Jazzyk: A programming language for hybrid agents with heterogeneous knowledge representations

Peter Novák

Department of Informatics
Clausthal University of Technology
Julius-Albert-Str. 4, D-38678 Clausthal-Zellerfeld, Germany
peter.novak@tu-clausthal.de

Abstract. Different knowledge representation tasks require different knowledge representation techniques. Agent designers should therefore be able to easily exploit benefits of various knowledge representation technologies in a single agent system.
I describe here a modular agent programming language *Jazzyk* based on the programming framework of *Behavioural State Machines* (*BSM*). *BSM* framework, and thus also *Jazzyk*, draws a strict distinction between a knowledge representational and a behavioural level of an agent program. It supports a high degree of modularity w.r.t. employed KR technologies, and at the same time provides a clear and concise semantics.

1 Motivation

No single knowledge representation (KR) technology offers a range of capabilities and features required for different application domains and environments agents operate in. For instance, purely declarative KR technologies offer a great power for reasoning about relationships between static aspects of an environment, like e.g. properties of objects. However, they are not suitable for representation of topological, arithmetical, or geographical information. Similarly, a relational database is appropriate for representation of large amounts of searchable tuples, but it does not cope well with representing exceptions and default reasoning. Hence, an important pragmatic requirement on a general purpose AOP framework is *an ability to integrate heterogeneous KR technologies* within a single agent system. An agent programming framework should not commit to a single KR technology. The choice of an appropriate KR approach should be left to an agent designer and the framework should be *modular* enough to accommodate a large range of KR techniques, while at the same time providing flexible means to encode agent's behaviours.

I recently proposed a framework of *Behavioural State Machines* (*BSM*) [13, 12], a general purpose computational model based on the Gurevich's *Abstract State Machines* [4], adapted to the context of agent oriented programming. The

BSM framework is a culmination of our previous efforts ([14] and [15]) to propose a solid theoretical basis for a lightweight, yet highly modular agent programming language. It treats heterogeneous knowledge bases of an agents on a par, i.e. does not prefer one over another thus allowing programmers to exploit strengths of various KR approaches in an agent system.

The main purpose of this paper is to describe *Jazzyk* (Section 3), a programming language based on the theoretical framework of *Behavioural State Machines* (Section 2), together with details of its implemented interpreter. Development of the *BSM* framework is an application driven research, therefore I furthermore provide a sketch of *Jazzbot* (Section 4), a case study demo application implemented in *Jazzyk*. The paper concludes with a discussion of *Jazzyk* (Section 5), related work and future development of this line of research (Section 6).

2 Behavioural State Machines

Before introducing the details of *Jazzyk*, first I briefly introduce its theoretical basis: the framework of *Behavioural State Machines* (*BSM*). *Behavioural State Machine* computational model is heavily inspired by the Gurevich's *Abstract State Machines* [4] framework.

The underlying abstraction is that of a transition system, similar to that used in most logic based state-of-the-art BDI agent programming languages *AgentSpeak(L)/Jason*, *3APL*, or *GOAL* [2, 6]. States are agent's mental states, i.e. collections of agent's partial knowledge bases, or *KR modules*. The state of the environment is treated as a KR module as well. Transitions between the agent's mental states are induced by *mental state transformers* (atomic updates of mental states). An agent system semantics is, in operational terms, a set of all enabled paths within the transition system, the agent can traverse during its lifetime. To facilitate modularity and program decomposition, *BSM* provides also a functional view on an agent program, specifying a set of enabled transitions an agent can execute in a given situation.

Behavioural State Machines draw a strict distinction between the *knowledge representational layer* of an agent and its *behavioural layer*. To exploit strengths of various KR technologies, the KR layer is kept abstract and open, so that it is possible to plug-in different heterogeneous KR modules as agent's knowledge bases. The main focus of *BSM* computational model is the highest level of control of an agent: its *behaviours*.

I introduced *BSM* framework in [12] and [13], therefore some technical details are omitted here and I mainly focus on a description of the most fundamental issues. Moreover, the Subsection 2.2 introduces a reformulated version of the original *BSM* semantics equivalent to the one originally published in [12] and [13].

2.1 Syntax

A BSM agent consists of a set of partial knowledge bases handled by so called *KR modules*. A KR module is supposed to store agent's knowledge e.g. about

its environment, itself, or other agents, or to handle its internal mental attitudes relevant to keep track of its goals, intentions, obligations, etc. However, because of the openness of the BSM architecture, no specific structure of an agent is prescribed and thus the overall number and ascribed purpose of particular KR modules is kept abstract. The formal definitions capture only their fundamental characteristics.

A KR module has to provide a language of query and update formulae and two sets of interfaces: *query* operators for querying the knowledge base and *update* operators to modify it.

