I T H E A

International Journal

INFORMATION THEORIES & APPLICATIONS

2009 Volume 16 Number 2

International Journal
INFORMATION THEORIES & APPLICATIONS
Volume 16 / 2009, Number 2

IJ ITA is official publisher of the scientific papers of the members of the ITHEA® International Scientific Society

IJ ITA welcomes scientific papers connected with any information theory or its application.

IJ ITA rules for preparing the manuscripts are compulsory.
The **rules for the papers** for IJ ITA as well as the **subscription fees** are given on *www.ithea.org* .
The camera-ready copy of the paper should be received by http://ij.ithea.org.
Responsibility for papers published in IJ ITA belongs to authors.

General Sponsor of IJ ITA is the **Consortium FOI Bulgaria** (www.foibg.com).

International Journal "INFORMATION THEORIES & APPLICATIONS" Vol.16, Number 2, 2009

Printed in Bulgaria

Edited by the **Institute of Information Theories and Applications FOI ITHEA®**, Bulgaria,
in collaboration with the V.M.Glushkov Institute of Cybernetics of NAS, Ukraine,
and the Institute of Mathematics and Informatics, BAS, Bulgaria.

Publisher: **ITHEA®**
Sofia, 1000, P.O.B. 775, Bulgaria. www.ithea.org, e-mail: info@foibg.com

ISSN 1310-0513 (printed) **ISSN 1313-0463 (online)** **ISSN 1313-0498 (CD/DVD)**

PROGRAMMING OF AGENT-BASED SYSTEMS

Dmitry Cheremisinov, Liudmila Cheremisinova

Abstract: The purpose of the paper is to explore the possibility of applying the language PRALU, proposed for description of parallel logical control algorithms and rooted in the Petri net formalism for design and modeling real-time multi-agent systems. It is demonstrated with a known example of English auction on how to specify an agent interaction protocol using considered means. A methodology of programming agents in multi-agent system is proposed; it is based on the description of its protocol on the language PRALU of parallel algorithms of logic control. The methodology consists in splitting agent programs into two parts: the block of synchronization and the functional block.

Keywords: multi-agent system, interaction protocol, BDI agent, parallel control algorithm.

ACM Classification Keywords: I.2.11 [Computer Applications]; Distributed Artificial Intelligence, Multiagent systems; D.3.3 [Programming Languages]: Language Constructs and Features – Control structures, Concurrent programming structures

Introduction

In recent years one of the most rapidly growing areas of interest for distributed computing is that based on the concept of agents and multiagent systems (MAS). The first MAS applications have appeared in the mid-1980s. Now they are becoming one of the most important topics in distributed and autonomous decentralized systems. There are increasing attempts to use agent technologies in variety of domains, ranging from manufacturing to process control, air traffic control and information management. Programming technology based on the use of interacting agents is considered the most promising tool of modern programming.

MAS is a computational system which consists of a number of agents that operate together in order to perform a set of tasks or to achieve a set of goals [Burmeister, 1992, Jennings, 2001, Lesser, 1999, Subrahmanian, 2000]. There exists variety of definitions of the notion of agent. The most of researchers adhere to the definition of M. Wooldridge and N.R. Jennings: agent is a hardware or software-based computer system that enjoys the following properties:

– *autonomy*: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal states;

– *social ability*: agents interact with other agents (and possibly humans) via some kind of *agent-communication language*;

– *reactivity*: agents perceive their environment, and respond in a timely fashion to changes that occur in it;

– *pro-activeness*: agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative.

MAS is usually specified as a concurrent system that consists of autonomous, reactive and internally-motivated agents acting in a decentralized environment. One key reason of the growth of interest in MAS is that the idea of an agent as an autonomous system, capable of interacting with other agents, is a naturally appealing one for software designers. Reactive agents do not have representations of their external environment and act using a stimulus response type of behavior, they respond to the present state of the environment. Robots are examples

of artificial agents. ARCHON [Burmeister, 1992] is the most well-known commercial system constructed on the basis of the conception of agents and designed for controlling the process of manufacturing articles.

The history of the development of the agent theory began with the problem of modeling the properties of living systems and dates back to the work of W. Pitts and W. MacCulloch on formal neurons, John von Neumann on self-reproducing automata, A.N. Kolmogorov on the theory of complexity, H. von Forster and Ilya Prigogine on the theory of self-organization, William Ashby on models of homeostasis, W.G. Walter on reactive robots, and John Holland on genetic algorithms.

Designing and building agent systems is difficult. They have all the problems associated with building traditional distributed, concurrent systems and have the additional difficulties that arise from having flexible and sophisticated interactions between autonomous problem-solving components. The big question then becomes how effective MASs can be designed and implemented.

The core concept of multi-agent systems is interaction; it is the foundation for cooperative or competitive behavior among several autonomous agents. Agent interactions are established through exchanging information in the form of messages that specify the desired performatives of interacting agents. Formalization of the concept of interaction as a method of transmitting messages from one dispatcher to several recipients through a transmission environment derives from Claude Shannon's work on communication theory [Shannon, 1948]. The subsequent formalization of this concept employed the theory of speech acts [Searle, 1986]. In this theory communication between agents is considered as a form of behavior, since particular types of sentences of natural language are of the nature of actions (called speech acts) and presume a "rational effect." Speech acts form the basis of the communication languages used by agents (e.g., the KQML and FIPA-ACL languages [ARPA, 1993, Finin, 1997, FIPA, 2002]); these languages also define the sets of permissible speech acts and semantics associated with these sets.

A set of interrelated messages forms conversations in which agents play different roles as a functions of their individual or common goals. Conversations in multiagent systems are based on interaction protocols that define all possible flows of conversations. Agent system can operate if agents have a common understanding of the possible types of messages, then they must know which messages they can expect in a particular situation and what they may do when they got some message. So messages exchanged between agents in some multi-agent system need to follow some standard patterns which are described in agent interaction protocol. Protocols play a central role in agent communication; they specify the sequences in which messages should be arranged in agent's interactions.

At this time, there are two major technical obstacles to the widespread adoption of multiagent technology: 1) the lack of systematic methodologies enabling designers to clearly specify and structure their applications as MASs and 2) the lack of widely available MAS toolkits. Flexible sets of tools are needed that enable designers to specify an agent's problem-solving behavior and to specify agents interaction, and then visualize and debug the behavior of the agents and the entire MAS. That is because designing agent systems is difficult enough: they have all the problems associated with building traditional distributed, concurrent systems and have the additional difficulties that arise from having sophisticated interactions between autonomous problem-solving components.

