Framework for model-based design of multi-agent systems

František Zbořil Jr.*, Vladimír Janoušek, Radek Kočí, František Zbořil and Zdeněk Mazal

Department of Intelligent Systems
Brno University of Technology, Czech Republic
E-mail: zborilf@fit.vutbr.cz
E-mail: janousek@fit.vutbr.cz
E-mail: koci@fit.vutbr.cz
E-mail:zboril@fit.vutbr.cz
E-mail: mazal@fit.vutbr.cz
*Corresponding author

Abstract: When developing a Multi-Agent System (MAS), we need, among others, to build agent architectures together with some decision algorithms and communication protocols. Before such systems are realised in a real world, we should sufficiently test and verify their behaviour. The presented text describes a new approach to building artificial agents and MASs using the methodology of model-based design. In this methodology some models are used for testing the behaviour of particular elements as well as for testing the behaviour of the system as a whole – in both cases by simulation. Our aim is to develop a tool that would allow model-based design of systems with artificial agents. For this reason we have been building an application called Tool for Multi-Agent System Simulation (T-Mass) which is aimed at model-based development. As a part of the tool we also developed a language called Agent Low-Level Language (ALLL), by which agents' behaviour is controlled.

Keywords: system modelling; agent systems; model-based design; programming languages.

Reference to this paper should be made as follows: Zbořil, F., Jr., Janoušek, V., Kočí, R., Zbořil, F. and Mazal, Z. (2009) 'Framework for model-based design of multi-agent systems', *Int. J. Autonomic Computing*, Vol. 1, No. 2, pp.140–162.

Biographical notes: František Zbořil Jr. is an Assistant Professor at the Faculty of Information Technology, Brno University of Technology, Czech Republic. His major interests comprise several areas related to artificial intelligence and modelling, mainly artificial agent architectures, multi-agent systems modelling and the realisation of such systems in the environment of Wireless Sensor Networks (WSNs). His actual research is aimed at the realisation of intelligent agent-based systems for runtime risk analysis through some particular reputations of WSN nodes.

Copyright © 2009 Inderscience Enterprises Ltd.

Vladimír Janoušek received his PhD degree from the Faculty of Information Technology, Brno University of Technology, Czech Republic in 1999. He is an Assistant Professor in the Department of Intelligent Systems at the same faculty. His research focuses on simulation-driven development, pure object orientation and reflective architectures.

Radek Kočí received his PhD degree from the Faculty of Information Technology, Brno University of Technology, Czech Republic in 2004. He is currently an Assistant Professor at the faculty. His research interest includes modelling and simulation in the context of software engineering, especially an application of formal models in the systems design methodology. He also cooperates on the modelling and simulation of agents and multi-agent systems.

František Zbořil is an Associate Professor of Computer Science at the Brno University of Technology, Czech Republic. He received his PhD degree in 1978 from the same university. His current research is focused on artificial intelligence, soft computing, and mathematical modelling and simulation.

Zdeněk Mazal received his MS degrese in Computer Science and Engineering from Brno University of Technology in Brno, Czech Republic in 2005. He is currently pursuing his PhD degree at the same university. He has published research papers at well-recognised international conferences, including the Intelligent Systems Design and Applications (ISDA) 2007 and 2009. His research interests include modelling multi-agent systems, Petri nets, the semantic web, *etc*.

1 Introduction

When working with Multi-Agent Systems (MASs), we usually have to deal with some kind of distributed system with nodes that show some appearance of intelligence. Recent understanding of the MAS is that there exists a net of agent platforms interconnected with some communication and transport channels. Such a network consists of an agent-friendly environment where intelligent agents can reside and act, eventually travelling among particular platforms.

The subject of artificial agents is not quite new. Artificial intelligence, as well as many other scientific areas, has taken artificial agents as part of their systems for more than 20 years. However, different areas understand the term ‘agent’ slightly differently. Here we deal with the artificial intelligent agent as it is understood in computer society. The contribution of this text is to introduce the principles of a new tool called Tool for Multi-Agent System Simulation (T-Mass) and demonstrate that such an approach to MAS modelling could bring advances in MAS development.

We first explain some basics of artificial agents and MASs in Section 2. We also discuss here some tools for MAS development and point out some advantages and weak spots of these tools. Then we introduce the T-Mass tools in Section 3. It contains sections about model elements and their functionalities as well as structures of MAS models when they are made in T-Mass. This section also describes an agent control language called Agent Low-Level Language (ALLL). Section 4 describes the principles of multi-agent model simulations and usage for model-based design. Section 5 gives some examples of

how to programme agents in ALLL, specifically, concretely how to make programs for agents driven by intentions. Then we conclude with some remarks about our expectations of the tool and outline our future work in this area.

2 Artificial agents, multi-agent systems and their models

To describe artificial agents, we use the definition that agents are autonomous entities working in mostly unknown and highly dynamic environments to achieve some objectives. An agent senses the environment through its sensors and acts in the environment rationally through its effectors. This means that its actions have a motivation, which is given by a goal. On the other hand, its architecture should not be very complex. In fact, an agent should have a rather simple structure that has full control over its behaviour and has the ability to act flexibly, swiftly and rationally.

To introduce today's most popular agent architectures, we distinguish between reactive and deliberative agents. Reactive agents in their pure form were introduced by Brooks (1991). He argued that there need not be a symbolic reasoning process to make agents act with some intelligence. Such agents use some models of behaviours which they perform when some relevant situations appear in the environment. Other approaches to artificial agent realisations use the so-called mental states. Reasoning based on agents' beliefs and obligations is used for the agent called Agent-0 by Shoham (1990). Also there are architectures based on intentions (originally developed by Bratman, 1987), extended into today's popular Belief-Desire-Intention (BDI) agent computational architectures (Rao and Georgeff, 1995; Rao, 1996).

In the field of MASs it is somehow quite difficult to distinguish between (multi)agent modelling tools and tools for agent system development. For example, 3APL and the 2APL system (its successor), the JADE framework, CYBELE and many others can be used when one develops systems with agents as well as when a multi-agent model is built. Of course, there are many other modelling techniques for the purposes of discrete parallel systems modelling, which are quite close to the MAS. Next we describe some popular tools and then outline our motivation for creating another model-based framework for the development of systems with artificial agents.

2.1 Overview of current tools for MAS modelling and development

First we discuss some widely known tools for agent development which we have mentioned already.

We start with the 2APL and JAM! tools. Both tools are based on the BDI theory. 2APL allows the making of plans in the form of production rules that could be applied to some goals or subgoals. This tool is also equipped with a platform corresponding to the Foundation for Intelligent Physical Agents (FIPA) specification.[1] JAM! is a language with a syntax similar to the Java language and it also allows the making of agents based on the BDI principles. These tools are suitable mainly when some agents with intention-driven behaviour should be realised and their behaviour in a multi-agent community checked out.