Definition 1. ***(KR module)*** *A knowledge representation module* $\mathcal{M} = (\mathcal{S}, \mathcal{L}, \mathcal{Q}, \mathcal{U})$ *is characterized by*

- *a set of states* $\mathcal{S}$*,*
- *a knowledge representation language* $\mathcal{L}$*, defined over some domains* $\mathcal{D}_1, \ldots, \mathcal{D}_n$ *(with* $n \geq 0$*) and variables over these domains.* $\underline{\mathcal{L}} \subseteq \mathcal{L}$ *denotes a fragment of* $\mathcal{L}$ *including only ground formulae, i.e. such that do not include variables,*
- *a set of query operators* $\mathcal{Q}$*. A query operator* $\models \in \mathcal{Q}$ *is a mapping* $\models : \mathcal{S} \times \underline{\mathcal{L}} \rightarrow \{\top, \bot\}$*,*
- *a set of update operators* $\mathcal{U}$*. An update operator* $\oplus \in \mathcal{U}$ *is a mapping* $\oplus : \mathcal{S} \times \underline{\mathcal{L}} \rightarrow \mathcal{S}$*.*

KR languages are compatible on a shared domain $\mathcal{D}$*, when they both include variables over* $\mathcal{D}$ *and their sets of query and update operators are mutually disjoint. KR modules with compatible KR languages are compatible as well.*

From the definition we have, that a KR language not including variables is compatible with any other KR language.

Each query and update operator has an associated identifier. For simplicity, these are not included in the definition, however I use them throughout the text. When used as an identifier in a syntactic expression, I use informal prefix notation (e.g. $\models \varphi$, or $\oplus\varphi$), while when used as a semantic operator, formally correct infix notation is used (e.g. $\sigma \models \varphi$, or $\sigma \oplus \varphi$). Additionally, when the evaluation of a query formula φ by a query operator $\models$ on a state σ results in $\top$, i.e. $(\sigma \models \varphi) = \top$, we simply write $\sigma \models \varphi$, otherwise when $(\sigma \models \varphi) = \bot$, we use notation $\sigma \not\models \varphi$.

Query formulae are the syntactical means to retrieve information from KR modules:

Definition 2. ***(query)*** *Let* $\mathcal{M}_1, \ldots, \mathcal{M}_n$ *be a set of compatible KR modules. Query formulae are inductively defined:*

- *if* $\varphi \in \mathcal{L}_i$*, and* $\models \in \mathcal{U}_i$ *corresponding to some* $\mathcal{M}_i$*, then* $\models \varphi$ *is a query formula,*
- *if* ϕ_1, ϕ_2 *are query formulae, so are* $\phi_1 \wedge \phi_2$*,* $\phi_1 \vee \phi_2$ *and* $\neg\phi_1$*.*

The informal semantics is straightforward: if a ground language expression $\varphi \in \underline{\mathcal{L}}$ is evaluated to true by a corresponding query operator $\models$ w.r.t. a state of the

corresponding KR module, then $\models \varphi$ is true in the agent's mental state as well. Note, that non-ground formulae have to be first ground before their evaluation (Subsection 2.2).

Subsequently, I define *mental state transformer*, the principal syntactic construction of *BSM* framework.

Definition 3. ***(mental state transformer)*** *Let $\mathcal{M}_1, \ldots, \mathcal{M}_n$ be a set of compatible KR modules. Mental state transformer expression (mst) is inductively defined:*

1. **skip** *is a mst (primitive),*
2. *if $\oplus \in \mathcal{U}_i$ and $\psi \in \mathcal{L}_i$ corresponding to some $\mathcal{M}_i$, then $\oplus\psi$ is a mst (primitive),*
3. *if ϕ is a query expression, and τ is a mst, then $\phi \longrightarrow \tau$ is a mst as well (conditional),*
4. *if τ and τ' are mst's, then $\tau|\tau'$ and $\tau \circ \tau'$ are mst's too (choice and sequence).*

An update expression is a primitive mst. The other three (conditional, sequence and non-deterministic choice) are compound mst's. Informally, a primitive mst is encoding a transition between two mental states, i.e. a primitive behaviour. Possibly labeled compound mst's introduce modularity and code re-use to the *BSM* framework. A standalone mental state transformer is also called an *agent program* over a set of KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$.

A mental state transformer encodes an agent behaviour. I take a radical behaviourist viewpoint, i.e. also internal transitions are considered a behaviour. As the main task of an agent is to perform a behaviour, naturally an agent program is fully characterized by a single mst (agent program) and a set of associated KR modules used in it. *Behavioural State Machine* $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$, i.e. a collection of compatible agent KR modules and an associated agent program, completely characterizes an agent system $\mathcal{A}$.

