions, obligations, etc. Unlike other agent oriented languages, the *BSM* framework abstracts from a particular purpose a KR module can be made to serve. Agents can have any number of such KR modules and an agent designer can ascribe any appropriate purpose to these modules (such as a belief, or a goal base). Formally, a KR module $\mathcal{M} = \langle \mathcal{S}, \mathcal{L}, \mathcal{Q}, \mathcal{U} \rangle$ is characterized by a set of states $\mathcal{S}$ the module can be in, a KR language $\mathcal{L}$ and two sets of query and update operators denoted $\mathcal{Q}$ and $\mathcal{U}$ respectively. A query operator $\models \in \mathcal{Q}$ is a function evaluating truth value of a query formula from the KR language against the current state of the KR module, i.e. $\models: \mathcal{S} \times \mathcal{L} \rightarrow \{\top, \bot\}$. An update operator $\oslash \in \mathcal{U}$ is a mapping $\oslash : \mathcal{S} \times \mathcal{L} \rightarrow \mathcal{S}$ facilitating transitions between agent's mental states induced by applying update formulae from the KR language: updating the current mental state σ by an update operator $\oslash$ and a formula ϕ results in a new state $\sigma' = \sigma \oslash \phi$. In a *BSM* agent $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ we additionally require that the KR languages (and consequently the set of query and update operators) of any two modules are disjoint, i.e. $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset$.

Syntax A primitive query $\phi = (\models \varphi)$ consists of a query operator $\models \in \mathcal{Q}$ and a formula $\varphi \in \mathcal{L}$ of the same KR module. Arbitrary queries can be composed by means of conjunction $\wedge$, disjunction $\vee$ and negation $\neg$.

Mental state transformers, syntactical counterparts to KR module updates, enable transitions from one state to another. A primitive *mst* $\rho = \oslash\phi$, constructed from an update operator $\oslash \in \mathcal{U}$ and a formula $\phi \in \mathcal{L}$, is an update on the state of the corresponding KR module of a mental state. Conditional mst's are of the form $\phi \longrightarrow \tau$, where ϕ is a query and τ is a mst. Such a conditional mst allows to make the application of mst τ conditional on the evaluation of query ϕ. Mst's can be combined by means of the choice $|$ and the sequence $\circ$ syntactic constructs.

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

1. $\mathbf{skip}$ *is a* primitive *mst,*
2. *if $\oslash \in \mathcal{U}_i$ and $\psi \in \mathcal{L}_i$, then $\oslash\psi$ is a* primitive *mst,*
3. *if ϕ is a query, and τ is a mst, then $\phi \longrightarrow \tau$ is a* conditional *mst,*
4. *if τ and τ' are mst's, then $\tau|\tau'$ is an mst (*choice*) and $\tau \circ \tau'$ is an mst (*sequence*).*

Semantics The *BSM* semantics is defined using a semantic calculus similar to that used for Gurevich's *Abstract State Machines* [8]. This formalism provides a *functional*, rather than an operational, view on mental state transformers. The *yields* calculus, introduced below, specifies an update associated with executing an mst. It formally defines the meaning of the state transformation induced by executing an mst in a mental state.

Formally, a mental state σ of a *BSM* agent $(\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ consists of the corresponding states $\langle \sigma_1, \ldots, \sigma_n \rangle$ of its KR modules. To specify the semantics of a *BSM* agent, first we need to define how queries are evaluated and how a state is modified by applying updates to it. A primitive query $\models_i \varphi$ in a state $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ evaluates the formula $\phi \in \mathcal{L}_i$ using the query operator $\models_i \in \mathcal{Q}_i$ in the current state σ_i of the corresponding KR module $\langle \mathcal{S}_i, \mathcal{L}_i, \mathcal{Q}_i, \mathcal{U}_i \rangle$. That is, $\sigma \models (\models_i \varphi)$ holds in a mental state σ iff $\sigma_i \models \varphi$, otherwise we have $\sigma \not\models (\models_i \varphi)$. Given the usual meaning of Boolean operators, it is straightforward to extend the query evaluation to compound query formulae. Note that a query $\models \phi$ does not change the current mental state σ.

The semantics of a mental state transformer is an *update set*: a set of (possibly sequences of) *updates*. The same notation $\oslash\psi$ ($\mathbf{skip}$) is used to denote a simple update as well as the corresponding primitive mst. It should be clear from the context which of the two is intended. 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.

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 agent $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$, denoted by $\sigma \bigoplus \rho$, is a new state $\sigma' = \langle \sigma_1, \ldots, \sigma_i', \ldots, \sigma_n \rangle$ where $\sigma_i' = \sigma_i\rho = \sigma_i \oslash \psi$ and σ_i, $\oslash$, and ψ correspond to one and the same $\mathcal{M}_i$ of $\mathcal{A}$. Applying the special update* $\mathbf{skip}$ *to a state σ results in the same mental state $\sigma = \sigma \bigoplus \mathbf{skip}$.*

The result of applying an update of the form $\rho_1 \bullet \rho_2$ to a state σ, i.e. $\sigma \bigoplus (\rho_1 \bullet \rho_2)$, is the new state $(\sigma \bigoplus \rho_1) \bigoplus \rho_2$.

