

УНИВЕРЗИТЕТ У НОВОМ САДУ  
ПРИРОДНО-МАТЕМАТИЧКИ ФАКУЛТЕТ

ПРИМЉЕНО: 20 АРГ 2003	
ОРГАНИЗ ЈЕД	БРОЈ
0603	331/1

Инв. бр. 3917



# Contents

<b>CONTENTS</b> .....	<b>I</b>
<b>FOREWORD</b> .....	<b>V</b>
<b>1 AGENTS</b> .....	<b>1</b>
<b>1.1 Introduction</b> .....	<b>2</b>
1.1.1 A Weak Notion of Agency .....	2
1.1.2 A Stronger Notion of Agency .....	3
<b>1.2 Classifications of Agents</b> .....	<b>4</b>
1.2.1 Type .....	4
1.2.2 Mobility .....	4
1.2.3 Size and Intelligence .....	5
1.2.3.1 Big-Sized Agents .....	5
1.2.3.2 Middle-Sized Agents .....	6
1.2.3.3 Micro-Agents .....	6
1.2.4 Learning .....	6
1.2.5 Architecture .....	7
1.2.6 Relationship with Other Agents .....	7
<b>1.3 Agent Applications</b> .....	<b>8</b>
1.3.1 Cooperative Problem Solving and Distributed Artificial Intelligence .....	9
1.3.2 Personal Digital Assistants and Intelligent Interfaces .....	9
1.3.3 Agents for Information Retrieval .....	10
1.3.4 Believable Agents .....	10
1.3.5 Electronic Commerce .....	10
1.3.6 Business Process Management .....	11
<b>1.4 Theories</b> .....	<b>11</b>
1.4.1 Logics for Agent Specification .....	11
1.4.2 Modal Operators Suitable for Agent Specification .....	12
1.4.3 Possible Worlds Semantics .....	13
1.4.4 Speech Act Theory .....	13
1.4.5 Logics for Agent Specification .....	14
1.4.5.1 Bell's Logic .....	14
1.4.5.2 Logic for Reasoning with Belief Contexts .....	14
1.4.6 Theory and Practice .....	15
<b>1.5 Architectures</b> .....	<b>15</b>
1.5.1 Deliberative Agent Architecture .....	15
1.5.1.1 One Deliberative Architecture .....	16
1.5.2 Reactive Agent Architectures .....	17
1.5.2.1 Subsumption Architecture .....	17
1.5.3 Hybrid Agent Architectures .....	18
1.5.4 Architectures of MAS .....	18
1.5.4.1 Communication .....	19
1.5.4.1.1 Agent Negotiation .....	19
1.5.4.2 The Blackboard Architecture .....	20
<b>1.6 Agent-Oriented Software Engineering</b> .....	<b>20</b>
1.6.1 The Gaia Methodology .....	20
1.6.2 MaSE – The Multiagent Systems Engineering Methodology .....	21
<b>1.7 Learning and Adaptation</b> .....	<b>21</b>
1.7.1 Why is Learning so Important? .....	21
1.7.2 A Definition and Classifications of Machine Learning .....	22
1.7.3 Q-Learning .....	23
1.7.4 Examples of Agent Learning .....	24
1.7.4.1 Agent's Knowledge Refinement using a Refinement Facilitator .....	24
1.7.4.2 Learning from Observation .....	24

1.7.4.3 Learning using Genetic Algorithms .....	25
1.7.4.4 Reinforcement Learning of Competitive Agents .....	25
1.7.4.5 Reinforcement Learning of Optimal Condition-Behavior Pairs .....	26
1.7.4.6 Case-Based Learning and the Contract Net Protocol .....	26
1.7.4.7 Q-Learning in Semi-Competitive Domain .....	26
1.7.4.8 Adaptation in Semi-Competitive Domain .....	27
1.7.4.9 RoboCup Soccer Tournaments .....	27
<b>2 OVERVIEW OF AGENT-ORIENTED PROGRAMMING LANGUAGES AND TOOLS .....</b>	<b>29</b>
2.1 AGENT0 .....	30
2.2 PLACA .....	32
2.3 Concurrent MetateM .....	33
2.4 AgentSpeak .....	35
2.5 JACK .....	36
2.6 ZEUS .....	37
2.7 HOMAGE .....	40
2.8 COOL .....	40
2.9 SICSLOG .....	42
2.10 KIDSIM .....	42
2.11 KQML and FIPA ACLs .....	43
2.12 Mobile Agent Tools and Languages .....	44
2.13 Java .....	45
<b>3 AJA FEATURES .....</b>	<b>46</b>
3.1 The Ideas behind AJA .....	47
3.1.1 New Agent-Programming Language is used Together with Java .....	47
3.1.2 The infrastructure for agent programming .....	47
3.1.3 Agents as a Vehicle for AI .....	48
3.1.4 Inter-Agent Communication is Compound .....	49
3.1.5 Inter-Agent Communication is Secure .....	49
3.1.6 Agent acts Reactively as well as Goal-Oriented .....	50
3.2 AJA Agent Architecture .....	50
3.2.1 Beliefs .....	51
3.2.1.1 Java Values .....	51
3.2.1.1.1 Java+ Constructs .....	52
3.2.1.2 Adaptable Parameters .....	52
3.2.1.2.1 Java+ Constructs .....	53
3.2.1.3 Dependant Values .....	53
3.2.1.3.1 Java+ Constructs .....	54
3.2.2 Actions .....	54
3.2.2.1 Java+ Constructs .....	55
3.2.3 Reflexes .....	55
3.2.3.1 Java+ Constructs .....	56
3.2.4 Negotiations .....	57
3.2.4.1 Java+ Constructs .....	62
3.2.5 WWW Negotiation .....	64
3.2.5.1 Java+ Constructs .....	66
3.2.6 Initialization .....	67
3.2.7 GUI .....	68
3.2.7.1 Java+ Constructs .....	68
<b>4 HADL AND JAVA+ .....</b>	<b>70</b>
4.1 HADL Grammar .....	71
4.1.1 Agent Program .....	71



4.1.2 Import .....	72
4.1.3 Declaration of Beliefs .....	73
4.1.4 Declaration of Actions .....	74
4.1.5 Declaration of Requesting Negotiations .....	75
4.1.6 Declaration of Responding Negotiations .....	75
4.1.7 Declaration of WWW Negotiation .....	76
4.1.8 Declaration of Reflexes .....	76
4.1.9 Initialization .....	77
4.1.10 All Grammar Production Rules .....	77
<b>4.2 Java+ .....</b>	<b>80</b>
4.2.1 Java+ Constants .....	80
4.2.2 Java+ Constructs for Beliefs .....	81
4.2.3 Java+ Constructs for Actions .....	82
4.2.4 Java+ Constructs for Negotiations .....	83
4.2.5 Java+ Construct for Reflexes .....	86
4.2.6 Java+ Constructs for WWW Negotiation .....	86
4.2.7 Java+ Constructs for GUI .....	88
4.2.8 Remaining Java+ Constructs .....	90
<b>5 AI CONSTRUCTS IN AJA .....</b>	<b>92</b>
<b>5.1 Adaptable Parameters .....</b>	<b>93</b>
5.1.1 Implementation .....	93
5.1.1.1 Initialization .....	94
5.1.1.2 Negative Reinforcement: \$AP_HIGHER .....	94
5.1.1.3 Negative Reinforcement: \$AP_LOWER .....	95
5.1.1.4 Negative Reinforcement: \$AP_BAD .....	96
<b>5.2 Dependant Values .....</b>	<b>97</b>
5.2.1 RPROP .....	99
5.2.2 Offline Training: \$DV_OFFLINE_TRAINING .....	102
5.2.3 Firing the Network: \$GET_BEL .....	102
5.2.4 Online Training: \$DV_SHOULD_BE .....	102
<b>6 A CASE STUDY – MULTI-AGENT SYSTEM IMPLEMENTED IN AJA .....</b>	<b>104</b>
<b>6.1 What does the MAS do? .....</b>	<b>105</b>
<b>6.2 Beliefs .....</b>	<b>105</b>
6.2.1 timeTable .....	106
6.2.2 eventAlertTime .....	106
6.2.3 eventAlertTimeToBackup .....	107
6.2.4 engToAlert .....	107
6.2.5 birthdaysTomorrowToAlert and birthdaysTodayToAlert .....	107
6.2.6 consultationDuration .....	107
6.2.7 consultationDurationToBackup .....	108
<b>6.3 Actions .....</b>	<b>108</b>
6.3.1 Timetable Manipulation .....	109
6.3.2 Alerting the User .....	109
6.3.3 Backup .....	109
6.3.4 GUI .....	110
<b>6.4 Reflexes .....</b>	<b>111</b>
6.4.1 Alerting Reflexes .....	111
6.4.2 Reflexes for Backup .....	111
6.4.3 Reflexes for Timetable Maintenance .....	112
<b>6.5 Negotiations .....</b>	<b>112</b>
6.5.1 EngagementInitReqNeg and EngagementInitResNeg .....	113
<b>6.6 WWW Negotiation .....</b>	<b>120</b>
<b>6.7 Initialization .....</b>	<b>122</b>

<b>7 RELATED WORK .....</b>	<b>124</b>
7.1 HOMAGF .....	125
7.2 JACK.....	125
7.3 LASSMachine and LASS.....	125
7.4 COOL .....	125
7.5 Subsumption Architecture .....	126
7.6 AI in Programming Language .....	126
<b>8 CONCLUSION AND FUTURE WORK.....</b>	<b>127</b>
8.1 The Goal of the Thesis .....	128
8.2 The Work Done .....	128
8.3 Future Work .....	129
<b>APPENDIX A - THE IMPLEMENTATION OF AJA TO JAVA TRANSLATOR.....</b>	<b>130</b>
<b>aja.framework .....</b>	<b>131</b>
Beliefs.....	132
Actions.....	133
Reflexes.....	134
Requesting and Responding Negotiations.....	135
Agent-to-Agent Communication .....	136
WebNegotiation .....	138
Built-in GUI .....	139
<b>aja.translator .....</b>	<b>140</b>
Translation .....	142
<b>APPENDIX B - AJA 1.0 INSTALLATION AND USAGE.....</b>	<b>143</b>
Installation and the Directory Structure.....	143
Preparing AJA for the First Use.....	144
<b>APPENDIX C - HOW TO TRANSLATE, COMPILE, RUN, AND USE THE EXAMPLE AGENTS.....</b>	<b>148</b>
Agents in the System and the Locations of their Files .....	148
Starting the Agents .....	148
Using Example Agents.....	155
Managing Colleagues.....	155
Managing Available Times .....	158
Managing Engagements.....	161
Access via Internet Browser .....	165
Other features.....	171
<b>REFERENCES .....</b>	<b>173</b>
<b>FIGURES.....</b>	<b>182</b>
<b>TABLES .....</b>	<b>184</b>
<b>INDEX.....</b>	<b>185</b>
<b>SAŽETAK.....</b>	<b>186</b>
<b>KRATKA BIOGRAFIJA .....</b>	<b>189</b>

## Foreword

Agents, agent-oriented programming (AOP), and multi-agent systems (MAS) are a relatively new field in computer science. It introduces new and unconventional concepts and ideas, which could sparkle a new software revolution in the near future.

Due to the immaturity of the field, there is still no generally accepted definition of the term 'agent'. There are many notions of this term. However, all agents have one property in common:

- *An agent acts on behalf of its user.*

Additional agent features *may* include the following properties:

- *An agent communicates with other agents in a multi-agent system.*
- *An agent acts autonomously.*
- *An agent is intelligent.*
- *An agent learns from experience.*
- *An agent acts pro-actively as well as reactively.*
- *An agent is modeled and/or programmed using human-like features, such as beliefs, intentions, goals, actions, etc.*
- *An agent is mobile.*

After more than a decade of intensive scientific work in the field, the next challenging step is evidently, to bring agents into mainstream programming. This is the crucial step. If project managers and programmers in the industry do not accept agent concepts, the agent field will have no future.

Agent programs are usually relatively complex ones. To design and implement agents from scratch, especially the ones in a multi-agent system, is a time-consuming process. Beside the implementation issues, e.g. network communication among agents, there are many design and higher-level problems to be solved. For example, it has to be chosen, how agent-to-agent messages should look like, how should they be sent: synchronously or asynchronously, etc. Additionally, an average mainstream programmer does not have any experience with agent-oriented programming. Almost all significant agent projects so far have been research projects at universities and other research institutions.

One way to facilitate agent transition from research to mainstream programming is to build and provide agent-oriented programming tools such as agent-oriented programming languages, agent-oriented integrated development environments, (IDEs) and agent-oriented design tools. Using such tools, the design and development of agent-oriented software systems would be faster and simpler.

Moreover, a mainstream programmer would not have to make important theoretical decisions alone. Using a tool, a programmer can successfully implement agent concepts that require a deep knowledge of agent theory, despite the fact that he/she does not possess this knowledge. In addition, a tool can significantly help in the implementation of system infrastructure (e.g. network communication, security, etc.).

The main goal of this thesis is the creation of one such tool for agent-oriented programming. The tool is called AJA, which is an acronym for **Adaptable Java Agents**. AJA consists of two programming languages:

- A higher-level language used for the description of main agent parts. This language is called HADL, which is an acronym for **H**igher **A**gent **D**efinition **L**anguage.
- A lower-level language is used for programming of the agent parts defined in HADL. This language is called Java+. It is actually Java enriched with constructs for accessing higher-level agent parts defined in HADL.

A translator from AJA to Java is implemented in the practical part of the thesis.

AJA agents have the following features:

- An agent communicates with other agents using a construct called negotiation. The messages sent can be encrypted or digitally signed in order to ensure the security of the system.
- An agent possesses adaptable parameters and neural nets that adapt themselves when the environment changes.
- An agent has reflexes, which are the reactive component of the agent architecture.
- An agent can perform its actions in parallel. Action executions are synchronized.
- An agent is accessible via the Internet, because it acts as a simple HTTP server. People can use this method to communicate with an agent.
- An agent has a Java Swing based graphical user interface. Its owner uses this interface to communicate with the agent.
- Because of the fact that Java+ language extends Java, it is possible to employ all useful Java features in the implementation of AJA agents (e.g. JDBC for database access).

The thesis also presents an original approach to integrating artificial intelligence techniques, such as neural nets, with a programming language. Having artificial intelligence components as a part of the programming language runtime environment makes their use straightforward. A programmer uses the language constructs that are implemented using artificial intelligence without the need to understand their background and theory.

The thesis is organized as follows. There are eight chapters and three appendixes.

In the first chapter, an overview of agents and multi-agent systems is given. The overview starts with agent definitions. Next, agents and multi-agent systems are analyzed and described from various perspectives. This chapter can be used as an introduction into the state-of-the-art in the field.

The second chapter surveys existing agent-oriented programming languages and tools.

The third chapter introduces AJA and describes the architecture of AJA agents. All components of AJA agents are explained.

The syntax and semantics of AJA languages HADL and Java+ are described in the fourth chapter.

The fifth chapter presents AI constructs in AJA in more detail.

To demonstrate and test the created tool, a case-study multi-agent system has been implemented in AJA. There are four personal digital assistant agents in the system. The sixth chapter describes the example agents and evaluates the tool.

In the seventh chapter related work and tools are analyzed and compared to AJA.

The last chapter concludes the thesis. It is pointed out that the development of the case-study multi-agent system has shown several important advantages of using AJA for the implementation of multi-agent systems.

The thesis has three appendixes. The first one describes implementation details of the AJA to Java translator. The second appendix is a guide for the installation and usage of the implemented AJA to Java translator. Finally, the third appendix describes step by step how to translate, compile, run, and use the example agents.

For an introduction to this new and extremely interesting field, for valuable help, suggestions, continual support and patience I am especially indebted to my supervisor Mirjana Ivanović.

Thanks also go to Zoran Budimac. His stimulating suggestions contributed to the improvement of this thesis.

I would also like to express my gratitude to Dušan Tošić and Živko Tošić, for reading my thesis and giving me useful comments on how to improve it.

I would like to thank to Prof. Dr. sc. Hans-Dieter Burkhard for inviting me to be a member of his team during my one semester stay at Humboldt University in Berlin in 1998/99. The valuable knowledge and experience I got while working together with professor Burkhard at the Artificial Intelligence Department have been very significant for this thesis.

At last but not least, I am also thankful to Miloš Radovanović for reading an early version of the thesis and correcting grammatical errors.

Novi Sad, 12 June, 2003.

*Mihal Badjonski*

# 1 Agents

Introduction .....	2
Classifications of Agents .....	4
Agent Applications .....	8
Theories .....	11
Architectures .....	15
Agent-Oriented Software Engineering .....	20
Learning and Adaptation .....	21



This chapter surveys the field of agents and multi-agent systems (MASs). The first section introduces agents and lists several agent definitions. The following section presents some classifications of agents. The third section describes agent applications. The theoretical 'tools' for agent specification and modeling are described in the fourth section. Agent architectures are reviewed in the following, fifth section. The sixth section describes agent-oriented software engineering. The final, seventh section of the chapter is concerned with various approaches to agent learning.

## 1.1 Introduction

The word agent has many meanings. Even if we restrict ourselves to computer science, the word agent is not uniquely defined.

Intuitively, an agent is an entity that acts on someone's behalf. An agent is autonomous in its decision-making. It possesses some intelligence and knowledge about the problem domain it is used for. An agent can behave reactively in its environment as well as proactively, in order to satisfy its goals.

An important feature of agents is their ability to communicate. An agent communicates with its user and/or other agents.

A system with two or more agents that cooperate or compete with each other in order to solve some problem or perform some task(s) is called a *multi-agent system* (MAS).

There are two notions of computer agents: a *weak notion* and a *strong notion*.

### 1.1.1 A Weak Notion of Agency

The weak notion of agent is, among other areas, often used within a new software engineering approach called agent-based software engineering.

Wooldridge and Jennings [105], [106] define an agent as

*"... a hardware or (more usually) 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 state;*
- social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;*
- reactivity: agents perceive their environment (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), 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."*



Michael Coen [48] defines agents as:

*"... programs that engage in dialogs and negotiate and coordinate transfer of information."*

The definition above puts very small restrictions on a program that is considered to be an agent.

In IBM [45], intelligent agents are defined as:

*"...software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires."*

In [92], two features of agents are stressed: persistence and special purpose.

