FRAMEWORK FOR MODEL-BASED DESIGN OF MULTI-AGENT SYSTEMS

František Zbořil jr.

Brno University of Technology, Faculty of Information Technology

zborilf@fit.vutbr.cz (František Zbořil)

Abstract

Artificial agents and multi-agent systems are attractive area which touches many other areas of interest including non-informatics ones like sociology or economy as well as many topics relating to computers, for example artificial intelligence or system design. When we have to develop a multi-agent system we need, among others, to build agent architectures together with some decision algorithms and communication protocols. Before such systems are realized in a real world we should sufficiently test and verify them. Presented text describes a new approach to building artificial agents and multi-agent systems using methodology of model-based design. In this methodology some models are used during the design process. The models are used for testing of particular elements behavior as well as for testing of behavior of the system as whole – in both cases by their simulation. Environment surrounding the elements is also simulated in the model and particular systems are tested whether they fulfills given objectives in proper way. When all the system elements work well in the model then they could be realized in some real environment. Our effort is to develop a tool that would allow model-based design of systems with artificial agents. For this reason we have been building application called T-Mass (Tool for Multi-agent System Development) which is aimed right on the model-based development of such systems and provides some important facilities for building rational agents. As a part of the tool we developed language called t-Sapi by which agents' behavior is controlled. Also we made two-phase synchronous algorithms for control of the model run. This paper shows how some popular agent architectures could be built with the t-Sapi language and how then they are used in the multi-agent simulation model. We also present some remarks about usage of the modeled agents and their consequent realization in real multi-agent applications.

Keywords: Simulation models, Agent control language, Reactive and BDI agents, Model-based design.

Presenting Author's biography

František Zbořil jr. is assistant professor at Faculty of Information technology, Brno University of Technology. His major interests comprise several area related to artificial intelligence and modeling, mainly artificial agent architectures, multi-agent systems modeling as well as computer vision and robotics. His actual research is aimed onto realization of intelligent agent-based systems for runtime risk analysis and management.

ISBN 978-3-901608-32-2 *Copyright © 2007 EUROSIM / SLOSIM*

1 Introduction

If we work with multi-agent systems (MAS) we usually have to deal with some kind of distributed system with nodes that show some appearances of intelligence. Recent understanding of the MAS is that there exist a net of agent platforms interconnected with communication and transport channels. Such a network consists of agent-friendly environment where intelligent agents can reside and act, eventually travel among particular platforms.

Subject of artificial agents are not new. Artificial intelligence as well as many other branches takes artificial agent as a part of their systems for more than twenty years. However, different areas understand the term '*agent*' slightly different. Here we will deal with artificial intelligent agent as it is understand in computer society. Contribution of this text should be introduction of principles of a new tool called T-Mass and demonstration that such approach to MAS modeling could bring advances in MAS development.

First we explain some basics of artificial agents and MAS in chapter 2. We also discuss here some tools for MAS development and we point out some advantages and weak spots of these tools. Then we introduce the T-Mass tools in chapter 3. It will include agent control language called t-Sapi and principles of multi-agent model simulations. Chapter 4 gives some examples of agents programmed in the t-Sapi language and chapter 5 provides description of T-Mass usage for model based design. 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 definition which says that agents are autonomous entities working in mostly unknown and highly dynamic environment for the reason of some objectives achievement. Agent senses environment through its sensors and acts in the environment rationally. It means that its doing has a motivation. On the other hand its architecture should not be very complex. In fact agent is rather simply structure that has full control of its behavior and has ability to act flexible, swiftly and rationally.

To introduce today most popular agent architectures we distinguish between reactive and deliberative agents. Reactive agents in their pure form have been introduced by Brooks [2]. Brooks argued that there need not be symbolic reasoning process to make agents act with some intelligence. Such agents have only some models of their behaviors that are used when relevant situation appears in the environment. Other approaches to artificial agent realizations use so-called mental states. Reasoning based on agent's beliefs and obligations is used for Shoham's agent called Agent-0 [9]. Also there are architectures based on intentions (originally developed by Bratman [1]) extended later to today's popular Belief-Desire-Intention agent architectures [4,7].

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

2.1 Overview of current tools for MAS modeling and development

First we introduce some widely known tools for agent development.

We start with the 2APL and JAM! tools. Both tools are based on the BDI theory. 2APL allows making plans in a form of production rules that could be applied for some goals or sub-goals. This tool is equipped with FIPA specification corresponding platform [13].

