# Behavioural State Machines: Programming Modular Agents

**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 application domains 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 an agent programming framework of *Behavioural State Machines*, with *Jazzyk*, an implemented programming language interpreter for *BSM*. The presented framework 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.

## Motivation

An agent is a *situated* and *embodied* software entity, which *autonomously* acts in its environment, *proactively* pursues its (or its user's) goals and *reacts* to changes of, and events in, its environment (Wooldridge 2000). In open multi-agent systems agents operate in highly *dynamic* and often *unstructured* environments with *incomplete information* about, and at best only a *partial control* of it. Therefore, similarly to creation of intelligent robots, design and implementation of agent systems capable of operating in such environments poses many challenges to AI research.

According to the Shoham's seminal paper (Shoham 1993), a complete *AOP programming framework* should include three main components: 1) a formal language for describing a mental state of an agent, 2) an interpreted programming language for defining agent programs faithful to the semantics of mental states, and 3) an "agentifier" converting neutral devices into programmable agents. Obviously, the central theme, around which the other components are built, is a language for representing agent's knowledge about the world, i.e. its mental state.

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.

The dynamics of an environment leads to difficulties with control of an agent. Unexpected events and changes can interrupt the execution of complex behaviours, or even lead to a failure. Therefore an agile agent has to be able to reactively switch its behaviours according to the actual situation. While achievement of long-term goals requires rather algorithmic behaviours, reaction to interruptions has to be immediate. Moreover, due to only partial accessibility of the environment, some situations can be indistinguishable to the agent. Hence it is vital to allow reactive non-deterministic choice between several potentially appropriate behaviours, together with arbitration mechanisms for steering the selection.

An agent programming language is a *glue* for assembling agent's behaviours which facilitate an efficient use of its knowledge bases and interface(s) to the environment. 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 high requirements on modern programming languages. Programs have to be easily readable and understandable. In order to allow for a rigorous study of theoretical properties of resulting agent systems, the semantics must be simple and transparent, without hidden mechanisms, yet the language has to be flexible enough to allow a strong support for program decomposition and code re-use.

I take a liberal engineering stance to the design of agent programming frameworks. In this report I describe a generic agent oriented architecture of *Behavioural State Machines* (Novák 2007), together with a brief discussion of *Jazzyk*, an implemented programming language for *BSM*. Finally, I also discuss appropriateness of the *BSM* framework for agent programming and provide a brief overview of an ongoing and future work on it. The introduced ideas are illustrated on a running example of an office space robot using

Answer Set Programming (see e.g. (Baral 2003)) to represent its beliefs about the environment.

## Behavioural State Machines

*Behavioural State Machine* is a general purpose computational model based on the Gurevich's *Abstract State Machines* (Börger & Stärk 2003), adapted to the context of agent oriented programming.

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*, or *3APL* (Bordini *et al.* 2005). States are agent's mental states, i.e. collections of partial states of its *KR modules* (agent's partial knowledge bases) together with a state of the environment. Transitions 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*.

*BSM* framework was first presented in (Novák 2007), therefore some technical details are omitted here and I focus on a description of the most fundamental issues.

### Syntax

Because of the openness of the introduced architecture, knowledge representation components of an agent are kept abstract and only their fundamental characteristics are captured by formal definitions.

Basically, a KR module has to provide a language of query and update formulae and two sets of interfaces: *entailment* 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 $\mathcal{L}$ over some domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ 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, when they include variables over the same domain $\mathcal{D}$ and their sets of query and update operators are mutually disjoint. KR modules with compatible KR languages are compatible as well.

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, infix notation is used (e.g. $\sigma \models \varphi$, or $\sigma \oplus \varphi$).