2.2 Semantics

The underlying semantics of *BSM* is that of a transition system over agent's mental states.

Definition 4. ***(state)*** *Let $\mathcal{A}$ be a BSM over KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$. A state of $\mathcal{A}$ is a tuple $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ of KR module states $\sigma_i \in \mathcal{S}_i$, corresponding to $\mathcal{M}_1, \ldots, \mathcal{M}_n$ respectively. $\mathfrak{S}$ denotes the space of all states over $\mathcal{A}$.*

$\sigma_1, \ldots, \sigma_n$ are partial states of σ. A state can be modified by applying primitive updates on it and query formulae can be evaluated against it. Query formulae cannot change the actual agent's mental state.

According to the Definition 1, to evaluate a formula in a state by query and update operators, the formula must be ground. Transformation of non-ground formulae to ground ones is provided by means of *variable substitution*. A variable substitution is a mapping $\theta : \mathcal{L} \rightarrow \underline{\mathcal{L}}$ replacing every occurrence of a variable

in a KR language formula by a value from the corresponding domain. Variable substitution of a compound query formula is defined by usual means of nested substitution. Note however, that a variable can be substituted in sub-formulae of a compound formula only when languages of the corresponding sub-formulae share the domain of the variable in question. A variable substitution θ is *ground* w.r.t. ϕ, when the instantiation $\phi\theta$ is a ground formula.

Informally, a primitive ground formula is said to be true in a given *BSM* state w.r.t. a query operator, iff an execution of that operator on the state and the formula yields $\top$. The evaluation of compound query formulae inductively follows usual evaluation of nested logical formulae.

Notions of an *update* and *update set* are the bearers of the semantics of mental state transformers. An update of a mental state σ is a tuple $(\oplus, \psi)$, where $\oplus$ is an update operator and ψ is a ground update formula corresponding to some KR module. The syntactical notation of a sequence of mst's $\circ$ corresponds to a sequence of updates, or update sets, denoted by the semantic sequence operator $\bullet$. Provided ρ_1 and ρ_2 are updates, also a sequence $\rho_1 \bullet \rho_2$ is an update. Additionally, there is a special no-operation update **skip** corresponding to the primitive mst **skip**.

A simple update corresponds to semantics of a primitive mst. Sequence of updates corresponds to a sequence of primitive mst's and is a compound update itself. An update set is a set of updates and corresponds to a mst encoding a non-deterministic choice.

Given an update, or an update set, its application on a state of a *BSM* is straightforward. Formally:

Definition 5. ***(applying an update)*** *The result of applying an update $\rho = (\oplus, \psi)$ on a state $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ of a BSM $\mathcal{A}$ over KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$ is a new state $\sigma' = \sigma \bigoplus \rho$, such that $\sigma' = \langle \sigma_1, \ldots, \sigma'_i, \ldots, \sigma_n \rangle$, where $\sigma'_i = \sigma_i \oplus \psi$, and both $\oplus \in \mathcal{U}_i$ and $\psi \in \mathcal{L}_i$ correspond to some $\mathcal{M}_i$ 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 $\sigma' = \sigma \bigoplus \rho_1$.

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 6 (yields calculus). *A mental state transformer τ yields an* update *ρ in a state σ under a variable substitution θ, iff $yields(\tau, \sigma, \theta, \rho)$ is derivable in the following calculus:*

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

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

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

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

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

The mst **skip** yields the update **skip**. Provided a variable substitution θ, similarly, a primitive update mst $\oslash\psi$ yields the corresponding update $(\oslash, \psi\theta)$. In the 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 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.

In the Definition 6 we assume that the variable substitution θ is ground w.r.t. all the formulae occurring in the considered mst τ.

The calculus defining the *yields* predicate provides a *functional view* on a mst and it is the primary means of compositional modularity in *BSM*. Mental state transformers encode functions yielding update sets over states of a *BSM*. The collection of all the updates yielded w.r.t. the Definition 6 comprises an update set of an agent program τ in the current mental state σ.

Finally, the operational semantics of an agent is defined in terms of all possible computation runs induced by a corresponding *Behavioural State Machine*.

Definition 7. ***(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' (induces a transition $\sigma \rightarrow \sigma'$), if there exists a ground variable substitution θ, s.t. the agent program $\mathcal{P}$ yields a non-empty update set ν in σ under θ and $\sigma' = \sigma \bigoplus \rho$, where $\rho \in \nu$ is an update.*

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 $\mathcal{A}$.