JADE is a Java framework and also an agent platform for building MASs in accordance with FIPA specifications. Besides the FIPA agent platform, it provides tools for the implementation of communication protocols, ontologies, behavioural procedures, *etc*. It is quite nice and probably the most popular tool today, but in its original version it does not support the development of mature agent architectures. Some extensions, such as JADEX, solve such disadvantages and introduce mechanisms for implementing, for example, the intention-driven agents.

Along with the original tools for agent development, the Petri nets could be mentioned here. At least we mention these nets for the reason that they represent a major stream in distributed system modelling. Today there is some effort to use Petri nets in agent modelling and realisation. For example, they are used by Ferber (1999) to demonstrate algorithms for distributed reasoning in multi-agent communities, agent communication protocols, coalition forming principles, *etc*. There is also a Petri net-based system for MAS realisation called MULAN, developed at the University of Hamburg (Cabac and Dörges, 2007). Furthermore there is an effort to develop rational agents with object-oriented Petri nets at Brno University of Technology (Zbořil and Kočí, 2007). But some disadvantages can be identified when Petri nets are brought into the world of rational agents, mainly that the expressivity of the Petri net model better suits distributed systems than particular agent architecture realisations. Moreover, agent systems need to have strict interfaces between agents and their environment. Also, some symbolic reasoning mechanisms used often in the field of artificial agents are not native to Petri nets.

2.2 Motivation for another agent tool development

In this section we outline the reasons why we have developed another agent systems modelling and development tool. We do this by summarising some points which are important for agents that are considered to work in systems like WSN. We start with the classical Wooldridge and Jennings (1995) characteristic of agents. We know that rational agents should be autonomous, proactive and possibly with some social capabilities. Taking into account assumed realisation in WSN the sensor nodes, which are usually called 'motes', we should add the following demands:

- The agent control programme should be as small as possible due to the minimal size of the motes dynamic memory (for example, the radio boards MICAz used by CrossBow motes contain an ATMEL128L microcontroller with just a 4kB dynamic memory).
- The agent control programme should be easily interpretable for the machine. Due to previous and to current development tools for the WSN like TinyOS, the agent program should be at quite a low level of abstraction.
- Its expressivity should correspond to today's modern approaches to agent development. The language should comprise agents' practical reasoning, communication among agents, reactivity to system changes and data storage and manipulation.
- The program and also the architectures of the agent and agent platform should reflect issues of the agents' mobility.

Our approach corresponds to these demands. Whole MAS models are built at a low level of abstraction and they are aimed to provide the possibility of creating many kinds of agents. The language called ALLL is easily interpretable and it has, as we show further, many features suitable for agent programming. Finally, the language allows straightforward implementation of platforms for such agents and the implementation can vary from microcontrollers, intelligent sensor nodes and embedded systems to classical computer-implementation environments.

3 T-Mass: a tool for multi-agent simulation

The previous sections have outlined that there are many tools for modelling and developing agents and that, despite that, we have made another one. Now we are going to introduce the basics of our discrete event modelling system, which we call T-Mass. Because this is a modelling tool, we start with some formal definitions of the models for which the tool is being developed.

3.1 Principles of T-Mass model

The classical approach to system modelling considers that a system is composed of a set of elements (universe) and a set of relations among these elements (characteristic). The model of a system is also a system, so the structure is the same. Naturally the situation is the same in the case of models for T-Mass. We first need to specify whatever is present in the model and then how these elements are interconnected. We use the usual way of formally describing a system. The system is a tuple

$$S = (U, R) \tag{1}$$

where U is the universe and R is the characteristic.

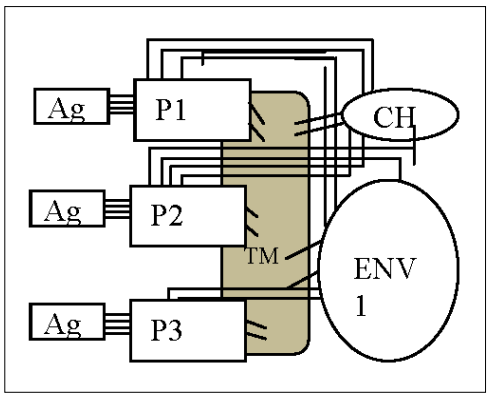
In fact, the whole model can be divided into several subsystems of two possible types. The first subsystem type is, according to FIPA, an agent platform. The second subsystem type is the environment. These two subsystems are both open – the platform system to the environment system and vice versa. Let us denote these subsystems as S^P for platforms and S^E for environments. Then one or more platforms and one or more environments denoted as $S_1^P, S_2^P ... S_n^P$ and $S_1^E, S_2^E ... S_m^E$ can exist in the T-Mass model. The whole model is then a system containing these subsystems which are interconnected with some input/output channels. We put (U_I^P, R_I^P) or (U_I^E, R_I^E) instead of the subsystem symbols, the symbol R^C for the denotation of a set of relations among particular subsystems and one special element U^T for a T-Mass control element. Now we can define the whole model as:

$$\begin{aligned} S = (&U^T \cup \bigcup_{i=1..n} U_i^P \cup \bigcup_{i=1..m} U_i^E, \\ &\bigcup_{i=1..n} R_i^P \cup \bigcup_{i=1..m} E_i^E \cup R^C). \end{aligned} \tag{2}$$

Such a system is a general specification of the MAS model. To demonstrate how we model the MAS for concrete problems, we need to closely describe particular subsystems, elements and structures of the system.

But first we introduce a model that can be created with the above mentioned elements. In fact, there can be interconnections only between a platform subsystem and an environmental subsystem. We do not allow interconnections of two environmental subsystems nor of two platform subsystems. An example of a model is provided in Figure 1.

Figure 1 Example of a model (see online version for colours)



This model is composed of one control element, two environmental elements and three platform subsystems. Each platform subsystem contains one platform element and one agent. There are two platforms interconnected with a communication element CH1. Every platform is connected to the environment ENV1. This means that only the agents Ag1 and Ag2 can communicate together, but all the agents share their environment.

3.2 The T-Mass control element

The T-Mass control element is responsible for controlling the whole simulation run. Every subsystem must be interconnected with this element and it executes these subsystems by sending some events to them. We will discuss the element closely in the section describing the T-Mass simulation run.