**Example 1** ***(running example)*** *Consider an office space robot using two knowledge bases $\mathfrak{B}$ and $\mathfrak{G}$ to keep track of its beliefs and goals implemented using Answer Set Programming and Prolog respectively. Additionally, the robot features an interface $\mathfrak{C}$ to its body realized in Java.*

*$\mathfrak{B} = (2^{AnsProlog^*}, AnsProlog^*, \{\models_{ASP}\}, \{\oplus_{ASP}, \ominus_{ASP}\})$ is a KR module realizing an ASP knowledge base. The underlying language is $AnsProlog^*$ (Baral 2003). It includes variables over atoms and function symbols. A set of states are all well-formed $AnsProlog^*$ programs (sets of clauses). There is a single query and two update operators. Query operator $\models_{ASP}$ corresponds to the skeptical version of entailment in ASP, i.e. $P \models_{ASP} \varphi$ iff $\varphi$ is true in all answer sets of the program $P$. The two primitive update operators $\oplus_{ASP}$ and $\ominus_{ASP}$ stand for an update by and retraction of an $AnsProlog^*$ formula (a partial program) to/from the knowledge base.*

*A KR module connecting the agent with its environment (its body) $\mathfrak{C} = (\Sigma_{JavaVM}, Java, \{\models_{eval}\}, \{\oplus_{eval}\})$ is a formalization of an interface to a running Java virtual machine. The set of Java KR module states $\Sigma_{JavaVM}$ is an abstraction of all states of memory of a running VM (initialized by loading of a Java program). Both query and update operators $\models_{eval}$, $\oplus_{eval}$ take a Java expression and execute it in the context of the VM. The query operator $\models_{eval}$ returns $\top$ iff a Java expression $\phi$ evaluates to True, otherwise it returns $\bot$. Only expressions not modifying the actual module state are allowed as query formulae.*

*Finally, $\mathfrak{G} = (2^{Prolog}, Prolog, \{\models_{Prolog}\}, \{\oplus_{Prolog}\})$ is a Prolog-based KR module capturing robot's goals. The set of all Prolog programs represents the set of states. Both the entailment operator $\models_{Prolog}$ and the update operators $\oplus_{Prolog}$ correspond to the usual Prolog query evaluation.*

*Modules $\mathfrak{B}$, $\mathfrak{C}$ and $\mathfrak{G}$ are compatible. Both $AnsProlog^*$ and Prolog include variables over terms, i.e. strings of alphanumeric characters, which is also a basic type of Java.*

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 $\phi_1, \phi_2$ 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 language expression $\varphi \in \mathcal{L}$ 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 that state as well.

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. $\mathbf{skip}$ *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 $\phi$ is a query expression, and $\tau$ is a mst, then $\phi \longrightarrow \tau$ is a mst as well (conditional),*
4. *if $\tau$ and $\tau'$ 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. 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 $\mathcal{A}$.

## Semantics

As was sketched above, 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 $\sigma$. 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, i.e. they are side effects free.

To evaluate a formula in a *BSM* 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 its corresponding domain. A variable substitution $\theta$ is *ground* w.r.t. $\phi$, 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 $\sigma$ is a tuple $(\oplus, \psi)$, where $\oplus$ is an update operator and $\psi$ 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 $\rho_1$ and $\rho_2$ are updates, also a sequence $\rho_1 \bullet \rho_2$ is an update. A sequence of update sets $\nu_1 \bullet \nu_2$ yields all possible sequential combinations of primitive updates from these sets $\nu = \{\rho_1 \bullet \rho_2 | (\rho_1, \rho_2) \in \nu_1 \times \nu_2\}$.

A simple update corresponds to the semantics of a primitive mst. Sequence of updates corresponds to a sequence of primitive mst's. 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 $\mathcal{A}$.*

*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 semantics of a mst in terms of an update set is determined by the following calculus:

$$\frac{\top}{yields(\mathbf{skip}, \sigma, \theta, \emptyset)} \qquad \frac{\top}{yields(\oplus\psi, \sigma, \theta, \{(\oplus, \psi\theta)\})} \quad \text{(primitive)}$$

$$\frac{yields(\tau, \sigma, \theta, \nu),\ \sigma \models \phi\theta}{yields(\phi \longrightarrow \tau, \sigma, \theta, \nu)} \qquad \frac{yields(\tau, \sigma, \theta, \nu),\ \sigma \not\models \phi\theta}{yields(\phi \longrightarrow \tau, \sigma, \theta, \emptyset)} \quad \text{(condition)}$$