Note, that since we assume disjoint sets of query/update operators for different KR modules, a formula $\oslash\psi$ unambiguously corresponds to a single KR module.

The meaning of a mental state transformer in state σ, formally defined by the *yields* predicate below, is the update it yields in that mental state. For the purpose of this paper, we introduce a slightly simplified, more convenient definition of the *yields* calculus originally published in [16, 17].

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:*

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

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

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

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

The mst $\mathbf{skip}$ yields the update $\mathbf{skip}$. Similarly, a primitive update mst $\oslash\psi$ yields the corresponding update $\oslash\psi$. In case the condition 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 $\mathbf{skip}$ 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 collection of all the updates yielded w.r.t. the Definition 3 comprises an update set of an agent program τ in the current mental state σ. The semantics of the agent $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ is then defined as a set of all, possibly infinite, *computation runs* $\sigma_1, \sigma_2, \ldots, \sigma_k, \ldots$ the agent can take during its lifetime, s.t. for each pair σ_i, σ_{i+1}, there exists an update ρ which is yielded by τ in σ_i (i.e. $yields(\tau, \sigma, \rho)$) and $\sigma_{i+1} = \sigma \bigoplus \rho$.

2.1 Jazzyk

Jazzyk is an interpreter of the *Jazzyk* programming language implementing the computational model of the *BSM* framework. In the examples later in this paper, we use a more readable notation mixing

the syntax of *Jazzyk* with that of the *BSM* mst's introduced above. **when** ϕ **then** τ encodes a conditional *mst* $\phi \longrightarrow \tau$. Symbols ; and , stand for choice | and sequence ∘ 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[2], a state-of-the-art macro preprocessor. Macros are a powerful tool for structuring and modularizing and encapsulating the source code and writing code templates. GNU M4 macros are defined using a statement **define**(<identifier>, <body>) and expanded whenever a macro identifier is instantiated in the source code. Before feeding the *Jazzyk* agent program to the language interpreter, first all the macros are expanded.

For further details on the *Jazzyk* programming language and the macro preprocessor integration with *Jazzyk* interpreter, consult [17]. Examples throughout this paper will use macros implementing parts of the *Jazzbot* agent program as standalone mental state transformers.

3 DESIGN & IMPLEMENTATION

The architecture of agents as *Behavioural State Machines* splits the agent program into two distinct layers: the *knowledge representation layer* and the *behavioural layer*. While the concern dealt with in the KR layer is modeling agent's beliefs about its environment and its own mental attitudes, the *BSM* computational model facilitates implementation of agents behaviours. The two are coupled by invocation of query and update operators of KR modules.

In the following we discuss considerations and issues we faced when developing a BDI inspired cognitive agent in the *BSM* framework. We accompany our discussion with examples adapted from the *Jazzbot* project implementation.

3.1 Jazzbot

To demonstrate the applicability of the framework of *Behavioural State Machines* and the *Jazzyk* language, we implemented *Jazzbot*, a virtual agent embodied in a simulated 3D environment of a first-person shooter computer game *Nexuiz*[3].

In [17], we introduce the architectural details of the *Jazzbot* project. *Jazzbot* is a goal-driven agent. It features four KR modules representing *belief base*, *goal base*, and an interface to its *virtual body* in a *Nexuiz* environment respectively. While the goal base consists of a single knowledge base realized as an ASP logic program, the belief base is composed of two modules: *Answer Set Programming* [3] based one and a *Ruby*[4] module for representing the map of the bot's environment. The interface to the environment is facilitated by a *Nexuiz* game client module. The Figure 1 depicts the structure of the *Jazzbot* application.

Jazzbot's behaviours are implemented as a *Jazzyk* program. *Jazzbot* can fulfill e.g. search and deliver tasks in the simulated environment, it avoids obstacles and walls. Figure 1 depicts the architecture of *Jazzbot* and features an example *Jazzyk* code chunk implementing a simple behaviour of picking up an object by a mere walk through it and then keeping notice about it in its ASP belief base. Note that all the used KR modules are compatible with each other, since they share the domain of character strings. Hence all the variables used in *Jazzbot*'s programs are meant to be character string variables.