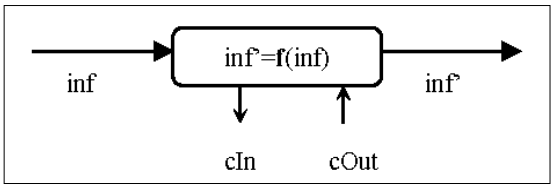
3.3 Environmental models

Environmental subsystems are mostly abstract to the MAS model designer. Except for several predefined environmental subsystems, it is considered that these subsystems are implemented with another tool. Each environmental subsystem is treated as one element with a set of input/output pair gates and one *cIn* and *cOut* gate that must be interconnected with the T-Mass control element.

One of the predefined environmental elements is the element for communication channel modelling. Although in classical models the interconnections among elements are just mathematical relations, in the world of WSN some computational process during communication should be performed – for example, for noise enhancement, energy consumption or packet losses.

Communication channels play an important role in the model, because they are the only element with which particular agent platforms can be interconnected. The developer can now use one of the special environmental elements for interconnection between two agent platforms and she or he can specify the process that manipulates the data during their transfers.

Figure 2 Communication channel model element



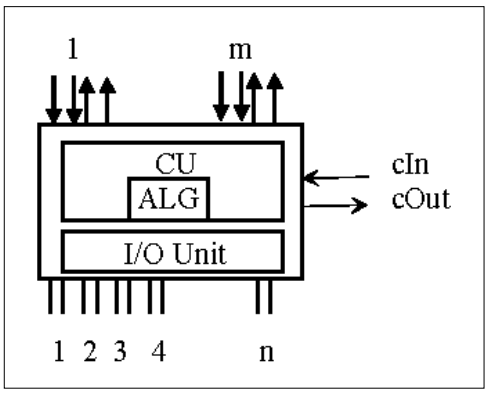
The behaviour of such an element is quite simple. It transfers an input from its input gate to its output gate and modifies it by some function. Each element or subsystem also has two additional gates *cIN* and *cOut*, which are connected to the T-Mass control element.

3.4 Platform element

In our case the agent platform differs from the platform specified in the FIPA standard. We can afford this because we are making just a MAS model and we do not need all the functionalities that are required from a FIPA-based agent platform in real MASs. We also build the agents on a lower level of abstraction, on which some issues like the ACL language need not be presented.

In real systems one layer of the platform is a physical device in which the agents can reside. The model of MAS treats the platforms rather as an interface between the agents and an environment. It also provides some advanced functionalities, such as the storage of data, which are global for the whole platform. Moreover, it provides a set of algorithms that can be used by the agents. Both are exploitable by ALLL agents using an action that will be introduced as 'platform service execution'. The structure of the platform element is shown in Figure 3.

Figure 3 Structure of an agent platform element



The agent platform element has three types of gates. Every such element has a *cIn* and *cOut* gate that are connected to a central T-Mass control element. Then there is a set of foursomes of gates. Each such foursome includes one input gate and one output gate for the corresponding agent in/out gates. The other two gates are there for the control of a particular agent. If an agent is assigned to the platform, all the corresponding in/out cIn/cOut gates must be interconnected. We call such a foursome to be one agent slot.

Each platform also includes a set of gate tuples, which are available for the interconnection of the platform with environmental subsystems. In Figure 3 there are *m* agent slots and *n* environmental gate tuples.

3.5 Platform subsystem

Every platform subsystem contains elements described in the previous sections. Concretely there is always one platform element and at least one agent element interconnected with it. The maximum number of agents is restricted by the number of platform slots.

3.6 Architectures for ALLL-based agents

In general, the behaviour of the agent as a system element is driven by ALLL. Before we introduce ALLL, we need to discuss two issues of importance for the language semantics. From our point of view the agent is a program running at a computer-based device(s). But that is not necessarily right because there are some pure hardware agents that have no programmable control unit, for example the situated automata. Nevertheless, most of the agents are controlled by some form of robot loop – from elementary reactive rule-based agents to complex intention-driven architectures. Thus the modelling system will contain a generic agent architecture and a language that would allow implementation of various agent control algorithms.

The architecture is very similar to the Von Neumann computer architecture. In general the architecture comprises data stored in a database, a language interpreter and a set of registers. Furthermore, there is an input/output interface managing incoming and outgoing messages. Finally, there is a pair of control gates available for the connection from/to the T-Mass control element, which will be closely described in the section about T-Mass modelling principles.

The database is logically divided into three main parts. Its first part is reserved for the agent's knowledge base. The agent's knowledge base is a database of grounded predicates of First-Order Predicate Logic (FOPL). We will denote the language of grounded predicate $\mathbf{L_{gp}}$ as a subset of the FOPL language.

The second part of the agent architecture is the agent's plan base, where some plans written in ALLL can be stored. Finally, there is another base for incoming messages and it is called input buffer. The input buffer is a database of predicate lists which are members of language $\mathbf{L_{list}}$, which is a set defined as:

$$\forall f_1 \ldots f_n \in \mathbf{L_{gp}}\ (n \geq 1, (f_1, \ldots f_n) \in \mathbf{L_{list}}). \tag{3}$$

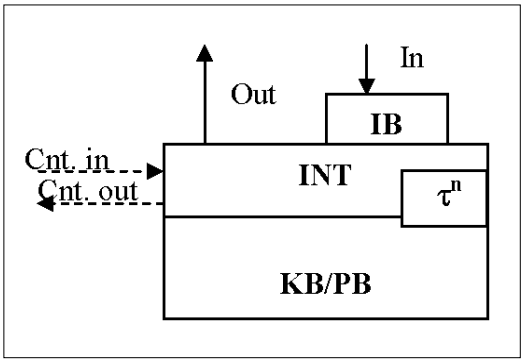
The register base is a vector of individual registers. Each register contains a set of predicates, which means that the state of the register is similar to the state of the knowledge base; or it could contain a plan, and so its state is a plan. So the register has a value from the set:

$$(\mathbf{L}_{\mathbf{list}} \cup \mathbf{L}_{\mathbf{ALLL}})^{\mathbf{n}}. \tag{4}$$

Although there could be several registers as a part of the agent, we will expect only one register for the purposes of this text.

The general ALLL-based agent architecture with all the components is shown in Figure 4.

Figure 4 Generic architecture for ALLL-based agents



Thus, the agent state is given by the states of its particular parts. Thus the function of agent behaviour could be defined as follows:

$$\sigma\colon \mathbf{U} \rightarrow \mathbf{L}_{\mathbf{GP}} \times \mathbf{L}_{\mathbf{list}}^{*} \times \mathbf{L}_{\mathbf{ALLL}}^{*} \times (\mathbf{L}_{\mathbf{list}} \cup \mathbf{L}_{\mathbf{ALLL}}). \tag{5}$$

and it maps particular agents to their state, which is a relation among the states of their knowledge base, input buffer, plan base and registers.