*"Let us define an agent as a persistent software entity dedicated to a specific purpose. 'Persistent' distinguishes agents from subroutines; agents have their own ideas about how to accomplish tasks, their own agendas. 'Special purpose' distinguishes them from entire multifunction applications; agents are typically much smaller."*

The Software Agents Group at MIT [46] compares software agents to conventional software and emphasizes the following differences: *"Software agents differ from conventional software in that they are long-lived, semi-autonomous, proactive, and adaptive."*

A collection of various agent definitions (including those listed above) can be found in [37]. All of these definitions are based on the weak notion of agency. Authors of [37] gave their own definition of an autonomous agent:

*"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."*

### **1.1.2 A Stronger Notion of Agency**

In order to make the understanding, modeling, programming, controlling, analyzing, and debugging of MASs easier, these systems are often specified using human mental categories. An agent can be described with the categories such as: *beliefs, plans, goals, intentions, desires, commitments*, etc. Shoham [91] claims that the use of mental categories in agent specification is justified only if the following three conditions are satisfied:

- mental categories are precisely defined using some formal theory,
- an agent has to obey that theory,
- every mental category used in an agent specification has to be of some benefit to the design, execution, debugging or modeling of the MAS.

The stronger notion of agency is mostly used in the field of artificial intelligence (AI). A definition of an agent in the sense of this notion is [106]:

*"An agent is a computer system that, in addition to having the properties identified in the definition of weak agent, is either conceptualized or implemented using concepts that are more usually applied to humans (knowledge, obligations, beliefs, desires, intentions, emotions, human-like visual representation, etc.)."*

## 1.2 Classifications of Agents

There are many features that might be used for the classification of agents. The ones used in this chapter are: the type of agents (hardware or software), mobility, size, intelligence, ability to learn, architecture, relation to other agents in the MAS, and types of applications where they are used.

### 1.2.1 Type

There are two types of agents: *hardware agents* and *software agents*.

Hardware agents are robots. They have sensors for observing their environment and effectors with which they perform physical actions.

Software agents are programs that satisfy one or more definitions given above. They became interesting with the spreading of computer networks. The ideas upon which a new programming paradigm, *agent-oriented programming*, is based are:

- distribution of a software system onto several autonomous components (agents) connected with a net, and
- problem solving using their communication.

### 1.2.2 Mobility

Agents can be *static* or *mobile*. Static agents are permanently located at some place, while mobile agents can change their location.

It is not so interesting to characterize hardware agents as mobile or static. A robot's ability to move is just one of its possible actions. On the other hand, software agents that can traverse computer networks set up a new computational paradigm that is quite different from the one introduced by the use of static software agents.

When a static agent wants some action to be executed at some remote site, it will send a message to an agent at that location with the request for that action. A mobile agent acts differently. In the situation described above, a mobile agent would transmit itself to the remote site and invoke the action execution.

At first glance, it is not quite clear, what are the benefits of the use of mobile software agents. It seems that everything that can be achieved with mobile agents can also be achieved with communicating static agents. In [42] an answer to this question is given. There are many benefits we can get from mobile agents [42]:

- they can provide better support for mobile devices (a mobile agent can be sent from a temporarily connected Palmtop computer or a PDA to a host to execute a task there.),

- they facilitate semantic information retrieval (they can search for particular information more efficiently than static agents can),
- real-time interaction with a server,
- better support for a heterogeneous environment,
- agent-based queries and transactions are more robust and flexible (because they are executed locally at the server side, without sensitive remote communication),
- they avoid process state preservation (because agents are persistent entities),
- they enable electronic commerce with electronic money,
- applications scale better,
- a user can personalize a server's behavior (the user's own interface to the server), and
- they enable intelligent mail handling.

As pointed out in [42], almost each one of the above features can be obtained using other techniques, without mobile agents. However, if we wanted to get all of these benefits without using mobile agent, it would require a large amount of work and would be practically impossible.

Many discussions about the usefulness of mobile agents can be found in [2]. As pointed out in one message in [2], it is true that mobile agents are not necessary, but it is also true that we do not need high-level programming languages at all, because every program can be written in assembler.

### 1.2.3 Size and Intelligence

Agents can be of various sizes and can possess various amounts of intelligence. The intelligence of a software agent is proportional to its size. The intelligence of an agent is usually achieved by encoding some knowledge. The bigger the knowledge base, the greater the agent's intelligence. The classification of software agents regarding their intelligence is the same as the one that is based on their size. Software agents can be coarsely grouped into three categories: *big-sized agents*, *middle-sized agents* and *micro-agents*. It is difficult to make clear boundaries among these categories, because there is no clear distinction between the adjectives big, middle-size and small.

#### 1.2.3.1 Big-Sized Agents

A big-sized agent occupies and controls one or more computers. It possesses enough competence to be useful even when it acts alone, without other agents in a MAS. Cooperating with other agents in a MAS gives some additional abilities to a big-sized agent.

A big-sized agent can be as big and as intelligent as an expert system is. An example

of such agents is given in [50], where a system for distributed medical care is described. A pharmacy agent is intelligent enough to prescribe medicine for a patient. But if the pharmacy agent cooperates with other agents (expert systems), a whole medical care system can be covered with a MAS.

Agents for plane ticket reservation are also big-sized agents. Each such agent is located at one airport. A ticket reservation for a flight from an agent's airport does not require communication with agents from other airports. But in order to make a reservation for flights from other airports, an agent needs to cooperate with these agents.

### 1.2.3.2 Middle-Sized Agents

A middle-sized agent is the one that is not useful alone without other agents in a MAS or without additional software. However, it is able to perform some non-trivial task(s). A user-interface agent that acts alone without other agents (it is not part of any MAS) and performs some simple actions can be classified as a middle-sized agent.

Examples of middle-sized agents can be found in [6], [9], [7], and [10].

Mobile agents are also middle-sized agents. They travel through a computer network and it is convenient to make them as small as possible. There is not enough space for a massive agent's knowledge base. Some applications (for example [65]) only extend a conventional programming language (Java, C, C++) with the ability of a program to stop its execution at one site, transfer itself to some other site and continue its execution at another machine.

Agents in the KIDSIM system are also middle-sized. KIDSIM is described in 2.10.

### 1.2.3.3 Micro-Agents

The term agent is also used in [75]. These agents (also called the Society of Mind agents) do not possess any intelligence. Minsky uses them for the explanation of how human intelligence is achieved. The intelligence emerges as a global effect of the overall activity of many simple and unintelligent agents. Unfortunately, there is still no theory that explains how this system works and which can be used for the development of such systems.

## 1.2.4 Learning

The ability of an agent to learn and to adapt to changes in its environment is a desired property. It makes an agent robust when unexpected changes occur. Regarding learning and adaptation, there are two types of agents: agents able to learn and agents unable to learn. Agents use learning capabilities:

- for adapting to non-predicted changes in the environment that is being constantly changed,
- in the training process that takes place prior to productive agent usage.

The training process is often used as the last phase in the development of a personal digital assistant (PDA<sup>1</sup>). In this case, knowledge can also be written directly without learning. However learning is a much better choice (see [71]), because most users find it very difficult to program their PDAs.

A PDA is an agent that acts like a secretary in an office. It learns by observing the actions of its user. Using agents as PDAs is described in subsection 1.3.2.

Agents unable to learn have to be programmed in such a way that nothing can surprise them. In some domains, this is not so hard to achieve. An example of such an agent is Library helper in [95], which is programmed in the agent-oriented programming language PLACA. Library helper has its rules that encompass every situation when it should react.

### 1.2.5 Architecture

The classification of agents regarding their architecture divides agents into three groups [106]:

- agents with a *deliberative architecture*,
- agents with a *reactive architecture*, and
- agents with a *hybrid architecture*.

As mentioned in the first of the given agent definitions, an agent should possess reactivity and pro-activity. These two features determine the above three classes of agent architectures. If an agent is more interesting in pursuing its own goals rather than immediately responding to events in its environment, its architecture is a deliberative one. The agents whose emphasis is on reactivity are built using reactive architectures. Hybrid architectures are those which equally implement deliberative (or pro-active) and reactive properties of agents.

Architectures of agents are described in more detail in the fifth section of this chapter.

### 1.2.6 Relationship with Other Agents

This aspect is more related to MASs than to particular agents. There are four types of agents:

- *Cooperative agents*
- *Competitive agents*
- *Semi-competitive agents*
- *Single agents*

---

<sup>1</sup> The term PDA has two meanings. In this context, the word is used to denote a program, i.e. a software agent that acts as a personal assistant of its user. Nowadays however the term PDA is also used to denote a type of popular small electronic devices.

Cooperative MASs are the ones where agents cooperate in their work. Every agent helps another agent if help is required and possible.

Competitive MASs are those where agents compete between themselves. Winner-agents are allowed to perform their actions and to influence the behavior of a system.

Semi-competitive MASs are somewhere between the above given cases. Agents in some semi-competitive MAS can be selfish, but they are willing to cooperate because their gain will be greater if they cooperate than it would be without cooperation. An example of a semi-competitive domain is Iterated Prisoner Dilemma (see 1.7.4.7).

A single agent is an agent that does not belong to any MAS and therefore does not communicate with other agents.

### 1.3 Agent Applications

An agent can be classified regarding the domain of application it is used in. In general, the MAS approach and agent-oriented programming are suitable for open system programming [22]. Open systems have the following features [22]:

- continuous availability,
- extensibility,
- decentralized control,
- asynchronicity,
- inconsistent information,
- arms-length relationships.

Examples of open systems, given in [22], are: traffic control, transport companies, plane ticket reservation, offices, robots, virtual reality, artificial life, computer games, etc. There are many applications of agents and some of them are described in the following subsections.

However, it has to be noted, that most applications that currently use agents could be built using non-agent techniques. As pointed out in [63], "*... the mere fact that a particular problem domain has distributed data sources or involves legacy systems does not necessarily imply that an agent-based solution is the most appropriate one – or even that it is feasible.*" Agent-based solutions could lead to a number of problems, common to all agent-based applications, e.g. (from [63]):

- *No overall system controller.* An agent-based solution may not be appropriate for domains in which global constraints have to be maintained, in domains where a real-time response must be guaranteed, or in domains in which deadlocks or livelocks must be avoided.
- *No global perspective.* An agent's actions are, by definition, determined by that agent's local state. However, since in almost any realistic agent



system, complete global knowledge is not a possibility, this may mean that agents make globally sub-optimal decisions. The issue of reconciling decision making based on local knowledge with the desire to achieve globally optimal performance is a basic issue in multi-agent systems research.

- *Trust and delegation.* For individuals to be comfortable with the idea of delegating tasks to agents, they must first trust them. Both individuals and organizations will thus need to become more accustomed and confident with the notion of autonomous software components, if they are to become widely used. Users have to gain confidence in the agents that work on their behalf, and this process can take time. During this period, the agent must strike a balance between continually seeking guidance (and needlessly distracting the user) and never seeking guidance (and exceeding its authority). Put crudely, an agent must know its limitations.

### **1.3.1 Cooperative Problem Solving and Distributed Artificial Intelligence**

The MAS approach is suitable for many problems within distributed artificial intelligence. Distributed artificial intelligence (DAI) is a part of AI relating to all aspects of building systems that possess more than one entity performing intelligent actions.

A manufacturing plant with robots can be seen as a MAS. Each robot is an agent. If there is a central computer that controls the overall manufacture, it can also be seen as an agent. These agents communicate in their language. Each agent is autonomous and has its goals. The actions performed by an agent are determined by the internal (mental) state of the agent and the message(s) it has received. A construction of such a system is facilitated when the MAS approach is used. Standard MAS techniques and some agent-oriented programming language for agent programming are a better choice than ad hoc solutions.

An example of a MAS used in DAI is a system for distributed medical care described in [50].

### **1.3.2 Personal Digital Assistants and Intelligent Interfaces**

With the expansion of the Internet, another type of agent is becoming more and more popular. Personal digital assistant (PDA) agent is a computer program aimed at performing simple and time-consuming tasks on behalf of its user. A PDA may handle e-mail of its user or it may monitor or find interesting newsgroups or web sites on the Internet and filter 'interesting' information [71]. A PDA may maintain the appointment schedule of its user and independently make or cancel appointments [71], [73]. A PDA may communicate with other PDAs and perform some tasks that would otherwise have to be performed by the user. As proposed in [71], a personal assistant should be able to improve its knowledge in the following ways:

- by observing user actions,

- by user programming (the user can present examples of good actions),
- by receiving feedback from user,
- by sharing experiences with other agents.

A PDA acts like an intelligent interface to some application or to the Internet. An Intelligent interface adapts itself to the preferences of the user and makes the user work more conveniently and efficiently. One such interface to the Internet that uses one agent (or softbot – software robot) is described in [29]. Another interesting approach is given in [111]. Here an agent is located in a mobile device. The agent uses the information about its geographical coordinates in order to limit the search results of an Internet search engine.

An adaptive user interface, which consists of many competitive agents, is described in [66]. Various settings for the interface are represented with various agents. The agents with the best user feedback are active and determine the interface.

Examples of agents embedded in interfaces are the Microsoft Excel Chart Wizard for graphic creation, and Microsoft Wizard for Windows 2000 Internet settings.

### **1.3.3 Agents for Information Retrieval**

Agents can be used for information retrieval. Suppose that every electronic source of information has an agent attached. When a user asks his/her agent for particular information, the agent will start a searching process. It will use a computer network to communicate with agents associated with ftp sites, databases, etc. In [106], the following example is given:

*"A typical scenario is that of a user who has heard about somebody at Stanford who has proposed something called agent-oriented programming. The agent is asked to investigate, and, after a careful search of various FTP sites, returns with an appropriate technical report, as well as the name and contact details of the researcher involved."*

### **1.3.4 Believable Agents**

Believable agents are the agents used in the entertainment industry. Computer games and animated films exploit believable agents to achieve human-like or animal-like features of their characters.

As pointed out in [14], emotions play a more important role than intelligence in believable agents creation. A believable agent should be emotionally sensitive and its emotions must be expressed faithfully.

A reactive architecture based on behaviors and emotions is described in [27].

### **1.3.5 Electronic Commerce**

In [62], beside personal assistants, two additional possible software agent applications are mentioned.



In the first one, agents are proposed for an electronic marketplace. An agent could be sent to an electronic shop with some amount of electronic money and orders from its user. It would be welcomed by a shop-agent. If everything was in order, they would trade. An agent could also go to (or call) a restaurant and negotiate with a restaurant-agent about reservation and meals.

An agent web-based marketplace has been developed at MIT Media Lab [25]. People who sell goods create selling agents, while people interested in buying goods create buying agents. After the creation of an agent, its selling or buying strategy is specified and the agent is ready for negotiation with agents of the opposite type, on behalf of its owner.

### **1.3.6 Business Process Management**

According to [62], agents could be used in business process management as well. When two companies are going to collaborate, the procedure that usually has to be performed is rather complex. A lot of paperwork has to be done between various departments in both companies (legal department, technical department, market department, etc.). Software agents might do a great part of this work.

Papers [60] and [61] describe an agent-based business process management system developed for British Telecom. Instead of using a fixed inflexible workflow, negotiating agents are used.

An approach to business process management with mobile agents is described in [21].

## **1.4 Theories**

Every MAS has some purpose. Agents should behave as their creators wanted to and they should fulfil some predefined tasks. In order to prove that agent behavior will be the desired one and in order to give some specification of the agents, many MAS theories have been developed.

The stronger notion of agency (see 1.1.2) demands for the use of human attributes in agent specification. Agents are often specified in terms of their beliefs, desires, intentions, plans, commitments, etc. Mental categories are chosen because they are the most natural and suitable way for agent description. Not only programmers use these terms. An agent can also model another agent with the mental categories. For example, agent A may believe that agent B intends to open the door at time t.

As mentioned in [91], some theory has to be used for a formal definition of the mental categories used in agent specification.

### **1.4.1 Logics for Agent Specification**

If we want to specify a MAS, the classical predicate or propositional logic does not suffice. We can use well-formed formulas of the predicate logic in the representation of agent beliefs. However, there is a need for a distinction among formulas describing beliefs of various agents. Furthermore, within one agent there will be a

group of formulas describing agent beliefs, a group for agent intentions, a group for its desires, etc. The predicate logic does not enable such grouping of formulas. It does not enable the representation of nested beliefs as well.

The situation when agent A believes that agent B believes that agent C wishes the agent A to plan to turn the machine on, cannot be represented with predicate logic.

The problem with the predicate logic is in the definition of the atomic formula. The arguments of any predicate are terms. A term can be a constant, a variable, or the value of a function whose arguments are terms. An argument of a predicate cannot be another predicate. The following atomic formula is therefore not valid in the predicate logic:

**believe(a, love(b, a))**

In the above example, **believe** is not a predicate. This formula is valid in another type of logic – modal logic.

Modal logic extends the predicate logic with modal operators. **believe** in the above example can be such an operator. Every mental category used in agent description can be represented with an appropriate modal operator.

Every MAS exists in some period of time. Beliefs, intentions, and other mental categories of agents change as time goes on. If agents need to synchronize their actions (in most cases they do), they will use their clocks or a common clock. Since time is an important factor in MAS specification, modal logic used for MAS specification is always a temporal one.

#### **1.4.2 Modal Operators Suitable for Agent Specification**

In [91], the following modal operators are proposed:

- $B_a^t e$  - agent a believes in formula e at time t,
- $OBL_{a,b}^t e$  - agent a is obliged to agent b for e at time t,
- $DEC_a^t e$  - at time t, agent a has a decision expressed with formula e,
- $DEC_a^t e$  is defined as  $OBL_{a,a}^t e$ ,
- $CAN_a^t e$  - agent a can do e at time t,
- $ABLE_a e$  is defined as  $CAN_a^{time(e)} e$  – agent a is able to do e iff it will be able to do e at the time when event e will occur.

Other researchers [94], [103] have used more or less modified sets of modal operators.

Usage of modal operators introduces some requirements that have to be satisfied. Shoham [91] listed the following ones:

- *Internal consistency*

- for every time  $t$  and for every agent  $a$ , the set  $\{e \mid B_a^t e\}$  has to be consistent. This means that an agent cannot believe in some fact and in the negation of that fact at the same time.
- for every time  $t$  and for every agent  $a$ , the set  $\{e \mid OBL_{a,b}^t e \text{ for some } b\}$  has to be consistent. An agent cannot be obliged to do something and not to do the same thing at the same time.
- *good faith*
  - for every time  $t$ , for all agents  $a$  and  $b$  and for every formula  $e$ ,
$$OBL_{a,b}^t e \Rightarrow B_a^t ((ABLE_a e) \wedge e)$$

An agent has to believe that it is able to perform all its obligations.
- *introspection*
  - for every time  $t$ , for all agents  $a$  and  $b$  and for every formula  $e$ ,
$$OBL_{a,b}^t e \Leftrightarrow B_a^t OBL_{a,b}^t e$$

If an agent is obliged to do something, then it believes that it is obliged to do that thing and vice versa.
  - for every time  $t$ , for all agents  $a$  and  $b$  and for every formula  $e$ ,
$$\neg OBL_{a,b}^t e \Leftrightarrow B_a^t \neg OBL_{a,b}^t e$$

### 1.4.3 Possible Worlds Semantics

Beliefs of an agent can be defined using possible worlds semantics. In almost every situation, an agent has only partial knowledge about the world it inhabits. It does not know all the facts in its current world. Therefore there may be several worlds possible (they do not conflict with agent's partial knowledge about the current world).