[2] `http://www.gnu.org/software/m4/`
[3] `http://www.alientrap.org/nexuiz/`
[4] *`http://www.ruby-lang.org/`*

```
Agent program:
when believes goals(Obj) [{find(Obj)}] and                    (1)
     believes brain(Obj) [{see(Obj)}] and                     (2)
     query map(Object, Dist) [{Dist=get_distance_of(Obj)}]    (3)
then {
     act body(Dist) [{move forward Dist}] ,                   (4)
     update brain(Obj) [{keeps(Obj)}]                         (5)
}
```

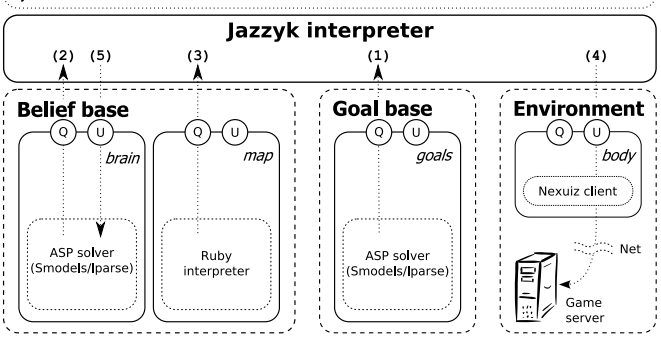

Figure 1. Scheme of *Jazzbot*

Below we describe our considerations while designing and implementing the *Jazzbot* softbot. For closer details on the architectural design of the *Jazzbot*'s components consult [17].

3.2 Knowledge representation layer

Our aim is to demonstrate the flexibility of the *BSM* framework in a BDI-inspired agent system. As described already above, *Jazzbot* features two independent knowledge bases: a *belief base* and a *goal base*. Additionally the embodiment of the bot requires an *interface to its body* (and thus to the environment). *Jazzbot* thus features KR modules labeled *beliefs* ($\mathcal{B}$), *goals* ($\mathcal{G}$) and *body* ($\mathcal{E}$). While the first two are implemented as logic programming based knowledge bases in *Answer Set Programming* [3], the last one is realized as a connector to a *Nexuiz* game server. Formally, the body is represented by a KR module $\mathcal{E} = (\mathcal{L}_{Nexuiz}, \{\models_{\mathcal{E}}\}, \{\oslash_{\mathcal{E}}\})$. It uses a special purpose language $\mathcal{L}_{Nexuiz}$ for query/update formulae and two query/update operators $\models_{\mathcal{E}}$, $\oslash_{\mathcal{E}}$ accepting formulae from this language and evaluating them against the simulated environment represented by the game server.

Belief base *Jazzbot*'s belief base $\mathcal{B}$ contains a logic program describing agent's beliefs about its environment and itself. It is supposed to closely reflect agent's perceptions of the environment, i.e. updates of belief base correspond to agent's perceptions, while queries can also include higher level consequences of primitive perceptions w.r.t. the logic program in $\mathcal{B}$.

In *Jazzbot* we exploit the power of non-monotonic reasoning for capturing relations and interactions between various beliefs an agent can hold. Formally, the *Jazzbot*'s belief base $\mathcal{B} = (AnsProlog^*, \{\models_{\mathcal{B}}\}, \{\oplus_{\mathcal{B}}, \ominus_{\mathcal{B}}\})$ uses $AnsProlog^*$ [3], the language of ASP logic programs and features an entailment query operator $\models_{\mathcal{B}}$ evaluating a query formula φ true iff it holds in all answer sets of the logic program representing the actual belief base. The updates $\oplus_{\mathcal{B}}$ and $\ominus_{\mathcal{B}}$ correspond to trivial assert and retract of a formula respectively[5].

[5] In the long run we consider more complex belief revision operators realizing extension and contraction operators similar to those used in *Dynamic Logic Programming* [14].

Listing 1 *Jazzbot*'s belief base implementation in $AnsProlog^*$.

```
% Initially the bot does not hold the box %
% The bot can later hold other objects as well %
¬hold(box(42)).

% Reasoning about the health status %
alive :− health(X), X > 0.
dead :− health(X), X <= 0.
attacked :− health(X), X <= 90.
wounded :− health(X), X <= 50.

% Reasoning about friendliness of other players %
friend(Id) :− see(player(Id)), not attacked, player(Id).
enemy(Id) :− see(player(Id)), not friend(Id), player(Id).

player(1..5).
```

The Listing 1 shows a logic program representing a part of the agent's initial belief base. The *Jazzbot*'s belief base facilitates reasoning about objects the bot believes to posses, its health status and other players. The bot perceives its health status as a numeric value, from which it derives its own state in the game. It is also able to perceive objects and other players in the environment and it reasons about them as well. For example, it considers a player it actually sees as a friend, if it does not feel threatened by him. In an extended example, the bot could also reason about its roles in the game and keep long-term beliefs about other players.

To represent the bot's information about the topology of its environment, we employ a module implemented in an object oriented scripting language *Ruby*. Since a description of it's functionality is not essential for the purposes of this paper, we do not provide a closer description of its internal functionality.

Goal base Goals are usually meant to provide a declarative description of situations (states) an agent desires to believe to be in (goals *to-be*), or activities it desires to perform (goal *to-do*). Each goal triggers a certain *behaviour* of the agent designed to satisfy it. Under some conditions w.r.t. the state of agent's beliefs, the agent adopts a goal and according to its type, it might eventually drop it, i.e. a goal also comes with an associated *commitment strategy*.