For correct agent functionality, it is considered that such an agent would be situated in a platform. In general, the platform provides some services to all the agents residing in the platform. Among others, the most important service is that the platform manages the communication among agents. Moreover, the platform provides some computational services to the agents. Formally, let there be a set **FNC** which is the set of all functions that the platform provides to the agent, and then each function from the set $f \in \mathbf{FNC}$ is defined as:

$$f\colon \mathbf{L}_{\mathbf{list}}^{*} \cup \mathbf{L}_{\mathbf{ALLL}} \rightarrow \mathbf{L}_{\mathbf{list}}^{*} \cup \mathbf{L}_{\mathbf{ALLL}}. \tag{6}$$

This means that the functions' input is either a plan or a list of predicates and their output is again a plan or a predicate list.

Now all the basic constructions have been defined and we can introduce ALLL itself.

3.7 The ALLL specification

Instead of a strictly formal syntax and semantic specification of the language, we provide some informal notation with an intuitively semantic definition. The formal syntax and semantics of the original language can be found in the dissertation by Zbořil (2004), but in the Czech language only. Here the language syntax will be presented as some constructions with predefined sublanguages. The semantics will then be illustrated by possible changes of the model and agent state.

3.7.1 Identifiers and predicates

Some of the syntax construction was shown in the previous sections. The syntax of $\mathbf{L_{ALLL}}$ will be shown using languages $\mathbf{L_{GP}}$ and $\mathbf{L_{list}}$. In addition we will need some other sets/languages. $\mathbf{L_{ID}}$ will be the language of the identifier defined by the following regular expression:

$$\mathbf{L_{ID}}=[\#]\{[a..z][A..Z][0..9]\}^{+}. \tag{7}$$

Now we define $\mathbf{L_{GP}}$ as a language given by the following formula:

$$\forall ps,f_1\ldots f_n\in\mathbf{L_{ID}}(n\geq 0,(ps,f_1\ldots f_n) \in \mathbf{L_{GP}}). \tag{8}$$

Note that the predicate is written a little differently than usual and that the predicate symbol is the first element of the sequence. The general list with predicates can contain predicates or sublists:

$$\forall s_1\ldots s_n\in\mathbf{L_{ID}}\cup\mathbf{L_{GLIST}}(n\geq 1,(s_1\ldots s_n)\in\mathbf{L_{GLIST}}). \tag{9}$$

In a similar way we will use another language $\mathbf{L_p}$, which is the language of predicates with possible anonymous variables instead of terms:

$$\forall f_1\ldots f_n\in\mathbf{L_{ID}}\cup\{_\}(n\geq 1,(f_1,f_2\ldots f_n)\in\mathbf{L_P}). \tag{10}$$

The anonymous variables are those used in the Prolog language and are also denoted by the underscore symbol ("_"). Its usage will be explained in the sections about language semantics.

3.7.2 The ALLL plan

The plan is the basic structure expressible in the ALLL language. The plan is a sequence of actions. In this case it is in the form of a linear list enclosed in brackets. If there is a set of all possible actions (written as a corresponding sentence of a language of actions denoted as $\mathbf{L_{ACT}}\subseteq\mathbf{L_{ALLL}}$) then the ALLL plan is any string from the set given by the formula:

$$\forall a_1,a_2\ldots a_n\in\mathbf{L_{ACT}}(n\geq 1,\hat{}(a_1,a_2,\ldots a_n)\in\mathbf{L_{ALLL}}). \tag{11}$$

The execution of the plan is performed action by action till the plan fails or it is successfully executed. It also gets us closer to the semantics of the language. In fact each action could either succeed or fail.

If an action fails, then the agent's register is set to a constant '*fail*' and the rest of the plan into which the failed action belonged is skipped. But the upper-level plan is not skipped if the failed plan had been executed as a lower-level one.

After the performance of the action, the model state can change in many ways, which depends on the action type. In general, all parts of an agent's internal state could change as well as the universe of the model.

Let us start with two special actions ***succeed*** and ***fail***, which immediately cause the successful or unsuccessful termination of the plan. Naturally there could be more actions within the plan. In the following sections all the remaining action types will be shown and their functionality/semantics will be defined.

3.7.3 Internal actions

In general, internal actions are those that do not affect an environment but only the agent's internal state. The first two actions are those that manipulate the knowledge and plan base. If there is a grounded predicate d and a predicate q a with a possible anonymous variable, then the action has a form with the following definition:

$$\forall d \in \mathbf{L_{GP}}\ \forall q \in \mathbf{L_P}\ (+d \in \mathbf{L_{ACT}} \wedge -q \in \mathbf{L_{ACT}}). \tag{12}$$

The semantic of the action is then as follows:

$$\mathbf{KB'} = \mathbf{KB} \cup d \tag{13}$$

$$\mathbf{KB'} = \mathbf{KB} - \{d \bullet \exists \sigma (d{=}q\ \sigma)\}. \tag{14}$$

It means that if some data are added into the knowledge base, the base is simply extended by the data. If there is a deletion action (which always succeeds), all the predicates unifiable with the action parameter are deleted from the base.

In a very similar manner there are actions of addition and deletion into/from the plan base. If there is a plan written in ALLL, then the addition of such a plan has the syntax

$$\forall plan \in \mathbf{L_{ALLL}}\ (+\hat{}plan \in \mathbf{L_{ACT}}) \tag{15}$$

The deletion of a plan is performed when the plan's name is mentioned as an action parameter. Then the action from the set given by the formula:

$$\forall name \in \mathrm{L}_{\mathbf{ID}}\ (-\hat{}(name) \in \mathbf{L_{ACT}}) \tag{16}$$

deletes a plan with the corresponding name if such a plan exists in the plan base.

$$\mathbf{PB'} = \mathbf{PB} - \hat{}plan \bullet plan{=}(name,(\mathrm{Body})). \tag{17}$$

If there is no plan with the given name, the action succeeds anyway.

3.7.4 Communication

On the other hand, communication is considered to be the only possible external action that an ALLL-based agent can make. In fact, a real agent has some effectors which it uses to affect the surrounding environment. But we omit such an action in our model because the agent is interconnected only with its platform.