The other difficulty of existing methodologies of MAS designing consists in follows. In the majority of MAS models autonomous behavior of agents is described in terms of belief, desires and intentions (BDI) [Burmeister, 1992], and their communications are specified in terms of protocols which have no a direct relation with the first formalism. The behavior of agents is described in terms of formalisms of a high level abstraction (such as temporal logic), but the communication is specified in the concepts close to realization. The independent behavior of agents in the majority of models of multi-agent systems is described by means of formalisms of high level abstractness, but the communication is specified by the concepts close to realization. The difference of levels of

the description does not allow model communications between agents at the level in which their independent autonomous behavior is described. This problem arises because of absence of agent models that unify all aspects of local behavior and the communications (BDI-behavior and communication), and will be suitable for automated implementation. The main reason for the absence of such a unifying model is that there is no general conceptual basis that combines all the abstractions related to agents.

Representation languages from the theory of agents are used to overcome these methodological difficulties. Such languages are based on some formalism and a programming system that together specify semantics of the language used in programming agents. Though ever newer agent programming languages have been proposed in the literature, only some of them are entirely intelligible from the semantic point of view.

This paper explores the possibility of applying existing formal theories of description of distributed and concurrent systems to interaction protocols for real-time multi-agent systems. In particular it is shown how the language PRALU [Zakrevskij, 1996, Zakrevskij, 1999, Zakrevskij, 2001] proposed for description of parallel logical control algorithms and rooted in the Petri net formalism, can be used to describe agent interaction protocols. The described approach can be used for modeling complex, concurrent conversations between agents in a multi-agent system. It can be used to define protocols for complex conversations composed of a great number of simpler conversations. With the language PRALU it is possible to express graphically concurrent characteristics of a conversation, to capture the state of a complex conversation during runtime, and to reuse described conversation structure for processing multiple concurrent messages. It is demonstrated with a known example of English auction [FIPA, 2002] on how to specify an agent interaction protocol using considered means. Finally, using PRALU language we can verify some key behavioral properties of our protocol description that is facilitated by the use of existing software for the language PRALU [Cheremisinov, 1986, Cheremisinova, 2002, Zakrevskij, 1999].

Then, we introduce a methodology of programming agents in MAS; it is based on the language PRALU. When more complicated, multilayered and concurrent conversations take place among groups of agents, PRALU approach that we use appears to offer advantages over the colored Petri net techniques that are proven to provide the most powerful mechanism for specifying interaction protocols up until now [Bai, 2004]. Further, PRALU as the agent-oriented programming language has such an advantage that its semantics is based on logic formalism, this language allows a simple realization. We show how the behavior of MAS can be simulated entirely in the language PRALU. The offered methodology is based on two-block architecture of organization of the description and realization of MAS that consists of the block of synchronization and the functional block. The first one coordinates performance of parallel processes of agent program, that is, it controls the agent behavior. The second part operates data and carries out calculations connected with complex information structures. The distinctive feature of the methodology is that it allows separating development of the synchronizing part of agent programs from the functional one.

An Agent-Oriented Model

The application domains of MASs are getting more and more complex. Firstly, many current application domains of MASs require agents to work in changing environment (or world) that acts on or is acted on by the system. A closed system is one that has no environment; it is completely self-contained in contrast to an open (uncertain) system, which interacts with its environment. Any real system is open. The MAS must decide what to do and develop a strategy in order to achieve its assigned goals, within open environment. For this, the MAS must have a representation or a model of the environment within which it evolves. The environment is composed of situations. A situation is the complete state of the world at an instant of time.

The application of multi-agent systems to real-time environments can provide new solutions to very complex and restrictive systems such as real-time systems. A suitable method for real-time multi-agent system development must take into account the intrinsic characteristics of systems of this type. As a rule they are distributed, concurrent systems with adaptive and intelligent behavior. For agent-based systems to operate effectively, they must understand messages that have a common ontology underlying them. Understanding messages that refer to ontology can require a considerable amount of reasoning on the part of the agents, and this can affect system performance.

Taking the view of agents as practical systems the predominant approach to specifying agents has involved treating them as intentional systems that may be understood by attributing to them mental states such as beliefs, desires, and intentions [Wooldridge, 1995]. Following this idea, a number of approaches for formally specifying agents have been developed, which are capable of representing the following aspects of an agent-based system:

– the beliefs that agents have – the information they have about their environment, which may be incomplete or incorrect;

– the goals that agents will try to achieve;

– the actions that agents perform and the effects of these actions;

– the ongoing interaction that agents have – how agents interact with each other and their environment over time.

BDI agents are systems that are situated in a changing environment, receive continuous perceptual input, and take actions to affect their environment, all based on their internal mental state. In practical terms, beliefs represent the information an agent has about the state of its environment. It is updated appropriately after each action. The desires denote the objectives to be accomplished, including what priorities are associated with the various objectives. Intentions reflect the actions that must be fulfilled to achieve the goal (the rules to be fired).

Multi-agent conversations are built upon two components:

– agent communication language;

– interaction protocol.

There are a number of agent communication languages, such as Knowledge Query and Manipulation Language (KQML) [Finin, 1997] and Agent Communication Language (ACL), proposed by the Foundation for Intelligent Physical Agents (FIPA) [FIPA, 2002], and others designed for special purposes and that are like mentioned ones. These agent communication languages specify a domain specific vocabulary (ontology) and the individual messages that can be exchanged between agents.

Interacting agents should comply with an interaction protocol in order to engage permissible sequences of message exchange. When agent sends a message it can expect a response to be among a set of messages indicated by the accepted protocol. The interaction protocol can be assigned by the designer of the multi-agent system otherwise an agent needs to indicate the protocol that it wants to follow before it starts to interact with other members of the system.

Agent Interaction Protocols

Interaction protocols [FIPA, 2002] specify the sequences in which messages should be arranged in agent interactions. Protocol constrains number of sequences of allowed messages for each agent at any stage during a communicative interaction, i.e. it describes some standard pattern messages exchanged between agents need to follow. By the very nature of protocols as public conventions, it is desirable to use a formal language to represent them. When agents are involved in interactions where no concurrency is allowed, most conversation protocols are traditionally specified as deterministic finite automata (DFA) [Pitt, 1999] of which there are numerous

examples in the literature. DFA consists of a set of states, an input alphabet and a transition function, which maps every pair of states and input to the next state. In the context of interaction protocols, the transitions specify the communicative actions to be used by the various agents involved in a conversation. A protocol based on such a DFA representation determines a class of well-formed conversations. Conversations that are defined in this way have a fixed structure that can be laid down using some kind of graphical representation. However, they are not enough expressive to model complex interactions, especially those with some degree of concurrency.