$$\frac{yields(\tau_1, \sigma, \theta, \nu_1),\ yields(\tau_2, \sigma, \theta, \nu_2)}{yields(\tau_1 | \tau_2, \sigma, \theta, \nu_1 \cup \nu_2)} \quad \text{(choice)}$$

$$\frac{yields(\tau_1, \sigma, \theta, \nu_1 \neq \emptyset),\ \forall \rho \in \nu_1 : yields(\tau_2, \sigma \bigoplus \rho, \theta, \nu_\rho)}{yields(\tau_1 \circ \tau_2, \sigma, \theta, \bigcup\limits_{\forall \rho \in \nu_1} \{\rho\} \bullet \nu_\rho)} \quad \text{(sequence)}$$

$$\frac{yields(\tau_1, \sigma, \theta, \emptyset),\ yields(\tau_2, \sigma, \theta, \nu_2)}{yields(\tau_1 \circ \tau_2, \sigma, \theta, \nu_2)} \quad \text{(sequence)}$$

A mental state transformer $\tau$ of a *BSM* $\mathcal{A}$ yields an update set $\nu$ in a state $\sigma$ under a variable substitution $\theta$, iff $yields(\tau, \sigma, \theta, \nu)$ is derivable.

The *functional view* on a mst is the primary means of compositional modularity in *BSM*. Mental state transformers encode functions yielding update sets over states of a *BSM*. Primitive mst $\mathbf{skip}$ results in an empty update set, while a proper update expression yields a singleton update set. The semantics of a conditional mst is provided for two cases according to a validity of the query condition. If the left hand side query condition holds, the resulting update set corresponds to that of the right hand side mst. Otherwise, the semantics of a conditional mst is equivalent to $\mathbf{skip}$. Conditional mst's provide a means for mst specialization (w.r.t. their applicability) and facilitate syntactical nesting of *BSM* code blocks.

A non-deterministic choice of two, or more mst's denotes a function yielding a unification of their corresponding update sets. Additionally, a sequence of mst's allows a fine-grained steering of the update selection.

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

**Definition 6** ***(BSM semantics)*** *A BSM* $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ *can make a step from state* $\sigma$ *to a state* $\sigma'$ *(induces a transition* $\sigma \rightarrow \sigma'$*), if there exists a ground variable substitution* $\theta$*, s.t. the agent program* $\mathcal{P}$ *yields a non-empty update set* $\nu$ *in* $\sigma$ *under* $\theta$ *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*.

## Implementation

In order to practically experiment with *BSM* agents, I developed *Jazzyk*, an implemented programming language of *Behavioural State Machines* with an accompanying interpreter.

Each *BSM* KR module provides a set of named query and update operators, identifiers of which are used in primitive query/update expressions. Invocations of KR module operators take the form "`<operation> <module> '[{' <expression> '}]'` ". Compound query expressions are constructed from primitive ones and brackets.

The core of *BSM* syntax are conditional nested rules 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. Finally, syntactic sugar of "`when-then-else`" conditional mst is introduced as well.

Figure 1 lists an example of a *Jazzyk* code for the robot from Example 1. In the normal mode of operation, the robot moves randomly around and when interrupted by somebody, it smiles and utters a greeting. When it detects low battery alert, it switches off all the energy consuming features and tries to get to a docking station, where it can recharge.