Now we provide the syntax and semantics of the agent's actions. An agent (let it be named 'agent1') performs the communication act towards another agent if it performs an action with a syntax defined by the formula:

$$\forall rcv \in \mathbf{L_{ID}}\ msg \in \mathbf{L_{list}}(!(rcv,msg) \in \mathbf{L_{ACT}}) \tag{18}$$

where rcv is the identifier and msg is a list. This action succeeds if there is an agent with the corresponding name, otherwise it fails. In the successful case, the receiver's input buffer is extended with a tuple containing the sender's name and the message:

$$\mathbf{IB}_{\mathrm{receiver}}\text{'}{=}\mathbf{IB}_{\mathrm{receiver}} \cup (\text{'agent1'}, msg). \tag{19}$$

Analogously, an action which withdraws messages from the agent's input buffer uses an identifier snd and has a syntax constructed as:

$$\forall snd \in \mathbf{L_{ID}}(?(snd) \in \mathbf{L_{ACT}} \vee ?(_) \in \mathbf{L_{ACT}}) \tag{20}$$

with semantic

$$\tau' = n \bullet (snd, n) \in \mathbf{IB} \text{ or } \tau' = n \bullet (_, n) \in \mathbf{IB} \quad (21)$$

$$\mathbf{IB'} = \mathbf{IB} - n. \quad (22)$$

It means that the register will contain a message from some specified sender or messages from any sender stored in the input buffer. The message is subsequently deleted from the input buffer.

There are two other internal action types. The first one is for belief base testing and has the form of a predicate (with a possible anonymous variable). Such an action succeeds if there is at least one unifiable predicate, otherwise it fails. The result of the action is a list of all unifiable predicates in the knowledge base. So if there is, for example, a testing action using a predicate with the predicate symbol *quer* $\in \mathbf{L_P}$ *y*, then the register's value would change as follows:

$$\tau' = \{data \bullet \exists \sigma (data = query\ \sigma)\}. \quad (23)$$

The last action type which is considered to be internal is the 'function call' action type. It has the same syntax as the testing action. To distinguish it from the testing action, the interpreter checks whether the agent platform provides a function with the same name as the predicate symbol of the predicate. If there exists such a function, then it is executed with predicate terms as its parameters. Then, after the execution, the result is stored in the register. For example, if there is an action '(factorial,2)' and the agent platform provides the function named *factorial*, then the result of such an action is the value 2 in the register τ.

3.7.5 Subplan execution

Subplan execution and some other actions should also be considered to be internal actions. Although they can produce only internal changes, we describe them in separate sections. In principle, this section as well as the following two sections is about the expansion of the plan base.

First we focus on subplan executions. By the term 'subplan execution' we mean that we expand the currently running plan by inserting another plan. There are two ways of executing a subplan: the direct way and the indirect way. If the direct execution is used, then the subplan is written inline in the execution action. The syntax of the direct execution action type is as follows:

$$\forall act_1, act_n \in \mathbf{L_{ACT}} (n \geq 1, @(act_{1 \ldots} act_n) \in \mathbf{L_{ACT}}). \quad (24)$$

Here act1, and act2 and so on are some other actions constituting the plans. The semantics of this kind of action is shown in Figure 5a.

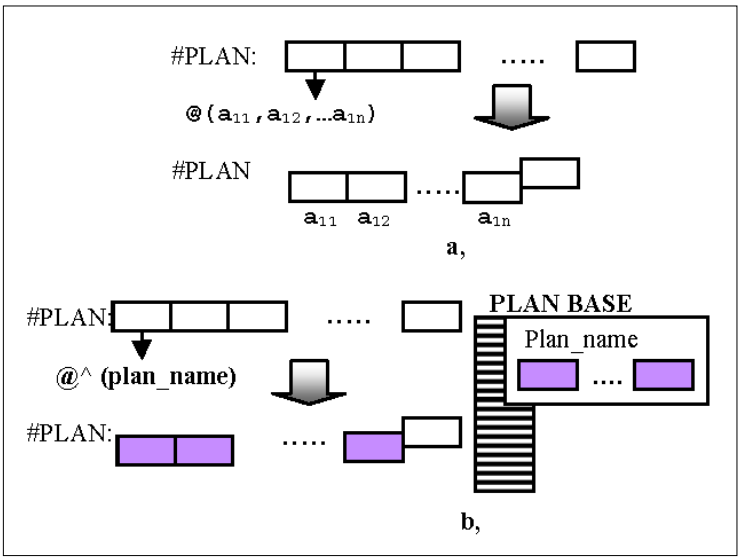
After the execution action (which always succeeds), the agent's plan is expanded with the given subplan. The advantages of such a construction is that even though the subplan fails, the original plan does not fail but continues with the action following the subplan execution action.

An indirect plan execution is similar to the direct execution, but the plan is not written inside the action. Instead of the plan script, a plan name is mentioned as an action parameter:

$$\forall plan_name \in \mathbf{L_{ID}}(@\hat{}(plan_name) \in \mathbf{L_{ACT}}). \tag{25}$$

Then the original plan is expanded by a relevant plan from agent's plan base. The new plan is added to the beginning of the original plan. The process of indirect plan execution is shown in Figure 5b.

Figure 5 Direct and indirect subplan execution (see online version for colours)



3.7.6 Plan instance execution

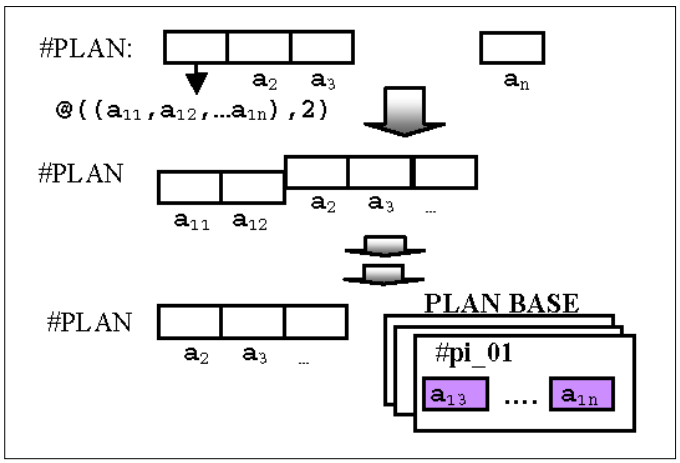
This section as well as the next one is an extension of the original language, as proposed in the dissertation (Zbořil, 2004). Experiments and practical usage of the original language brought a need for upgrading the language in some aspects. One of these aspects was a need for a meta-reasoning capability of the language. The original language was then upgraded with actions that allowed executing a plan for a given number of steps. Such executions have the syntax:

$$\begin{aligned}
&\forall act_1 \ldots act_n \in \mathbf{L_{ACT}}, m \in \aleph, name \in \mathbf{L_{ID}}, iname \in \mathbf{L_{ID}} \\
&\quad (n \geq 1,\ m \geq 0 \\
&\quad @((act_1 \ldots act_n), m) \in \mathbf{L_{ACT}} \\
&\quad \vee @((act1, act2 \ldots), \#iname, m) \in \mathbf{L_{ACT}} \\
&\quad \vee @\hat{}(name, m) \in \mathbf{L_{ACT}} \\
&\quad \vee @\hat{}(name, \#iname, m) \in \mathbf{L_{ACT}}).
\end{aligned} \tag{26}$$

Strings *name* and *iname* stand for some identifiers. The first two actions are for direct execution and the rest is for indirect execution. The plan is then executed for *n* steps. If it does not finish after the given amount of steps, the rest of it is stored in the plan base. But there are two ways of naming the plan. If a name with preposition # is mentioned and the

name is originally in the plan base, then that name is used. If a plan with the same name is already stored, then some implicit name generated by an interpreter is used. The implicit name is also used when there is no mention of a name in the action. The principle of plan instance execution is shown in Figure 6.