Protocols can be represented as well in a variety of other ways. The simplest is a message flow diagram, as used by FIPA [FIPA, 2002]. More complex protocols will be better represented using a UML sequence (Unified Modeling Language) [Booch, 1999] and AUML (Agent UML) [Bauer, 2001, Odell, 2001, Winikoff, 2005], interaction diagram, statechart [Harel, 1998] and Colored Petri Net (CPN) [Bai, 2004, Jensen, 1992].

UML is one of the currently most popular graphical design languages that is de facto standard for the description of software systems. AUML extends UML sequence diagrams to support the specification of interactions between agents. The advantage of AUML is its visual representation as well as statechart [Harel, 1998], but unfortunately, AUML suffers from two issues. Firstly, the notation itself is not formally and precisely defined; and secondly, tool support for AUML is largely non-existent [Cost, 1999]. AUML diagrams only offer a semi-formal specification of interactions. This weakness may generate several problems. Indeed, the lack of formal semantics in AUML, such as in UML, can lead to several incoherences in the description of a MAS's behavior. It is difficult, especially in the case of complex MAS, to detect this kind of defects [Poutakidis, 2002].

A CPN model of a system describes the states, which the system may be in, and the transitions between these states. CPNs provide an appropriate mathematical formalism for the description, construction and analysis of distributed and concurrent systems. CPNs can express a great range of interactions in graphical representations and well-defined semantics, and allow formal analysis and transformations [Bai, 2004, Murata, 1989]. By using CPNs, an agent interaction protocol can be modeled as a net of components, which carry the protocol structure. Using CPNs to model agent interaction protocol, the states of an agent interaction are represented by CPN places. Each place has an associated type determining the kind of data that the place may contain. Data exchanges between agents are represented by tokens, and the colors of tokens indicate the data value of the tokens. The interaction policies of a protocol are carried by CPN transitions and their associated arcs. A transition is enabled if all of its input places have tokens, and the colors of these tokens can satisfy constraints that are specified on the arcs. A transition can be fired, which means the actions of this transition can occur, when this transition is enabled. When a transition occurs, it consumes all the input tokens as computing parameters, conducts conversation policy and adds new tokens into all of its output places. After a transition occurs, the state (marking) of a protocol has been changed and a protocol will be in terminal state when there is no enabled or fired transition.

There are a number of works using Petri Nets or CPNs to model agent interaction protocols [Cost, 1999, Poutakidis, 2002], there have been also some works on the investigation of flexibility, robustness and extensibility of protocols [Hutchison, 2002]. Today, only Petri Net models are considered to be one of the best ways to model agent interaction protocols. However the notion of an agent executing an action with Petri Net is not explicit in the notation [Paurobally, 2002]. A different PN can be assigned to each agent role, raising questions about how the entire protocol is inferred and the reachability and consistency of shared places. If a single Petri net is partitioned for each role, this leads to a complex diagram where a partition is required for each agent identified. Furthermore, alternative actions and states either agree or reject but not both, cannot be expressed in standard Petri nets.

It is necessary for any protocol itself to be correct and verifiable. If it is not correct then the agents that follow it may perform contradictory and unexpected actions leading to possible breakdown of the interaction. The central problem of the verification of interactions that take place in open (not being cooperative) systems is the problem

of conformance inspection between behavior of agents and interaction protocol. That is the protocol must be understandable by all agents of the system and they must behave according to this protocol. Thus a language used to represent MAS protocol must be clear enough to allow means for protocol verification. We consider that a language for developing protocols is needed which can ideally meet the following requirements:

1) provide a graphical representation for ready perception of structure by MAS designer;

2) have an unambiguous formal specification with clear semantics for verification;

3) be close to an executable language for implementation purposes;

4) for relative tractability, maintain a propositional form for a formal language;

5) provide well-defined program logic for ensuring complete protocols and validating the properties of a protocol;

6) exhibit enough expressiveness for agent interactions and nested interactions.

Keeping in mind complex systems characterized by complex interaction, asynchronism and concurrency we propose to use for the purposes of their description a special language PRALU [Zakrevskij, 1996, Zakrevskij, 1999, Zakrevskij, 2001] satisfying all mentioned requirements. It has its background in the Petri net theory (expanded nets of free choice – EFC–nets investigated by Hack [Hack, 1972]) but possesses special means for keeping track of the current states of the conversation, receiving messages and initiating responses. The language combines properties of "cause-effect" models with Petri nets. It is intended for a wide application in engineering practice and is well suited for representation of the interactions involved in concurrent system; synchronization among them and then it is simple enough for understanding. The language PRALU supports hierarchical description of the algorithms that is especially important in the case of complex systems. At last a powerful software has been developed that provides correctness verifying, simulation, hardware and software implementation of PRALU-descriptions [Cheremisinov, 1986, Cheremisinova, 2002]. The review of obtained results in this field one can find in [Cheremisinova, 2002].

PRALU Language

Any algorithm in PRALU consists of sequences of operations to be executed in some pre-determined order. Two basic operations are used in PRALU: acting "$\rightarrow A$" and waiting "$-p$" operations. The first one changes the state of the object under control, whereas the second one is passive waiting for some event without affecting anything. In simple case A and p are conjunctive terms, so acting and waiting operations can be interpreted as waiting for event $p = 1$ and producing event $A = 1$. But A can be understood too as a formulae defining operations to be performed and p as a predicate defining condition to be verified. For example, acting and waiting operations could be specified by the expressions such as

$-(a > b + c)$ and $\rightarrow (a := b + c)$.

The sequences consisting of action and waiting operations are considered to be linear algorithms. For instance, the following expression means: wait for p and execute A, execute B, then wait for q and execute C:

$-p \rightarrow A \rightarrow B - q \rightarrow C$.

In general, a control algorithm can be presented as an unordered set of chains α_j in the form

$\mu_j : - p_j L_j \rightarrow \nu_j$,

where L_j is a linear algorithm, μ_j and ν_j denote the initial and the terminal chain labels being some subsets of integers from the set $M = \{1, 2, ..., m\}$: $\mu_j, \nu_j \subset M$ and the expression "$\rightarrow \nu_i$" presents the transition operation: to the chains with labels from ν_j.

Chains can be fulfilled both serially and in parallel. The order in which they should be fulfilled is determined by the variable starting set $N_t \subseteq M$ (its initial value $N_0 = \{1\}$ as a rule): a chain $\alpha_j = \mu_j : - p_j L_j \to \nu_j$ (that was passive) is activated if $\mu_j \subseteq N_t$ and $p_j = 1$. After executing the operations of the algorithm L_j, N_t gets a new value $N_{t+1} = (N_t \setminus \mu_j) \cup \nu_j$. The algorithm can finish when some terminal value of N is reached (one-element as a rule), at which time all chains became passive. But the algorithms can also be cyclic; they are widely used when describing production processes.