Even though the introduced semantics of *Behavioural State Machines* speaks in operational terms of sequences of mental states, an agent can reach during its lifetime, the style of programming induced by the formalism of mental state transformers is rather declarative. Primitive query and update formulae are treated as black-box expressions by the introduced *BSM* formalism. On this high level of control, they rather encode *what* and *when* should be executed, while the issue of *how* is left to the underlying KR module. I.e., *agent's deliberation abilities reside in its KR modules, while its behaviours are encoded as a BSM.*

Figure 1 lists a pseudocode of the abstract interpreter cycle straightforwardly following from the introduced BSM semantics. In a single deliberation cycle 1) the agent program interpreter computes the update set ν corresponding to the agent program $\mathcal{P}$ according to the Definition 6, 2) non-deterministically chooses an update ρ from ν, and finally 3) updates the current mental state by applying the update ρ to it. Under _ in the *yield*(...), we denote a substitution of the set of all the free variables used in the encoding of the agent program $\mathcal{P}$.

Algorithm 1 Abstract *BSM* interpreter
input: agent program $\mathcal{P}$, initial mental state state σ_0

```
σ = σ_0
loop
  compute ν = {ρ|yields(P, σ, _, ρ)}
  if ν ≠ ∅ then
    non-deterministically choose ρ ∈ ν
    σ = σ ⊕ ρ
  end if
end loop
```

Additionally, the non-deterministic choice of the abstract *BSM* interpreter fulfils the *weak fairness condition*, similar to that in [11], for all the induced runs.

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

The *BSM* framework assumes that the mental state of an agent, including its environment, changes only between the single executions of the deliberation cycle. Therefore in order to implement agile agents which act in their environments reasonably quickly w.r.t. the speed of change of the environment, the query and update operators should be computable procedures invocations of which shouldn't take too long w.r.t. the application domain.

3 Jazzyk, the language and interpreter

In order to practically test the *BSM* approach to programming agent systems, I designed and implemented a programming language *Jazzyk* and an interpreter for it. *Jazzyk* closely follows the *BSM* framework, i.e. 1) the syntax allows for one to one encoding of mental state transformers in the language and 2) the interpreter closely follows the *BSM* semantics with only minor discrepancies aimed at making the interpreting of programs more efficient. The syntax and the precise *Jazzyk* interpreter semantics, as well as all deviations from the formal semantics are discussed in this section. Finally I also briefly sketch technical details of the *Jazzyk* interpreter implementation.

3.1 Syntax

Figure 1 lists the EBNF of *Jazzyk*, which straightforwardly follows from the syntax of *BSM* introduced in Subsection 2.

According to the BSM syntax, a *Jazzyk* program is a mental state transformer. However to allow for such programs, few technical issues have to be handled as well. The KR modules have to be declared and subsequently bound to the corresponding plug-ins implementing their functionality in a KR language

```
program       ::= (statement)*
statement     ::= module_decl | module_notify | mst

module_decl   ::= 'declare' 'module' <moduleId> 'as' <KRModuleType>
module_notify ::= 'notify' <moduleId> on
                  ('initialize' | 'finalize' | 'cycle') formula

mst           ::= 'nop' | 'exit'  | '{' mst '}' |
                  update | conditional | sequence | choice
sequence      ::= mst ',' mst
choice        ::= mst ';' mst
conditional   ::= 'when' query_expr 'then' mst ['else' mst]

query_expr    ::= query 'and' query | query 'or' query |
                  not 'query' | '(' query ')'
query         ::= 'true' | 'false' |
                  <operatorId> <moduleId> [variables] formula
update        ::= <operatorId> <moduleId> [variables] formula

formula       ::= '[{' <arbitrary string> '}]'
variables     ::= '(' (<identifier> ',')* <identifier> ')' | '(' ')'
```

Fig. 1. *Jazzyk* EBNF.

of choice. Before a first update operation is invoked on a KR module, it should be initialized by some initial state. This state is encoded as a corresponding KR language formula, i.e. code block. Similarly, when a module is being shut down, it might be necessary to perform a cleanup of the knowledge base handled by the module. In order to allow for a KR module initialization and shut-down (finalization), so called notifications KR modules are introduces. They take a form of a statement declaring a formula/code block to be executed when the KR module is loaded (i.e. before the program interpretation) and when it is being unloaded (i.e. after either a call of special purpose mst '`exit`', after an error during program interpretation, or after the last deliberation cycle was performed). Additionally, as a purely technical feature, also a notification after each deliberation cycle is provided. It should serve to strictly technical purposes like e.g. possible cleaning of a query cache, in the case a KR module implements such an optimization technique.

The core of *Jazzyk* syntax are rules of conditional nested mst's of the form $query \longrightarrow mst$. These are translated in *Jazzyk* as "`when <query> then <mst>`". Mst's can be joined using a sequence '`,`' and choice '`;`' operators corresponding to *BSM* operators $\circ$ and $|$ respectively. The operator precedence can be managed using braces '{', '}', resulting in an easily readable nested code blocks syntax. The query formulae are a straightforward translation of *BSM* query syntax.