Because of the separation of concerns between the agent's belief and goal bases in the *BSM* framework, the interactions between belief base conditions and goals have to be expressed explicitly in the form of internal behaviours (causal updates). Therefore, rather than providing a concrete fixed logic-based semantics for goals and their associated commitment strategies, we propose viewing these as mere context holders, or *behaviour drivers*. The main purpose of a goal is then to *implicitly* represent a condition on beliefs which is to be achieved (or avoided) and enable execution of a behaviour designed to eventually satisfy this condition in the future. In this view, goals of an agent are only loosely coupled with its beliefs.

Formally, the *Jazzbot*'s goal base $\mathcal{G} = (AnsProlog^*, \{\models_{\mathcal{G}}\}, \{\oplus_{\mathcal{G}}, \ominus_{\mathcal{G}}\})$ is technically equivalent to the belief base $\mathcal{B}$, however because the BSM framework requires disjoint KR modules, we use a special subscript for its operators. Besides holding a set of currently adopted goals (primitive facts), the agent can thus also reason about their interactions and derive non-trivial goals, or subgoals from the more primitive ones.

The Listing 2 provides a logic program encoding a part of agent's initial goal base. Initially the bot has two maintenance goal: to survive and to be happy as well as a single achievement goal to get the box identified as box(42). To satisfy the goal to be happy, the bot activates behaviours which are triggered by tasks to communicate and wander around the environment. Survival requires the bot to explore its environment as well as to seek safety and energy sources.

Listing 2 *Jazzbot*'s goal base implementation in $AnsProlog^*$.

```
% Initially the bot has two maintenance goals and %
% a single achievement goal. %
maintain(happy).
maintain(survive).
achieve(get(box(42))).

% Subgoals of the goal maintain(happy) %
task(communicate) :− maintain(happy).
task(wander) :− maintain(happy).

% Subgoals of the goal maintain(survive) %
task(wander) :− maintain(survive).
task(safety) :− maintain(survive).
task(energy) :− maintain(survive).

% Subgoals of the goal achieve(get(Object)) %
task(search(X)) :− achieve(get(X)), not achieve(get(medikit)), item(X).
task(pick(X)) :− achieve(get(X)), not achieve(get(medikit)), item(X).
task(deliver(X)) :− achieve(get(X)), not achieve(get(medikit)), item(X).

% Specialized subgoals of the goal achieve(get(medikit)) %
task(search(medikit)) :− achieve(get(medikit)).
task(pick(medikit)) :− achieve(get(medikit)).

% Ressurect after being killed %
task(reborn) :− achieve(reborn).

% Definition of items %
item(medikit).
item(X) :− box(X).
box(1..50).
```

In the course of its lifetime, the bot might adopt goals regarding getting objects from the environment. Unless some exceptional conditions are met, e.g. for whatever reason the bot desperately looks for a *medikit* object, the goal to get an object from the environment activates also goals to find, pick and deliver the object. The order of execution of the three subgoals will be specified implicitly by encoding of the bot's individual behaviours. Searching for the *medikit* object is defined using specialized rules, as this special object does not have to be delivered anywhere.

Similarly to the bot's belief base, the goal base contains a logic program for reasoning about agent's goals. In the case a larger program is contained in the bot's goal base, this raises a question which parts of the particular goal base language (literals used in the logic program) are to be interpreted as behaviour triggers and which serve only for reasoning about interactions between goals. To solve this problem, we divide the goal base language into two parts: *declarative goals handling* and *handling of tasks*. The goal base then facilitates a breakdown of declarative achievement goals, such as achieve(get(X)), into tasks, behaviour triggers, such as task(search(X)). This way, we moved the reasoning about goal interactions completely into the goal base, instead of handling it inside the agent program, as it is done in other agent programming languages, such as *Jason* [7]. In the following we will show, that this technique results in implementation of behaviours, execution of which is triggered by merely checking the associate trigger literal in the goal base.

3.3 Behavioural layer

The choice of agent's KR modules and their ascribed purposes drives the implementation of agents behaviours. These can be either *exogenous* - resulting in selecting an action (or a sequence of actions) to be executed in the agents environment, or *endogenous* - implementing interactions between agent's knowledge bases. Analysis of information flows between agent's KR modules straightforwardly leads to identification of the individual compound behaviours.

The *Jazzbot*'s KR layer consists of the following KR modules: *beliefs*, *goals* and the bot's *body*. This gives rise to three information flows in the agent system: *body* $\longrightarrow$ *beliefs*, *beliefs* $\longrightarrow$ *goals*, *goals* $\longrightarrow$ *body*. In accord with the usual understanding of BDI model of rationality [18], these represent respectively the following principles: 1) an agent senses its environment and reflects its perceptions in its beliefs (*perceptions*); 2) because it believes to be in a certain situation, it updates its goals (*commitment strategies*); and finally 3) to achieve the adopted goals, it acts in the environment (*action selection*).

In the following we discuss these individual behaviors in detail.