When the conditions $\mu_j \subseteq N_t$ ($|\mu_j| > 1$) and $p_j = 1$ are satisfied for several chains simultaneously these chains will be fulfilled concurrently. On the contrary chains with the same initial labels are alternative (only one of them can be fulfilled at a time), they are united in a sentence with the same label as will be shown below.

Thus PRALU allows concurrent and alternative branching, as well as merging concurrent and converging alternative branches. These possibilities are illustrated with the following examples of simple fragments [Zakrevskij, 2000]:

Concurrent branching	Merging concurrent branching	Alternative branching	Converging alternative branching
1: ...→2.3	2: ...→4	1: – *a*...→2	2: ...→4
2: ...	3: ...→5	– $\bar{a}$ ...→3	3: ...→4
3: ..	4.5: ...		4: ...

There exist in PRALU two syntactic constraints on chains that restrict concurrent and alternative brunching. If some chains are united in the same sentence (they have the same initial labels) they should have orthogonal predicates in the waiting operations opening the chains:

$(i \neq j)$ & $(\mu_i \cap \mu_j \neq \varnothing) \to (p_i$ & $p_j = 0)$.

The other constraint is similar to the corresponding condition specific for extended nets of free choice (Hack [Hack, 1972]):

$(i \neq j)$ & $(\mu_i \cap \mu_j \neq \varnothing) \to (\mu_i = \mu_j)$.

PRALU language has some more useful properties that can be usefull for description of complex interaction protocols.

1. PRALU algorithms can be expressed both in graphical and symbolic forms.

2. PRALU language permits hierarchical description of protocols. The two-terminal algorithms (having the only initial and the only terminal chain labels) may be used as blocks (invoked as complex acting operations) in hierarchical algorithms.

3. In PRALU there exist some additional interesting operations that can be useful when describing interaction protocol. Among those are suppression operations that provide response on special events that can take place outside or within control systems. Suppression operations

"$\to *$", "$\to *\gamma$", "$\to '\gamma$" (where $\gamma \subseteq M$)

interrupt the execution of all concurrently executed algorithm chains (in the case of "$\to *$"), only those ones mentioned in γ (in the case of "$\to *\gamma$") or are not mentioned in γ (in the case of the operation "$\to '\gamma$"). These operations break the normal algorithm flow.

4. In addition to logical variables, arithmetic variables are also used in the PRALU language. In particular, among arithmetic operations ought to be mentioned the following operations are introduced:

– timeout operation "– *n*" – delay for *n* unit times;

– counting operations that count event occurrences:

"($x = n$)" – assignment of a natural value n to a multivalued variable x;

"(x +)" and "(x –)" – assignment of unit positive and unit negative increment in value;

"–($x = n$)" – awaiting the start of an event: the value of x is equal to n.

Representing Agent Interaction Protocols in PRALU

As an example of an interaction protocol let consider English auction [FIPA, 2002]. The auctioneer seeks to find the market price of a good by initially proposing a price below that of the supposed market value and then gradually raising the price. Each time the price is announced, the auctioneer waits to see if any buyers will signal their willingness to pay the proposed price. As soon as one buyer indicates that it will accept the price, the auctioneer issues a new call for bids with an incremented price and continues until no buyers are prepared to pay the proposed price. If the last price that was accepted by a buyer exceeds the auctioneer's (privately known) reservation price, the good is sold to that buyer for the agreed price. If the last accepted price is less than the reservation price, the good is not sold.

In the case of the auction there are participants of two types: the Auctioneer and Buyers. So, we have two kinds of interaction protocols – those of Auctioneer and of Buyers. So, we have two kinds of interaction protocols – those of Auctioneer and of Buyers. The last participants are peer and should be described with identical interaction protocols.

Interaction protocol as a whole can be represented in PRALU as three complex acting operations – blocks that are exchanging with values of logical variables, only such variables are mentioned in them. Each block has some sets of inputs and outputs that are enumerated in brackets following the block name (the other variables of a block are its internal). Initialization of a block algorithm is depicted by the fragment such as "$\rightarrow *$Buyer". The operation Buyer exists in as many copies as the number of participants of the auction, the copies differ in their indexes only.

The modeling of the process of auction begins with the execution of "Main_process" triggering event that initiates the interaction protocol execution. Here the processes Auctioneer and Buyer$_n$s are executed concurrently. For the sake of simplicity we limit the number of buyers to two. The process Auctioneer starts with sending the first message (start_auction) that is waited by others participants to continue communication.

Below PRALU description of the auction interaction protocol is shown. Here we show the only block Buyer$_n$, but for real application (intending to simulate the process of auction, for example) we should have as many proper copies as it has been used (in our case – two). Here through "′var1" the inversion of the Boolean variable var1 is denoted.

Main_process ()

1: $\rightarrow$ 2.3.4

2: $\rightarrow *$Auctioneer $\rightarrow$ 5

3: $\rightarrow *$Buyer$_1$ $\rightarrow$ 6

4: $\rightarrow *$Buyer$_2$ $\rightarrow$ 7

5.6.7: $\rightarrow$ **.**

Buyer$_n$ (start_auction, price_proposed, end_auction / accept_price$_n$, not_understand)

1: – start_auction $\rightarrow$ 2

2: – price_proposed $\rightarrow *$Decide (/ decision_accept, decision_reject) $\rightarrow$ 3

– end_auction → .

3: – decision_accept → accept_price$_n$ → 4

– decision_reject → ′accept_price$_n$ → 4

– not_understand → 4

4: – timeout → 2

Auctioneer (accept_price$_1$, accept_price$_2$, not_understand / start_auction, price_proposed, end_auction)

1: → start_auction → 2

2: → ′accept_price$_1$.′accept_price$_1$.′not_understand →∗Price_propose (/price_proposed) → 3

3: – not_understand → 2

– accept_ price$_1$ → 4

– accept_ price$_2$ → 4

–′not_understand.′accept_ price$_1$.′accept_ price$_2$ → end_auction →
∗Is_reservation_price_exceeded (/ is_exceeded) → 6

4: →′price_proposed.′win$_1$.′win$_2$ → 5

5: – accept_ price$_1$ →win1 → 2

– accept_ price$_2$ →win2 → 2

6: – is_exceeded →good_sold → 7

– ′is_exceeded →′good_sold → 7

7: → .