JAM! is a language with syntax similar to the Java language and also allows to make agents based on BDI principles. These tools are suitable mainly when some agents with intention-driven behavior should be realized and their behavior verified in a multi-agent community.

JADE is a Java framework and platform for building multi-agent systems in accordance with FIPA specifications. Beside FIPA agent platform it provides tools for implementation of communication protocols, ontology, behavioral procedures etc.. It is quite nice and probably the most popular tool today but in its original version it does not support development of mature agent architectures. Some extensions as is JADEX solve such disadvantage and introduce mechanisms how to implement for example the intention driven agents.

Finally we discuss the Petri nets. Along with original tools for agent development the Petri-nets could be mention here. At least we mention these nets for the reason that they represent major stream in distributed system modeling. Today there is some effort to use Petri nets in agent modeling and realization. For example they are used by Feber [5] to demonstrate distributed algorithms for distributed reasoning in multiagent community, communication protocols, coalition forming etc. There is also a Petr-net based system for MAS system realization called MULAN developed at University of Hamburg [3]. At last there is an effort to develop rational agents with object

 Copyright © 2007 EUROSIM / SLOSIM

oriented Petri nets at Brno University of Technology. But some disadvantages can be identified when Petri nets are brought into the world of rational agents. Mainly the expressivity of the Petri net model better suits to distributed systems than particular agent architecture realizations. Furthermore agent systems need to have strict interface between agents and their environment and also some symbolic reasoning mechanisms used often in the field of artificial agents are not native to the Petri nets.

2.2 Motivation for another tool development

There are many other tools which could be useful for agent systems development. The reason why we develop another modeling and development tool is that T-Mass is found at low-level of abstraction and it is aimed to provide possibility to create many kinds of agents. It is easily interpretable and has many features suitable for agent programming. Finally the language allows straightforward implementation of platform for such agents for devices varying from microcontrollers, intelligent sensors and embedded systems to classic computer implementation environments.

3 A tool for multi-agent simulation

The previous sections outlined that there are many tool for modeling and developing agents and that despite of this we made another one. Now we are going to introduce basics of our system. Because this is a modeling tool we start first with some formal definitions of the models for which the tool is being developed.

3.1 T-Mass model

As usual, the T-Mass model is a tuple $\mathbf{M}=(\mathbf{U},\mathbf{R})$ where $\mathbf{U}$ is an universe and $\mathbf{R}$ is a characteristic. Universe is a set of all elements within the system. In T-Mass the universe is a set of t-Sapi based agents. Characteristic is a set of all relations among the agents. Behavior of the system is determined by the system state changes during the simulation run. The state of the system is then given by state of all particular agents and the agent states are given by particular states of some parts of the agent architecture, which will be shown next.

3.2 Architectures for t-Sapi based agents

In general, behavior of the agent as a system element is driven by the t-Sapi language. Before we introduce t-Sapi language we need to discuss two issues important for the language semantic. From our point of view the agent is a program running at a computer-based architecture. But that is not necessarily right because there are some pure hardware agents that have not programmable control unit, for example situated automata. Nevertheless, most of agents are controlled by some form of robot loop – from elementary reactive agents to complex intention driven architectures. Thus the modeling system will contain generic agent architecture and a language that would allow creation of various agents. For this reason we show architecture for t-Sapi based agents first and then we introduce platform in which such agent must be situated.

The architecture is very similar to the Von Neumann computer architecture. In general the architecture comprises of data stored in database, language interpreter and a set of registers. Furthermore there is an input/output interface managing incoming and outgoing messages. There also can be some control signals that will be closely described in the section about T-Mass modeling principles.

The database is logically divided into three main parts. Its first part is reserved for agent's knowledge base. Agent's knowledge base is a database of grounded predicates of first-order predicate logic (FOPL). We will denote language of grounded predicate $\mathbf{L}_{\mathbf{gp}}$ as a subset of FOPL language.

Second part of the agent architecture is agent's plan base where some plans written in the t-Sapi language $\mathbf{L}_{\mathbf{tsapi}}$ 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 $\mathbf{L}_{\mathbf{list}}$ that is a set defined as

$$\forall f_1 \dots f_n \in \mathbf{L}_{\mathbf{gp}}\ (n \geq 1, (f_1, \dots f_n) \in \mathbf{L}_{\mathbf{list}}) \tag{1}$$

Register base is a vector of individual registers. Each register contains a set of predicates, it means that state of the register is similar to the state of knowledge base, or it could contain a plan, then its state is a plan. So the register has a value from the set

$$(\mathbf{L}_{\mathbf{list}} \cup \mathbf{L}_{\mathbf{tsapi}})^{\mathbf{n}} \tag{2}$$

Although there could be several registers we will expect only one register for purposes of this text.