Beliefs of an agent can be defined as those facts that exist in every possible world. Using possible world semantics, a logic can be made that defines agent beliefs.

A drawback of this logic is that it requires from an agent to be logically omniscient. This means that every agent must believe in every logical consequence of its beliefs. A logically omniscient agent must be aware of every tautology, among other formulas. Such an agent cannot be created within the finite size of computer memory.

### 1.4.4 Speech Act Theory

Types of messages that agents send to each other are often derived from the speech act theory [3].

According to this theory, messages that are sent are actions in the same way that physical actions are. If an agent sends a message to another agent, then it wants to

change the state of affairs in the world. It wants to change the mental state of the receiver agent and thus make it act in the desired way.

Speech acts may fail as well as physical actions. The receiver of the message does not have to change the behavior as the sender expected.

The most widely used speech acts are messages of the following types:

- **inform,**
- **request,**
- **ask,**
- **refrain,**
- **decline, etc.**

### **1.4.5 Logics for Agent Specification**

As already mentioned, relatively many logics for agent specification have been made. Two following subsections briefly describe two of them.

#### **1.4.5.1 Bell's Logic**

Beliefs, obligations, and capabilities must remain unchanged as time passes, unless the agent has changed them. If agent **Driver** believes that its car does not move at time  $t$ , then it should believe the same at time  $t+1$ , unless it start to drive at time  $t+1$ . The problem of beliefs persistence, obligations persistence, etc. is similar but even more difficult than the frame problem in situation representation in traditional AI.

John Bell analyzed this problem and made an appropriate logic for its surpassing [15]. He developed a many sorted first order modal language with explicit reference to time points and intervals.

#### **1.4.5.2 Logic for Reasoning with Belief Contexts**

Sometimes, it is expected from agents to reason about other agents' mental states. In [26] and [16], A. Cimatti, L. Serafini and M. Benerecetti gave a logic and architecture suitable for this requirement. If an agent-reasoning system is built upon this theory, then any agent will be able to reason about the reasoning of another agent. It will also be able to reason about another agent reasoning about the reasoning of the third agent and so on.

The logic is quite simple and extends the classical logic with contexts of beliefs and bridge rules. Bridge rules enable a reasoner to change the context of its reasoning. This logic also demands for logical omniscience.

## 1.4.6 Theory and Practice

Shoham [91] demands from a MAS to obey its theory. Unless the MAS is a very simple one, this request is hardly achievable. Usually, the theory that faithfully describes a MAS is very huge and awkward.

In order to avoid complications with theory, Rebecca Thomas made a compromise in her Ph.D. thesis [94]. She made the agent-oriented programming language PLACA (see 2.2), which is a descendant of AGENT0 [91] (see 2.1). The theory she made for the description of PLACA features has some limitations. An agent programmed in PLACA does not behave exactly by the logic in some situations. However, the logic is very convenient and comprehensive. There are some small parts in the logic that the language does not obey. R. Thomas has pointed to these parts in the language specification. She chose this approach with a good reason. The alternative approach with the logic faithfully describing the language PLACA would involve a huge and awkward logic.

The obvious requirement that agents programmed in PLACA cannot always achieve is logical omniscience. The logic she has made, as well as any other logic based on possible worlds semantics will always require from agents to be logically omniscient.

## 1.5 Architectures

The architecture of an agent and the architecture of a MAS should enable the implementation of the concepts proposed by the MAS theory.

The first three subsections of this section concern with various architecture types of a single agent. The last subsection discusses architectures of MASs.

### 1.5.1 Deliberative Agent Architecture

Deliberative architectures are inherited from traditional AI. Deliberative agents have their own world model represented with symbols. They have their goals that they try to achieve. They use plans. If they plan, they use their internal, symbolic model of the world. Planning is based on a search for the appropriate sequence of actions. Once the outside world is represented with symbols, it is ignored. Only the internal model is examined.

Most of the work with agents today is based on deliberative agent architecture. Examples of such architectures are IRMA [17] and GRATE\* [55].

Traditional AI has failed to fulfill the expectations. Two main reasons for this failure are (from [106]):

- The *transduction problem*: that of translating the real world into an accurate, adequate symbolic description, in time for that description to be useful.
- The *representation/reasoning problem*: that of how to symbolically represent information about complex real-world entities and processes, and



how to get agents to reason with this information in time for the results to be useful.

Deliberative architectures also possess the above drawbacks and that is why some alternative approaches are proposed.

### 1.5.1.1 One Deliberative Architecture

An agent architecture and an agent specification language are given in [67]. That agent architecture is given in Figure 1. An agent contains a knowledge base (KB) with facts describing the environment. An agent is situated in the real world. The KB Manager maintains the knowledge base in order for the KB to represent the current situation in the real world. The KB Manager also maintains the consistency of the KB. There are several types of facts in the KB. **Fluents** are facts with the shortest duration. Some facts exist in the KB by **default**. Some facts can be changed and some cannot. Using the information about fact types, the KB Manager determines which facts will remain in the KB when an inconsistency is detected, and which ones will be deleted. Knowledge in the KB is represented with extended logic programming rules.

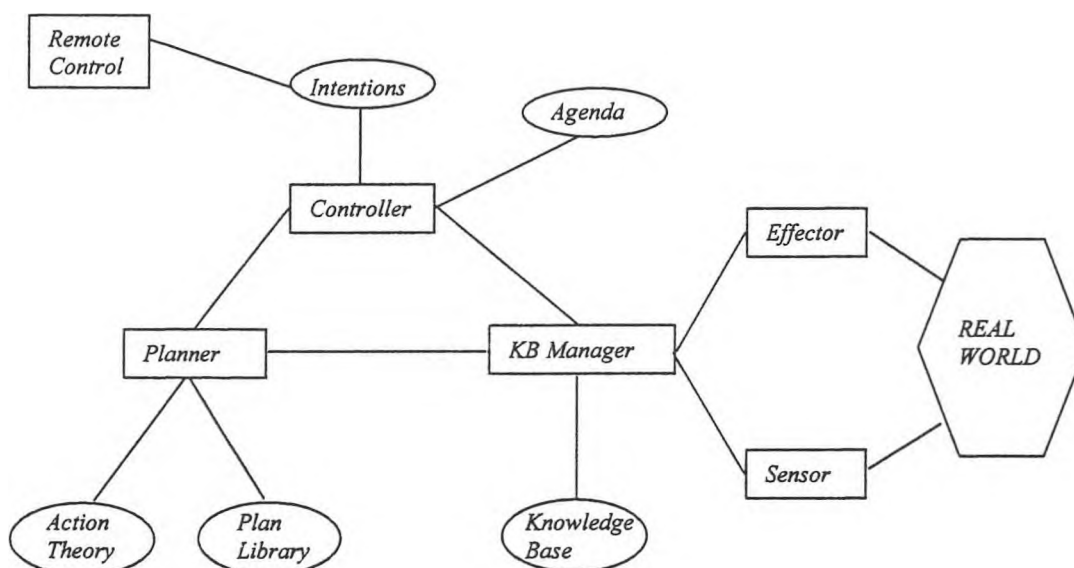


Figure 1 The architecture of a knowledge-based situated agent.

Slika 1 Arhitektura situiranog agenta zasnovanog na znanju.

An agent has intentions. Setting agent intentions, one can determine agent behavior. An intention can be fulfilled if there is a plan for that intention. Some plans may be written in advance during agent programming. These plans are placed in the plan library and they enable an agent to response reactively in certain situations. When there is no plan in the library associated with some intention, a planner will generate a plan. In plan generation, the planner uses the information from the **Action Theory**. A plan consists of a sequence of actions.

The **Action Theory** is a part of agent architecture, where every action available to the agent is described. An action is described with its name, parameters, necessary



conditions that must be satisfied before the action takes place, duration of the action and consequences of the action.

## 1.5.2 Reactive Agent Architectures

Reactive agents give a fast response to every change in their environment. They do not waste time with long reasoning processes. Nevertheless, they may be able to perform complex tasks.

An approach that exploits situated automata for a reactive agent architecture is described in [64].

Pattie Maes [69] has developed a reactive agent architecture that is composed of modules organized into a network. Each module has some competence. A module can be active or inactive. Its activity depends on its activation value. The activation value is determined with the current situation, pre-conditions for module activation and the activation value of the modules that support the activation of the observed module.

The subsumption architecture, which is based on 'behaviors', is the best-known reactive architecture.

### 1.5.2.1 Subsumption Architecture

Rodney Brooks from MIT advocates a new approach to AI. In his work (including [19] and [20]), he criticizes the traditional AI, which is based on the physical-symbol system hypothesis [82] and on the search as the most important mechanism. He has built many robots using his approach, which is based on four principles [19]:

- *Situatedness* - The robots are situated in the world – they do not deal with abstract descriptions, but with the here and now of the world, directly influencing the behavior of the system.
- *Embodiment* - The robots have bodies and experience the world directly – their actions are part of a dynamic system with the world, and they have immediate feedback through their own sensors.
- *Intelligence* - They are observed to be intelligent – but the source of intelligence is not limited to just the computational engine. It also comes from the situation in the world, the signal transformations within sensors, and the physical coupling of the robot with the world.
- *Emergence* - The intelligence of the system emerges from the system's interactions with the world and, sometimes, from indirect interactions between its components – it is sometimes hard to point to one event or place within the system, and say that is why some external action was manifested.

Brooks' robots are capable of performing complex tasks, with relatively simple programming, based on the subsumption architecture. The subsumption architecture [18] is composed of several levels of behaviors. Behaviors on lower level have higher priority than behaviors on higher levels. Lower-level behavior can be, for

instance, used for avoiding obstacles, while higher-level behavior can be the one that is used for going from one room to another.

These systems are highly reactive. As a result of an external input, the appropriate behavior (or behaviors) is (are) selected and used.

Despite the fact that Rodney Brooks' work is concerned with robots (hardware agents), Pattie Maes has shown that same ideas can also be exploited in the design of software agents [70].

### **1.5.3 Hybrid Agent Architectures**

Hybrid architectures possess both deliberative and reactive parts. They exploit good features of both deliberative and reactive architectures.

An example of a hybrid architecture is PRS [39]. PRS (Procedural Reasoning System) is a belief-desire-intention architecture. It contains knowledge areas (KAs) that can be used as plans for intentions (deliberative feature) but which can also be activated when some event occurs (reactive feature).

### **1.5.4 Architectures of MAS**

In the three previous sections, possible agent architectures were described. This section is concerned with the architectures of MASs. There is no recipe that can tell us how to put agents together and make a MAS.

As already mentioned in 1.2.6, three types of MAS organization can be distinguished:

- Cooperative MASs
- Competitive MASs
- Semi-competitive MASs

Cooperative MASs are the ones where agents cooperate in their work. Every agent will help another agent if help is required and possible. Most of the MASs developed are of this type.

Competitive MASs are those where agents compete between themselves so as to obtain control over some object. Winner-agents are allowed to perform their actions and to influence the system's behavior. An example of a competitive MAS is described in subsection 1.7.4.4.

Semi-competitive MASs are somewhere between the above given cases. Agents in some semi-competitive MAS can be selfish, but they are willing to cooperate because their gain will be greater if they cooperate than it would be without cooperation. An example of semi-competitive domain is Iterated Prisoner Dilemma (subsection 1.7.4.7).

A common feature to every MAS is communication. In every MAS, agents communicate.



An architecture that may be suitable for some MASs creation is the Blackboard Architecture. The Blackboard Architecture can be used for all three types of MAS domains (cooperative, competitive, and semi-competitive).

#### 1.5.4.1 Communication

The architecture used has to enable communication.

Communication can be realized by *broadcasting*, when every message is broadcast to all agents. The message has to contain the address of the receiver. Every agent checks the address and uses the message if it recognizes itself as the receiver. This type of message passing is an expensive one. It is hard to implement this way of communication. However, this type of communication is often used for the Contract Net Protocol.

In the Contract Net Protocol, an agent that needs to fulfill some task broadcasts the task announcement to every agent in the MAS. Agents that received the task announcement send a bid to the agent that announced the job. The task-announcer analyzes every bid it received and selects the most appropriate agent for the task execution. It then sends a response message to every bid-sender. Only one agent will be allowed to execute the job. Other agents will be denied.

Another type of communication is *point-to-point* communication. In this type of communication, exactly one agent receives a message. If there is no direct connection between two agents and they have to communicate, then the message will travel via various agents until it reaches the destination agent. Point-to-point communication is less useful than broadcasting, but it has lower communication costs.

A hybrid type of communication can be made as a combination of the two above types of communication. One hybrid communication proposal is given in [35].

##### 1.5.4.1.1 Agent Negotiation

Automated agent negotiation is proposed as a key form of interaction in systems composed of multiple autonomous agents [30]. There are a lot of papers describing automated agent negotiation and agent negotiation strategies, e.g. [30], [32], [31], [59], [68], [93], [51], etc. Despite the fact that formal models describing automated negotiation have been developed, still there are no convincing real world examples of multi-agent systems with autonomous agent negotiation. The two main problems in the implementation of automated agent negotiation are:

- All possible options an agent has in a negotiation have to be evaluated. Automated negotiation formal models presume that the usability of a negotiation option can be numerically measured. This is, however, usually not the case in practice. There are too many factors determining the long-term reward of a particular decision in a negotiation.
- The most autonomous agent negotiation examples are from the e-commerce domain. When an agent negotiates, usually it buys or sells goods for its owner. Its job is very responsible. If it makes a mistake, it could cause financial damage to its owner. Due to this fact and considering

the state-of-the-art agents, it is hard to believe, that someone will trust his/her agent enough to leave it alone to buy and sell goods.

#### 1.5.4.2 The Blackboard Architecture

Some MASs are organized with the blackboard architecture. The blackboard architecture had been used in expert systems (ES) development [97], before it was used in the MAS domain. The Blackboard architecture is composed of knowledge sources (in ES domain) or agents (MAS domain) that communicate by message sending and receiving. An agent sends a message to the blackboard. The receiver will read the message from the blackboard. All agents use the same blackboard. The blackboard architecture is used in [83].

## 1.6 Agent-Oriented Software Engineering

Agent-based computing has the potential to significantly improve the theory and practice of modeling, designing, and implementing software systems [56], [57], [58]. However, before this happens, agent-oriented software engineering methodologies have to be developed.

The two most popular agent-oriented software engineering methodologies so far are Gaia [108], [109] and MaSE [28], [102]. In addition, approaches are being made in extending UML with agent-oriented constructs [79], [110], x[40]. Surveys of existing agent-oriented software engineering methodologies can be found in [96] and [53].

### 1.6.1 The Gaia Methodology

Gaia [108], [109] is a general methodology that supports both micro-level (i.e. agent structure) and macro-level (i.e. multi-agent system organization structure) of agent development. Gaia can be used for a broad range of multi-agent systems. It requires, however, static inter-agent relationships at run-time. Due to this fact, it cannot be used for open multi-agent systems, where system organization changes at run-time.

In order to use Gaia in an open and unpredictable domain such as Internet applications, Gaia authors propose some modifications and an extension of the methodology in [112].

The Gaia methodology includes agent-oriented analysis and agent-oriented design.

The Gaia analysis process consists of two steps:

1. defining the *roles* in the system, and
2. modeling the *interactions* between the roles.

A role has the following attributes:

- *permissions* – define what the role is allowed to do and what information it is allowed to access,
- *responsibilities* – there are two types of responsibilities:

- *safety properties* – prevent and disallow something bad to happen to the system, and
- *liveness properties* – the role has to add something good to the system.
- *protocols* – the patterns of interactions.

Gaia has formal operators and templates for representing roles and their belonging attributes, it also has schemas that can be used for the representation of interactions.

In the Gaia design process, roles are mapped into agent types, and then instances of each agent type are created. The next step is to determine the services model needed to fulfill a role in one or several agents, and the final step is to create the acquaintance model for the representation of communication between agents.

Authors of Gaia, Jennings und Wooldridge, list the common pitfalls and mistakes in agent-oriented software engineering in their paper [107].

### 1.6.2 MaSE – The Multiagent Systems Engineering Methodology

The application domain of MaSE [28], [102] is similar to the application domain of Gaia. MaSE also requires static inter-agent relationships in run-time. An additional constraint in MaSE is point-to-point communication between agents, i.e. multicasting is not allowed. In comparison to Gaia, MaSe is more oriented towards automated code generation through the MaSE tool.

## 1.7 Learning and Adaptation

The remaining part of this chapter is concerned with agent learning.

### 1.7.1 Why is Learning so Important?

As mentioned in [22], multi-agent systems are mostly used as open systems. Beside other features, open systems should possess robustness and graceful degradation. This means that new agents should be easily added to the system. Furthermore, some existing agents should be modified or removed from the system without significant performance degradation. The agent environment may change unpredictably and an agent has to react appropriately.

In order to achieve robustness and graceful degradation of a MAS, agents should be able to learn and adapt themselves to new circumstances.

In [100], *adaptation* is defined as self-modifications that enable a system to survive in a changed environment. An agent should learn about its environment in order to modify itself appropriately.

## 1.7.2 A Definition and Classifications of Machine Learning

Learning in MAS is based on machine learning techniques in AI. There are many definitions of learning. In [99], Weiss claims that there is no satisfying definition of learning. For example, one definition (given in [99]) is:

*Learning is the process of acquiring new knowledge, refining motor and cognitive skills, and integrating the acquired knowledge and the refined skills into future problem solving and planning activities.*

In [99] and [100], Weiss classifies machine learning (ML) in two dimensions: by learning strategy and by learning feedback.

Regarding learning strategy, ML can be divided into:

- *rote learning* (agent simply memorizes a given recipe),
- *learning from example* (agent uses given positive and negative examples to obtain new knowledge),
- *learning by analogy* (new knowledge is obtained by reasoning that exploits analogies in problem domain),
- *learning from observation and by discovery* (agent discovers new facts using its knowledge)

The simplest type of learning is rote learning. The types of learning listed above are sorted in ascending order by efforts needed to implement the corresponding learning type. The most difficult learning is learning from observation and by discovery.

Using learning feedback as a criterion for the classification of learning, there are three ML types:

- *supervised learning* – The agent that gives feedback to a learning agent acts as a teacher. After every action of the learning agent, the most appropriate action that should have been performed is told to the learning agent.
- *reinforcement learning* – The agent that gives feedback acts as a critic. With its feedback, it only informs the learning agent about the effects of a performed action.
- *unsupervised learning* – The learning agent does not receive any feedback after it has performed some action.