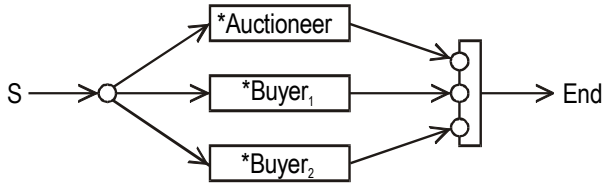
Fig. 1 depicts graphical schemes of three mentioned PRALU-blocks: Main_process, Auctioneer and Buyer$_n$.

It is assumed that all unformalized operations are referred to as acting operations that set values of logical variables assigned to them. For example, Buyer's operation "Decide" decides for accepting or rejecting the announced price. Depending on adopted decision, it outputs true value of logical variable "decision_accept" or of logical variable "decision_reject". In a similar, Auctioneers operation "Price_propose" proposes an initial price or increments the charged price outputting true value of logical variable "price_proposed"; the operation "Is_reservation_price_exceeded" verifies if the price accepted by a buyer exceeds the auctioneer's reservation price outputting true or false value of logical variable "is_exceeded".

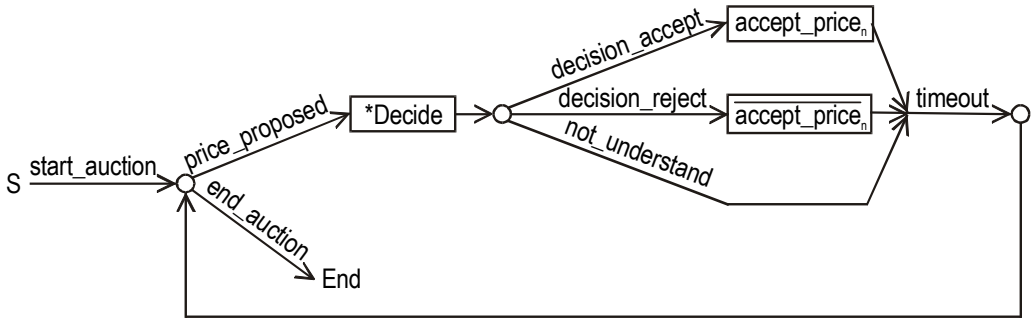
The operation "– timeout" (where "timeout" is an integer number) means waiting for "timeout" unit times before doing something followed it. The operation "→." is interpreted as the transition to an end of the process described by the block. When the processes of Auctioneer and all of Buyers reach their end in the Main_process the transition to its finish is executed.

BDI Architecture for the Language PRALU

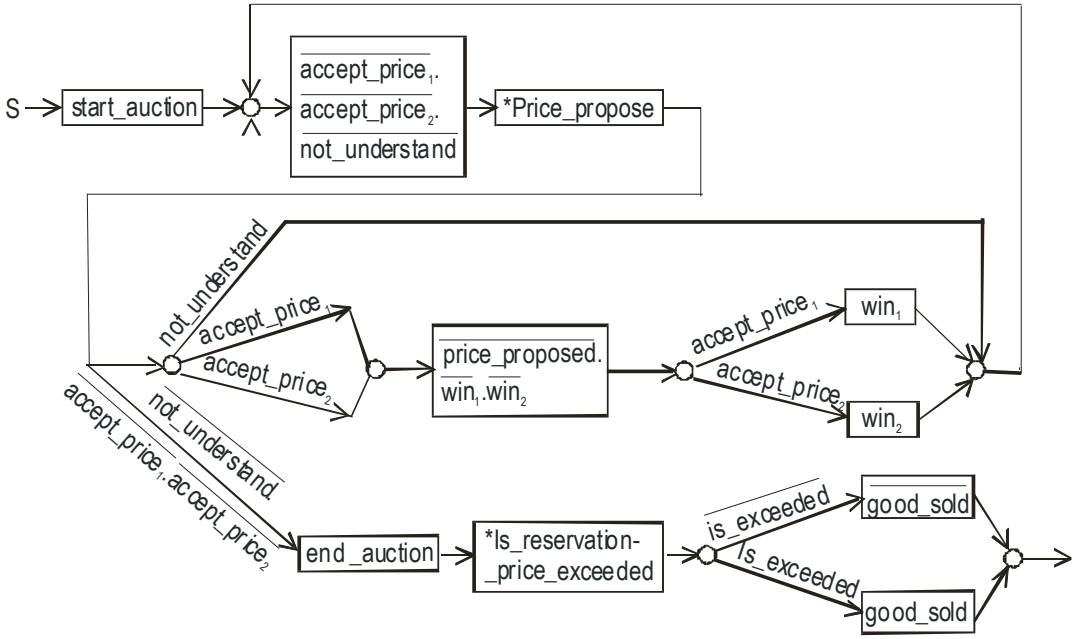
In practical programming, an agent is a well-formed entity incorporated in a computer system designed to perform flexible, independent actions for the purpose of attaining specified goals. Agents differ from ordinary software by the complexity of the interaction and communication scenarios. Any physical multiagent system is open, i.e., the agents function in a varying environment, which affects the agents and varies as a consequence of their operations. The system must arrive at decisions so as to attain the objectives assigned to it, and for this purpose it must possess a model of the environment in which its behavior evolves.



Main_process ()



Buyer$_n$ (start_auction, price_proposed, end_auction / accept_price$_n$, not_understand)



Auctioneer (accept_price$_1$, accept_price$_2$, not_understand / start_auction, price_proposed, end_auction)

Fig. 1. English auction interaction protocol in PRALU

Over the last decade, the specification and application of BDI agents (belief, desire, intention) [Burmeister, 1992] have received a great deal of attention. Within the BDI architecture agents are associated with beliefs (typically about the environment and other agents), desires or goals to achieve, and intentions or plans to act upon to achieve its desires. These three components completely set a state of the "mind" of the agent.

In programming terms, beliefs represent knowledge (information) a BDI agent has about the state of the environment; that is updated after each action. The desires denote the objectives to be accomplished, taking into account their priorities. Intentions reflect the actions that should be fulfilled to achieve the goal (the rules to be fired). Interaction protocols reduce search space of possible decisions, owing to limited range of answers to possible messages for the given situation. Ontology [Gruber, 1993] as a formal specification of terms of a subject domain and relations between them is almost equivalent to the notion of semantics of programming language.

In our case ontology will define the terms of BDI architecture in terms of the language PRALU. It is possible to consider, that BDI agent consists of the sets of beliefs *B*, plans *P*, situations *E*, actions *A* and intentions *I*. When the agent notes a change in its environment, it decides, that there an event takes place representing some situation from *E*. Registration of the event consists in changing a state of the agents "mind": a choice of some belief from *B*. According to it and the desire (defined by some plan from *P*) the agent intends to execute some intention representing some sequence of actions from *A* to achieve the goal chosen from *I*. Thus, planned actions are defined by the chosen plan from *P*. After they were carried out, the current situation of the environment is changed.