General t-Sapi based agent architecture with all the components is shown in Fig. 1.

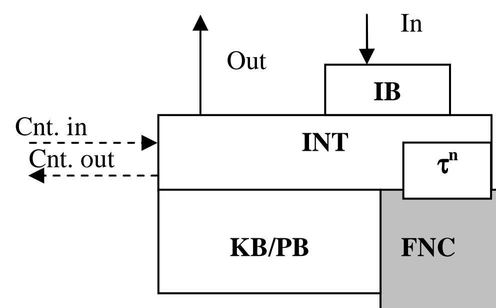

Fig. 1 Generic architecture for t-Sapi based agents

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

$$\sigma\text{: } \mathbf{U} \rightarrow \mathbf{L}_{\mathbf{GP}} \times \mathbf{L}_{\mathbf{list}}^{*} \times \mathbf{L}_{\mathbf{tsapi}}^{*} \times (\mathbf{L}_{\mathbf{list}} \cup \mathbf{L}_{\mathbf{tsapi}}) \tag{3}$$

 Copyright © 2007 EUROSIM / SLOSIM

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

For correct agent functionality it is considered that such 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 communication among agents. Moreover the platform provides some computational services to the agents. Formally, let there be a set **FNC** which is 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

$$f\colon \mathbf{L_{list}}^{*} \cup \mathbf{L_{tsapi}} \rightarrow \mathbf{L_{list}}^{*} \cup \mathbf{L_{tsapi}} \quad (4)$$

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

Now all the basics constructions have been defined and we can introduce the t-Sapi language itself.

3.3 t-Sapi language specification

Instead of strict formal syntax and semantic specification of the language we provide some informal notation with intuitively semantic definition. Formal syntax and semantic of original language could be found in dissertation [11], but in Czech language only. Here the language syntax will be presented as some constructions with predefined sub-languages. Semantic will be then illustrated by possible changes of the model and agent state.

3.3.1 Identifiers and predicates

Some of syntax construction was shown in the previous sections. The syntax of $\mathbf{L_{tsapi}}$ language 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 language of identifier defined by regular expression

$$\mathbf{L_{ID}}=[\#]\{[a..z][A..Z][0..9]\}^{+} \quad (5)$$

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

$$\forall ps,f_1\ldots f_n \in \mathbf{L_{ID}}(n\geq 0,(ps,f_1\ldots f_n)\in \mathbf{L_{GP}}) \quad (6)$$

Note that the predicate is written little bit different and that the predicate symbol is the first element of the sequence. General list with predicates can contain predicates or sub-lists.

$$\forall s_1\ldots s_n \in \mathbf{L_{ID}}\cup\mathbf{L_{GLIST}}(n\geq 1,(s_1\ldots s_n)\in \mathbf{L_{GLIST}}) \quad (7)$$

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

$$\forall f_1\ldots f_n \in \mathbf{L_{ID}}\cup\{_\}(n\geq 1,(f_1,f_2\ldots f_n)\in \mathbf{L_P}) \quad (8)$$

The anonymous variable is those used in the Prolog language and is also denoted by underscore symbol "_". Its usage will be explained in the sections dealing with language semantic.

3.3.2 t-Sapi plan

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

$$\forall a_1,a_2\ldots a_n \in \mathbf{L_{ACT}}(n\geq 1,\hat{}(a_1,a_2,\ldots a_n)\in \mathbf{L_{tsapi}}) \quad (9)$$

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 semantic of the language. In fact each action could either succeed or it could fail.

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

After action performance the model state can change in many ways which depends on the action type. In general, all parts of 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 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 will be defined.