A well-known problem in ML is the *Credit Assignment Problem*. This problem was formulated by Marvin Minsky [74]. It is the problem of determining an action in a sequence (or group) of actions that causes the increase (or decrease) of system performance.

The Credit Assignment Problem can be divided into [100]:

1. the assignment of credit or blame to external actions due to overall performance change (the contribution of every performed action has to be determined),
2. the assignment of credit or blame to the corresponding internal decisions that cause an action execution (the contribution of every decision involved in the action selection has to be determined).

In MAS learning, 1. is called the *inter-agent credit-assignment problem* and 2. is called the *intra-agent credit-assignment problem*. The Inter-agent credit-assignment problem is more difficult than the assignment of credit or blame to external actions of single agent systems, because a performance change can be caused by a group of actions performed by several agents.

Learning in a MAS inherits all problems of one-agent ML and is even more difficult. Weiss and Dillenbourg analyze in [101] the significant differences between one agent learning and multi-agent learning. They identify three features that may occur in multi-agent learning, but do not occur in one agent learning:

- *multiplication* – all agents in the system use the same learning algorithm, but their learning processes are independent of each other. If the learning process of an agent fails, it does not significantly impact the overall system performance.
- *division* – agents in the system can distribute the learning process among themselves. Each agent is responsible for one part of the learning algorithm.
- *interaction* – agents can communicate during the learning process in a multi-agent system and in this way accelerate and improve the learning process.

### 1.7.3 Q-Learning

Q-learning is one of the well-known reinforcement learning algorithms. It is often used for learning in MASs [41], [86].

Q-learning can be applied if there is a finite number of states in the system and if there is a finite number of possible actions in every state. To every pair (*state*, *action*), a number (Q-value) is assigned. In every state of the system, the action with the highest Q-value has the highest probability of being chosen for execution. Q-values are determined in the following way:

1. All Q-values are initialized to 0.
2. From the current state *s*, select an action *a*, receiving immediate payoff *r* and arriving at next state *s'*.
3. Update *Q(s, a)*, based on this experience as follows:

$$\Delta Q(s, a) = \alpha[r + \gamma(\max_b Q(s', b) - Q(s, a))]$$

where  $\alpha \rightarrow 0$  is the learning rate and  $0 \leq \gamma < 1$  is the discount factor.

4. Go to step 2.

The above formula is based on the idea that a change of a Q-value should be determined by the immediate reward received and by the reward that is supposed to be received in the future if all future selected actions are the optimal ones.

In step 2, action  $a_t$  will be selected with the probability:

$$p(a_t) = \frac{e^{Q(s,a_t)/t}}{\sum_a e^{Q(s,a)/t}}$$

As the time goes on, the constant  $t$  (temperature) should slowly decrease to small positive value ( $t \rightarrow 0^+$ ). This ensures that the probability of the selection of the action with the highest Q-value increases as time passes (the system is more prone to experiment with less promising actions at the beginning than it is later).

## 1.7.4 Examples of Agent Learning

Examples of various agent learning methods are given in the remainder of this subsection.

### 1.7.4.1 Agent's Knowledge Refinement using a Refinement Facilitator

In [24], a framework for knowledge refinement in a group of agents is presented. The test domain used consists of several hunter agents that hunt large and small preys. Every hunter has its knowledge base. A hunter's knowledge base contains knowledge about other hunters, rules for making commitments to large prey hunting, knowledge about environment, prey, etc. When hunters fail in their cooperative hunt, one of them has to refine its knowledge base. The problem is in the determination of the agent whose knowledge is inaccurate. Byrne and Edwards solved this problem by introducing an external entity named the *refinement facilitator*. Every agent involved in an unsuccessful hunt generates possible refinements of its knowledge. It does that using special rules. The refinement facilitator uses its rules and determines which agent caused the failure. That agent will refine its knowledge. It will use its recent experience and apply generalization and/or specialization to the appropriate part of its knowledge base.

### 1.7.4.2 Learning from Observation

An interesting proposal is made in [40] for utilizing an available distribution system in the US Department of Defense to move its personnel, equipment and supplies in support of military operations worldwide.

Force projection is becoming a serious problem in the US Army. Centralized planning appeared to be a bottleneck in force distribution. Every shipment of personnel, equipment, food and other supplies is planned and controlled in one center. The approach presented in [40] is aimed to overcome this drawback. As opposed to centralized control, a MAS approach is proposed. According to [40], every shipment would have a small computer attached. The computer would act as a



mobile agent. It would monitor the state of the shipment and report that state to the nearest static agent. Static agents would be large computers, placed in facilities such as: depots, seaports, operation centers, etc. The inexpensive communication would be organized using Low Earth Orbit satellite network. Mobile agents would be programmed with relatively simple programs. Static agents would make decisions. Exceptions would occur when the appropriate static agent does not know how to handle a shipment. In that case, the agent would ask a human to take control.

Learning of static agents would have several steps:

1. seeding the knowledge base,
2. induction (observing how man solves problems and remembering the conclusions)
3. testing (agent also solves problems and man evaluates agent solutions)
4. independent work.

Authors of [40] suggest neural nets for knowledge representation in static agents and also propose genetic algorithms for the search of the optimal solution.

Learning from observation, as one source of agent knowledge, is proposed by Pattie Maes in [71] as well (see 1.3.2).

#### 1.7.4.3 Learning using Genetic Algorithms

In [43], genetic algorithms are used. Authors of [43] chose the predators and prey domain for the experiments with agent strategies. Agents in [43] do not communicate. Every agent (predator or prey) has its strategy written as a Lisp program. They used strongly typed genetic algorithms for the search for the best strategies (programs). The system was repeatedly executed with various strategies. Strategies with better results had a greater chance to survive. In the end, the best strategies for predators and for prey were found.

#### 1.7.4.4 Reinforcement Learning of Competitive Agents

Intelligent agents are often used as interfaces. An interface can be seen as an agent. However, in [66] an intelligent interface is presented, which consists of a group of agents. More precisely, it consists of a group of groups of competitive agents. This interface is intended for manipulation with 3D graphical scenes. The interface receives commands such as: *"move the chair to the left, a bit less, ..."*. Every agent represents possible user preferences for some aspect of the scene. In each group of agents, only one agent is active. For instance, in the group for moving objects, there can be agents that would move the chair 0.5 meters to the left, 1 meter, 3 meters, etc., if the command was *"move the chair to the left"*. However, only the active agent makes decisions. If the movement was not as the user planned, the active agent would receive a negative reinforcement. This would decrease his bid value. Perhaps it would not be the agent with the highest bid value any more. These bid values are used in the Contract Net Protocol (see 1.5.4.1) manner for active agent selection. An active agent is determined for every action.

#### 1.7.4.5 Reinforcement Learning of Optimal Condition-Behavior Pairs

In [72] M. J. Mataric uses a behavioral approach to AI (see 1.5.2.1) and presents a new type of robot learning. She used four robots with an identical architecture. Every robot possesses the following behaviors:

- avoiding (...other robots if it possesses the puck)
- dispersing (moving away from the crowd)
- searching (...for the puck)
- homing (going home with the puck)
- resting (after a long day)

The learning space of a robot is determined with the boolean variables:

- *have-puck?* (**true** if robot owns the puck, **false** otherwise)
- *at-home?* (**true** if robot is at home, **false** otherwise)
- *near-intruder?* (**true** if near some other robot, **false** otherwise)
- *night-time?* (**true** if it is time for resting, **false** otherwise)

The goal of each robot is to possess the puck and to rest during nighttime. Robots use estimation functions to obtain reinforcement in learning the most appropriate condition-behavior pairs.

#### 1.7.4.6 Case-Based Learning and the Contract Net Protocol

Case-based reasoning can also be applied in MAS learning. In [78], case-based learning is used for the refinement of the Contract Net Protocol (see 1.5.4.1). Instead of a task announcement being sent to all existing agents in the MAS, the task announcement can be sent only to agents with good performance in executing similar tasks to the current task. Each agent maintains his own case base and uses it for the determination of appropriate agents.

System testing confirmed the expectations of the authors. The communication costs were significantly reduced, without a decrease in system performance.

#### 1.7.4.7 Q-Learning in Semi-Competitive Domain

Until now, only learning in cooperative MAS domains (agents cooperate) and competitive domains (agents compete) have been presented. The semi-competitive domain is somewhere between cooperative and competitive domain.

An example of the semi-competitive domain is Iterated Prisoner Dilemma (IPD). Prisoner Dilemma (PD) is a game with two players. Each player has two actions available: to cooperate with another player or to defect. Their actions are performed simultaneously. Rewards received by both players are given in Table 1. The first value is the reward of the row player and the second value is the reward of the column player.



row player \ column player	<i>cooperate</i>	<i>defect</i>
<i>cooperate</i>	0.3 / 0.3	0.0 / 0.5
<i>defect</i>	0.5 / 0.0	0.1 / 0.1

Table 1 Rewards in Iterated Prisoner Dilemma.  
Tabela 1 Nagrade u iteriranoj dilemi zatvorenika.

As can be seen, an agent will gain the most, if the opponent cooperates and the agent defects. However, the opponent agent may not cooperate and both may receive a small reward (0.1). Therefore, the best option for both is to cooperate and get the reward 0.3.

IPD is the game in which PD is repeated many times in sequence. The aim of the players is to get as much rewards as possible.

In [86] two agents play IPD. They learn their action selection using Q-learning. Experiments have also been made when the first agent had a predefined strategy and the second agent used Q-learning. In all cases Q-learning gave satisfying results and thus proved itself as an appropriate technique for semi-competitive domains.

#### 1.7.4.8 Adaptation in Semi-Competitive Domain

Another paper describing semi-competitive domains is [90]. In [90], a MAS consists of:

- *philanthropic agents* - they will always help to any agent if he ask them,
- *selfish agents* - they always ask for help, but never help to any agent,
- *reciprocal agents* - they use reciprocity in helping others agents,
- *individual agents* - they never ask for help and never help any one.

Using experiments, authors of [90] found that reciprocal agents will always adapt themselves to any circumstances. In a heterogeneous system, reciprocal agents will benefit more than other agents. The greater the number of reciprocal agents in the system, the greater the amounts of their rewards will be.

#### 1.7.4.9 RoboCup Soccer Tournaments

RoboCup [49] is an international joint project with the goal of promoting AI, robotics, and related fields. It is an attempt to foster AI and intelligent robotics research by providing a standard problem where a wide range of technologies can be

integrated and examined. RoboCup chose to use the soccer game as a central topic of research, aiming at innovations to be applied for socially significant problems and industries.

There are five RoboCup leagues: four robot leagues and one software simulation league. RoboCup competitions provide an excellent opportunity to directly compare and evaluate various agent technologies. Robots or computer programs in one soccer team represent a multi-agent system. Each player is one agent. Together, they have one main goal: to win the game.

RoboCup tournaments have shown that machine learning is extremely important in the RoboCup domain. The team (MAS) with better machine learning algorithms wins.

There are many papers describing agent learning in the RoboCup domain. For example, in [12] it is described how an agent in the simulation league can learn to kick the ball optimally.

The ultimate goal of the RoboCup project, by 2050, is to develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer.

## 2 Overview of Agent-Oriented Programming Languages and Tools

AGENT0 .....	30
PLACA.....	32
Concurrent MetateM.....	33
AgentSpeak .....	35
JACK .....	36
ZEUS .....	37
HOMAGE .....	40
COOL .....	40
SICSLOG .....	42
KIDSIM.....	42
KQML and FIPA ACL.....	43
Mobile Agent Tools and Languages .....	44
Java .....	45

Software agents are computer programs. As well as every other program, a software agent is implemented in a programming language. A hardware agent also needs a program that controls it.

Desirable properties of an agent programming language depend on the agent type. It is very convenient if a language possesses some of the theoretical concepts used for agent specification and modeling. This enables higher-level programming.

Some of the most popular and most well-known programming languages and development tools used for agent implementation are the subject of this chapter. There are, however, many other tools and languages, which could not be described here due to limited space.

## 2.1 AGENT0

Agent-Oriented Programming (AOP) is the expression invented by Yoav Shoham [91]. Shoham sees AOP as a subset of concurrent object oriented programming. He insists on the usage of mental categories in AOP languages. He developed the language AGENT0 in order to demonstrate his idea.

Shoham compared AOP and concurrent OOP in [91] (Table 2).

	OOP	AOP
Basic Unit	object	agent
Parameters defining state of basic unit	unconstrained	beliefs, commitments, capabilities, choices, ...
Process of computation	message passing and response methods	message passing and response methods
Types of message	unconstrained	inform, request, offer, promise, decline
Constraints on methods	none	honesty, consistency

Table 2 OOP and AOP.

Tabela 2 OOP i AOP.

According to Shoham in [91], every AOP system should be composed of three parts:

- A language for agent mental space description.
- A language for programming of agents.
- An 'agentifier', which converts devices such as a robot, camera, etc. into programmable agents.

In his significant work [91], Shoham proposes the following scenario for agent execution:

1. Read current messages and update mental state, including beliefs and commitments.

2. Execute commitments for current time, possibly resulting in further belief change.
3. Go to 1.

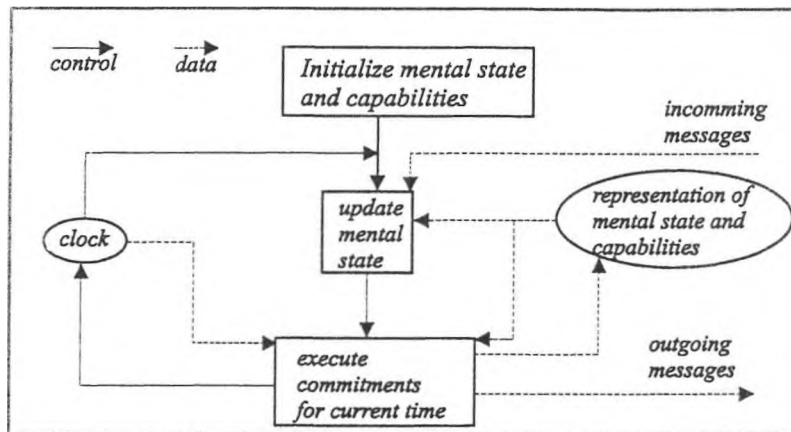


Figure 2 Generic Agent Interpreter.  
Slika 2 Opšti interpreter agenta.

An agent program is obliged for the execution of the step 1. Step 2 does not depend on the agent program.

Shoham also gave a flow chart of a generic agent interpreter (Figure 2).

According to Shoham, agent commitments are determined by its mental state and incoming messages.

In AGENT0, Shoham allows four types of communicative actions, which are derived from speech-acts theory.

- (INFORM t agent fact)
- (REQUEST t agent action)
- (UNREQUEST t agent action)
- (REFRAIN action)

A part of AGENT0 syntax is given below. Terminal symbols are written in bold style. Star (\*) represents repetition of an item (any number of times, including 0 and 1).

```

<program> ::= timegrain := <time>
             CAPABILITIES := ( <action> <mntcond> ) *
             INITIAL BELIEFS := <fact> *
             COMMITMENT RULES := <commitrule> *

```

An AGENT0 program consists of three parts.

In the first part agent capabilities are listed. Each agent action is accompanied by its mental conditions. An action will not be executed unless the mental conditions are satisfied.

In the second part of a program, agent's initial beliefs are specified.

Commitment rules are the main part of a program.

```
<commitrule> ::=  
    (COMMIT <msgcond> <mntlcond>(agent action)* )
```

Commitment rules are the core of an AGENT0 program. They determine the behavior of an agent. When an agent receives a message, it tries to apply one or more of its commitment rules. A rule will be applied if the message satisfies the message condition of the rule and the mental state of the agent is as described in the rule. If both conditions are satisfied, the agent will commit itself for actions listed in the rule.

While performing actions, an agent sometimes has to modify its mental state. A new belief or commitment that will be added to the mental state of an agent should be consistent with the previous beliefs and commitments. If this is not the case, there are two possible actions:

- to delete the old belief or commitment and add the new one,
- to leave the mental state unchanged.

The verification whether the mental space is consistent is sometimes very hard to perform. In [91], three ways of beliefs and commitments maintaining are proposed:

1. formal methods of mathematical logic,
2. heuristic methods,
3. making the language for mental space description as simple as possible, thus enabling trivial verification.

The third solution is used in AGENT0.

Variables can also be used in rules. Every literal beginning with a question mark is a variable (for instance ?x). A variable can get its value in a similar way PROLOG variables are instantiated.

## 2.2 PLACA

AOP language PLACA [94], [95] is a direct descendant of AGENT0. It is a language similar to AGENT0, but it brings some improvements. PLACA possesses planning facilities. This feature significantly reduces the frequency of communication in comparison with AGENT0 programs. In PLACA, an agent does not have to send a request for every action that it wants another agent to perform. It can send only one request message to another agent – a request for a desired state of

affairs. The receiver agent will check its rules. If both rule conditions are satisfied, it answers that it will plan to achieve the desired state of affairs. The receiver agent will use its planning abilities. He will not receive a message for every single action.

Due to the introduction of plans, mental categories in PLACA are different from those in AGENT0. The syntax has also been changed. Syntax for a rule in PLACA is as follows:

```
<rule> ::= (<message-condition>, <mental-condition>,
           <mental-changes>, <message-list>)
```

If the received message satisfies <message-condition> and if the current mental state of an agent satisfies <mental-condition> then the agent's mental state will be changed as specified in <mental-changes> and the agent will send messages listed in <message-list>. An Example of a mental change rule (from [95]) is:

```
( (TO Fred, FROM boss, REQUEST, ?x),
  (),
  ((ADOPT (INTEND ?x))),
  ((TO boss, FROM Fred, INFORM, (*now* (intend ?x))))
)
```

When a boss requests something from Fred, whatever current Fred's mental state is, it will adopt an intention to do what boss has requested and it will inform boss about its new intention.

## 2.3 Concurrent MetateM

Both mentioned languages, AGENT0 and PLACA, are tightly connected with the appropriate logics describing their mental categories. In Concurrent MetateM [34], [35], [104], the connection between the logic and the language is even stronger. Concurrent MetateM uses a specification of a MAS with formulas of a propositional linear temporal logic named PML (Propositional MetateM Logic).

In Concurrent MetateM, every agent is defined by its interface and its computational engine. An interface consists of an identifier of an agent, a set of symbols it recognizes and a set of symbols it can send. An example of an interface is given in [104]:

```
stack (pop, push) [popped, full]:
```

The agent *stack* recognizes *pop* and *push* and sends *popped* and *full*.

The computational engine contains rules of the form:

*antecedent about past*  $\Rightarrow$  *consequent about present and future*

PML contains temporal connectives listed in the Table 3.