In traditional parallel programming languages the basic concepts are data and calculation control. Data are represented by values of variables, and the control is specified by a set of processes which transform local memory states defining variable values. The concept of the intellectual agent introduces new ideas of data manipulation in parallel programming. Agent's states are set by more complex information structures of predicates logic of the first order or modal logic. Calculations as transformations of memory states are controlled by protocols specifying communications between agents.

The program of a BDI agent is a set of plans defining actions by means of which the agent should reach a goal of its functioning. The plan consists of head labels, a body and tail labels. The body of the plan is a sequence of actions, by means of which the agent should reach a goal of its functioning, and conditions which the agent should check up. The head and tail labels of the plan symbolize intentions. Critical concept for the behavior of the agent is the concept of active intention. The plan will be carried out only when all intentions from its head are active. After executing the plan all intentions from the head become inactive and intentions from the tail quite the contrary become active. The current set of active intentions always is not empty, the body of a plan and a set of tail intentions can be empty. Such a model of BDI agent can be easily described on PRALU: a plan has a direct analogy with a chain α_j (in the form μ_j: $-p_j L_j \to \nu_j$, L_j: "$-p \to A -q \to B -r \to C$…"). The action leading to some goal can be described by acting operation "$\to g$" (done *g*) that may be executed, if some belief has appeared correct. Check of such condition is described by waiting operation "– a" (happens a). An agent carries out the action following waiting operation only after the moment when "belief *a*" becomes true.

Methodology of Programming Agents on PRALU

We propose the natural methodology of designing agent program based on splitting the program into two parts: synchronization and functional blocks (Fig. 2). The first block coordinates performance of parallel processes of the agent program, that is, it controls the agent behavior. The synchronization block should be described on PRALU language. The functional block operates data and carries out calculations connected with complex information structures (formulas of predicate or modal logic). This block is realized in programming language.

Such a splitting is carried out at the level of the project statement of MAS. The functional part is presented by predicates that describe memory states of the agent program (or/and external environment) or prescribe performance of some actions. In PRALU-description the appropriate logical variable is introduced for each predicate, setting true value to this variable starts the process of the predicate calculation. At the stage of verification of logical consistency it is better to interpret predicates as independent logical variables.

For example, the predicate *Decide* is used in the description of the behavior of agent Buyer. In the synchronization block in Fig. 3 it is represented by the Boolean variable *Decide*. In the functional block, the meaning of a predicate is indicated by pseudo-code, which shows the use of the variable *Decide* and rules for computing two other Boolean variables, *decision_accept* and *decision_reject*.

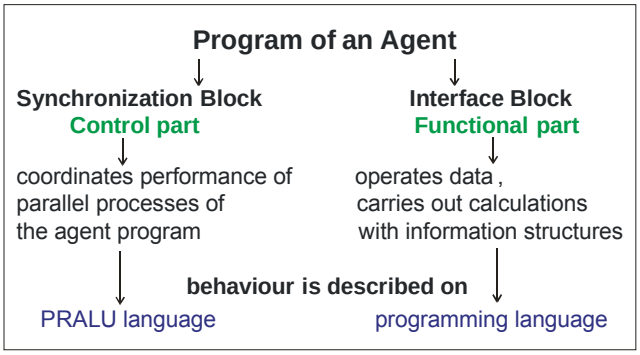


Figure 2. Splitting an agent program

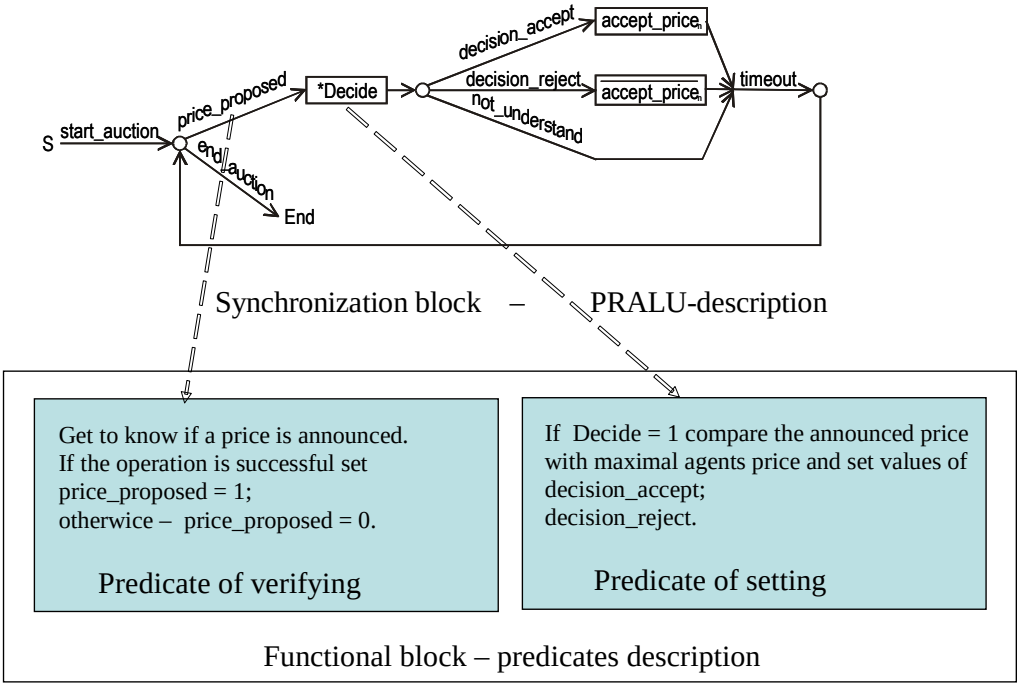


Figure 3. The program of the agent Buyer in English auction

Such an approach allows separating development of the synchronization part of the PRALU-description from the functional one. When designing MAS, the implementation of the functional part can be delayed concentrating designer's attention on working out the synchronization block implementing interaction protocol. This may significantly simplify the process of verifying the logical consistency of the behaviors of agents.

Program Implementation of PRALU-Descriptions

Programming in PRALU, the designer of the agent specifies sets of its belief, plans and intentions. The process of designing an agent program consists of the following stages.

1. Splitting at functional level the specification of MAS to be designed into synchronization and functional parts.

2. Development and analysis of PRALU-description of the synchronization part of MAS, being based on informal specification of its interaction protocol.

3. Verification of logic consistency of the MAS behavior by checking correctness and simulation of PRALU-description. Elimination of mistakes found out.

4. Program implementation of the functional part of MAS.

5. Development of computer program implementing behavior of MAS: translating PRALU-description of MAS on programming language and binding it with the programs of the functional part.