3.3.3 Internal actions

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

$$\forall d\in \mathbf{L_{GP}}\ \forall q\in \mathbf{L_P}\ (+d \in \mathbf{L_{ACT}} \wedge -q \in \mathbf{L_{ACT}}) \quad (10)$$

are valid t-Sapi actions. Their semantic is then as follows.

$$\mathbf{KB'} = \mathbf{KB}\cup d \quad (11)$$

$$\mathbf{KB'} = \mathbf{KB} - \{d\bullet\exists\sigma(d=q\ \sigma)\} \quad (12)$$

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

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

$$\forall plan \in \mathbf{L_{tsapi}}\ (+\hat{}plan \in \mathbf{L_{ACT}}) \quad (13)$$

 Copyright © 2007 EUROSIM / SLOSIM

Deletion of a plan is performed when plan's name is mentioned as action parameter. Then action form the set given by formula

$$\forall name \in \mathbf{L_{ID}} \; (\text{-}^\wedge(name) \in \mathbf{L_{ACT}}) \tag{14}$$

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

$$\mathbf{PB'} = \mathbf{PB}\text{-}^\wedge plan \bullet plan=(name,(\text{Body})) \tag{15}$$

If there is not plan with given name the action succeeds anyway.

3.3.4 Communication

On the other hand, communication is considered to be only possible external action that t-Sapi based agent can make. In fact, real agent has some effectors which use to affect surrounding environment. But we omit such acting in our model.

Now we provide syntax and semantic of the agent's act actions. Agent (let it be named 'agent1') makes communication act toward another agent if it performs action with syntax defined by the formula

$$\forall rcv \in \mathbf{L_{ID}} \; msg \in \mathbf{L_{list}}(!(rcv,msg) \in \mathbf{L_{ACT}}) \tag{16}$$

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

$$\mathbf{IB}_{receiver}{'}=\mathbf{IB}_{receiver} \cup (\text{'agent1'},msg) \tag{17}$$

Analogously, action that which withdraw messages from agent's input buffer uses identifier *snd* and has syntax constructed as

$$\forall snd \in \mathbf{L_{ID}}(?(snd) \in \mathbf{L_{ACT}} \vee ?(_) \in \mathbf{L_{ACT}}) \tag{18}$$

with semantic

$$\tau' = \text{n} \bullet (snd,\text{n}) \in \mathbf{IB} \text{ or } \tau' = \text{n} \bullet (_,\text{n}) \in \mathbf{IB} \tag{19}$$

$$\mathbf{IB'} = \mathbf{IB}\text{-n} \tag{20}$$

It means that the register will contain a message from specified sender or a 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. First one is for belief base testing and has a form of predicate (with possible anonymous variable). Such action succeeds if there is at least one unifiable predicate, else it fails. Result of the action is then a list of all unifiable predicates in the knowledge base. So if there is for example a testing action with predicate *quer*$\in \mathbf{L_P}$ *y*, then the registers wallue would change as follows

$$\tau' = \{\text{data} \bullet \exists \sigma(\text{data}=query\ \sigma)\} \tag{21}$$

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

3.3.5 Sub-plan execution

Sub-plan execution and some other actions should be also 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 expansion of the plan base.

First we focus on sub-plan executions. By the term '*sub-plan execution*' we mean that we expand currently running plan by inserting another plan. There are two ways how to execute sub-plan – direct way and indirect way. If the direct execution is used then the sub-plan is written inline in the execution action. The syntax of direct execution action type is as follows

$$\forall \text{act}_1,\ldots\text{act}_n \in \mathbf{L_{ACT}}(\text{n} \geq 1, @(\text{act}_1 \ldots \text{act}_n) \in \mathbf{L_{ACT}}) \tag{22}$$

Here the act1 and act2 and so on are some other actions constituting of the plans. Semantic of this kind of action is shown in Fig. 2a.

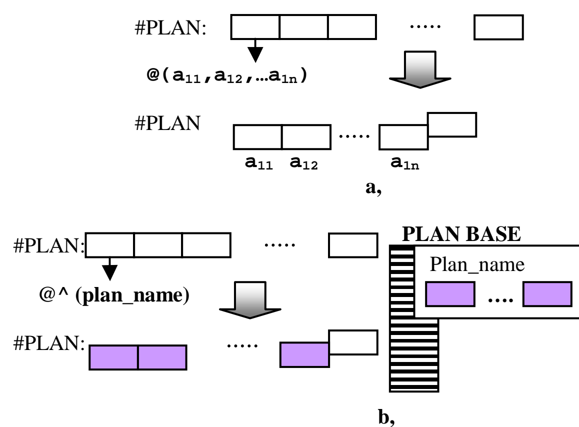

Fig. 2 Direct and indirect sub-plan execution

After action execution that always succeeds the agent's plan is expanded with given sub-plan. Advantages of such construction is that even the sub-plan fails the original plan does not fail but continues with action following the sub-plan execution action.