<u>Strict past time connectives:</u>	<u>Present and future time connectives:</u>
○ - weak last	○ - next
● - strong last	◇ - sometimes
◆ - was	□ - always
■ - heretofore	U - until
S - since	W - unless
Z - weak since	

Table 3 Connectives in PML.  
Tabela 3 Veznici u PML-u.

An agent program is executed in the following cycle:

1. Adding new messages to the history.
2. Retrieving the rules that have to be performed.
3. Concurrent execution of the rules and commitments from the past (this is the most difficult step).
4. Go to 1.

An example of one MAS specification is given below [104]. It consists of three agents. One is a resource producer and remaining two are resource consumers.

```
rp (ask1, ask2) [give1, give2]:
  ● ask1 ⇒ ◇ give1;
  ● ask2 ⇒ ◇ give2;
  start ⇒ □ ¬ (give1 ∧ give2)

rc1 (give1) [ask1]:
  start ⇒ ask1;
  ● ask1 ⇒ ask1

rc2 (ask1, give2) [ask2]:
  ● (ask1 ∧ ¬ ask2) ⇒ ask2
```

**rp** recognizes two symbols: **ask1** and **ask2**. It can send two symbols as well: **give1** and **give2**. The first rule written commits **rp** to send **give1** now or somewhere in the future if it received symbol **ask1** somewhere in the past. The second **rp**'s rule is similar.

**rc1** recognizes only the symbol **give1** and can send only the symbol **ask1**. Its two rules force it to constantly transmit the symbol **ask1**.

**rc2** sends the symbol **ask2** only once, after **rc1** has sent its symbol **ask1**.

The specification of a MAS is executed without any translation. The connection of Concurrent MetateM with logic makes it a very interesting language. However, its small expressive power makes it inconvenient for practical applications.

## 2.4 AgentSpeak

In [98] the language AgentSpeak is described. Creators of AgentSpeak see AOP as a new programming paradigm that will introduce DAI concepts, such as mental categories, into programming of multi-agent systems. They tried to join the Beliefs–Desires–Intentions (BDI) architecture and Object Based Concurrent Programming. The aim was to bring together MAS concepts (mental categories, reactive and proactive behavior, communication using speech acts, distribution over a wide area network, real-time response, concurrent execution of plans in and among agents, meta-level reasoning, etc.) and OO programming. As a result of their work, the language AgentSpeak was created.

AgentSpeak syntax reminds of C++. Instead of a class, AgentSpeak uses an agent-family. Object fields in C++ correspond to relations in AgentSpeak. Methods correspond to services. Every agent family has its public and private parts. Nevertheless, there are some essential differences between AgentSpeak and object oriented languages. There are only three types of services:

- to achieve a certain relational tuple (i.e. to achieve a certain state of the world),
- to query the existence of a particular relational instance (i.e. to check whether something is true),
- to send a message with a particular relational instance to another agent (i.e. to tell something to another agent).

The procedural part of the language is placed in plans. Plans are made for services and are invoked on service activation. The syntax for plan declaration is:

```
plan <plan_name> {  
  invoke on <service_statement>  
  with context <abs_situation>  
  perform <goal_statement> {; <goal_statement>}  
  finally assert <abs_situation> }
```

Where <abs\_situation> is the conjunction of the disjunction of relational instances and <goal\_statement> is defined as below:

```
<goal_statement> ::=  
  <assign_stat> |  
  <while_stat> |  
  <if_then_else_stat> |  
  <non_deterministic_stat> |  
  <service_statement> |  
  <speech_act_stat>  
  
<speech_act_stat> ::=  
  <inform-act-statement> |  
  <request-with-wait-statement> |  
  <request-with-no-wait-statement>
```

When `<service_statement>` in the `invoke on` slot is executed and `<abs_situation>` in the `with context` slot is satisfied, the plan may be selected for execution. If that happens, statements in the `perform` slot will be executed and after the execution the contents of the `finally assert` slot will be added to the agent's mental state.

After an agent receives a speech act that triggers some service statement, it selects all the relevant plans. The *applicable plans* are those relevant plans, which satisfy the context condition. One *applicable plan* will be chosen for execution. The chosen plan is called *intention*.

At any time, an agent can have several intentions that execute simultaneously. Every message in an agent's mailbox has its priority. Message priorities determine the order of message processing.

## 2.5 JACK

JACK Intelligent Agents™ [47], [23] is an agent-oriented development environment created by Agent Oriented Software Pty. Ltd in Australia. It is built on top of and fully integrated with the Java programming language. Similar to agents in AgentSpeak (2.4), JACK agents are also BDI agents.

Authors of JACK describe its relationship to Java with the analogy to the relationship between the C++ and C languages. C was developed as a procedural language and subsequently C++ was developed to provide programmers with object-oriented extensions to the existing language. Similarly, JACK has been developed to provide agent-oriented extensions to the Java programming language.

JACK source code is compiled into regular Java code before being executed.

JACK agents are autonomous, reasoning entities capable of making pro-active decisions while reacting to events in a real-time environment. They have the Belief Desire Intention (BDI) architecture. Following the BDI model, JACK intelligent agents are autonomous software components that have explicit goals to achieve or events to handle (desires). To describe how they should go about achieving these desires, these agents are programmed with a set of plans. Each plan describes how to achieve a goal under varying circumstances. Set to work, the agent pursues its given goals (desires), adopting the appropriate plans (intentions) according to its current set of data (beliefs) about the state of the world. This combination of desires and beliefs initiating context-sensitive intended behavior is part of what characterizes a BDI agent.

Each JACK agent has:

- a set of beliefs about the world (its data set);
- a set of events that it will respond to;
- a set of goals that it may desire to achieve (either at the request of an external agent, as a consequence of an event, or when one or more of its beliefs change); and

- a set of plans that describe how it can handle the goals or events that may arise.

The JACK Agent Language is a super-set of Java – encompassing the full Java syntax while extending it with constructs to represent agent-oriented features (from [1]):

- It defines new base classes, interfaces and methods.
- It provides extensions to the Java syntax to support new agent-oriented classes, definitions and statements.
- It provides semantic extensions (runtime differences) to support the execution model required by an agent-oriented software system.

The JACK Agent Language introduces five main class-level constructs. These constructs are (from [1]):

- *Agent* – The agent construct is used to define the behavior of an intelligent software agent. This includes capabilities an agent has, what type of messages and events it responds to and which plans it will use to achieve its goals.
- *Capability* – The capability construct allows the functional components that make up an agent to be aggregated and reused. A capability can be made up of plans, events, beliefsets and other capabilities that together serve to give an agent certain abilities. An agent can, in turn, be made up of a number of capabilities, each of which has a specific function attributed to it.
- *BeliefSet* – The beliefset construct represents agent beliefs using a generic relational model. It has been specifically designed so that a beliefset can be queried using logical members. Logical members are like normal data members, except that they follow the rules of logic programming (as in programming languages like PROLOG).
- *View* – The view construct allows general-purpose queries to be made about an underlying data model. The data model may be implemented using multiple beliefsets or arbitrary Java data structures.
- *Event* – The event construct describes an occurrence in response to which the agent must take action.
- *Plan* – An agent's plans are analogous to functions. They are the instructions the agent follows to try to achieve its goals and handle its designated events.

## 2.6 ZEUS

ZEUS [44], [4], [77], [76] is an agent building environment and component library created at British Telecommunications Labs. ZEUS is one of the most complete and

most powerful agent tools. It provides far more than a set of Java classes for agent implementation (what is the case with many other agent tools).

ZEUS agents are collaborative agents. Each ZEUS agent consists of three layers:

1. *definition layer* - The definition layer represents the agent's reasoning abilities, its goals, resources, skills, beliefs, preferences, etc.
2. *organizational layer* - The organizational layer describes the agent's relationships with other agents.
3. *co-ordination layer* - The co-ordination layer describes the co-ordination and negotiation techniques the agent possesses.

Communication protocols are built on top of the co-ordination layer and implement inter-agent communication. Beneath the definition layer is the API.

ZEUS provides a library of software components and tools that facilitate the rapid design, development and deployment of agent systems. The three main functional components of the ZEUS toolkit are (adapted from [44]):

- The Agent Component Library
- The Agent Building Tools
- The Visualization Tools

The Agent Component Library is a collection of various software components, e.g.:

- A TCP/IP-based message passing mechanism capable of transmitting FIPA ACL performatives.
- Implementations of IIOP and HTTP protocols.
- A library of predefined co-ordination strategies, represented in the form of recursive transition network graphs; these include several variants on contract-net, and auction protocols for more commercially oriented behavior.
- A co-ordination engine that drives agent interactions by executing co-ordination strategies.
- Support for several types of organizational relationships within agent societies.
- A general purpose planning and scheduling mechanism to support goal-driven intelligent behavior.
- Support for agent competencies in terms of primitive actions, summary plans, forward chaining rules and self-executing behavior scripts.
- Representations to store and exchange information on tasks and ontology concepts.

- An agent-to-legacy system interface to facilitate inter-operability with existing software systems.
- Full implementations of three different utility agents that provide runtime support services: agent name-to-network location resolution (Name Servers), service discovery (Facilitators) and persistent storage (Database Proxies).

The Agent Building Tools provide an integrated suite of editors that guide developers through the stages of the comprehensive agent development methodology. During this process developers describe the agents within their application, how they interact, and the tasks they perform. Amongst the tools provided are:

- An Ontology Editor for defining the concepts, attributes and constraints within a domain.
- An Agent Definition Editor for describing agents logically, e.g. their tasks, initial resources, planning abilities etc.
- A Task Description Editor for describing the attributes of tasks and for graphically composing summary tasks.
- An Organization Editor for defining the organizational relationships between agents, and agents' beliefs about the abilities of other agents.
- A Co-ordination Editor for selecting the set of co-ordination protocols with which each agent will be equipped, and the strategies that influence the agent's behavior.

The Visualization Tools collect information on agent activity, interpret it and display various aspects in real-time. The following tools are included:

- A Society Viewer that shows all known agents, their organizational relationships and the messages they exchange.
- A Reports Tool that shows the society-wide decomposition/distribution of active tasks and the execution states of the various tasks.
- A Statistics Tool that displays individual agent and society-wide statistics in a variety of formats.
- An Agent Viewer that enables the internal states of agents to be observed and monitored.
- A Control Tool that is used to remotely review and/or modify the internal states of individual agents.
- And each tool is supported with an online hypertext-based help system.

ZEUS includes a code generator tool, which automatically converts agent definitions into executable Java source code. The code produced by the generator tool is created in the form of call-back methods, this allows the developer to integrate agents with

application specific code. An "Application Programmers' Interface" (API) is provided that allows existing software to be connected to ZEUS agents.

## 2.7 HOMAGE

In [81], a multi language environment is presented. It is called HOMAGE and is aimed at the development of MAS applications. HOMAGE has two programming levels. The lower level is object oriented. It is used for the definition of objects in object oriented (OO) languages C++, Common Lisp and/or Java. Objects defined at lower level are used in a higher, agent-oriented level. The higher level contains almost all of the essential features of AOP defined in [91]. It contains many additional features as well. Nevertheless, it does not use mental categories. Authors from [81] use objects from the lower level for agent internal state representation and agent action definitions. Object fields represent the internal state of an agent, and methods represent parts of the actions.

Agent rules are similar to commitment rules from [91]. They have the following syntax:

```
rule_name
  msg_pattern
  precondition
  [disable disable_list]
  service
  [enable enable_list]
```

The rules are grouped and a group of rules can be enabled or disabled by rule execution.

HOMAGE enables agents to communicate using the Internet. They can use TCP/IP, HTTP, and e-mail protocols.

HOMAGE is made for efficient programming of multi-agent systems. Its authors did not include mental categories. They used well-known OO programming, which proved to be efficient and with rich expressive capabilities. Programming in terms of mental categories requires additional learning and adaptation for programmers. Creators of HOMAGE probably wanted to make a programming environment that would be immediately applicable for MAS programming.

## 2.8 COOL

Every AOP language described so far is made for MASs where agents work without communication most of their time. They communicate only occasionally and their conversations are relatively simple ones. One question and one response are the most common case. An AOP language that is aimed at MASs with complex conversations is described in [13]. Barbuceanu and Fox have developed the



language COOL (COOrdination Language), that enables definitions of conversation classes. An example of a conversation class in COOL is [13]:

```
(def-conversation-class 'customer-conversation
  :name 'customer-conversation
  :content-language 'list
  :speech-act-language 'kqml
  :initial-state 'start
  :final-states '(rejected failed satisfied)
  :control 'interactive-choice-control-ka
  :rules '((start cc-1) (proposed cc-13 cc-2)
          (working cc-5 cc-4 cc-3)
          (counterp cc-9 cc-8 cc-7 cc-6) (asked cc-
10)
          (accepted cc-12 cc-11)
          )
)
```

Every conversation class has its name. There are two languages assigned to every conversation. The content language is an 'internal' language used for the specification of message contents. The speech act language is an 'external' language used for the specification of additional information to the information represented by the content language. In the above example the external language is KQML. Additional information may describe the purpose of the message, the type of the answer expected, etc.

Beside other information, a class definition specifies a finite state automaton (initial-state, final-state, and rules slots define the automaton). A conversation is comprised of several message passings between two participants. Every received message triggers a rule, which executes some actions (usually including message sending) and changes the current state in the automaton. The automaton is the main part of the conversation class. When it reaches some final state, the conversation is finished. Example [13] of rule definition used in a conversation class is given below.

```
(def-conversation-rule 'con-1
  :current-state 'start
  :received '(propose :sender customer
                   :content '(customer-order
                              :has-line-item
?li))
  :next-state 'order-received
  :transmit '(tell :sender ?agent
                  :receiver customer
                  :content '(working on it)
                  :conversation ?conv)
  :do '(put-conv-var ?conv ?order
        (cadr (member :content ?message)))
  :incomplete nil)
```

The above rule is named **con-1**. It can be applied in the **start** state of the automaton, when the message in the **received** slot arrives. The rule will transmit the message in the **transmit** slot, do the action in the **do** slot and change the state of the automaton to the **order-received** state.

When the A wants to use conversation class *c1* for communication with agent B, it sends the following message to agent B [13]:

```
(propose                ;; new communicative action
  :language list
  :sender A
  :receiver B
  :content (or (produce 200 widgets)
              (produce 400 widgets))
  :conversation c1      ;; first new slot
  :intent (explore fabrication possibility)
  .....;;second new slot
)
```

Agent A uses *conversation* slot to propose the communication class for the started communication. The *content* slot of this message contains a content language expression and *intent* slot explains to the B agent what is the intention of the conversation.

Modified KQML (see 2.11) is used as a speech-act language in the above example. Authors of [13] extend KQML language with the *propose* communication action and with two additional slots: *conversation* slot and *intent* slot.

## 2.9 SICSLOG

The language used for programming of agents described in 1.5.1.1 (Figure 1) is named SICSLOG. This is also a language based on the stronger notion of agency.

Similar to AgentSpeak (2.4), plans of SICSLOG agents are the only part of the language, which may contain procedural elements. A plan consists of an action sequence. Two standard control structures can be used in plans: the *conditional action* and the *looping action*. A plan can also generate new intentions, using the *stip action* ('seeing-to-it-that' action).

## 2.10 KIDSIM

KIDSIM [92] is a tool for the creation of computer simulations. It was made as a part of the research that had the aim to make computer programming easier and thus available to a greater population of people. The main reason for seeing computer programming difficult is the distance between human problem representation and the representation of the same problem in some programming language. The greater the distance, the more difficult it is to program. In KIDSIM, every element of the simulation is visible on screen all the time. These elements are agents. They are persistent in time and they have their rules, which determine their behavior. Rules are created graphically. KIDSIM uses pictures instead of text. Agents can be copied. This operation is also visible graphically and can be performed simply with a mouse. Once the agents have their rules and are placed in their positions on the screen, the simulation can begin.

Testing with KIDSIM has shown that almost every child was able to make a complex simulation after a short period of learning. Children liked KIDSIM and enjoyed the work. The test with conventional programming languages gave opposite results.

## 2.11 KQML and FIPA ACLs

There are researchers demanding for use of a common language for agent communication [80]. Usage of an agent communication language (ACL) should distinguish agents from other programs. As described in [38], an ACL should be domain independent and should enable agents to send and receive messages whose content is written in some other language.

The most well known ACLs are KQML [33] and FIPA ACL [36].

KQML (Knowledge Query and Manipulation Language) is developed at the DARPA Knowledge Sharing Initiative External Interfaces Working Group.

KQML uses speech-act performatives such as `reply`, `tell`, `deny`, `untell`, etc. Every KQML message consists of a performative and additional data written in several slots. Some slots are `:content`, `:in-reply-to`, `:sender`, `:receiver`, `:ontology`, etc. An example of a KQML message is:

```
(tell
  :sender Adam
  :content (equal 5 (+ 2 3))
  :receiver Bill
  :in-reply-to math-question
  :language KIF
  :ontology maths)
```

Agent `Adam` tells agent `Bill` that  $5 = 2+3$ . It uses the KIF language for the content slot value. This message is a reply to the message that had `math-question` in its `reply-with` slot. Ontology should help `Bill` in understanding the expression in the content slot.

KIF is a first-order predicate language written in a Lisp-like manner. Creators of KQML propose KIF as an appropriate language for content slot expressions. However, they do not disapprove with the usage of other languages instead of KIF.

The set of performatives in KQML and their slots should be general enough to enable agent communication in every agent application. Whether the performatives chosen possess such an expressive power or not, is still an open question. Some claim [2] that there might be some problems with the semantic of performatives. Various agents may interpret the same performative in various ways.

Despite the fact that KQML design has not been completed yet, KQML has been used in some agent application and tools. For example the language COOL (2.8) uses KQML for message passing between agents.

FIPA ACL is a standard ACL defined by FIPA - Foundation for Intelligent Physical Agents. The syntax and semantics of FIPA ACL are very similar to the syntax and semantics of KQML. Time will show which one of these two standards will prevail.

## 2.12 Mobile Agent Tools and Languages

There are many agent development kits made for programming of mobile software agents. They are mainly written in Java. These tools and runtime environments facilitate the transfer of Java mobile objects (agents) from one computer to another, where they continue their execution.

Some mobile agent kits are listed in the Table 4.

Product	Company / University	Language
Aglets	IBM Japan	Java
Concordia	Mitsubishi Electric, Information Technology Center America	Java
Gossip	Tryllian	Java
Grasshopper	IKV++	Java
Jumping Beans	Ad Astra Engineering, Inc.	Java
Voyager	Object Space	Java
Agent Tcl	Dartmouth University	Tcl
Gypsy	Technical University of Vienna	Java
Knowbot System Software	The Corporation for National Research Initiatives (CNRI)	Python
Mole	University of Stuttgart	Java
NOMADS	Institute for Human and Machine Cognition, University of West Florida	Java
Odyssey	General Magic	Telescript
Ara	University of Kaiserslautern	C/C++, Java, Tcl
Tacoma	Norway & Cornell	C
MOA	OpenGroup, UK	Java

The Tube	David Halls, UK	Scheme
Ajanta	Minesota University	Java
Kafka	Fujitsu Laboratories, Japan	Java

Table 4 Mobile agent programming tools.  
Tabela 4 Alati za programiranje mobilnih agenata.