6. Testing of the generated programs.

The use of the PRALU language as a tool for representing the specifications of agents as well as the executability of these specifications makes the process of designing a multiagent system a structured process. All formal tasks may be implemented in the language and software tools for the implementation of these tasks are available. Moreover, there is the powerful theory supported with software tools for verification, simulation, hardware and software implementation of the descriptions of algorithm entity incorporated in a computer system in the PRALU language [Cheremisinov, 1986, Cheremisinova, 2002, Zakrevskij, 1999].

In the process of program implementation of PRALU-description it is translated on the intermediate language applied in the simulation program [Cheremisinov, 1986] too. The program that is generated by the PRALU compiler consists of two interacting blocks – 1) calculation of responses, and 2) control of sensors and servo mechanisms of the system. The control structure of the response computation program consist in an infinite cycle involving input of signals from the agent's sensors that constitute its beliefs into the internal memory, computation of the agent's responses from the values of these signals, and output of signals to the agent's servo mechanisms. All communication related to control and to data between the response computation block, sensor control block, and the servo mechanisms are input into the representation of the PRALU description implemented in the intermediate language.

To implement a parallel algorithm on the only processor [Cheremisinov, 1986], it is necessary to order the operations of the intermediate language by means of the PRALU intermediate language planner, which has its own properties and methods. The properties of the planner consist in the presence of two queues, a queue of waiting and a queue of ready branches. The methods of a program planner consist in triggering, halting, and pausing subprocesses. In the planner's initial state the queue of ready branches contains the first operation of the algorithm while the queue of waiting branches is empty. The program automatically orders the sequence of execution of operations that are in the process of being executed and there is no need for preliminary planning.

In each cycle the planner extracts the next branch from the queue of ready branches, executes a corresponding chain of algorithm in PRALU, and places all the branches that must be executed following the given branch into the waiting queue. If the planner discovers that the queue of ready branches is empty, (1) it transfers the contents of the queue of waiting branches to the queue of ready branches, rendering the queue of waiting branches empty; and (2) executes information exchange with the environment. Such a sequence of execution of chains of an algorithm in PRALU guarantees that operations of the initial algorithm that may be executed in parallel are all of the same length. Thus, the program implementation of PRALU possesses the semantics of measurable time that satisfy a rendezvous condition [Cheremisinov, 2008].

In the two-block model of the program, the predicates of the functional part are considered as a “supplementary tool” of sensors and servo mechanisms of the agent that generates logical signals or is triggered by a signal. If the results of the execution of computational operations must be taken into account in the control process, the supplementary tool will generate logical signals expressly for this goal. In the program that controls the sensors and servo mechanisms, this supplementary tool is described in the form of expressions in the ordinary programming language by the designer of the agent’s program.

This strategy of programming of agents is especially simple where the Forth programming language is used as a platform [Moore, 1974]. In this case it is only necessary to define a single word of Forth for each operator of an intermediate language.

The principle of implementation of the Forth language, i.e., threaded code, may be used for programming of agents in a C (or C++) language platform. The threaded code, a strategy used in the Forth language, consists in representation of a program in which nearly all elements are denoted by procedure calls (some procedures presuppose that data are located behind them in the threaded code). A stack is used to implement procedure calls; to distinguish it from a data stack, such a stack is referred to as a return stack and is used for storage of procedure data. Queues of ready and waiting branches of the intermediate language are implemented by means of return and data stacks, respectively.

The transformation of a representation of an algorithm in PRALU into a C program is executed by a single-pass text translator that converts the operators of the intermediate language into procedure calls in C language syntax. The efficiency of a program constructed in this way may be estimated using as an example the implementation of one of the most difficult operations of the intermediate language, the pause operation. The job executed by this operation consists in storage of the address of the next procedure call in the data stack and a transfer to execution of a procedure the address of which is extracted from the return stack. The C compiler constructs a fragment of code consisting of only two machine instructions for execution of these operations.

When programming agents on the base of a C language platform, procedures defined by predicates should be written in C using the descriptions like in Fig. 3 as a model.

Most of the well-known systems of logical programming of agents use a computation model from the Prolog language [Endriss, 2004]. Prolog is based on first-order predicate logic and the objects which the program manipulates are symbols without any meaningful interpretation. Execution of a Prolog program involves a process called unification and consists in scanning a database of facts and selecting values that satisfy some query. Unification is based on syntactic identity. In order to find a set of solutions, Prolog traverses a search tree, tests a set of variants, most of which do not occur in a solution, and, if unsuccessful, returns to a previous state and tests a different branch. For complex problems this search process requires great expenditure of memory and time, which is one reason for the computational inefficiency of Prolog.

In comparing the proposed PRALU-based logical programming system with other systems, for example, Prolog-based systems [Endriss, 2004], the improved computational efficiency of PRALU-based systems is apparent; in fact, the computational efficiency of PRALU-based systems is comparable with that of C language programs.

Software tools for automatic development, debugging, hardware and software implementation have been developed for the PRALU language [Cheremisinov, 1986, Cheremisinov1, 1986, Cheremisinova, 2002]. In particular, a modeling program may be used to display the behavior of new protocols written in PRALU. Display of behavior consists in exhibiting the control states and states of the variables of an algorithm in PRALU. The control states are exhibited by distinguishing operations in the text of the algorithm that are being executed at a given time (display of activity points), as it is done in most debuggers.

Conclusion

This paper has addressed the need for formalized and more expressive logical and graphical methodologies for specifying (and then validating) interaction protocols in multi-agent systems. Towards this, it was proposed to use the formal language PRALU intended for the representation of complex interactions involved in concurrent system, being in need of synchronization among these interactions. It was demonstrated as well how PRALU algorithms could be used for the specification of multi-agent interaction protocols by the example of English auction. In favor of using the language PRALU is the existence of a great deal of methods and software developed for simulation of PRALU algorithms as well as for their hardware and software implementation.

The methodology is suggested that ensures structuring the process of MAS designing on the base of PRALU language by means of separating calculation part of MAS specification from its control part. That allows the designer to concentrate on more complex stage of MAS designing – its interaction protocol. It is shown that language PRALU is very suitable for specifying interaction protocols of MAS and for implementation of suggested methodology. The obtained results are directed towards the usage when designing parallel/distributed technical systems too.

Bibliography

[ARPA, 1993] Specification of the KQML Agent-Communication Language, ARPA Knowledge Sharing Initiative, External Interfaces Working Group, July 1993.

[Bai, 2004] Quan Bai, Minjie Zhang Khin, Than Win. A Coloured Petri Net Based Approach for Multi-agent Interactions. In: The 2nd International Conference on Autonomous Robots and Agents, Palmerston North, New Zealand, 13 – 15 December 2004.