Perceptions *Jazzbot* agent roams around the simulated 3D environment and keeps track of its perceptions regarding its surroundings and a state of its own body. The actual state of *beliefs* is supposed to reflect its perceptions about the world and relations between them. In general, provided a sensory information φ from the agent's environment (body), following the information flow notation used above, we can encode a corresponding update ψ of an agent's belief base as a conditional mental state transformer

$$\models_{\mathcal{E}} \varphi \longrightarrow \oslash_{\mathcal{B}} \psi \tag{1}$$

A set of such conditional mst's captures the relations between various perceptions and their belief counterparts. Now, according to the chosen model of perception, a designer can form a proper *BSM* mst τ_{perc} from this set by joining the mst's by either non-deterministic choice $|$, or sequence $\circ$ operators. In a more advanced setting, beyond the scope of this paper, the designer can even choose to further structure them using nested conditional mst's.

Listing 3 Implementation of *Jazzbot*'s perceptions handling.

```
define('PERCEIVE','
{
    /* Check the health sensor */
    when |=E [{ body health X }] then
    {
        /* Before updating with the new value, retract the old one */
        when |=B [{ health(Y) }] then (-)B [{ health(Y). }] ,
        (+)B [{ health(X). }]
    } ,

    /* Check whether the bot still sees an object it remembers */
    when |=B [{ see(Id, Type) }] and not |=E [{ see(Id, Type) }]
    then (-)B [{ see(Id, Type) }] ,

    /* Check the camera sensor */
    when |=E [{ eye see Id Type }] then (+)B [{ see(Id, Type). }] ,

    ...
}
')
```

The Listing 3 shows an example encoding an mst implementing the bot's perception of its own health in the game as well as recognition of objects in the vicinity of the bot. Note, that the *Jazzbot* checks its sensors in a sequence. Perception can be considered a safe sequential behaviour, as checking sensors takes only a little time and since the bot does not act in the environment, this behaviour can always finish without an interruption.

Goal commitment strategies A specific goal can be adopted because an agent believes its adoption is appropriate. Similarly, because of a certain state of beliefs, the agent might decide to drop a goal: for instance, a goal can be satisfied, or believed to be impossible to achieve, etc. Informally, a set of such internal behaviours related to a single goal implements a *commitment strategy* associated with it. Moreover, a designer can implement different commitment strategies w.r.t. various goals. Commitment strategies thus realize the second component of the information flow between the agent's knowledge bases: given a condition on agent's beliefs φ, the agent updates its goal base by a goal formula ψ. The corresponding mst loosely follows the scheme

$$\models_{\mathcal{B}} \varphi \longrightarrow \oslash_{\mathcal{G}} \psi \tag{2}$$

Here, the entailment operator $\models_{\mathcal{B}}$ represents the belief base entailment operator and $\oslash_{\mathcal{G}}$ is a corresponding goal base update operator (in the *Jazzbot* setting $\oslash_{\mathcal{G}} \in \{\oplus_{\mathcal{G}}, \ominus_{\mathcal{G}}\}$). Similarly to perceptions, the agent designer can join and structure the mst's realizing commitment strategies of individual goals the agent can adopt during its lifetime using *BSM* composition operators. τ_{cs} will denote the resulting mst.

Listing 4 Implementation of *Jazzbot*'s goal commitment strategies handling.

```
define('HANDLE_GOALS','
{
    /* Adoption and dropping of the goal to get medikit */
    when |=B [{ wounded }] then (+)G [{ get(medikit). }] ;

    when |=G [{ get(medikit) }] and not |=B [{ wounded }]
    then (-)G [{ get(medikit). }] ;

    /* When the bot receives a user command, it obeys */
    when |=B [{ command(get(X)) }] then (+)G [{ achieve(get(X)). }] ;

    when |=G [{ achieve(get(X)) }] and |=B [{ holds(X) }]
    then (-)G [{ achieve(get(X)). }] ;

    /* When the bot finds out it was killed, it resurrects in the game */
    when |=B [{ dead }] then (+)G [{ achieve(reborn). }] ;

    when |=B [{ alive }] then (-)G [{ achieve(reborn). }] ;

    ...

}
')
```

The Listing 4 provides an example of implementation of commitment strategies w.r.t. bot's achievement goals. When the bot believes it was wounded in the game, it adopts a goal to get the *medikit* object to refresh its health. Sometime after it starts to believe that it found it, it drops the goal. Similarly, the bot implements custom handling for each achievement goal it can deal with. It is responsive to user commands and is able to get an item on request and finally when it detects that it was terminated in the game, it adopts a goal to get back into it. Adopted goals, subsequently trigger behaviours which should achieve them. Note, that the individual mst's implementing goal commitment strategies are joined together using the non-deterministic choice operator. This way, the bot is allowed to adopt, or drop a goal only once per an execution cycle.

Goal oriented behaviours: action selection The core task of an agent, is to perform behaviours (actions) in the environment. In the setting introduced above, behaviours have a purpose: satisfaction of adopted goals. We speak therefore about *goal oriented behaviours*. In general, following the information flow notation, they amount to choosing an appropriate action ψ for achieving a goal φ, the agent currently pursues:

$$\models_{\mathcal{G}} \varphi \longrightarrow \oslash_{\mathcal{E}} \psi \tag{3}$$

with $\models_{\mathcal{G}}$ representing an entailment operator on the goal base and $\oslash_{\mathcal{E}}$ being the body/environment update operator.