## 2.13 Java

General-purpose languages are also used for agent implementation. The most used general-purpose language for agent programming is Java. Other general-purpose languages used include C++, Perl, Scheme, etc.

Java is an object-oriented language developed at Sun. Soon after its release, this language has become one of the most popular programming languages today. It is suitable for agent programming because of several characteristics:

- high level support for network programming (Internet applications),
- remote method invocation (RMI) facilitates the development of distributed applications,
- concurrency (multiple threads of executions),
- device independence (every platform that can run the Java Virtual Machine is able to execute the same Java classes),
- Java applets are used for some mobile agent implementations,
- object-oriented programming is a powerful programming paradigm and is especially suitable for agent programming (in [91] agents are defined as objects that satisfy some constraints).

Many agents have been directly implemented in Java. In addition, almost all agent development tools and languages have also been implemented in Java.

The software developed as a part of this thesis is also written in Java.

### 3 AJA Features

The Ideas behind AJA .....	47
AJA Agent Architecture .....	50

This chapter introduces the agent building tool AJA, which is implemented in the practical part of this thesis. An overview of the AJA agent architecture is given describing all AJA agent components and how do they work together.

AJA is the acronym for Adaptable Java Agents. The design of AJA is based upon the positive and negative experience obtained in designing and implementing the agent-oriented language LASS ([5], [8], [10], [11]) as well as upon the analyses of the existing agent building tools and environments.

## **3.1 The Ideas behind AJA**

AJA tool is based on the ideas summarized in this section.

### **3.1.1 New Agent-Programming Language is used Together with Java**

A few agent-oriented languages have been created so far, mainly as the results of agent-research projects. The purpose of these languages, such as AGENT0 (2.1), PLACA (2.2), Concurrent MetateM (2.3), AgentSpeak (2.4), is to introduce a new programming paradigm, to show new agent-oriented programming concepts and language constructs. However, these languages are not well suited for the commercial projects, because they miss the conventional features, such as database programming support, GUI programming support, etc. However, as already said, the purpose of aforementioned languages was not even to be used in commercial projects at the first place.

The goal of this thesis was to create an agent-development tool that also introduces new agent concepts at the programming language level. In addition, another goal was to create a tool that can be used in the implementation of the enterprise-scaled real-world multi-agent systems. In other words, the tool introduces new language, but at the same time it should be powerful enough to support all standard features (e.g. database access, windows-based GUI, etc.) that can be found in popular programming languages, such as Java.

To design and implement a new programming language powerful enough to be used in commercial software implementation requires far more resources than a Ph.D. project can afford. Furthermore, the existing programming languages, first of all Java, already supports all the required features: database programming, network programming, security, multimedia, etc. It would make no sense to implement everything again in an agent-oriented language. Instead, two languages can be used together: an agent-programming language to specify high-level agent parts and Java can be used to implement these components.

### **3.1.2 The infrastructure for agent programming**

Programming a software agent from scratch, e.g. in Java, requires a certain amount of programming skills and time. For example, agent program should be able to communicate over the network with user(s) and other agents, so the network programming is required. Consequently, a security issue has to be handled



appropriately. In addition, an agent can execute several activities and/or communicate with several other agents at the same time, which raises the thread synchronization issue. The infrastructure parts in an agent program, such as network-programming and agent communication, security, and thread synchronization can take more programming hours and days than programming of so called "higher-level" agent parts. Due to this reason, the language or tool used for agent implementation should provide the infrastructure for agent programming enabling thus the agent programmer to concentrate on other, more important parts of agent program.

### 3.1.3 Agents as a Vehicle for AI

Artificial intelligence (AI) research in 70's and 80's caused great expectations from AI in the near future. However, AI technologies such as machine learning, planning, etc. are still not being used in the mainstream programming. There are two main reasons why is it so:

- In order to implement an AI technique from the scratch, one needs deep background knowledge of the corresponding AI field. An average mainstream programmer is usually not familiar with AI.
- The nature of most nowadays information systems is rather "static". In other words, the order of the program execution steps has been exactly specified in advance at the programming time. In such an approach there is no space for AI.

The latter of the above two problems could be overcome introducing "dynamic", agent-based information systems instead of current "static" ones. Despite the fact that "static" solution is the only possible solution for some applications (e.g. the banking software for the money transfer), there are many situations, where the "dynamic" approach is more suitable. For example, modern companies and organizations today have their information resources connected through the large corporate Intranets. Programs in the system are interrelated and depend on each other. The overall system is harder to control and maintain if the programs in the system are inflexible, written in the classical "static" manner, without agents and AI<sup>2</sup>.

As suggested in [89] and [88], intelligent agents can be a vehicle for AI-related technologies. If intelligent agents and agent-based systems become a mainstream, then AI can indirectly also become a mainstream.

However, the former of the above two problems remains. Even if agents become the mainstream and average software designer and developer start to think in agent terms they still miss AI background. A programmer cannot program planning component of an agent, if he/she does not know how to do it.

Fortunately, this problem can also be overcome. Instead of programming AI components from the scratch, a programmer can use already implemented freely or commercially available pluggable AI components and embed them into program.

---

<sup>2</sup> An IBM research team analyses this problem in [52] and suggests a radically new, AI-based approach in building the information systems called Autonomic Computing.

This could be done in the similar manner the JavaBeans or ActiveX components are used. A programmer does not have to understand how the AI component is implemented. All he/she needs to know is how to use the component.

Because of the fact that pluggable AI components (e.g. JavaBean for neural machine learning) are still not generally available, an alternative approach is to embed AI techniques into agent-programming language. For example, a language could contain a construct that corresponds to neural network. A programmer uses this construct in his/her program and the runtime environment of the agent-programming language creates and controls the neural network during the program execution. This approach is used in AJA.

### **3.1.4 Inter-Agent Communication is Compound**

Almost all agent-building tools facilitate the communication between agents. There is a support for textual message sending over the network and for receiving a message. This means that an agent developer does not have to program at the network socket level, because the tools already implement this part of agent infrastructure. But, agent communication usually consists of several interrelated message sending and message receiving. However, agent languages and tools mostly do not help in composing together the single interrelated communication acts. One exception is the language COOL (see 2.8), where the communication between two agents is modeled with an automaton. After sending or receiving a message the automaton changes its state. The communication is finished when the automaton has reached the final state. This approach is also used in AJA.

### **3.1.5 Inter-Agent Communication is Secure**

The security of the system is often the most important subject in commercial software projects. The most vulnerable part of MAS, from the security perspective, is its inter-agent communication. An agent development tool can therefore never be seriously accepted from the mainstream programmers community, if it does not support secure agent-to-agent communication. In addition, the security implementation should be based on the standard and well-known algorithms and concepts that have already been proved and verified as secure.

The most secure communication, encrypted communication, can be achieved using e.g. Secure Socket Layer (SSL) communication protocol with both client and server authentications. The encryption and decryption of messages require however some amount of computing resources, what cannot be ignored in some situations.

Sometimes however the message content is not secret, so it does not have to be encrypted. Nevertheless, it can be important for the message receiver agent to be sure to know who has sent the message and to be sure that the message has not been tampered on its way through the network. This can be achieved with the digital signature of the messages. Signing of the message is not so expensive as the message encryption, but it also requires certain amount of the processor time.

If a MAS is isolated or secured on some other way, or if the security is not the issue in a MAS, then the message sending without encryption or digital signing should be used in order to avoid overhead.

AJA supports all three abovementioned security levels of agent-to-agent messaging.

### 3.1.6 Agent acts Reactively as well as Goal-Oriented

An agent performs actions in order to reach given goals i.e. it acts goal-oriented. This feature is not common only to agents, but it is a feature of conventional programs as well. Nevertheless, in contrast to conventional programs, an agent should also act reactively. This means, parallel to pursuing its goals, agent should "sense" its environment and react when it is appropriate.

## 3.2 AJA Agent Architecture

An AJA agent consists of the following parts:

- *Beliefs* – primitive values, data structures, and the values generated by AI components. Agent beliefs define its internal state.
- *Actions* – blocks of code, possibly with parameters, that can be seen as atomic agent actions. AJA provides the synchronization mechanism for the parallel actions execution.
- *Reflexes* – condition-action(s) pairs. Reflex is a reactive component. If the condition part of a reflex is satisfied, the action(s) part consisting of a sequence of actions will be executed. Reflex conditions are checked periodically using a specified time period. Condition checking can also be forced at any time in any place in agent program.
- *Negotiations* – represent the inter-agent communication as automaton. A negotiation automaton includes states, transitions, message sending, and message receiving. There are three types of negotiations in AJA:
  - *requesting negotiation* – used when agent initiates the communication,
  - *responding negotiation* – used when agent responds to the communication request initialized by another agent,
  - *WWW negotiation* – used when a user communicates with an agent via Internet browser.
- *Initialization part* – AJA agent executes its initialization part immediately after it has been started.

Two languages are used for the programming of AJA agents:

- **HADL** (Higher Agent Definition Language) – contains the constructs for the higher-level declaration of agent beliefs, actions, negotiations, reflexes, and initialization.

- **Java+** - is used for the implementation of higher-level agent parts defined in HADL. Java+ extends Java with the constructs for accessing agent beliefs, actions, negotiations, reflexes, and GUI. Because of the fact that Java+ extends Java, Java+ makes the agent integration with the legacy Java software straightforward. Any Java class can be used in any Java+ part of AJA agent.

Agent program written in HADL and Java+ is translated into Java. Chapter 4 describes HADL and Java+ in more details.

Regarding various agent classifications presented in section 1.2, AJA agents can be classified as:

- agents based on the stronger notion of agency,
- software agents,
- static agents,
- middle-sized and big-sized agents,
- agents that can learn,
- agents with hybrid architecture, and
- agents for both cooperative and competitive systems multi-agent systems.

### 3.2.1 Beliefs

Beliefs of an AJA agent represent the information it has about the world. Agent beliefs define its internal state. There are three types of beliefs in AJA:

- *Java values* (both primitive and compound).
- *Adaptable parameters*.
- *Dependant values*.

#### 3.2.1.1 Java Values

This type of AJA beliefs resembles global variables in traditional programming languages. They store primitive or compound values. The listing below shows the declaration of three such beliefs.

```
<<
import demo.TimeTable;
import java.util.Vector;
>>
BELIEFS

timeTable : TimeTable;

birthdaysTodayToAlert : Vector ;

toBackup : boolean = << false >>;
```

The type of `timeTable` belief is `demo.TimeTable`. The type of `birthdaysTodayToAlert` is `java.util.Vector`, and `toBackup` is a `boolean` value.

### 3.2.1.1.1 Java+ Constructs

Java values beliefs could be accessed in Java+ using the constructs:

- `$GET_BEL(belName)` – returns the value of the belief *belName*,
- `$SET_BEL(belName, newValue)` – sets a new value to the belief *belName*.

The following listing shows how can this be done:

```
***
TimeTable tt = $GET_BEL(timeTable);
Vector engagements = tt.getEngagements();
***
$SET_BEL(toBackup, true);
***
```

### 3.2.1.2 Adaptable Parameters

An agent program usually contains constants whose optimal values are not known in advance, because they depend on user preferences or some other unpredictable values.

For example, an appointment scheduling PDA agent reminds its user before the appointment starts. However, there is one problem here. The optimal default reminding time for appointments is not known at agent programming time. It depends on the preferences of particular user.

Of course, one solution to this problem would be to ask user for the default reminding time. However, the user-specified default reminding time does not have to be the optimal one. Furthermore, asking the user to specify every parameter in an agent application is not in the agent manner.

Another solution to the optimal constant value problem is to use adaptable parameters instead of constants. Adaptable parameter adjusts its value at run-time according to the feedback received. AJA supports adaptable parameters as a type of agent beliefs. The value of an adaptable parameter is a real number (Java type `double`). The listing below shows an example of adaptable parameter declaration.

```
BELIEFS
***
eventAlertTime : ADAPTABLE LBOUND << 0 >> = << 15 >>;
***
```

`eventAlertTime` is an adaptable parameter with the lower bound 0 and without the upper bound. The initial value of the parameter is 15.

Adaptable parameter in AJA has two optional attributes:

- lower bound,
- upper bound.

### 3.2.1.2.1 Java+ Constructs

The following Java+ constructs are used with adaptable parameters:

- `$GET_BEL(belName)` – returns the current value of the adaptable parameter *belName*.
- `$AP_BAD(belName)` – negative reinforcement for the current value of the adaptable parameter *belName*.
- `$AP_HIGHER(belName)` – negative reinforcement for the current value of the adaptable parameter *belName*. The value should be higher.
- `$AP_LOWER(belName)` – negative reinforcement for the current value of the adaptable parameter *belName*. The value should be lower.
- `$SET_BEL(belName, newValue)` – sets a new current value of the adaptable parameter *belName*. This construct is used only in case when the optimal value of the parameter has become known.
- `$AP_TO_FILE(belName, fileName)` – stores adaptable parameter to a file.
- `$AP_FROM_FILE(belName, fileName)` – loads adaptable parameter from a file.

The chapter 5 describes the implementation details of adaptable parameters in AJA.

### 3.2.1.3 Dependant Values

As already mentioned at the beginning of this chapter, agent-programming languages and tools could facilitate the use of AI by providing AI constructs at the agent-language level. AJA provides such a construct, namely dependant values – the third type of AJA beliefs.

Dependant value in AJA is implemented using artificial neural network (ANN). A programmer uses dependant values in AJA program without the need to understand how exactly an ANN learns and computes the output value from its input values.

The dependant values (i.e. ANNs) can be with no doubt very useful in programming of intelligent agents. In the example MAS, which is implemented in AJA in the practical part of this thesis, one dependant value belief is used. It computes the expected duration of consultation with students at the university depending on the number of appointed students and the number of remaining days before the next exams.

Dependant values are declared as shown in the following listing.

```
BELIEFS
```

```
***
```

```
consultationDuration :
    DEPENDS_ON
        numOfStudents MIN << 1 >> MAX << 30 >>,
        daysBefore MIN << 0 >> MAX << 50 >>
        MIN_VAL << 1 >>
        MAX_VAL << 240 >>
        EXAMPLES_FILE "nnsamples.txt"
        HIDDEN_LAYERS 5;
```

```
***
```

There are two input values for this dependant belief:

- **numOfStudents** with the lower bound 1 and the upper bound 30,
- **daysBefore** with the lower bound 0 and the upper bound 50.

The minimal output value is 1 and the maximal is 240. The examples for the ANN are in the file **nnsamples.txt**. The ANN has one hidden layer with five nodes.

### 3.2.1.3.1 Java+ Constructs

The following Java+ constructs are used with dependant values:

- `$GET_BEL(belName(param1, ...))` – fires the ANN with given input values and returns the output value.
- `$DV_OFFLINE_TRAINING(belName, maxCycles, maxAverageError)` – performs the supervised off-line learning of the ANN with the given stopping conditions.
- `$DV_SHOULD_BE(belName(param1, ...), value)` – supervised on-line learning. The *value* is the correct output for the given input values (*param1*, ...).
- `$DV_TO_FILE(belName, fileName)` – stores dependant value to a file.
- `$DV_FROM_FILE(belName, fileName)` – loads dependant value from a file.

The implementation details of dependant values in AJA are described in chapter 5.

## 3.2.2 Actions

An AJA action consists of a block of Java+ code, return value type, and parameters. AJA actions are similar to Java methods. However, there is also a big difference. In the declaration of AJA action it can also be specified which actions are not compatible with the action being declared. The AJA run-time system blocks the action execution until any of the incompatible actions is executed.

The listing below declares an AJA action.

```
ACTION void eventAlertAct()
    <<
```



```

AlertDialog ad =
    new AlertDialog($AG_JFRAME, $GET_BEL(engToAlert));
ad.show();
if (ad.earlier()){
    $AP_HIGHER(eventAlertTime);
}
else if (ad.later()){
    $AP_LOWER(eventAlertTime);
}

```

The action `eventAlertAct` does not have parameters and does not return a value. It pop-ups a dialog window that alerts user about the next incoming engagement in his/her calendar. The class `AlertDialog` is an ordinary Java class that extends `javax.swing.JDialog`. `$AG_JFRAME` is the reference of the agent window, which is a subclass of `javax.swing.JFrame`. If user gives negative reinforcement regarding the event alerting time, then the corresponding adaptable parameter will receive it. The action `eventAlertAct` has no incompatible actions.

### 3.2.2.1 Java+ Constructs

AJA action can be started using one of the following three Java+ constructs:

- `$EXEC(actionName, par1, par2, ...)` – executes the action `actionName` with the given arguments in current thread of execution. The action execution starts immediately after all, if any, incompatible actions finish their executions.
- `$EXEC_PARALLEL(actionName, par1, par2, ...)` – executes the action `actionName` with the given arguments in new thread of execution. The action execution starts immediately after all, if any, incompatible actions finish their executions.
- `$EXEC_AT(datetime, actionName, par1, par2, ...)` – executes the action `actionName` with the given arguments. The action execution starts at the specified time-point, immediately after, if any, incompatible actions finish their executions.

Furthermore, an AJA action can also be started as a result of reflex triggering.

In addition, there are two Java+ constructs that are also used with actions:

- `$IS_EXECUTING_ACTION(actionName)` – returns `true` if at least one instance of the action `actionName` is currently being executed, `false` otherwise.
- `$IS_WAITING_ACTION(actionName)` – returns `true` if the specified action is waiting to be executed, `false` otherwise. An action waits to be executed if the incompatible actions are being executed.

### 3.2.3 Reflexes

AJA reflex consists of two parts. The first part is an activation condition, whose value determines whether the second part of the reflex should be executed or not.

The second part is a sequence of action invocations. Reflex activation condition is checked periodically using a specified time period.