[Bauer, 2001] B. Bauer, J.P. Müller, J. Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. In: Agent-Oriented Software Engineering, Ed. P. Ciancarini and M. Wooldridge, Springer-Verlag, Berlin, 2001.

[Booch, 1999] G Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide, Addison Wesley, 1999.

[Burmeister, 1992] B. Burmeister, K. Sundermeyer. Cooperative problem-solving guided by intensions and perception. In: Demazean Decentralized A.I. 3. Ed. E. Werner and Y. Amsterdam, The Netherlands, North Holland, 1992.

[Cheremisinov, 1986] D.I. Cheremisinov. Implementation of parallel algorithms on a microprocessor. In: Programming, 1986, No 1 (in Russian).

[Cheremisinov1, 1986] D.I. Cheremisinov. LyaPAS-M Programming System Based Package for the Development of Microprocessor Programs. In: Upravlyayushchie sistemy i mashiny, 1986, No 1 (in Russian).

[Cheremisinov, 2008] D. Cheremisinov, L. Cheremisinova. Formalization of interaction events in Multi-agent Systems. In: Information Technologies & Knowledge (IJ ITK), 2008, Vol. 2, No. 2.

[Cheremisinova, 2002] L.D. Cheremisinova. Realization of parallel algorithms for logical control. Institute of Engineering Cybernetics of NAS of Belarus, Minsk, 2002 (in Russian).

[Cost, 1999] R. Cost. Modeling Agent Conversations with Coloured Petri Nets. In: Working Notes of the Workshop on Specifying and Implementing Conversation Policies, Seattle,Washington, 1999.

[Endriss, 2004] U. Endriss, N. Maude, F. Sadri, F. Toni. Logic-Based Agent Communication Protocols. In: Advances in Agent Communication, Ed. F. Dignum, Laboratory Notes in Artificial Intelligence, 2004, Vol. 2922.

[Finin, 1997] F. Finin, Y. Labrou, J. Mayfield. KQML as an agent communication language. In: Software Agents, Ed. J.M. Bradshaw, MIT Press, 1997.

[FIPA, 2002] Foundation for Intelligent Physical Agents (FIPA), Communicative Ac Library Specification, 2002; http://www.fipa.org/specs/fipa00037.

[Gruber, 1993] T.R. Gruber. A Translation Approach to Portable Ontologies. In: Knowledge Acquisition, 1993, Vol. 5, No 2.

[Hack, 1972] M. Hack. Analysis of production schemata by Petri nets. Project MAC-94, Cambridge, 1972.

[Harel, 1998] D. Harel, M. Politi. Modeling reactive systems with statecharts. McGraw-Hill, 1998.

[Hutchison, 2002] J. Hutchison, M. Winikoff. Flexibility and Robustness in Agent Interaction Protocols. In: Proceedings of the 1st International Workshop on Challenges in Open Agent Systems, Bologna, Italy, 2002.

[Jennings, 2001] N. R. Jennings. An Agent-Based Approach for Building Complex Software Systems. In: Communications of the ACM, 2001, Vol. 44, No 4.

[Jensen, 1992] K. Jensen. Colored Petri Nets – Basic Concepts. In: Analysis Methods and Practical Use, Springer-Verlag, Berlin, 1992, Vol. 1.

[Lesser, 1999] V. Lesser. Cooperative Multiagent Systems: A Personal View of the State of the Art. In: IEEE Trans. on Knowledge and Data Engineering, 1999, Vol. 11, No 1.

[Moore, 1974] C.H. Moore. FORTH: A New Way to Program a Mini-computer. In: Astro. Astrophys. Suppl., 1974, Vol. 5.

[Murata, 1989] T. Murata. Petri Nets: Properties, Analysis and Applications. In: Proceedings of the IEEE, 1989, Vol. 77, No 4.

[Odell, 2001] J.J. Odell, H.V.D. Parunak, B. Bauer. Representing Agent Interaction Protocols in UML. In: Agent-Oriented Software Engineering, Ed. P. Ciancarini and M. Wooldridge, Springer-Verlag, Berlin, 2001.

[Paurobally, 2002] S. Paurobally, J. Cunningham. Achieving Common Interaction Protocols in Open Agent Environments. In: Proceedings of 1st International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS'02), Bologna, Italy, July 2002.

[Pitt, 1999] J. Pitt, A. Mamdani. Protocol-based Semantics for an Agent Communication Language. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-1999), Morgan Kaufmann Publishers Inc., San Francisco, USA, 1999.

[Poutakidis, 2002] D. Poutakidis, L. Padgham, and M. Winikoff. Debugging Multi-Agent Systems Using Design Artefacts: the Case of Interaction Protocols. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi Agent Systems, Bologna, Italy, 2002.

[Searle, 1986] J.R. Searle. What is a Speech Act. In.: Novoe v zarubezhnoy lingvistike, 1986, issue 17 (in Russian).

[Shannon, 1948] C.E. Shannon. A Mathematical Theory of Communication. In: Bell Syst. Tech. J., 1948, Vol. 27.

[Subrahmanian, 2000] V.S. Subrahmanian, P. Bonatti, J. Dix et al. Heterogeneous Agent Systems. MIT Press, 2000.

[Winikoff, 2005] M. Winikoff. Towards Making Agent UML Practical: A Textual Notation and a Tool. In: Proceedings of the Fifth International Conference on Quality Software, September 19–20 2005,.

[Wooldridge, 1995] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. In: The Knowledge Engineering Review, 1995, 10(2).

[Zakrevskij, 1996] A.D. Zakrevskij. Parallel logical control algorithms: verification and hardware implementation. In: Computer Science Journal of Moldova, 1996, Vol. 4, No 1.

[Zakrevskij, 1999] A.D. Zakrevskij. Parallel algorithms for logical control. Minsk, Institute of Engineering Cybernetics of NAS of Belarus, 1999 (in Russian).

[Zakrevskij, 2000] A. Zakrevskij, V. Sklyarov. The Specification and Design of Parallel Logical Control Devices. In: Proceedings of PDPTA'2000, June, Las Vegas, USA, 2000.

[Zakrevskij, 2001] A. Zakrevskij, L. Zakrevski. Algorithms for logical control: their description, verification and hardware implementation. In: Proceedings of PDPTA'01, Las Vegas, 2001, Vol. 2.

Authors' Information

Dmitry Cheremisinov, Liudmila Cheremisinova – The United Institute of Informatics Problems of National Academy of Sciences of Belarus, Surganov str., 6, Minsk, 220012, Belarus, Tel.: (10-375-17) 284-20-76, e-mail: cher@newman.bas-net.by, cld@newman.bas-net.by