Indirect plan execution is similar to the direct execution, but the plan is not written inside the action. Instead of the plan script there is mentioned a plan name as action parameter.

$$\forall plan_name \in \mathbf{L_{ID}}(@^\wedge(plan_name) \in \mathbf{L_{ACT}}) \tag{23}$$

 Copyright © 2007 EUROSIM / SLOSIM

Then a plan from agent's plan base is included into the original plan if there is a plan with corresponding plan name. Process of indirect plan execution is shown in Fig. 2b.

3.3.6 Plan instance execution

This section as well as the next section is extension of the original language as it was proposed in dissertation [11]. Experiments and practical usage of the original language leaded to a need for upgrade of the language in some aspects. One of these aspects was a need of meta-reasoning capability provided by the language. Original language then was upgraded with actions that allow executing a plan for given number of steps. Such executions have syntax

$$\begin{aligned}&\forall act_1\ldots act_n\in \mathbf{L_{ACT}}, m\in \aleph, name\in \mathbf{L_{ID}}, iname\in \mathbf{L_{ID}}\\ &\quad (n\geq 1,\ m\geq 0\\ &\qquad @((act_1\ldots act_n),m)\in \mathbf{L_{ACT}}\\ &\qquad \vee @((act1,act2\ldots),\#iname,m)\in \mathbf{L_{ACT}}\\ &\qquad \vee @\hat{}(name,m)\in \mathbf{L_{ACT}}\\ &\qquad \vee @\hat{}(name,\#iname,m)\in \mathbf{L_{ACT}})\end{aligned} \tag{24}$$

Strings *name* and *iname* stand for some identifiers. First two actions are for direct execution and the rest is for indirect execution. Plan is executed for *n* steps. If it does not finish the rest of it is stored in the plan base. But there are two ways how the plan is named. If there is mentioned a name with preposition # and the name is original in the plan base then such name is used. If there is already stored a plan with the same name then some implicit name generated by interpreter is used. The implicit name is also used when there is not mentioned a name in the action. Principle of plan instance execution is shown in the following Fig. 3.

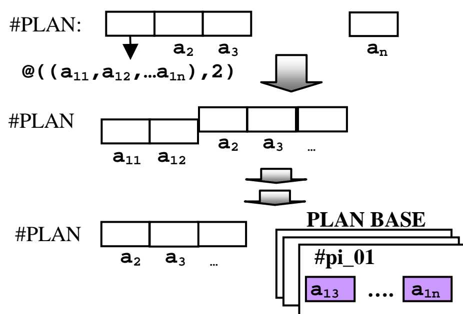

Fig. 3 Plan instance execution principle

Plans that arise by such 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 execution of the plan instance finishes, it is deleted from the plan base.

3.3.7 Cloning

Last but not least feature of the language is that it allows runtime cloning of the agents. It means that agent can make its copy or it can create an agent and supply it with a subset of its plan- and belief-base. Because the cloning will not be used further in this text we introduce it just briefly. Syntax of the cloning action type can be for example is as follows

@@^(clonecore,(newbase,_),^(!(a,b)))

Double 'at' symbol is followed by plan name (in this case clone_core) that will be the top-level plan of the new agent. Then list of predicates or plans (in the same form as the plan base and knowledge base testing actions types have) are used for creation of the agent's knowledge and plan base. This action is that one that changes universe of the model.

4 Agent implementation in t-Sapi

Here we show some implementation examples. In the following examples we will work with some functions that should be provided by agent platform. These function are well known from the LISP language, concretely we need functions *car*, *cdr* for reaching head (first element) and tail (rest of the list without the first element)