Figure 6 Plan instance execution principle (see online version for colours)



Plans that arise by such an execution are treated as some instances of the original plan. They could be executed again. But as they are processed, the corresponding plan instance in the plan base is modified immediately. When the execution of the plan instance finishes, it is deleted from the plan base.

3.7.7 Cloning

The last but not least feature of the language is that it allows runtime cloning of the agents. This means that the agent can make a copy of itself or it can create an agent and supply it with a subset of its plan base and belief base. Because the cloning will not be used further in this text, we introduce it just briefly. The syntax of the cloning action is as follows:

@@^(clone_core,(newbase,_),^(!(a,b))). (27)

The double 'at' symbol is followed by the plan name (in this case clone_core) that will be the top-level plan of the new agent. Then the list of predicates or plans (in the same form as the plan base and knowledge base testing actions types) is used for the creation of the agent's knowledge and plan bases. This action is the only one which changes universe of the model. Concretely it adds a new agent connected to the parental agent platform. If this is not possible, the action fails.

4 Model-based development of a multi-agent system using T-Mass

In this section we show how the process of model-based design can be realised with T-Mass. First we introduce the main stages of the model-based design and then focus on the simulation process.

4.1 The model-based design of MASs

The model-based development of MASs has three main phases – the creation of the model, design by simulation and the realisation of the system. In the following points we extend each phase for a closer insight into the process:

- Creation of the model
  - 1 definition of the agent roles necessary for the model
  - 2 definition of agents' behaviour (decision procedures, protocols, *etc.*)
  - 3 development or adoption of suitable agent architecture
  - 4 implementation of particular agents
  - 5 development of an environment model
- Model-based design
  - 6 Loop:
    - 6.1 Simulation of system run and checking of agent behaviour
    - 6.2 Identification of possible design mistakes
    - 6.3 Handling problems and redevelopment of the multi-agent model
- Realisation of the system
  - 7 realisation of the agents with identical behaviour to their models
  - 8 situating agents in the real environment
  - 9 validation of the model-based design results and possible reformation of the model (loop to 1).

The whole process of model-based design also includes stages that are out of the scope of the T-Mass tool. Especially the first three points are related more to software engineering, where methodologies like GAIA (Wooldridge *et al.*, 2000) could help. To propose agent roles and agents, one needs to delimitate responsibilities, competences and protocols and, for these, find out which kind of agent and which algorithms would be suitable.

We have already discussed Point 4 in the previous sections. Environmental modelling is currently of interest in the researchers' community. Thus we only remark that the model needs to be relevant and all important aspects for agent behaviour must be present in the model. So we move on to the process of simulation, which allows us to check the whole system's behaviour and tune particular agents' facilities.

4.2 Simulation with T-Mass v.2

The current version of T-Mass allows the execution of two-step synchronous simulation. In general there are two phases: the first one of the agents' action and the second one of environmental evaluation.

During the first phase, every platform executes one step of each agent's plan. Then the environment is evolved and the stimuli are sent to the agents' input buffers.

1 System initialisation

2 Evaluate all the platform subsystems

  2.1 Each agent puts an action to its 'out' gate

  2.2 The platform processes the actions

    2.2.1 The null action is ignored; it does not affect anything

    2.2.2 An action towards the platform services or communication with an agent residing on the same platform is processed and put to output/agent gates

    2.2.3 An action towards the environment (including communication with an agent residing on another platform) is propagated to the platform/environment output gates.

3 Evaluate all environmental subsystems

  3.1 They take agent actions from the platform output/environment gates

  3.2 Every environmental subsystem makes its evaluation and sets values to its output/platform gates

  3.3 The platforms possibly propagate the environment stimuli to the agents' input buffers.

4 GOTO 2

The simulator initiates the subsystems by sending appropriate signals to their *cIn* and it is informed that they have finished when it receives a signal from their *cOut* gates. The null action is a special agent's message towards the platform which means that the agent does not want to do anything towards the environmental systems.

4.3 Realisation of the ALLL agent

Finally, if we are satisfied with the agent's behaviour in the model, we would like to use it in a real environment. Our approach is to make platforms with ALLL interpret for each device in which the agent(s) should reside (computer operating systems, microcontrollers, *etc*.). Such platforms should also provide the basic algorithms that the agents use and furthermore should be able to interpret agent actions in the real environment. As we have shown before, the actions in the model are in the form of communication acts. But now the platform should execute some real actions when the agent makes some communication acts. However, there are some actions in the form of communication and there are some real actions toward the environment, but the agents do not distinguish among them.

The realised agents are then faced with a real environment and their success depends on how we modelled the environment. For some reason we find that there is something wrong in the systems. Then we need to check if the environment does not behave in a different way than we had expected. If such a situation appears, the agent and its capabilities should be redeveloped again.

5 ALLL agent realisation example

Here we show some implementation examples, where we will work with some functions that should be provided by the agent platform. These functions are well known from the LISP language. We need the functions *car*, *cdr* for reaching the head (first element) and the tail (the rest of the list).

5.1 Reactive agent implementation

First we demonstrate how purely reactive agents can be implemented. The main plan, named '#CORE', represents the general robotic loop that executes two subplans – one for event selection and another for selected event processing. Executed subplans may, but need not, fail; however, even if one of them fails, the main agent loop does not fail.

```
π(#CORE,(@(
            @^(select_event),
                @^(process_event),
              )
    @^(#CORE)
     )
)
```

The implementation of the subplans depends on event representation and the representation of relevant processes for given events. An agent expects the incoming events to be in its input buffer and the event itself has the form of a grounded predicate. For our purposes the events will be sent by an agent named GODI that transforms environmental changes to appropriate events, and the predicate symbol will then be 'event'. So if there is a tuple:

(GODI,(event,term1,term2…))

in the input buffer, then the agent tries to start a plan which is relevant to the event predicate. Let there be some lists stored in the knowledge base with the following form:

((event,terms…),(plan_name))

and for each such list there can also be a plan with the corresponding name stored in the plan base:

π(plan_name,(actions…)).