The listing below declares a reflex with only one action invocation in the EXEC block.

```
REFLEX eventAlertReflex
  CHECKING_PERIOD 10000 // every ten seconds
  CONDITION
  <<
    TimeTable tt = $GET_BEL(timeTable);
    Engagement nextEng = tt.nextEngagement();
    if (nextEng == null){
      return false; // no engagements to alert
    }
    else{
      Engagement last = $GET_BEL(engToAlert);
      if (nextEng.equals(last)){
        return false; //the user has been alerted already
      }
      else{
        long curTime = System.currentTimeMillis();
        long minToStart =
          (nextEng.getStart().getTime() - curTime) / 60000;
        if (minToStart > $GET_BEL(eventAlertTime)){
          return false; // it's too early
        }
        else{
          $SET_BEL(engToAlert, nextEng);
          return true;
        }
      }
    }
  >>
EXEC
  eventAlertAct();
```

The reflex `eventAlertReflex` alerts its user before the start of the next engagement in his/her timetable. The activation condition of this reflex is checked every ten seconds. If the timetable is empty, i.e. there is no next engagement, then the activation condition is **false**. Otherwise, if there is a next engagement, but the user has already been alerted to this engagement, then the activation condition is also **false**. Otherwise, if the time remaining to the start of the next engagement is greater than the value of the adaptable parameter `eventAlertTime`, then the return value is **false**. If not, then the next engagement is set as a value of the belief `engToAlert` and **true** is returned. The action `eventAlertAct`, which is called in the EXEC part of the reflex, is described in the section 3.2.2.

Each reflex executes in separate thread. If there are more than one reflexes being executed at the same time, then they execute concurrently in parallel threads of execution.

### 3.2.3.1 Java+ Constructs

There is only one Java+ construct used with reflexes:

- `$TRIGGER_REFLEXES` – fires all reflexes immediately, ignoring their activation checking period.

### 3.2.4 Negotiations

Requesting and responding negotiations are the construct an agent uses in order to communicate with other agents. If an agent initiates the communication with other agent(s), it does it with a requesting negotiation. In the opposite situation, when other agent has initiated the communication, the agent uses a responding negotiation to respond.

AJA negotiations are executed concurrently in separate concurrent threads. There can also be more than one instances of the same negotiation executed at the same time.

Both types of negotiations are represented as automata and usually consist of several message sending and message receiving between two agents or among more agents. Each message contains a speech act string and optionally an array of serializable Java objects.

Due to the different ways the requesting and the responding negotiations are started, they have slightly different syntax.

Each requesting negotiation consists of its name, return type, parameters, initialization part, and negotiation states. The first state in the negotiation is always called `START`. The negotiation ends, when a final state is reached. A final state has the keyword `FINAL` before the state name.

The following example shows a skeleton of one requesting negotiation. This negotiation is used in the MAS containing PDA agents for joint engagement scheduling.

```
REQUESTING_NEGOTIATION void EngagementInitReqNeg (
    Interval[] possibleStarts,
    int expectedDuration,
    String subject,
    String comment,
    Vector personsToInvite,
    int priority)

NEG_INIT // initialization part of requesting negotiation
<<
    // initialization of local variables
>>

START: // the first state in the negotiation automaton
<<
    // sending the first message to the agent
    // of every person in personsToInvite parameter

    // The message contains the speech act
    // "new engagement request" and all data about the potential
    // engagement, including the possible starting times.

    // If all agents has answered in one minute, then
    // the next state is the state DETERMINE_ENG_START.
    // Otherwise the next state is ERROR.
>>

DETERMINE_ENG_START:
<<
    // The answers obtained from other agents are analyzed.
```

```

// If not all the answers contain the speech act:
//      "here are my intervals"
// then the next state is the state NO_INTERSECTION.

// Otherwise the answers received contain the
// time intervals when the engagement could be started.
// The intersection of the suggested starting times
// is computed.
// If there is no intersection, then the next state
// is the state NO_INTERSECTION.

// Otherwise, the earliest starting time in the
// intersection is selected.

// The agent tries to reserve the engagement time in
// the timetable of its user. Due to concurrent access to
// the timeTable belief from other negotiations, reflexes,
// and actions, it is possible that the initially
// free time for the engagement is now occupied.
// If the time is occupied, the next state is
// the state NO_INTERSECTION.
// Otherwise the time for the engagement is reserved in
// the timetable and the
// next state is the state SEND_ENG_START.

// In case of any failure in the agent to
// agent communication an exception will be thrown
// and caught in this state. If this happens, the next
// state will be the state ERROR.
>>

NO_INTERSECTION:
//appropriate time for the engagement cannot be found
<<
// A message "abandon" is sent to all participating
// agents.

// The next state is the state REPORT_FAILURE
>>

SEND_ENG_START:
<<
// The selected starting time of the engagement
// is sent to all participating agents.

// Despite the fact that the starting time belongs
// to the intersection of the individually proposed
// starting times,
// it can happen that in the meantime one or more
// agents have reserved this time for other engagements.
// Because of this, each agent has to approve
// the starting time.

// If all agents have successfully reserved
// the proposed time for the engagement, then they
// reply with the "ok" message and the next
// state is the state CONFIRM_ENGAGEMENT.

// If there is an agent that has not approved the starting
// time, because this time had been occupied by another
// engagement in the meantime, then the next state is
// the state REPEAT_ALL.

// In case of any failure in the agent to
// agent communication an exception will be thrown
// and caught in this state. If this happens, the next
// state will be the state ERROR.

```

```

>>
REPEAT_ALL:
<<
    // The procedure has to be repeated with the new
    // starting time.

    // The time interval reserved in the timetable
    // for the engagement is released.

    // A message is sent to every participating agent
    // with the speech act "let's try again".

    // The agents reply with the new starting times proposals.

    // If all agents have answered in one minute, then
    // the next state is again the state DETERMINE_ENG_START.
    // Otherwise the next state is ERROR.
>>

```

```

ERROR:
<<
    // A communication error happened.
    // It can be a network error, the remote agent
    // is down or the digital signature is not valid.

    // A message with the speech act "engagement cancelled"
    // is sent to all accessible participating agents.

    // The next state is the state REPORT_FAILURE.
>>

```

```

FINAL REPORT_FAILURE:
<<
    // The user is informed about the failure in
    // the engagement creation.
>>

```

```

FINAL CONFIRM_ENGAGEMENT:
<<
    // All participating agents are informed that
    // the joint engagement has been created.

    // If the communication error happens during
    // the message sending to an agent, then the message sending
    // to this agent will be repeated later until it gets
    // the message. In any case, the engagement is officially
    // created and will not be canceled because of
    // the communication failure in this negotiation state.
>>

```

The requesting negotiation enlisted above is used for the creation of joint engagements (appointments). It returns no value (**void**) and has six parameters:

- The parameter **possibleStarts** specifies the allowed times for the engagement start.
- The parameter **expectedDuration** specifies the engagement duration.
- The parameter **subject** specifies the engagement subject.
- The parameter **comment** contains the comment about the engagement.
- The parameter **personsToInvite** is a **Vector** containing the persons to be invited.
- The parameter **priority** specifies the engagement priority.

The negotiation has the following eight states: **START**, **DETERMINE\_ENG\_START**, **NO\_INTERSECTION**, **SEND\_ENG\_START**, **REPEAT\_ALL**, **ERROR**, **REPORT\_FAILURE**, and

CONFIRM\_ENGAGEMENT. The final states are REPORT\_FAILURE and CONFIRM\_ENGAGEMENT

The Figure 3 shows the states in the automaton of the above requesting negotiation and the possible transitions.

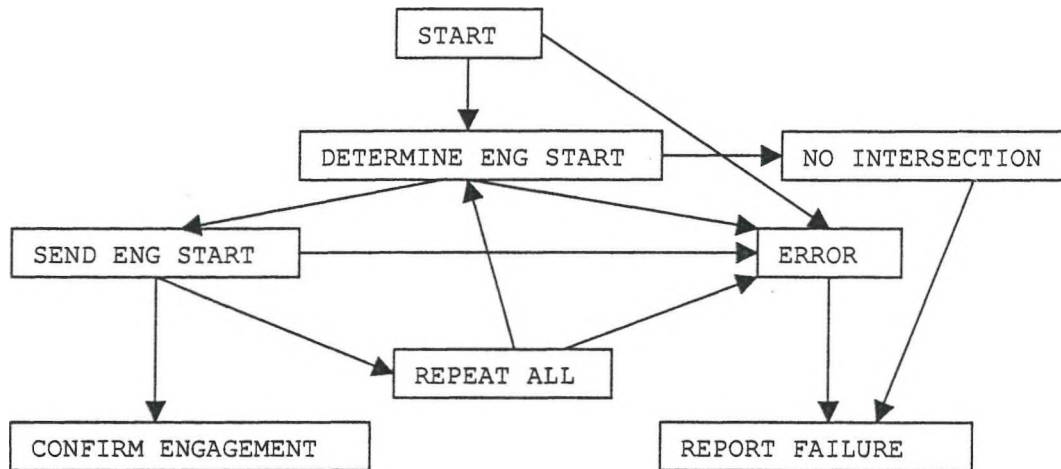


Figure 3 States and possible transitions in the requesting negotiation EngagementInitReqNeg.  
Slika 3 Stanja i moguće tranzicije u zahtevajućem pregovaranju EngagementInitReqNeg.

The complete source code of the **EngagementInitReqNeg** is given in 6.5.1.

The skeleton of the corresponding responding negotiation is given bellow. Responding negotiation has no parameters and no return value. Instead, it has the **ACTIVATION\_CONDITION** part. This part is executed when the first message is received in order to find out which responding negotiation should be started.

```

RESPONDING_NEGOTIATION EngagementInitResNeg

ACTIVATION_CONDITION(MessageData md)
<< //determines if the message received activates this
//responding negotiation
String spAct = md.getSpeechAct();
return spAct.equals("new engagement request");
>>

NEG_INIT
<<
// initialization of local variables
// with the values from the message received
>>

START:
<<
// Determining which of the proposed starting
// times are suitable for the user of this agent.

// If no starting time is suitable, then the message with
// the speech act "no time available" is sent back to the
// requesting agent and the next state is the
// state END_STATE
  
```

```

// Otherwise, the message containing the speech act
// "here are my intervals" and the possible starting
// times is sent to the requesting agent and the next
// state is the state SECOND_MESSAGE.

// If the communication error occurs, then the next state
// is the state ERROR.
>>

SECOND_MESSAGE:
<<
// The next message is received from the requesting agent.
// If the message contains the speech act
// "engagement cancelled" or "abandon" then the next
// state is the state END_STATE.

// Otherwise the message contains the proposed
// starting time of the engagement.
// The agent tries to reserve the time
// for the engagement in its timetable.
// If it succeeds, it sends an "ok" message and the
// next state is the state THIRD_MESSAGE.
// Otherwise it sends the message containing the speech
// act "cannot take part any more" and the next state
// is the state WAIT_REPEAT

// If the communication error occurs, then the next state
// is the state ERROR.
>>

WAIT_REPEAT:
<<
// The agent should receive the message
// with the speech act "let's try again".
// After it receives this message, the
// next state is the state START.

// If the communication error occurs, then the next state
// is the state ERROR.
>>

THIRD_MESSAGE:
<<
// The next message is received from the requesting agent.

// If the speech act is "engagement cancelled", then the
// next state is the state ERROR.

// Otherwise if the speech act is "let's try again",
// then the reserved time for the engagement is
// released and the next state is the state START.

// Otherwise the speech act is "engagement created"
// which means that the engagement has been created.
// The next state in this case is the state END_STATE.

// If the communication error occurs, then the next state
// is the state ERROR.
>>

FINAL_ERROR:
<<
// If the engagement has been registered, then it
// is removed.
>>

FINAL_END_STATE:

```

```

<<
  // An empty state.
>>

```

The responding negotiation enlisted above is used for the creation of joint engagements (appointments). It is activated when the message containing the speech act "new engagement request" is received. The negotiation contains six states: START, SECOND\_MESSAGE, WAIT\_REPEAT, THIRD\_MESSAGE, ERROR, END\_STATE. The states ERROR and END\_STATE are the final states.

The Figure 4 shows the states in the automaton of the above responding negotiation and the possible transitions.

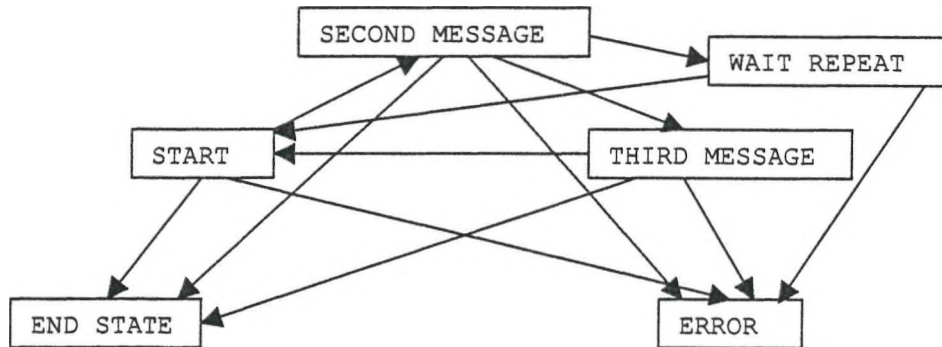


Figure 4 States and possible transitions in the responding negotiation EngagementInitResNeg.  
Slika 4 Stanja i moguće tranzicije u odgovarajućem pregovaranju EngagementInitResNeg.

The complete source code of the **EngagementInitResNeg** is given in 6.5.1.

As the two negotiations given above show, the negotiation construct in AJA enables complex and robust inter-agent communication. Whenever the communication error or an unexpected event happen, the automaton will simply end in an error state, where the cleanup and the release of resources take place.

The negotiations are the only language construct in AJA that can be used for the communication between two agents. Even if the communication is very simple one it still has to be implemented as a negotiation.

### 3.2.4.1 Java+ Constructs

The following Java+ constructs are used for starting requesting negotiations:

- `$NEGOTIATE(negName, par1, par2, ...)` – starts the requesting negotiation *negName* with the real parameters *par1, par2, ...* in the current thread. The value of this expression is the value returned from the negotiation.
- `$NEGOTIATE_PARALLEL(negName, par1, par2, ...)` – starts the requesting negotiation *negName* with the real parameters *par1, par2, ...* in a new thread. This expression has no value. The value returned from the negotiation is ignored.



- `$NEGOTIATE_AT(date, negName, par1, par2, ...)` – starts the requesting negotiation *negName* at the specified time-point *date* with the real parameters *par1, par2, ...* in a new thread. This expression has no value. The value returned from the negotiation is ignored.

The messages can be sent in a negotiation body with the Java+ constructs given below:

- `$SEND_FIRST(toAgentRMIURL, speechAct, obj1, obj2, ...)` – sends the first message to the agent *toAgentRMIURL* containing the speech act *speechAct* and the serializable objects *obj1, obj2, ...*. This is the first message in the communication session between two agents. The expression returns the universally unique session ID, which is used later to send other messages belonging to the same communication session between two agents.
- `$SEND_FIRST_SIGNED(toAgentRMIURL, speechAct, obj1, obj2, ...)` – this construct is similar to the previous one. The only addition is the digital signature of the agent that sends the message. The receiver agent knows who has sent the message, and that the message has not been tampered on the way.
- `$SEND_FIRST_ENCRYPTED(toAgentSSLAddress, speechAct, obj1, obj2, ...)` – this construct is similar to the previous two. The message is however sent using SSL protocol. This is the most secure way of message sending. The receiver agent knows who has sent the message and it can be sure that the message has not been tampered on the way, and that the message content has not be read from someone else.
- `$REPLY(toAgentURL, sessionID, speechAct, obj1, obj2, ...)` – used for sending a reply message in a communication session (i.e. the message that is not the first message in the session).
- `$REPLY_SIGNED(toAgentURL, sessionID, message, obj1, obj2, ...)` – similar to `$REPLY`, but the message is digitally signed. The receiver agent knows who has sent the message, and that the message has not been tampered on the way.
- `$REPLY_ENCRYPTED(toAgentSSLAddress, sessionID, message, obj1, obj2, ...)` – similar to `$REPLY` and `$REPLY_SIGNED` but the message is sent using SSL protocol. This is the most secure way of message sending. The receiver agent knows who has sent the message and it can be sure that the message has not been tampered on the way, and that the message content has not be read from someone else.

A message is received in the negotiation body using one of the following three constructs:

- `$GET_ANSW(sessionID, maxWaitMillis)` – returns the next message received in the communication session or null if the message is not received in the specified time.
- `$GET_ANSW_SIGNED(sessionID, maxWaitMillis)` – similar to `$GET_ANSW`, but the message has to be digitally signed.
- `$GET_ANSW_ENCRYPTED(sessionID, maxWaitMillis)` – similar to `$GET_ANSW` and `$GET_ANSW_SIGNED` but the message has to be sent using SSL protocol.

The following two Java+ expressions are also used with negotiations:

- `$STATE(newState)` – changes the state of the negotiation.
- `$IS_ACTIVE_NEGOTIATION(negName)` – returns true if at least one instance of the specified negotiation is currently active, false otherwise.

### 3.2.5 WWW Negotiation

WWW negotiation is a special type of responding negotiation. WWW negotiation makes an agent accessible via World Wide Web. Each AJA agent contains a simple HTTP server that listens to the port number specified in AJA agent program. When an agent executes its WWW negotiation, it does not communicate with other AJA agents, but with the Internet browser that has sent a HTTP request to the HTTP server component of the agent.

WWW negotiation consists of an initialization part and the negotiation states. The main difference between an ordinary negotiation and a WWW negotiation is that in a WWW negotiation no messages are sent to and received from other agents. Instead, the Java+ constructs are used for the sending HTML forms to browser and for the obtaining the answer from the web user.