In our experience, we quickly found that the agent's core behaviour, the action selection mechanism, often requires a more complex structuring than the, rather reactive, scheme 3 prescribes. First, there can potentially be several behaviours supposed to achieve the same goal in possibly different contexts, and second, in different contexts, a single behaviour might be appropriate for achieving several different goals. We therefore extend the scheme 3 above as follows:

$$\phi_{\mathcal{G}} \wedge \phi_{\mathcal{B}} \longrightarrow \tau \tag{4}$$

$\phi_{\mathcal{G}}$ represents a pursued goal query, $\phi_{\mathcal{B}}$ is a belief context guard, and τ is a possibly compound behaviour associated with (some of the) goal(s) represented by $\phi_{\mathcal{G}}$. To support source code modularity, the *Jazzyk* interpreter integrates a powerful macro preprocessor. Thus in different contexts re-usable mst's can be wrapped into named macros and simply expanded at places in the code, where they are applied.

Various behaviours of an agent are combined in various ways in different contexts and situations. In certain contexts (e.g. emergency situations), where a tight behaviour control is required, script-like compound behaviours (also called ballistic [1]) are more appropriate. However, more often we want the agent to interleave behaviours associated with orthogonal, not interfering, goals. The design choices are therefore application specific and left to the designer. We denote the compound mental state transformer implementing agent's action selection as τ_{act}.

Listing 5 Implementation of *Jazzbot*'s behaviour selection.

```
define('ACT','
{
    /* Behaviours for getting an item */
    /* The bot searches for an item, only when it does not have it */
    when |=_G [{ task(search(X)) }] and not |=_B [{ hold(X) }]
    then SEARCH('X') ;

    /* When a searched item is found, it picks it */
    when |=_G [{ task(pick(X)) }] and |=_B [{ see(X) }]
    then PICK('X') ;

    /* When the bot finally holds the item, it deliver it */
    when |=_G [{ task(deliver(X)) }] and |=_B [{ hold(X) }]
    then DELIVER('X') ;

    /* Simple behaviour triggers without guard conditions */
    when |=_G [{ task(reborn) }] then REINCARNATE ;

    when |=_G [{ task(wander) }] then WALK ;

    when |=_G [{ task(safety) }] then RUN_AWAY ;

    when |=_G [{ task(communicate) }] then SOCIALIZE ;

    ...
}
')
```

The Listing 5 provides an example code implementing selection of goal oriented behaviours, realized as parametrized macros, satisfying *Jazzbot*'s goals. While the bot simply triggers behaviours for reincarnation, walking around, danger aversion and social behaviour, the execution of behaviours finally leading to getting an item are guarded by belief conditions. This way, we introduce an order on these behaviours. Recall, that in the goal base, the goal to get an item triggers the tasks to search for it, pick it up and deliver it simultaneously. Using a different handling of goals directly in the goal base, we could implement ordering of the goals already there and then trigger the individual behaviours without a belief base guard condition.

3.4 Control cycle

Putting together the previously designed mst's implementing the agent's model of perception τ_{perc}, its goals commitment strategies τ_{cs} and the agent's behaviour, i.e. the action selection mechanism τ_{act}, we implement a control cycle of the *BSM* agent program. According to ordering and combination of the mst's a designer can 1) develop the control model of the agent, as well as 2) control determinism of the agent program. In the *Jazzbot* example we could consider either a case in which in a single computation step the bot non-deterministically either perceives, handles its goals, or acts: $\tau_{perc}|\tau_{cs}|\tau_{act}$, or sequentially executes all the stages: $\tau_{perc} \circ \tau_{cs} \circ \tau_{act}$. Different orderings of the mst's yield different overall behaviours of the agent as well.

As we already discussed above, according to the internal structure of the partial mst's of the *BSM* agent program, the agent can for example either check all its sensors in a single cycle, or consider only one of them non-deterministically. Similarly for the goal commitment strategies and action selection mechanism. Different structuring of the partial mst's inside τ_{perc}, τ_{cs} and τ_{act} allows a programmer to implement various control models. Below we provide few examples of partial control models expressed using an LTL-like notation [15]:

- $\Diamond(\models_{\mathcal{G}} \varphi_{\mathcal{G}} \rightarrow \tau_{\varphi})$ to eventually execute a behaviour τ_{φ} associated with the goal $\varphi_{\mathcal{G}}$ it is sufficient to use a corresponding conditional mst $\models_{\mathcal{G}} \varphi_{\mathcal{G}} \longrightarrow \tau_{\varphi}$ somewhere within τ_{act}.
- $\Box(\models_{\mathcal{G}} \varphi_{\mathcal{G}} \rightarrow \tau_{\varphi})$ to ensure a stricter version of the previous case, the mst τ_{act} has to be structured as a sequence of mst's triggering various goal oriented behaviours, where one of them takes the form of a conditional $\models_{\mathcal{G}} \varphi_{\mathcal{G}} \longrightarrow \tau_{\varphi}$.
- $\Box(\models_{\mathcal{E}} \varphi_{\mathcal{E}} \rightarrow \bigcirc \oplus_{\mathcal{B}} \varphi_{\mathcal{B}})$ adoption of a belief $\varphi_{\mathcal{B}}$, corresponding to a perception $\varphi_{\mathcal{E}}$, immediately after the agent perceives $\varphi_{\mathcal{E}}$, can be ensured by structuring the perception mst τ_{perc} as a sequence of conditional mst's. At the same time the mst's of the control cycle have to be combined into a sequence $\tau_{perc} \circ \tau_{cs} \circ \tau_{act}$.