Each KR module provides a set of named query and update operators, identifiers of which are used in primitive query and update expressions. To allow

the interpreter to distinguish between arbitrary strings and variable identifiers in primitive query and update expressions, *Jazzyk* allows optional explicit declaration of a list of variables used in them.

A standalone update expression is a shortcut for a *BSM* rule of the type $\top \rightarrow$ `<update>`. An obvious syntactic sugar of "`when-then-else`" conditional mst is introduced as well. Moreover, the syntax accepted by the *Jazzyk* interpreter includes a powerful macro language enabling support for higher level code structures, like e.g. named mst's with optional arguments. Such extended features will be discussed below in Subsection 3.3. Right hand side of Figure 2 provides short example of a *Jazzyk* program implementing a part of the *Jazzbot* agent described later in Section 4.

3.2 Interpreter

The semantics of the *Jazzyk* interpreter closely follows the *BSM* semantics shown in Algorithm 1 with only few deviations: 1) query expressions are evaluated sequentially from left to right, 2) the KR modules are responsible to provide a single ground variable substitution for declared free variables of a true query expression, 3) before performing an update, all the variables provided to it have to be instantiated. Additionally, query operator invocations are not supposed to change the agent's mental state, however this is not possible to ensure technically on the level of the *Jazzyk* interpreter implementation.

The above listed simplifications of the original *BSM* semantics were introduced in order to make the process of agent program interpretation more efficient and more transparent to the programmer. The most important deviation from the original *BSM* semantics is the treatment of variable substitutions. In order to make evaluation of mst queries straightforward and efficient, a KR module is required to provide only a single variable valuation for a provided primitive query formula, if such exists. In the case of more possible valuations of such a non-ground query formula, the KR module is free to pick a suitable one[1].

3.3 Extended features and the interpreter implementation

Jazzyk interpreter was designed to provide a lightweight modular agent oriented programming language. Except for the *vertical* modularity, i.e. modularity in terms of possibility to use, re-use, or replace heterogeneous KR languages to handle agent's underlying knowledge bases, *Jazzyk* implementation support a *horizontal* modularity in terms of modularity of the source code. For a robust programming language it is desirable to provide syntactical means to manipulate large pieces of code easily. Composition of larger programs from smaller components is a vital means for avoiding getting lost in the so called "spaghetti code".

[1] As far as the precise mechanism is well documented by the KR language plug-in developer.

Agent program before preprocessing with M4 syntax:

```
declare module brain as ASP
declare module goals as ASP
declare module body as Nexuiz

notify goals on initialize [{
  stay_healthy. find_box.
}]

define('perceive', '
  when sense body($3) [{$1}]
  then add brain($3) [{$2}]
')

perceive('sonar wall', 'inFrontOfWall') ;
when believes goals [{stay_healthy}] then {
  ...
  perceive('body health X', 'health(X)', 'X')
}
```

Resulting pure Jazzyk program after macro expansion:

```
declare module brain as ASP
declare module goals as ASP
declare module body as Nexuiz

notify goals on initialize [{
  stay_healthy. find_box.
}]

when sense body() [{sonar wall}]
then add brain() [{inFrontOfWall}]
;
when believes goals [{stay_healthy}]
then {
  ...
  when sense body(X) [{body health X}]
  then add brain(X) [{health(X)}]
}
```

Fig. 2. Example of macro preprocessing. Program is a part of *Jazzbot* agent.

To support this horizontal modularity, *Jazzyk* interpreter integrates GNU M4[2], a powerful macro preprocessor. Before a *Jazzyk* program is fed to the interpretation cycle (Algorithm 1), its source code is fed to GNU M4 preprocessor to expand and interpret all the M4 specific syntactic constructs. This way, the language of *Jazzyk* programs is extended by the full M4 language syntax.

In terms of source code modularity, by integration of GNU M4 macro preprocessor into the *Jazzyk* interpreter, it gains several important features almost "for free": definition of macros and their expansion in the source code, possibility of a limited recursive macro expansion, conditional macro expansion, possibility to create code templates, handling file inclusion in a proper operating system path settings dependent way, limited facility for handling strings, etc.

The Figure 2 provides an example of a macro expansion mechanism. A reusable mst `perceive` is defined and subsequently used in different contexts of an agent program.

To simplify debugging of agent programs, *Jazzyk* interpreter implements a full-featured error reporting following the GNU C++ Compiler[3] error and warning reporting format, what allows an easier integration of the interpreter with IDE frameworks, or programmers' editors like e.g. Eclipse, Emacs, or Vim.