Now we go on with the presentation of the processes for event selection and event processing.

First we implement the plan named 'select_event'. Its structure can be like this, for example:

```
π(select_event,(?(GODI),(cdr,τ),
            @((τ,_),succeed),
            @(select_event)).
```

The action withdraws an event from the agent's input buffer. If the action succeeds, the register τ contains an event list. Then the event list's presence in the agent's knowledge base is tested. The script contains a register symbol, which is replaced with the actual event list. In runtime the testing action will appear there. If the testing succeeds, the plan finishes with success and the event with the corresponding plan name appears in the register. If the testing action fails, the plan is executed again until it finds a suitable event or until the input buffer is empty.

π(process_event,((car, τ),@^τ)).

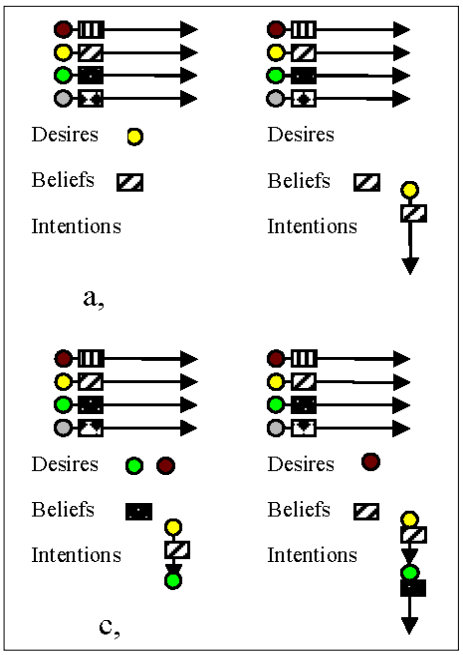
The plan for event processing is even simpler. It reaches the plan name using the *car* function and then executes the plan. After plan execution, whether it is successful or not, the control is returned to the main control loop and another event can be processed.

5.2 Intention-driven agents implementation

The second agent architecture that will be shown in this text is based upon the currently popular idea of intention-driven behaviour. In brief, these systems adopt some goal as its intention when they find that this goal can be achieved. Then they make plans in the form of so-called intention structures that are built with some predefined plans (sometimes also called 'acts') from the agent's plan library. When there is an intention structure, its lowest-level plans can be executed until the intention is fulfilled or all possible attempts to perform the intention structure for the given intention fail.

To make a program for such a system with the ALLL language, we use its meta-reasoning abilities. In fact, there will be two levels of processes. At the upper level, there will be goal and subgoal processing and intention structure fabrications. This process senses incoming events and tries to make the proper intention structure for these events. At the lower level there will be processes for achieving the intention itself. In Figure 7 an example of BDI agent reasoning is shown.

Figure 7 Example of BDI-based reasoning (see online version for colours)



At the very beginning one initial goal is stored in the agent's set of desires. The belief set is a state pictured with a rectangle (Figure 7a). There is a plan library with one plan suitable for the (yellow) goal and applicable in a given environment state. Such a plan is added into the intention set Figure 7(b). During execution of the plan, the belief state changes and two new desires arise. The first desire (green) is a subgoal of the only intention and the second (brown) is another top-level goal (Figure 7c). Thus there are two goals, but only one relevant and applicable plan (for the green subgoal), which is consequently added to the intention structure Figure 7(d).

Now we try to implement this system in ALLL. However, the implementation of a complete BDI agent is quite complex to fit it into this text. We focus only on the most important implementation parts. In the following paragraphs, some basic constructions for intention formation and execution will be shown.

The intention structure itself will be a structure of some ALLL plans. At the lower level, the plans stored in an intention structure will be executed. First we show how the plans and also some metadata could be stored in belief and plan bases. Each plan has defined purposes and conditions of usage. So in the belief base there are a set of tuples in the form:

((event,terms…),condition,plan_name).

Event and *condition* are predicates and *plan_name* is a string. This triple is used when a relevant (for an event) and applicable (in the actual state of the belief base) plan is searched. It is supposed that for each *plan_name* appearing in any such triple, there is a plan with the same *plan_name* stored in the plan base.

The BDI agent control loop will be similar to that of the reactive agent:

```
π(#CORE,( @(
            @^(select_event),
                @^(process_event),
                )
                @^(execute_is),
                @^(#CORE)
        )
)
```

The selection of an event is quite the same as before. The only change is that the sender is followed by an intention which raised the event. We will discuss this later in this section. So the event in general looks as follows:

(sender,PID,(event,term1,term2…)).

The event selection plan deletes the sender information and finishes with the list:

(PID,(event,term1,term2…)).

Now we aim to do event processing. We consider that there is an event in the register in the above-mentioned form and the agent needs to find a plan which is relevant and applicable. The agent first withdraws all the triples relevant to the event and then tests

the condition of usage. It does this until it finds a plan that is applicable or it finds that such a plan does not exist. The implementation of this is a little difficult in ALLL but still possible.

```
        π(process_event ,(
1,              -(pint,_),+(pint,τ),
2,              (cdr, τ),(car, τ),
3               (τ,_,_),…
```

First the register is stored in the knowledge base for further usage. All predicates with the same predicate symbol are deleted and the current event is stored (*viz.* Line 1). Subsequently the event is reached as the second element of the register list (Line 2). The following line tests the presence of a triple beginning with the event stored in the register. If this action succeeds, the process continues with these actions:

```
4,      (cdr, τ),-(pplan,_),+(pplan,τ),
5,      (car, τ), τ,
```

The three actions in Line 4 store the tuple (condition,plan) into the knowledge base. Then the condition itself is reached and the test action with the condition is executed. If the plan has not failed so far, then we have a plan that is relevant for the event and applicable in the given belief base state.

Now the reached subplan is executed and consequently the plan instance which had raised the event also may be executed. The rest of the 'process_event' plan then continues:

```
6,      (pplan,_),(cdr, τ),(car, τ),
7,      @^τ,
8,      (pint,_),(cdr, τ),(car, τ),
9,      @^τ
10,     ))
```

The actions in Lines 6 and 8 reach the plans stored during the previous process and execute an event-related plan at Line 7.

Another important issue of the BDI agent is how the plans for event processing are designed. In short, there are several classes of actions, which can be distinguished as internal actions, external actions and goal adaptation. Although internal actions and external actions are similar to the knowledge base manipulation and communication actions provided by ALLL, we only show here how goal adaptation can be realised.