Note, that we assume that the abstract interpreter of the *BSM* framework is fair, i.e. it is not the case, that an mst which is always enabled along an infinite computation trace will never be executed. This allows us to use modal operators, such as $\Diamond$, in our semi-formal specifications above.

Listing 6 Implementation of *Jazzbot*'s control cycle.

```
/* The actual Jazzbot agent program */
PERCEIVE , HANDLE_GOALS , ACT
```

Finally, the Listing 6 sums up the running example of this paper. It provides the implementation of the control cycle implemented in *Jazzbot* using the macros defined in the previous subsections. Note, that the bot executes in every step all the stages of its control cycle sequentially.

4 PUTTING IT TOGETHER

Finally, we can put together a set of more general design guidelines for development of embodied agent systems implemented in a rule-based languages similar to *BSM*. In this paper we focus on agents featuring belief and goal bases. The central element around which our design considerations revolved were *agent's goals*.

Goals determine currently active (enabled) behaviours and thus serve as a trigger for agent's exogenous behaviours, which are the

visible manifestations of its functionality. Additionally, a goal is associated with a commitment strategy steering its adoption and satisfaction, or dropping (both determined by conditions on agent's beliefs). This leads to a notion of *goal oriented behaviours*.

A set of goal oriented behaviours can be characterized by a tuple $(\phi, \kappa_{\oplus}, \kappa_{\ominus}, \Upsilon)$, where in the case of *Jazzbot* agent, $\phi \in \mathcal{L}_{ASP}$ is a goal, $\kappa_{\oplus}, \kappa_{\ominus} = \{\varphi \in \mathcal{L}_{ASP}\}$ are sets of its adopt and drop conditions w.r.t. the belief base $\mathcal{B}$ respectively, and $\Upsilon = \{\tau | \tau \textit{ is an mst}\}$ is a set of behaviours triggered by ϕ. Informally, a commitment strategy behaviour τ_{cs} then should contain conditional mst's of the form $\models_{\mathcal{B}} \varphi \longrightarrow \oslash_{\mathcal{G}} \phi$ with $\oslash_{\mathcal{G}} \in \{\oplus_{\mathcal{G}}, \ominus_{\mathcal{G}}\}$ and $\varphi \in \kappa_{\oslash}$ being either an adopt, or a drop condition. Υ then contains conditional mst's of τ_{act} similar to $\models_{\mathcal{G}} \varphi \longrightarrow \tau$, where $\tau \in \Upsilon$. A very similar view can be formulated for beliefs featuring belief adoption and drop conditions (w.r.t. agent's perceptions) and being loosely associated with adopt/drop conditions of agent's goals. In a consequence, such considerations would lead to a formal characterization of causal information flows between the agent's knowledge bases, a topic beyond the scope of the presented work.

By generalizing the presented approach to development of *Jazzbot* we arrive to the following methodological steps/guidelines for designing *BSM* agents:

1. identify the set of agent's goals and design their interactions w.r.t. the employed KR technology,
2. design a set of behaviours τ_{act} triggered by these goals (supposed to achieve them),
3. identify the adoption and satisfaction (drop) conditions for these goals and design concrete commitment strategies for them in τ_{cs},
4. identify the relevant part of agent's beliefs w.r.t. the conditions associated with the goals,
5. design the agent's belief base including appropriate belief relationships w.r.t. the employed KR technology,
6. design the model of perception τ_{perc} by identifying the percepts of the agent and link them to corresponding beliefs,
7. finally construct the global *BSM* agent program by appropriately structuring and combining the mental state transformers τ_{perc}, τ_{cs} and τ_{act} into a control cycle.

The presented guidelines, centered around the notion of a goal, loosely fit the general view of methodologies for agent-based systems like e.g. *Tropos*, or *MaSE* [4]. In these, one of the main results of the analytical stage of a single agent system are agent's goals, or tasks, associated with agent's roles. Such methodologies are usually not coupled to a particular agent architecture, the details of the agent design are therefore left to a particular platform. The guidelines proposed here thus informally fill this gap, at least w.r.t. the *BSM* framework. However, we are convinced that some of the considerations discussed above apply also to other agent oriented rule-based language, especially when considering heterogeneous KR technologies in a single agent.