The semantics of the *Jazzyk* interpreter is that of *BSM* with few simplifications to allow for an efficient computation: 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, and 3) before performing an update, all the variables provided to it must be instantiated.

```
/* Initialization */
declare module beliefs as ASP /* initialization omitted */
declare module goals as Prolog /* initialization omitted */
declare module body as Java /* initialization omitted */

/* Perceptions handling */
when sense body (X,Y) [{ GPS.at(X,Y)}]
then adopt beliefs [{ at(X,Y) }] ;

/* Default operation */
when sense body [{ (Battery.status() == OK) }] then {
    /* Move around */
    perform body [{ Motors.turn(Rnd.get(), Rnd.get()) }] ;
    perform body [{ Motors.stepForward() }]
} else
{
    /* Safe emergency mode – degrade gracefully */
    perform body [{ Face.smile(off) }] ;
    perform body [{ InfraEye.switch(off)}] ;
    update goals [{ assert(dock) }]
} ;

/* Goal driven behaviour */
when query goals [{ dock }] then {
        when query beliefs (X,Y) [{ position(dock_station, X, Y) }]
        then {
            perform body (X,Y) [{ Motors.turn(X,Y) }] ;
            perform body (X,Y) [{ Motors.stepForward() }]
        }
} ;

/* Commitment handling */
when query goals [{ dock }] and
      query beliefs [{ position(dock_station, X, Y), at(X,Y) }]
then update goals [{ retract(dock) }] ;

/* Interruption handling */
when sense body (X) [{ Visual.see(X) }] and
      query beliefs (X) [{ friend(X), not met(X) }]
then {
    perform body [{ Face.smile(on) }] ,
    perform body [{ Audio.say("Hello!") }] ,
    adopt beliefs (X) [{ met(X) }]
}
```

Figure 1: Example of an office space robot agent.

## Agent oriented programming

The plain formalism of *Behavioural State Machines*, does not feature any of the usual characteristics of rational agents (Wooldridge 2000), such as *goals*, *beliefs*, *intentions, commitments* and alike, as first class citizens. In the following, I discuss the example in Figure 1 and demonstrate how some features, desirable for rational agents, are implemented in *Jazzyk* using the *BSM* framework.

**Heterogeneous KR** The agent uses three KR modules corresponding to those introduced in the Example 1: `beliefs` - an ASP module representing agent's beliefs ($\mathfrak{B}$ in Example 1) with a query operator `query` and two update operators `adopt`, `drop`, corresponding to $\models_{ASP}$,

$\oplus_{ASP}$ and $\ominus_{ASP}$; `body` - a *Java* module for interfacing with the environment ($\mathfrak{C}$), providing query and update operators `sense` and `perform` ($\models_{eval}$ and $\oplus_{eval}$) ; and `goals` - a *Prolog* module to represent goals ($\mathfrak{G}$), with operators `query` and `update` ($\models_{Prolog}$ and $\oplus_{Prolog}$).

The robot in Figure 1 uses two different knowledge bases (KB) to store information representing its mental attitudes: beliefs and goals. While it employs only a single belief base to store the information about the world, in general it might be useful to employ several KBs using heterogeneous KR technologies. Instead of fixing the structure of an agent system, it should be a programmer who chooses a number, roles and relations between the KBs. The *BSM* framework allows an easy integration of heterogeneous KBs in a transparent and flexible way.

**Situatedness and embodiment** Interaction with an environment is facilitated via the same mechanism as handling of various knowledge bases. The essence of an interface to an environment are *sensor* and *effector* operators. The scheme is the same as for query and update interfaces of a pure KR module. Metaphorically, in line with behavioural roboticists, we could say that KR module for interfacing with an environment uses the world as its own representation (Brooks 1991). Moreover, given the flexibility of *BSM* framework, an agent can easily interface with various aspects of its environment using different modules. Social and communicating agents can use specialized modules to interface with social environments they participate in and at the same time perform actions in other environments they are embodied in.

In our example, the robot interacts with its environment via the module `body`. Figure 1 shows examples of perception handling and performing actions.

**Reactiveness, scripts and priorities** The model of *Behavioural State Machines* is primarily suitable for mixing of non-deterministic choice of reactive and script-like, so called ballistic (Arkin 1998), behaviours. An example of such is listed in Figure 1, in the part "Interruption handling", where the robot first smiles, says "*Hello!*" and finally records a notice about the event. Such sequential, or script-like behaviours can pose a problem if an agent performs more than one exogenous action in a sequence. If the subsequent action depends on the previous one, which can possibly fail, the whole script can fail.