Technically, *Jazzyk* interpreter is implemented in C++ as a standalone command line tool. The KR modules are shared dynamically loaded libraries installed as standalone packages on a host operating system. When a KR module is loaded, the *Jazzyk* interpreter forks a separate process to host it. The communication between the *Jazzyk* interpreter and a set of the KR module sub-processes is facilitated by an OS specific shared memory subsystem. This allows loading

[2] `http://www.gnu.org/software/m4/`.
[3] `http://gcc.gnu.org/`

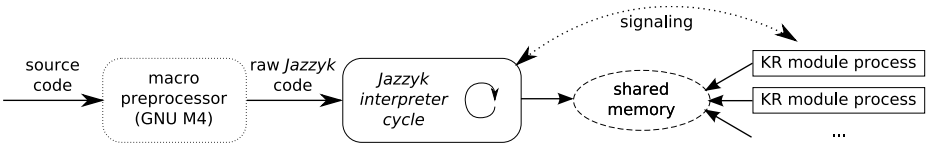

Fig. 3. *Jazzyk* interpreter scheme

multiple instances of the same KR module implemented in a portable way. The Figure 3 depicts the technical architecture of the *Jazzyk* interpreter.

The *Jazzyk* interpreter was implemented in a portable way, so it can be compiled, installed or relatively easily ported to most POSIX compliant operating systems. As of now, the interpreter was ported to Linux and Windows/Cygwin platforms. The *Jazzyk* interpreter was published under the open-source GNU GPL v2 license and is hosted at `http://jazzyk.sourceforge.net/`. To support implementation of 3rd party KR modules, I also published a KR module software development kit including template of a trivial KR module together with all compile/package/deploy scripts.

4 Jazzbot: demo application

To demonstrate the applicability of *Jazzyk* language and its interpreter and to further drive this line of research, we implemented *Jazzbot*, a virtual agent embodied in a simulated 3D environment of a first-person shooter computer game *Nexuiz*[4].

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 KB realized as an ASP logic program, the belief base is composed of two modules: ASP logic programming one and a *Ruby* module. The interface to the environment is facilitated by a *Nexuiz* game client module.

4.1 Answer Set Programming

Answer Set Programming module [8] provides the bot with non-monotonic reasoning capabilities. It is realized by a *Jazzyk* module which integrates an ASP solver *Smodels* [20] with accompanying logic program grounding tool *lparse* [19]. Hence the syntax and the semantics of logic programs the module handles, i.e. query/update formulae, is that accepted by *lparse* and *Smodels*. Query formulae query the answer sets (stable models) of the actual logic program in the knowledge base using two query operations: skeptic and optimistic. While the skeptic query requires a query formula to be true in all the models of the knowledge

[4] `http://www.alientrap.org/nexuiz`

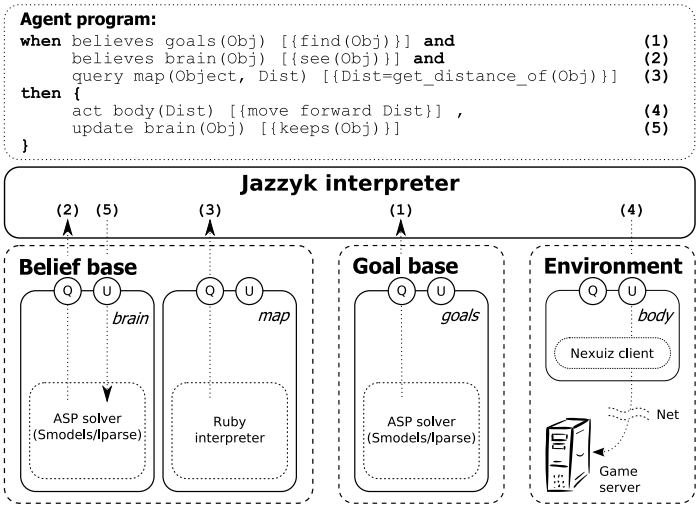

Fig. 4. Scheme of *Jazzbot*

base, the optimistic one requires only existence of at least one answer set satisfying the given query formula. The ASP KR module implements only a naive LP update mechanism based on updating facts.

4.2 Ruby

For representation of topological knowledge about the environment we chose an interpreted object-oriented programming language *Ruby*[5]. The *Ruby* module features a simple query/update interface allowing evaluation of arbitrary *Ruby* expressions. The functionality of the *Ruby* KR module resembles an interactive mode of the *Ruby* interpreter in which a user enters an arbitrary programming language expression on the command line and the interactive interpreter executes it and returns its value. The query/update formulae variables are bound to *Ruby* global name-space variables.