4.1 Reactive agent implementation

First we demonstrate how pure reactive agents can be implemented. Main plan named '#CORE' represents general robotic loop that executes two sub-plans – one for event selection and another for selected event processing. Executed sub-plans 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)
      )
)
```

Implementation of the sub-plans depends on event representation and representation of relevant processes for given events. Agent expects the incoming events to be in its input buffer and the event itself has a form of a grounded predicate. For our purposes the events will be sent by agent named GODI (we will discuss role of the GODI agent later in this text) and the predicate symbol will 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 should be also a plan with corresponding name stored in the plan base

π(plan_name,(actions…))

 Copyright © 2007 EUROSIM / SLOSIM

Now the processes for event selection and event processing can be shown.

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

$$\pi(\text{select_event},(?(\text{GODI}),(\text{cdr},\tau), @((\tau,_),\textbf{\textit{succeed}}), @(\text{select_event}))$$

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

$$\pi(\text{process_event},((\text{car}, \tau),@\hat{}\tau))$$

Plan for processing event is even simpler. It reaches plan name using the *car* function and then executes the plan. After plan execution, no matter if successful or not, the control is returned back to main control loop and another event can be processed.

4.2 Intention-driven agents implementation

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

To make such system with t-Sapi language we exploit its meta-reasoning abilities. In fact there will be two-levels of processes. At the upper level there will be process of goal and sub-goal processing and intention structure fabrications. This process senses for incoming events and would tries to make proper intention structure for these events. At the lower level will be the processes for intention achievements themselves. In Fig. 4 an example of BDI agent reasoning is shown.

At the beginning one goal is in agent's desire set and belief set is a state pictured with rectangle (a,). There is a plan library with one plan suitable for (yellow) goal and applicable in given environment state. Such plan is added into the intention set (b,). During execution of the plan the belief state changed and two new desires aroused. First desire (green) is a sub-goal of the only intention and second (brown) is another top-level goal (c,). Then there are two goals but only one relevant and applicable plan (for the green sub-goal) which is consequently added to the intention structure (d,).

Now we try to implement such system in t-Sapi. However the implementation of complete BDI agent is quite complex to fit it into this text we focus only at the most important implementation parts. In following paragraphs some basic constructions for intention forming and execution will be shown.

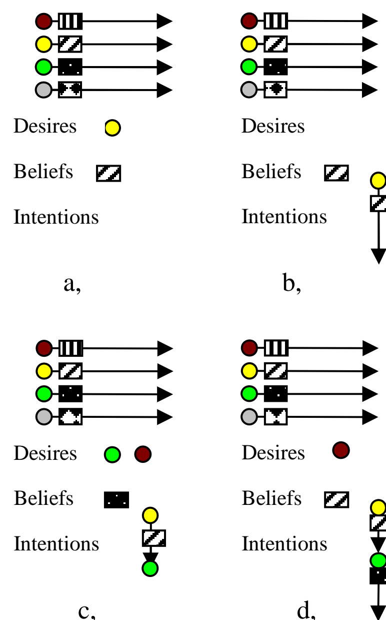

Fig. 4 Example of BDI based reasoning

The intention structure itself will be a structure of t-Sapi plans. At the lower level some plans stored in an intention structure would be executed. First we show how plans and some metadata could be stored in belief and plan bases. Each plan has defined purposes and condition of its usage. So in 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 relevant (for an event) and applicable (in actual state of 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.

BDI agent control loop will be similar to that of reactive agent.

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

 Copyright © 2007 EUROSIM / SLOSIM

Selection of an event is quite the same as before. The only change is that the sender is followed with 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 a list

(PID,(event,term1,term2…))

Now we aim our sights at event processing. We consider that there is an event in the register in the mentioned form and the agent needs to find out a plan which is relevant and applicable. The agent first withdraws all the triples relevant to the event and then it 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. Implementation of this is little bit difficult in t-Sapi 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 use. 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). Following line tests for presence of 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,
```

Three actions in line 4 stores the tuple (condition,plan) into the knowledge base. Then the condition itself is reached and test action with the condition is executed. If the plan does not fail so far then we have a plan that is relevant for the event and applicable in given belief base state.

Now the plan does the following: executes founded plan and possibly executes plan instance that raised the event. Rest of the "process_event" plan then continues with

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

Actions in lines 6 and 8 reach plans stored during the process and they execute event-related plan in line 7 and possibly executes plan instance which produced the event in line 9.

Another important issue of the BDI agent is how the plans for event processing are designed. In short there are several classes of actions distinguished to internal actions, external actions and goal statements. Although internal actions and external actions are similar to the knowledge base manipulation and communication actions provided by the t-Sapi language, we only show here how goal statements can be realized.

First we show goal *achievement* statement. It means that agent should actively behave to reach declared goal. Goal statement is in fact event processing. If a plan needs to set a sub-goal it simply send an event to its input buffer. But there must be also mention 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 $(a_1,a_2,\ldots a_n)$ and the first action should be a sub-goal statement. This can be implemented in the t-Sapi language with respect to the event-processing plan like this

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

Please note that symbols $a_2 \ldots a_n$ are again just some abstractions for some t-Sapi actions. But the principle is clear. Rest of the plan is executed for 0 steps. It means that just an instance called #plan1_i1 is created and stored in the plan base. Then a message is sent to itself (here we consider agent's name to be 'agent1') and if the specified event is processed by a sub-plan, the rest of the plan is executed as the instance.