A *BSM* agent program effectively forms a syntactical tree with inner nodes of *AND* and *OR* types, corresponding to operators $\circ$ and $|$ respectively. This allows efficient steering of the mst selection process and even introducing priorities.

Consider an agent program of the form:

**when** $\phi_1$ **then** $<mst_1>$ ,
**when** $\phi_2$ **then** {
  **when** $\phi_{2,1}$ **then** $<mst_{2,1}>$ ,
  **when** $\phi_{2,2}$ **then** $<mst_{2,2}>$
} ,
. . . ,
**when** $\phi_n$ **then** $<mst_n>$

At the top level, sequence of conditional mst's encodes an ordered set of behaviours applicable in possibly indistinguishable situations, ordered according to the priorities a programmer assigned to them. Additional nesting, as seen in the second mst, allows for even a finer grained control of the agent's deliberation cycle.

**Goal driven behaviour and commitment strategies** The robot in the Figure 1 uses goals to steer its behaviours. Goals are used to keep a longer-term context of the agent (docking), yet still allow it to react, i.e. change the focus to unexpected events (meeting a friend), in an agile manner.

Goals come with a certain commitment strategy. Different types of goals require different types of commitments (e.g. achievement vs. maintenance goals). The *BSM* model is quite flexible w.r.t. commitment strategy implementation. In the presented example, the commitment w.r.t. the goal $\varphi =$ `dock` can be informally written as an LTL formula $\Box(\mathfrak{G}\varphi \wedge \mathfrak{B}\varphi \Longrightarrow \Diamond\neg\mathfrak{G}\varphi)$.

Different commitment strategies can be implemented in the *BSM* model. The study of various types of commitment strategies and formal specification methods for *BSM* will be, however, a subject of our future work.

## Discussion and related work

The primary motivation for development of computational model of *Behavioural State Machines* is my research towards studying the applicability of non-monotonic reasoning techniques in the context of cognitive agent systems and cognitive robotics. Development of *BSM* framework was therefore driven by a resulting need for an architecture supporting 1) integration of heterogeneous knowledge representation technologies in a single agent system and 2) flexibility w.r.t. various types of behaviours.

The architecture of *BSM* is *open*, *modular* and *pragmatic*. I.e. one not dictating a programmer ways to implement an agent, especially w.r.t. internal structure of its mental state, yet allowing him to freely exploit techniques at hand, even if that would mean a bad practice. It should be a choice of KR technologies and a set of *methodological guidelines*, which lead a programmer to a design of a practical agent system, rather than a domain independent choice made by creators of a programming framework.

The *BSM* framework is a culmination of the line of research launched by *Modular BDI Architecture* (Novák & Dix 2006). It borrows the idea and the style of KR modules integration from this previous work (Novák & Dix 2006), while its theoretical foundations stem rather from the well studied general purpose computational model of *Abstract State Machines* (*ASM*) (Börger & Stärk 2003).

In particular, the *BSM* framework can be seen as a special class of *ASM*, with domain universes ground in KR modules, lacking parallel execution to maintain atomicity of transition steps and featuring specialized type of sequences of updates. The close relationship to the formalism of *ASM*, allows an easy transfer of various *ASM* extensions, such as turbo, distributed, or multi-agent *ASM* (Börger & Stärk 2003), to *BSM*

framework. Moreover, *ASM* formalism comes with a specialized modal logic for reasoning about *ASM* programs, what we hope to exploit in the study of formal specification as well as verification methods for *BSM*.

*IMPACT* (Subrahmanian *et al.* 2000) system features a similar degree of freedom, w.r.t. heterogeneous KR technologies, as the *BSM* framework. It was designed to support integration of heterogeneous information sources as well as agentification of legacy applications. *IMPACT* agent consists of a set of *software packages* with a clearly defined interface comprising a set of data types, data access functions and type composition operations. An agent program is a set of declarative rules involving invocations of the underlying components via the predefined data access functions. While a *BSM* program encodes a merely non-deterministic choice of conditional update expressions, and thus facilitates reactive behaviours of the agent, various semantics of *IMPACT* are strictly grounded in declarative logic programming. For a more thorough discussion of the related work see the original technical note (Novák 2007).