4.3 Nexuiz

The environment, *Jazzbot* operates in, is provided by a remote *Nexuiz* server. *Nexuiz* is an open-source 3D first-person shooter computer game based on the *Quake DarkPlaces*[6] engine. The *Nexuiz* KR module [10] implements a client functionality and facilitates the bot's interaction with the game server. *Jazzbot* can exploit several virtual sensors: gps, sonar, eye, compass, surface sensor and health status sensor, as well as effectors of its virtual body allowing it to move, jump, turn, use an item, attack, or utter a plain text message.

[5] `http://www.ruby-lang.org/`
[6] `http://icculus.org/twilight/darkplaces/`

Jazzbot is a client-side bot. That means, that in order to faithfully mimic the human player style environment for the bot, the sensory interface is designed so, that it provides only a (strict) subset of the information of that a human game player can access. For instance, *Jazzbot* can only check the scene in front of it using the directional sonar sensor. The rendering of a whole scene also is inaccessible to it, so only a single object can be seen at a time. Similarly to a human player, *Jazzbot* can reach only to the local information about its environment and information about objects which it cannot see, or are located behind the walls of the space it stands in, are inaccessible to it.

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 4 depicts the architecture of *Jazzbot* and features a *Jazzyk* code chunk implementing a simple behaviour of picking up an object by mere walk through it and then keeping notice about it in its ASP belief base. Note that all the three 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.

5 Discussion

In my view, an agent programming language is a *glue* for assembling agent's behaviours. Furthermore, it should facilitate an efficient use of its knowledge bases and interface(s) to the environment.

However, a programming language is a software engineering tool, in the first place. Even though its primary utilization is to provide expressive means for behaviour encoding, at the same time it has to fulfill requirements on modern programming languages. Programs have to be easily readable and understandable and the language semantics should be transparent to a programmer, i.e. as clear and simple as possible.

The *BSM* framework, and in turn *Jazzyk*, its implementation, is an attempt to satisfy these requirements in a working system obeying design principles of *simplicity*, *modularity* and *semantic transparency*.

1. *BSM* in the core allow implementation of agent programs in a form of simple non-deterministic reactive behaviours. Their precedence and relations can be steered by nesting of behaviours (mst's) and their combinations by operators of non-deterministic choice and chaining,
2. *Jazzyk* itself is a lightweight language. To support modularity and further extensibility, it exploits a power of a macro preprocessor allowing implementation of code templates and higher level syntactic constructs like e.g. general purpose perception or goal handlers (as sketched in the Figure 2),
3. finally, the proposed simple semantics of *BSM* stems from that of Gurevich's *Abstract State Machines* (*ASM*) framework, formerly known as *Evolving Algebras*. This relationship allows further transfer of *ASM* extensions and modeling tools, like logic for *ASM* to the *BSM* framework.

It can be argued that *Jazzyk* is oversimplified and does not follow the popular tradition of *BDI* [18] architectures. We already addressed these issues in [14]. There we showed how a *BDI* agent architecture can be implemented in a modular way in a framework close to *BSM* with an advantage, that an agent system designer has a freedom to implement a model of rationality suitable for the agent application instead of fixing it in the programming framework.

The syntactical structure of *BSM* closely resembles the one we introduced in [15]. *BSM* framework is indeed an evolution of our previous work. However the semantics of the language introduced in [15] was not simple enough and did not allow a straightforward implementation of a transparent language interpreter. Moreover, the concept of *mental state transformer* was still quite complex what led to problems with implementation of source code modularity in the language.

Our research project follows the spirit of [9], where Laird and van Lent argue that approaches for programming intelligent agents should be tested in realistic and sophisticated environments of modern computer games. To provide a substance to claims about practicality and applicability of *Jazzyk*, similarly to [21], we put *Jazzyk* to a test in such a challenging environment and we developed *Jazzbot*, a functional demonstration of a non-trivial virtual agent. We report on the details of the methodology of programming the *Jazzbot*'s behaviour and our experience with it more extensively in [16]. Because of *Jazzyk*'s modularity in terms of employed KR technologies, agent applications, such as *Jazzbot*, can be used as a test-bed for investigating applications of various KR technologies in the domain of agent systems.

6 Related work and conclusion

The landscape of agent programming frameworks is thriving (see e.g. a survey [1], or [2]). Most of the state-of-the-art frameworks like *3APL* [5], *Jason* [3], *GOAL* [6] and other provide a clear semantics of a resulting agent system. However, for representation of agent's beliefs, they usually provide a fixed, logic based knowledge representation technique (often *Prolog*). Following the *BDI* tradition, from the relation of agent's beliefs and goals stems a subsequent need to implement also the goal base using a related logic based KR technology. Unlike the *BSM* framework, which was designed with the motivation to allow a liberal combination of heterogeneous KR technologies in a single agent system, they do not allow a straightforward employment of e.g. an object-oriented KR approach (like *Ruby* in the case of *Jazzbot*) in one of an agent's knowledge bases.