Second possible goal declaration is called goal *testing*. In this case the agent waits whether declared goal is valid or not (the goal may occur for example by environment change - as an example let us mention goal 'is it night?'). In this case the test goal action could be implemented as

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

Based on similar principles like the achievement goal action the plan instance is used here. First a plan instance is made. Then a predicate is written into the knowledge base meaning that there is an instance waiting with test goal. The instance consists of predicate testing itself, execution of the rest of the plan and deletion of instance predicate from the knowledge base. Each time the main loop cycle reaches 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 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 here because it is similar to the “process_event” plan. Better after showing how some agent types can be implemented with t-Sapi we move forward and introduce T-Mass simulation principles.

5 Multiagent system development and simulation with T-Mass

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

5.1 Model-based design of MAS

Model based development of MAS has three main phases – creation of the model, design by simulation and realization of the system. In following points we extend each phase for closer insight into the process.

Creation of the model

1. Definition of agent roles necessary for the model.
2. Definition of agents' behavior (decision procedures, protocols, etc.)
3. Development or adoption of suitable agent architecture.
4. Implementation of particular agents.
5. Development of environment model.

Model-based design

6. Loop:
   6.1. Simulation of system run and checking of agent behavior.
   6.2. Identification of possible design mistakes.
   6.3. Handling problems and redevelopment of the multi-agent model.

Realization of the system

7. Realization of the agents with identical behavior as their models had.
8. Situating agents into the real environment.

The whole process of model-based design also includes stages that are out of the scope of the T-Mass toll. Especially the first three points are related rather to software engineering where methodology like GAIA [10] could help. To propose agent roles and agent one need to delimitate responsibilities, competences and protocols and for those find which kind of agent and which algorithms would be suitable.

The point 4 we have already discussed the in the previous sections. Environment modeling is of current interest in the researchers' community. Thus we only remarks that the model needs to be relevant and all important aspects for agent behavior must be present in the model. So we move on to the process of simulation that allows us checking whole system behavior and tuning particular agents' facilities.

5.2 Simulation with T-Mass

Current version of T-Mass allows making two-step synchronous si1mulation. In general there are two phases, first one of agents' acting and second one of environment evaluation.

5.2.1 The GODI agent

There is still the problem how to include agents' environment into the system model. As we outlined already the multi-agent community is in principal an open system influenced by a surrounding environment that can be affected by the agents. But the environment has not been introduced till now however it is important part of MAS.

In fact the environment is encapsulated in one of the agent's knowledge base and its evolution is under control of this agent. Such an agent is called GODI (General Object Design Interpreter). GODI is responsible for projection of other agents' actions to the environment, evolution of the environment due to the actions and exposition of the environment state to the other agents input buffers. The GODI maintains whole multi-agent model in its knowledge base that means that it has a model of environment as well as a model agent population. In each step the GODI can evolve the model and to send relevant messages to the agents. Furthermore also the communication among the agent is driven by the GODI and for this reason there are not direct interconnections among the particular agents. The only communication is between agents and the GODI and vice versa.

5.2.2 T-Mass Simulation loop

In recent realization of the T-Mass tool the simulation process runs in accordance with two phase's synchronous algorithm. In the first stage each agent runs their code until an external action in the form of communication act toward the GODI is performed. When every MAS agent has sent its external action to the GODI the second phase starts. GODI makes an evaluation of the MAS model then finds particular stimuli for every agent in the model and sends them their stimuli in a form of message.

The whole simulation process is driven by some signals which are sent to particular agent (recall Fig. 1). The signals used in the system are *Ready*, *ClearBuffer* (request for deleting the content of receivers input buffer), *Evolve* (continue process run), *Execute* (start process run) and *Terminate* (cancel process run). The behaviour of the simulator works by the following algorithm:

1. System Initialization, the agent population constitutes universe **U** = (GODI, $Agent_1$, $Agent_2$ … $Agent_n$)
2. T-Mass sends the *ClearBuffer* signal to every agent in the model