We start with the goal *achievement* statement. It means that the agent should actively behave in a way to reach the declared goal. Goal statement is in fact event generation. If a plan needs to set a subgoal, it simply sends an event to its input buffer. But there must also be mentioned an instance of the event producer. For this reason, some mechanisms introduced in the section about plan instances execution will be used. For example, there is a plan with actions (a_1,a_2,…a_n) and the first action should be a subgoal statement. This can be implemented in ALLL with respect to the event-processing plan, as follows:

```
π(plan1,
        @^((a2,…an),#plan1_i1,0),
        !(agent1,#plan1_i1,
                (event,terms…))
)
```

Please note that symbols a_2 … a_n are again just some abstractions for some ALLL actions. But the principle is simple. The rest of the plan is executed for 0 steps. This means that just an instance called #plan1_i1 is created and stored in the plan base. Then the agent sends a message to itself (here we consider the agent's name to be 'agent1') and if the specified event is processed by a subplan, the rest of the plan is executed as the instance.

The second possible goal declaration is called goal *testing*. In this case the agent waits to see whether the declared goal is valid or not (the goal may occur, for example, when the environment changes; as an example let us mention the goal 'is it night?'). In this case the test goal action could be implemented as:

```
π(plan2,
  (@(
        @(
            (test_predicate)
            -(waiting_inst,#plan2_i1))
          @(a2,…an)
          @plan2),
     #plan2_i1,0)
   +(waiting_inst, #plan2_i1)
)
```

The purposes of the plan instance are for processing of the achievement goals, which are used in the BDI systems. First a plan instance is made. Then a predicate is written into the agent's knowledge base and that means that there is an instance waiting with a test goal. The instance consists of the test predicate itself, execution of the rest of the plan and deletion of the instance predicate from the knowledge base. Each time the main loop cycle reaches the execution of the plan 'execute_is', the plan should execute one waiting plan instance, if there is any. When the test predicate passes successfully, then the rest of the plan instance is executed and the waiting predicate need not be tested anymore. If it fails, the instance is renewed by calling itself again.

We will not describe the 'execute_is' plan because it is very similar to the 'process_event' plan.

6 Conclusion and future work

We presented a new way to develop systems with agents. As the main advances of the language, we consider that it is easily interpretable, it allows implementation systems based on behaviour, like the intelligent agents are, and it is suitable for usage for modelling and simulation of the multi-agent systems. Finally, the language allows agents' mobility among platforms where the interpreter of the language and some basic functions are present. Because this language is based at a low level of abstraction and it is sometimes inconvenient to write codes directly in this language, our next effort is to develop a higher-level language. Such a language would allow easier implementation of agents and ALLL is then used as a destination language into which the agent code will be compiled.

At this time the tool is implemented in the Java language for the Eclipse system. Also, some experiments with agent models and models of MASs have been done. Now we are about to make some real applications in which artificial agents will be used. Specifically we intend to make a sensor network for runtime risk analysis and management. In this application the principles described in this paper will be used for the checking of argumentation protocols and distributed reasoning algorithms. We believe that this process proves the usefulness of T-Mass usage and may inspire us for further extension of this modelling.

Acknowledgements

This work was supported by the Czech Grant Agency under the contracts GP102/07/P306 and GP102/07/P431, and the Ministry of Education, Youth and Sports of the Czech Republic under the contract MSM 0021630528.

References

Bratman, M.E. (1987) *Intention, Plans and Practical Reason*, Cambridge, MA: Harvard University Press.

Brooks, R. (1991) 'Intelligence without reason', *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pp.569–595.

Cabac, L. and Dörges, T. (2007) 'Tools for testing, debugging and monitoring multi-agent applications', *Proceedings of the Workshop on PNSE'07*, Siedlce, Poland.

Ferber, J. (1999) *Multi-Agent Systems*, Great Britain: Addison-Wesley.

Rao, A. (1996) 'AgentSpeak(L): BDI agents speak out in a logical computable language', *Agents Breaking Away, Lecture Notes in Artificial Intelligence*, Amsterdam: Springer-Verlag, Vol. 1038.

Rao, A.S. and Georgeff, M.P. (1995) 'BDI agents: from theory to practice', *Proceedings of the 1st Conference of Multiagent Systems*, San Francisco, USA.

Shoham, Y. (1990) 'Agent-oriented programming', Technical Report STAN-CS-1335-90, Computer Science Department, Stanford University, Stanford, California.

Wooldridge, M.J. and Jennings, N.R. (1995) 'Intelligent agents: theory and practice', *The Knowledge Engineering Review*, Vol. 10, No. 2, pp.115–152.

Wooldridge, M., Jennings, N. and Kinny, D. (2000) 'The Gaia methodology for agent-oriented analysis and design', *Autonomous Agents and Multi-Agent Systems*, The Netherlands: Kluwer, Vol. 3, pp.285–312.

Zbořil, F. (2004) 'Plánování a komunikace v multiagentních systémech', Dissertation, FIT Brno University of Technology, Brno, Czech Republic, p.107.

Zbořil, F. and Kočí, R. (2007) 'Intention structures modeling using object-oriented Petri nets', *Proceedings of the 7th ISDA*, Los Alamitos: IEEE CS, pp.33–38, ISBN: 0-7695-2976-3.

Bibliography

Bellifemine, F., Poggi, A. and Rimassa, G. (1999) 'JADE – a FIPA-compliant agent framework', *Proceedings of PAAM'99*, London, pp.97–108.

Bordini, R.H., Hübner, J.F. and Wooldridge, M. (2007) *Programming Multi-Agent Systems in AgentSpeak Using Jason*, Wiley, ISBN: 978-0-470-02900-8.

Dastani, M., Hobo, D. and Meyer, J. (2007) 'Practical extensions in agent programming languages', *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07)*, ACM Press.

D'Iverno, M., Luck, M. and Georgeff, M. (2004) 'The dMARS architecture: a specification of the distributed multi-agent reasoning system', *Autonomous Agents and Multi-Agent Systems 9*, The Netherlands: Kluwer Academic Publisher, pp.5–53.

Huber, M. (2001) 'JAM agents in a nutshell', *Intelligent Reasoning Systems*, Oceanside, California, USA.

Zbořil, F. (2007) 'Framework for model-based design of multi-agent systems', *Proceedings of EUROSIM 07*, Vienna, p.11.

Zbořil, F. and Zbořil, F.V. (2008) 'Simulation of wireless sensor networks with intelligent nodes', *10th International Conference on Computer Modelling and Simulation*, Cambridge, GB: IEEE CS, p.6, ISBN: 0-7695-3114-8.

Note

1 Foudation for Intelligent Physical Agents, http://www.fipa.org.