5 DISCUSSION & CONCLUSION

To our knowledge, not too much was reported on implementation techniques and source code structuring of larger, non-trivial agent systems in agent oriented programming languages like *Jason*, or *3APL* [6]. Some sketchy notes on overall system design can be found in *Jason* and *2APL* team reports from *Multi-Agent Programming Contest* 2007 and 2008 [2, 12, 13], however no more general methodological considerations are discussed there and authors focus solely on design of their particular agent system. From the published source code of these projects[6], it can be seen that the authors extensively use escape calls into *Java* code and the *Java* implementation comprised a significant part of their system implementations: representation of the agent's environment, shortest point to point path planning, inter-agent coordination, etc.. The framework of *Behavioural State Machines* makes calls to external code a first class concept of an agent programming language and facilitates only interactions between the agent's knowledge bases.

This unconstrained approach allows for custom implementation of various strategies for handling agent's mental attitudes and control models. While the mainstream approach in agent programming languages is to choose a set of constraints an agent system must obey, our approach is different. We propose a generic and flexible agent programming framework, even capable of emulation of other agent programming languages (see [11]), and subsequently develop a set of methodological and design guidelines, so to say *rules of good practice*, for development of cognitive agents. We accompany the discussion by examples of code patterns supporting implementation according to these design guidelines.

The presented discussion provides a snapshot of our current experience and knowledge in programming cognitive agents with the framework of *Behavioural State Machines*. We discuss ideas and issues resulting from an ongoing work towards proposing a set of *design patterns for cognitive agent programming*. The main contribution of the presented work is an attempt to lift these considerations into a set of methodological steps, partly also applicable to implementation of agents in other rule-based agent oriented programming frameworks (*AgentSpeak(L)* family of languages). However, the *BSM* framework was specifically designed in a liberal way and thus it allows use of heterogeneous knowledge representation technologies together with implementation of arbitrary interactions among them. Therefore, because of additional constraints these languages (mostly BDI oriented) impose on agent programs, some of the presented interactions between agent's knowledge bases and implementation techniques (e.g. reasoning about agent's goals in $AnsProlog^*$) might be difficult to implement in them.

In the future research, we aim to study formal specification methods (like e.g. source code annotations) for cognitive agents, by trying to generalize examples of control models like those presented in Subsection 3.4. Subsequently we aim at characterizing more complex code structures and templates, by means of dynamic, or temporal logic adapted for the *BSM* framework. Our line of research follows a bottom-up approach: instead of proposing a way to design a system by analyzing it, we rather try to experiment with live implementations and collect experiences, which could later serve as a basis for a generalization.

REFERENCES

[1] Ronald C. Arkin. *Behavior-based Robotics*. MIT Press, Cambridge, MA, USA, 1998.

[2] L. Astefanoaei, C. P. Mol, M. P. Sindlar, and N. A. M. Tinnemeier. Going for gold with 2APL. In *Proceedings of Fifth international Workshop on Programming Multi-Agent Systems, ProMAS'07*, volume 4908 of *LNAI*. Springer Verlag, 2008.

[3] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[4] Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors. *Methodologies and Software Engineering for Agent Systems: The*

[6] Communicated over Agent Contest mailing lists: publicly available at `http://cig.in.tu-clausthal.de/agentcontest/`.

Agent-Oriented Software Engineering Handbook, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, June 2004.

[5] Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge J. Gomez-Sanz, João Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30:33–44, 2006.

[6] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Multi-Agent Programming Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, 2005.

[7] Rafael H. Bordini, Jomi F. Hübner, and Renata Vieira. *Jason and the Golden Fleece of Agent-Oriented Programming*, chapter 1, pages 3–37. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [6], 2005.

[8] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[9] Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Meyer. *Programming Multi-Agent Systems in 3APL*, chapter 2, pages 39–68. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [6], 2005.

[10] Frank S. de Boer, Koen V. Hindriks, Wiebe van der Hoek, and John-Jules Ch. Meyer. A verification framework for agent programming with declarative goals. *J. Applied Logic*, 5(2):277–302, 2007.

[11] Koen Hindriks and Peter Novák. Compiling GOAL Agent Programs into Jazzyk Behavioural State Machines. Submitted 2008.

[12] Jomi F. Hübner and Rafael H. Bordini. Developing a team of gold miners using Jason. In *Proceedings of Fifth international Workshop on Programming Multi-Agent Systems, ProMAS'07*, volume 4908 of *LNAI*. Springer Verlag, 2008.

[13] Jomi F. Hübner, Rafael H. Bordini, and Gauthier Picard. Using *jason* to develop a team of cowboys: a preliminary design for Agent Contest 2008. In *Proceedings of Sixth International Workshop on Programming Multi-Agent Systems, ProMAS 2008*, 2008.

[14] João Alexandre Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers of Artificial Intelligence and Applications*. IOS Press, 2003.

[15] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[16] Peter Novák. Behavioural State Machines: programming modular agents. In *AAAI 2008 Spring Symposium: Architectures for Intelligent Theory-Based Agents, AITA'08*, March 26-28 2008.

[17] Peter Novák. Jazzyk: A programming language for hybrid agents with heterogeneous knowledge representations. Sixth International Workshop on Programming Multi-Agent Systems, May 2008.

[18] Anand S. Rao and Michael P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In *KR*, pages 473–484, 1991.