 Copyright © 2007 EUROSIM / SLOSIM

3. T-Mass sends the *Execute* signal to the GODI
4. GODI evolves the system model and sends the proper stimuli to the model agents
5. GODI sends the *Ready* signal to the T-Mass
6. T-Mass sends the *ClearBuffer* signal to the GODI
7. T-Mass sends the *Execute* signal to the model agents
8. Agents run their plans until they execute an external action
9. Agents send the *Ready* signal to the T-Mass.
10. T-Mass either
    10.1. sends the *Terminate* signal to the model agents and terminates the simulation.
    10.2. or sends nothing to them now
11. T-Mass sends the *Evolve* signal to the GODI.
12. GODI evolves the system model and sends the proper stimuli to the model agents.
13. GODI sends the *Ready* signal to the T-Mass.
14. T-Mass sends the *ClearBuffer* signal to the GODI.
15. T-Mass sends *Evolve* signal to every model agent.
16. GOTO 8

In each step the system allows to check particular agent's bases states as well as all the messages that had been sent between agents and the GODI. So the simulation is done when all the agents work well and they satisfy the reasons for which they had been made.

5.3 Realization of t-Sapi agent

Finally, if we are satisfied with the agents behavior in the model we would like to use them in a real environment. Our approach how to do it is to make platforms with t-Sapi interpret for each device in which the agent(s) should reside (computer operating systems, microcontrollers, etc.). Such platform should also provide basic algorithms which the agents use and furthermore it should be able to interpret agent actions in the real environment. As we showed before, the actions in the model are in the form of communication acts. But now the platform should execute some real actions when agent makes some communication acts. So however there are some actions in the form of communication and some of real acting the agents does not distinguish among them in this sense.

Realized agents then are faced with real environment and their success depends on how we managed to model the environment. For some reasons we can find that there is something wrong in the systems. Then we need to check if the environment does not behave in different way than we had expected or if we have not omitted something to verify via simulation. If such situation appears the agent and their capabilities should be redeveloped again.

6 Acknowledgement

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

7 Conclusion and future work

We presented a new way how the systems with agents can be developed. As main advances of the language we consider that it is easily interpretable, allows implementation systems based on behavior like intelligent agents are and is suitable for usage in systems for MAS modeling and simulating. Finally the language allows their mobility among platforms where interpreter of the language and some basic functions are present. Because this is language based at low level of abstraction and it is sometime uncomfortable to write codes directly in this language our next effort is to develop higher-level language. Such language would allow easier implementation of agents and t-Sapi is then used as destination language into which the agent code will be compiled.

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

8 References

[1] M. E. Bratman: *Intention, Plans and Practical Reason*, Harvard University Press, Cambridge, MA, 1987

[2] R. Brooks: “Intelligence Without Reason”, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pp. 569-595, 1991

[3] L. Cabac, T. Dörges: “Tools for Testing, Debugging and Monitoring Multi-Agent Applications” *Proceedings of the Workshop on PNSE'07*, Siedlce, Poland, 2007

[4] M. d'Iverno, M. Luck and M. Georgeff.: “The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System”, *Autonomous Agents and Multi-Agent Systems 9*, pp.5-53, Kluwer Academs Publisher, Netherland, 2004

[5] J. Ferber: *Multi-Agent Systems*, Addison-Wesley, Great Britain, 1999

[6] M. Huber: “JAM Agents in a Nutshell”, *Intelligent Reasoning Systems*, Oceanside, CA, USA, 2001

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

ISBN 978-3-901608-32-2

Copyright © 2007 EUROSIM / SLOSIM

[8] A. S. Rao and M. P. Georgeff,: "BDI Agents: From Theory to Practice", *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, San Francisco, USA, 1995

[9] Y. Shoham: "Agent-oriented programming", *Technical Report STAN-CS-1335-90*, Computer Science Department, Stanford University, Stanford, CA 94305, 1990.

[10] M.Wooldridge, N. Jennings, D.Kinny: "The Gaia Methodology for Agent-Oriented Analysis and Design", Autonomous Agents and Multi-Agent Systems, 3, pp. 285-312, Kluwer, The Netherlands, 2000

[11] F. Zbořil: Plánování a komunikace v multiagentních systémech, Brno, Czech Republic, 2004

[12] F. Zbořil, R. Kočí: "Intention structures modeling using object-oriented Petri nets", accepted for conference ISDA'07

[13] Foundation for Intelligent Physical Agents, http://www.fipa.org

ISBN 978-3-901608-32-2 *Copyright © 2007 EUROSIM / SLOSIM*