Recently in [7], we showed that *GOAL* does not strictly commit to a single logic-based KR technology, such as e.g. *Prolog*. However, a question remains how difficult would it be to use heterogeneous KR technologies with *GOAL* as it is done in the *BSM* framework. Because of the model of rationality *GOAL* uses (blind commitment), there must be a close relationship between the KR languages of belief and goal bases. *BSM* do not require such a relationship to exist, it is rather a task of a programmer to encode such a relationship whenever

necessary. Macro `perceive` in the Figure 2 provides such an example: it relates a perception to its projection in the agent's belief base.

In [17], authors describe *Qsmodels* architecture based *Quake* bots implemented in plain ASP/*Smodels*. *Qsmodels* bots use planning as the primary approach to implementation of behaviours. I rather take a position that logic-based techniques are better suited for modeling static aspects of an environment, rather than for steering agents' behaviours. Unlike *Qsmodels* planning bot, *Jazzbot* is a rather reactive agent with a strong support for deliberative features.

The main contributions of this paper are a detailed description of the programming language *Jazzyk* together with its interpreter and a rough overview of the functionality of *Jazzbot*, a case study demonstrating applicability of *Jazzyk* language. The *Jazzbot* project is a driver for my future work. In this context I will focus on development of techniques for programming agents based on the template of *Jazzbot*, so that I can better understand a methodology for programming such systems. The aim is to design at least a fragmentary formal higher level specification language based on a flavor of modal logic, which would allow a straightforward translation (compilation) into raw *Jazzyk* programs. On a technical side, to complement the current family of *Jazzyk* KR modules, we plan to implement a *Prolog* module based on *SWI Prolog*[7] and a *Scheme* module based on *GNU Guile*[8]. We are also working on a module allowing inter-agent communication via an established MAS platform middleware.

7 Acknowledgments

I am grateful to Koen Hindriks for his support, constructive criticism and considerable contribution to simplification of the *BSM* semantics. Bernd Fuhrmann contributed to the development of the *Jazzyk* interpreter and Michael Köster with David Mainzer implemented the KR modules for the *Jazzbot* project.

References

1. 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.
2. 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.
3. 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* [2], 2005.

[7] `http://www.swi-prolog.org/`
[8] `http://www.gnu.org/software/guile/`

4. Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
5. 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* [2], 2005.
6. 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.
7. Koen Hindriks and Peter Novák. Compiling GOAL Agent Programs into Jazzyk Behavioural State Machines. Sixth German Conference on Multi-Agent system Technologies, MATES 2008, September 23-26 2008.
8. Michael Köster. Implementierung eines autonomen Agenten in einer simulierten 3D-Umgebung - Wissensrepräsentation. Master's thesis, 2008.
9. John E. Laird and Michael van Lent. Human-level AI's killer application: Interactive computer games. *AI Magazine*, 22(2):15–26, 2001.
10. David Mainzer. Implementierung eines autonomen Agenten in einer simulierten 3D-Umgebung - Interaktion mit der Umwelt. Master's thesis, 2008.
11. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.
12. Peter Novák. An open agent architecture: Fundamentals. Technical Report IfI-07-10, Department of Informatics, Clausthal University of Technology, November 2007.
13. 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.
14. Peter Novák and Jürgen Dix. Modular BDI architecture. In Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors, *AAMAS*, pages 1009–1015. ACM, 2006.
15. Peter Novák and Jürgen Dix. Adding structure to agent programming languages. In Mehdi Dastani, Amal El Fallah-Seghrouchni, Alessandro Ricci, and Michael Winikoff, editors, *ProMAS 2007*, Fifth International Workshop on Programming Multi-Agent Systems, Hawaii, USA, 2007.
16. Peter Novák and Michael Köster. Designing goal-oriented reactive behaviours. In *Proceedings of the 6th International Cognitive Robotics Workshop, CogRob 2008, July 21-22 in Patras, Greece*, July 2008.
17. Luca Padovani and Alessandro Provetti. Qsmodels: ASP planning in interactive gaming environment. In José Júlio Alferes and João Alexandre Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 689–692. Springer, 2004.
18. Anand S. Rao and Michael P. Georgeff. An Abstract Architecture for Rational Agents. In *KR*, pages 439–449, 1992.
19. T. Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report B18, Digital Systems Laboratory, Helsinki University of Technology, October 1998.
20. Tommi Syrjänen and Ilkka Niemelä. The Smodels System. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, pages 434–438. Springer, 2001.
21. Michael van Lent, John E. Laird, Josh Buckman, Joe Hartford, Steve Houchard, Kurt Steinkraus, and Russ Tedrake. Intelligent agents in computer games. In *AAAI/IAAI*, pages 929–930, 1999.