As far as the original Shoham's definition of an AOP framework is concerned, *BSM* framework intentionally avoids to deal with the language for describing agent's mental states. Instead it focuses more on the programming language for its behaviours. KR language, as well as the semantics of KR modules is left open. However, through the requirement of providing query operators, the existence of a well defined semantics of the underlying knowledge base is secured. The *BSM* programming language is merely a tool for design of a variety of interactions between the KR modules of an agent. Moreover, by adding an appropriate query/update interface to a legacy system, e.g. a relational database, it can be easily wrapped into a service agent envelope, or made a part of a more sophisticated agent system, i.e. "agentified" in the very sense of Shoham's *agentification*.

## Conclusion

In the presented report I describe the theoretical framework of *Behavioural State Machines* originally introduced in (Novák 2007). *BSM* is an architecture for programming flexible and robust hybrid agents, exploiting heterogeneous KR technologies and allowing an easy agentification of low level interfaces and information sources. Its semantics is primarily obeying principles of *KR modularity*, and *flexibility* in terms of encoding *reactive*, as well as *sequential* behaviours. It draws a strict distinction between the *representational* vs. *behavioural* aspects of an agent and primarily focuses on the later. While agent's deliberation resides in its KR modules, a *BSM* agent program encodes its behaviours.

I also discuss *Jazzyk*, an implemented programming language for *BSM* framework. The implemented interpreter, together with an initial set of KR modules was released in late 2007 under GNU GPL license. The project website is hosted at `http://jazzyk.sourceforge.net/`.

In order to substantiate the presented theoretical framework in a real world experiment and drive the future research, we are currently intensively working on a showcase agent system similar to the one described in (van Lent *et al.* 1999): a non-trivial BDI-based bot in a simulated 3D environment of a *Quake*-based computer game using ASP solver *Smodels* (Syrjänen & Niemelä 2001) in its belief base, while the bot's behaviours are implemented in *Jazzyk*. *Smodels* plug-in serves loosely as the agent's deliberation component and the control loop is facilitated by the *Jazzyk* interpreter.

In the future, I will focus on studying higher level formal agent specification methods based on modal logic, which allow automatic translation into the plain language of *BSM*. I briefly touched on this issue in the discussion of goal oriented behaviours and agent commitment strategies. This research will be pragmatically driven by the already mentioned showcase demo application.

## References


Arkin, R. C. 1998. *Behavior-based Robotics*. Cambridge, MA, USA: MIT Press.

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

Bordini, R. H.; Dastani, M.; Dix, J.; and Seghrouchni, A. E. F. 2005. *Multi-Agent Programming Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers.

Börger, E., and Stärk, R. F. 2003. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer.

Brooks, R. A. 1991. Intelligence without representation. *Artif. Intell.* 47(1-3):139–159.

Novák, P., and Dix, J. 2006. Modular BDI architecture. In Nakashima, H.; Wellman, M. P.; Weiss, G.; and Stone, P., eds., *AAMAS*, 1009–1015. ACM.

Novák, P. 2007. An open agent architecture: Fundamentals (revised version). Technical Report IfI-07-10, Department of Informatics, Clausthal University of Technology.

Shoham, Y. 1993. Agent-oriented programming. *Artif. Intell.* 60(1):51–92.

Subrahmanian, V. S.; Bonatti, P. A.; Dix, J.; Eiter, T.; Kraus, S.; Ozcan, F.; and Ross, R. 2000. *Heterogenous Active Agents*. MIT Press.

Syrjänen, T., and Niemelä, I. 2001. The Smodels System. In Eiter, T.; Faber, W.; and Truszczynski, M., eds., *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, 434–438. Springer.

van Lent, M.; Laird, J. E.; Buckman, J.; Hartford, J.; Houchard, S.; Steinkraus, K.; and Tedrake, R. 1999. Intelligent agents in computer games. In *AAAI/IAAI*, 929–930.

Wooldridge, M. 2000. *Reasoning about rational agents*. London: MIT Press.