The skeleton of one WWW negotiation is given below. This WWW negotiation is a part of the PDA agent belonging to a lecturer at University. The negotiation provides the information about the consultations with students and enables the on-line registrations for the consultations. The web-users are students interested for the consultations with the lecturer.

```
WWW_NEGOTIATION
NEG_INIT
<<
  // obtaining the reference of the timeTable belief
  // initialization of local variables
>>

START:
<<
  // If there are consultations planned then
  //   the next state is the state WHEN
  // else
  //   the next state is the state NO_CONSULTATIONS.
>>
WHEN:
```

```

<<
// Sends a html page with the list of consultation
// dates and the check boxes for the selection
// of the consultation dates.

// If a web user selects the 'cancel' button
// or if he/she does not select any consultation
// date, then the next state is the state CANCEL.

// Otherwise, the next state is the state PURPOSE.
>>
PURPOSE:
<<
// Sends a html page with the form for the entering
// the short description of the consultation purpose.

// A web-user enters the text and clicks one of the
// following three buttons: 'ok', 'back', or 'cancel'

// If the button clicked is the 'ok' button, then
// the next state is the state WHO
// else if the button clicked is the 'back' button, then
// the next state is the state WHEN
// else the button clicked is the 'cancel' button and
// the next state is the state CANCEL.
>>
WHO:
<<
// Sends a html page with the form for the entering
// the student name and ID.

// A web-user enters the text and clicks one of the
// following three buttons: 'ok', 'back', or 'cancel'

// If the button clicked is the 'ok' button, then
// the student is added to the list of students
// for the selected consultations and the next
// state is the final state DONE. However,
// one or more of the selected consultations
// could be deleted from the lecturer's timetable
// in the meantime. In this case, the student
// is not appointed for the consultations and the
// next negotiation state is the state FAILURE.
// Otherwise, if the button clicked is the 'back' button,
// then the next state is the state PURPOSE
// else the button clicked is the 'cancel' button and
// the next state is the state CANCEL.
>>
FAILURE:
<<
// The student is informed that the consultation times
// have been changed in the meantime. Two options are
// available:
// - to try to make the appointment one more time (in
// this case the new state is the state START),
// - to cancel the appointment making (in this case
// the new state is the state CANCEL).
>>
FINAL DONE:
<<
$WWW_DISPLAY_TEXT("OK. Have a nice day.");
>>
FINAL NO_CONSULTATIONS:
<<
$WWW_DISPLAY_TEXT("Sorrv, there will be no consultations "+
"          in the near future. Try later.");
>>
FINAL CANCEL:

```

```

<<
$WWW_DISPLAY_TEXT("Appointment has not been made. Bye.");
>>

```

Students who want to attend the consultations with the lecturer owning the AJA agent use the WWW negotiation enlisted above. WWW negotiation is activated whenever the HTTP request is made. If there are more than one concurrent web users, then each works with its own instance of the WWW negotiation. These instances are executed in separate concurrent threads. The negotiation contains eight states: START, WHEN, PURPOSE, WHO, FAILURE, DONE, NO\_CONSULTATIONS, CANCEL. The states DONE, NO\_CONSULTATIONS, and CANCEL are the final states.

The

Figure 5 shows the states in the automaton of the above WWW negotiation and the possible transitions.

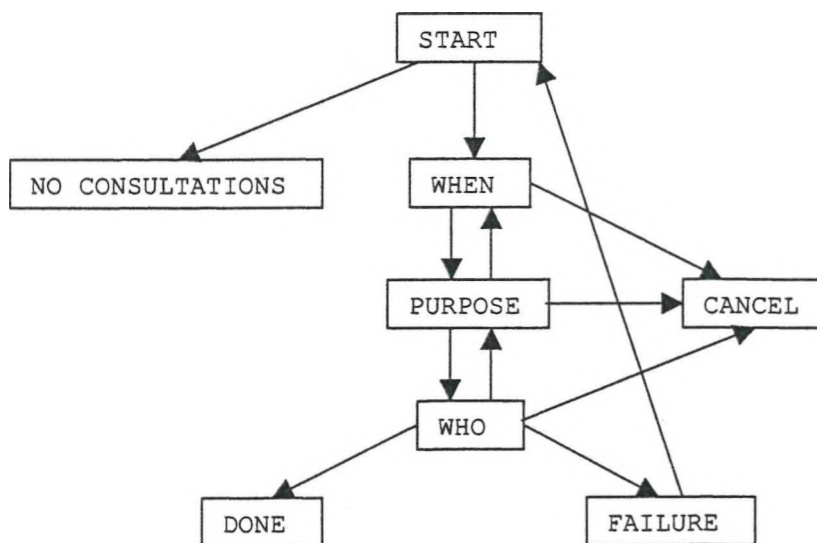


Figure 5 States and possible transitions in the WWW negotiation.  
Slika 5 Stanja i moguće tranzicije u WWW pregovaranju.

The complete source code of the presented WWW negotiation is given in 6.6.

### 3.2.5.1 Java+ Constructs

The Java+ constructs given below can be used inside a WWW negotiation body in order to send the HTML content to browser and to obtain the user input:

- `$WWW_DISPLAY_TEXT(text)` – sends a simple HTML page with the given text only.
- `$WWW_DISPLAY_TEXT(text, bNames)` – sends a HTML page with the text and buttons and returns a button selection.
- `$WWW_GET_LONG_TEXT(desc, bNames, initText)` – obtains a long text input and a button selection from a web user.

- `$WWW_GET_ONE_LINE_TEXT(desc, bNames, initText)` – obtains a one-line text input and a button selection from a web user
- `$WWW_GET_COMBO(desc, items, bNames, selected)` – obtains a combo box selection and a button selection from a web user.
- `$WWW_GET_LIST_SINGLE(desc, items, bNames, selected)` – obtains a single list selection and a button selection from a web user.
- `$WWW_GET_LIST_MULTIPLE(desc, items, bNames, selected)` – obtains a multiple list selection and a button selection from a web user.
- `$WWW_GET_CHECK_BOXES(desc, items, bNames, selected)` – obtains a check box selection and a button selection from a web user.
- `$WWW_GET_RADIO(desc, items, bNames, selected)` – obtains a radio button selection and a button selection from a web user.

Due to *client request – server response* nature of the HTTP communication protocol, there are few restrictions on the use of the above Java+ constructs:

- a) `$WWW_DISPLAY_TEXT(text)` has to be used at the end of the WWW communication and at the end only, because it sends a simple HTML text without HTML form to a browser. The web user has no possibility to get the next page in the same WWW negotiation instance.
- b) Each `$WWW` construct except `$WWW_DISPLAY_TEXT(text)` represents the following activities. First, a HTML page with an input form is sent to browser. The WWW negotiation is blocked and waits. After some time, the web user fills in the form and clicks a button. Then the new HTTP request is received containing the user input. In addition, the HTTP server inside AJA agents is multi-threaded. There can be many HTTP request received at the same time belonging to as many active WWW negotiation instances. Due to implementation issues, the `$WWW` constructs cannot be used inside any Java statement (e.g. inside a block statement `{ ... }`). `$WWW` constructs have to be used at the top level in the WWW negotiation states.
- c) AJA agent programmer has to be aware, that WWW negotiation does not always execute as originally planned, because a web user can use "Back" and "Forward" buttons of its Internet browser.

### 3.2.6 Initialization

The last part of an agent program in AJA is the initialization block. This is a place where e.g. beliefs are initialized and the GUI actions that communicate with user are started. The initialization part of an AJA program is executed immediately after the agent program has been started.

The initialization block starts with the keyword `INITIALIZATION` followed by the Java+ code that initializes the agent program

```
INITIALIZATION
<<
  //initialization code
>>
```

### 3.2.7 GUI

Because of the fact that AJA programs are translated into Java, the GUI in AJA is also implemented in Java. Java Swing components are used for the implementation of AJA GUI.

AJA programmers have two possibilities:

- a) to use the built-in GUI support in AJA, i.e. the Java+ constructs for the GUI communication with the user, or
- b) to implement GUI alone from the scratch using the Java Swing.

If the built-in AJA GUI is used, then the main agent window looks like the one on the Figure 6.

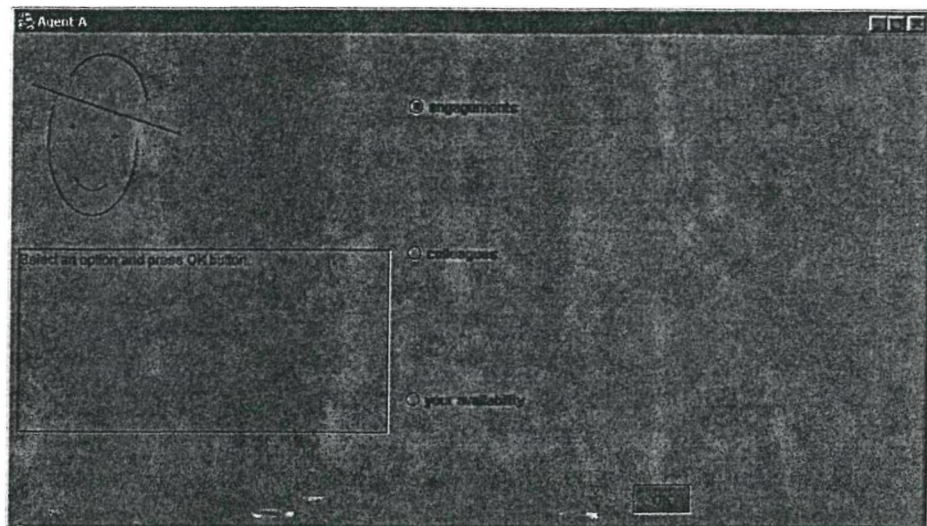


Figure 6 The appearance of the main window of an AJA agent.  
Slika 6 Izgled glavnog prozora AJA agenta.

#### 3.2.7.1 Java+ Constructs

The Java+ constructs implementing built-in GUI support in AJA are the following:

- `$REMOVE_TEXT` – removes all input-output components from agent window.
- `$CLEAR_STATUS_BAR` – clears the status bar of the agent window.
- `$WRITE_STATUS_BAR(text)` – writes text in the status bar of agent window.

- `$DISPLAY_TEXT(text)` – displays text in the agent window.
- `$DISPLAY_TEXT(text, bNames)` – displays text with buttons and obtains a button selection.
- `$GET_LONG_TEXT(desc, bNames, initText)` – obtains a long text input and a button selection.
- `$GET_ONE_LINE_TEXT(desc, bNames, initText)` – obtains a one-line text input and a button selection.
- `$GET_COMBO(desc, items, bNames, selected)` – obtains a combo box selection and a button selection.
- `$GET_LIST_SINGLE(desc, items, bNames, selected)` – obtains a single list selection and a button selection.
- `$GET_LIST_MULTIPLE(desc, items, bNames, selected)` – obtains a multiple list selection and a button selection.
- `$GET_CHECK_BOXES(desc, items, bNames, selected)` – obtains a check box selection and a button selection.
- `$GET_RADIO(desc, items, bNames, selected)` – obtains a radio button selection and a button selection.

If a programmer would prefer to implement the custom agent appearance and GUI, then he/she can obtain the reference of the main agent window with the Java+ construct:

- `$AG_JFRAME` – the value of this expression is the instance of `javax.swing.JFrame` representing the main agent window.

## 4 HADL and Java+

HADL Grammar .....	71
Java+ .....	80



The agent-building tool AJA combines two programming languages for agent programming. Higher-level agent features are specified in HADL (Higher Agent Definition Language). For the implementation of this features an extension of the programming language Java is used, named Java+.

The previous chapter contains several examples of HADL and Java+ code. This chapter presents the complete syntax of both languages and informally describes their semantics.

## 4.1 HADL Grammar

HADL grammar is a context-free grammar. It is enlisted here in this section using an EBNF-like language. The terminal symbols of the grammar are written in upper case letters. The nonterminal symbols are written in lower case letters enclosed with angle brackets.

### 4.1.1 Agent Program

The starting production rule of the grammar is the following one:

```
<program> =
  AGENT <name>
  LOCATED_ON <location> [RMI <port>] [HTTP <port>] [SSL <port>]
  [ PICTURE <fileName> ]
  KEYSTORE <fileName>
  KEY_PAIR ALIAS <alias>
  MAX_REPLY_TIME <hoursToWait>
  [ <javaImport> ]
  [ <beliefsDeclaration> ]
  [ <actionsDeclaration> ]
  [ <requestingNegotiationsDeclaration> ]
  [ <respondingNegotiationsDeclaration> ]
  [ <wwwNegotiationDeclaration> ]
  [ <reflexesDeclaration> ]
  [ <initialization> ]
  END
```

where:

<name> = identifier

<location> = string

<port> = positive integer number

<fileName> = string

<alias> = string

<hoursToWait> = positive integer number

An AJA agent program starts with the keyword **AGENT** followed by the agent name. The next mandatory word is the keyword **LOCATED\_ON** followed by the host name (or IP address) of the computer where the agent is located. Optionally, the keyword **RMI** can follow with the port number of the **rmiregistry** and/or the keyword **HTTP** with the port number for accessing the WWW negotiation via Internet browser

and/or the keyword **SSL** with the port number for the secure agent-to-agent communication.

If the RMI port is not specified, then the default port 1099 is used. If the HTTP port is not specified, then the default port 1971 is used. Similarly, if the SSL port is not specified, the default port 3456 is used.

The next optional part in the agent program is the keyword **PICTURE** and the name of the file containing the agent picture. The agent picture is shown in the upper left part of the default agent window. If the picture file name is omitted, then the default picture is used (see the Figure 6 on the page 68).

The next two obligatory parts specify the location of the keystore file and the keystore alias of the key-pair belonging to the agent. The keystore and the key-pair are used for the digital signing of the messages sent by the agent and for the verifying of received digital signatures.

Next, the keyword **MAX\_REPLY\_TIME** is required and the number of hours the agent program waits for the reply of the remote agent in communication session. This parameter is necessary for the deallocation of resources used by uncleanly finished communication sessions (e.g. the remote agent has been stopped in the middle of a negotiation).

The first few lines of an AJA agent program look like the ones on the example below:

```
AGENT Maxim LOCATED_ON "stribog.im.ns.ac.yu" RMI 1099 HTTP 2100
PICTURE "C:\\pictures\\Max.jpg"
KEYSTORE "..\\Max.ks"
KEY_PAIR_ALIAS "Max_key_pair"
MAX_REPLY_TIME 1 // wait maximal one hour for the
                // response of other agent
...

```

The remaining parts of AJA agent program are discussed in the following subsections.

### 4.1.2 Import

AJA program usually contains a lot of Java+ code. The Java+ code uses the Java classes and interfaces stored in various Java packages. In order to avoid writing the inconveniently long qualified names of this data types, they can be imported and thus the package name can be omitted in the source code.

The corresponding HADL syntax grammar rule is the following one:

```
<javaImport> = '<<' <JAVA+> '>>'
```

where <JAVA+> means the Java+ code, which in this case contains the Java import.

Example:

```
<<
import demo.*;
import java.io.*;
```

```

import java.text.*;
import java.util.Date;
import java.util.Vector;
import javax.swing.JOptionPane;
>>

```

### 4.1.3 Declaration of Beliefs

```

<beliefsDeclaration> = BELIEFS <belDecl> ';' { <belDecl> ';' }

```

The declaration of beliefs starts with the keyword **BELIEFS**. Each belief declaration is followed by a semicolon.

The belief declaration production rule is the following one:

```

<belDecl> =
  <belName> ':' (
    <JAVAtype> [= <initValue> ]      |
    ADAPTABLE [LBOUND <lowerBound>]
              [UBOUND <upperBound>]
              '=' <apInitValue>     |
    DEPENDS_ON <neurParamList>
              [MIN_VAL <lowerBound>]
              [MAX_VAL <upperBound>]
              EXAMPLES_FILE <fileName>
              [ <netConf> ]
    )

```

where:

```

<belName> = identifier

```

```

<initValue> = <expressionJAVA+>

```

```

<lowerBound> = <expressionJAVA+>

```

```

<upperBound> = <expressionJAVA+>

```

```

<apInitValue> = <expressionJAVA+>

```

```

<neurParamList> = <paramName>
                  [MIN <lowerBound>]
                  [MAX <upperBound>]
                  (',' <paramName>
                  [MIN <lowerBound>]
                  [MAX <upperBound>] ]

```

```

<netConf> = HIDDEN_LAYERS <nodesNum> [',' <nodesNum>]

```

```

<nodesNum> = non-negative integer number

```

As described in 3.2.1, there are three types of beliefs in AJA.

A belief of the first type, a Java value, is declared using the first branch in the belief declaration production rule.

A belief of the second type, an adaptable parameter, is declared using the second branch in the belief production rule.

Finally, a dependant value, the third belief type, is declared using the third branch in the belief production rule.

The section 3.2.1 contains examples of belief declarations for each belief type.

Adaptable parameters and dependant values are described in details in chapter 5.

#### 4.1.4 Declaration of Actions

```
<actionsDeclaration> = <actionDecl> { <actionDecl> }
```

The action declaration part of an AJA program consists of one or more action declarations. The action declaration syntax is specified by the following production rule:

```
<actionDecl> =  
  ACTION ( <returnType> | 'void' ) <actionName>  
          '(' [ <formalParams> ] )'  
          [ INCOMPATIBLE <actionName> { ',' <actionName> } ]  
          <body>
```

where:

```
<returnType> = <JAVAtype>  
<actionName> = identifier  
<formalParams> = <formalPar> { ',' <formalPar> }  
<formalPar> = <JAVAtype> <paramName>  
<paramName> = identifier  
<body> = '<<' <JAVA+> '>>'
```

If an action has a return type, then the Java+ code in the action body has to return a value of the specified type.

The action names following the keyword **INCOMPATIBLE** denote the actions whose executions block the execution of the action being defined.

The name of the action being defined can also be enlisted in its incompatible actions list. In such a case there can be at most one instance of this action being executed.

An example of the action declaration can be found in the section 3.2.2.

## 4.1.5 Declaration of Requesting Negotiations

The next part of an AJA agent program is the declaration of requesting negotiations.

```
<requestingNegotiationsDeclaration> = <reqNegDecl> { <reqNegDecl> }
```

Each requesting negotiation complies with the following production rule:

```
<reqNegDecl> =  
  REQUESTING_NEGOTIATION ( <returnType> | 'void' )  
  <negotiationName> '(' [ <formalParams> ] )'  
  <negBody>
```

where:

```
<negotiationName> = identifier  
  
<negBody> = <initPart> <startState> { <state> }  
  
<initPart> = NEG_INIT '<<' <JAVA+> '>>'  
  
<startState> = START ':' '<<' <JAVA+> '>>'  
  
<state> = [ FINAL ] <stateName> ':' '<<' <JAVA+> '>>'  
  
<stateName> = identifier
```

Similar to methods in Java, a requesting negotiation can return a value or it can be 'void'. Each requesting negotiation has its name and optionally a list of formal parameters.

The negotiation body starts with the initial part. The initial part is executed first and is a good place to declare and to initialize the local variables, which are visible in the remaining part of the negotiation body. After the initial negotiation part follows the first negotiation state. The first state is always the state with the name **START**. This is the starting state in the negotiation. The remaining negotiation states follow after the starting state. If a state is declared as **FINAL**, then the negotiation process ends after executing the state.

An example of the requesting negotiation declaration is given in the section 6.5.1.

## 4.1.6 Declaration of Responding Negotiations

```
<respondingNegotiationsDeclaration> = <respNegDecl> {<respNegDecl>}
```

A responding negotiation is declared using the following production rule:

```
<respNegDecl> =  
  RESPONDING_NEGOTIATION <negotiationName>  
  ACTIVATION_CONDITION '(' 'MessageData' identifier )'  
  << <JAVA+> >>  
  <negBody>
```

Responding negotiation cannot be invoked directly like a requesting negotiation. Instead, it is invoked when the agent receives a message that does not belong to any active communication session and the message satisfies the activation condition of the responding negotiation. Consequently, responding negotiations do not have parameters. They have however the activation condition part, which is similar to a method in Java returning `boolean` value. The activation condition Java+ block has a reference of the `aja.framework.MessageData` instance containing the message and the message description.

When a message is received that does not belong to any active communication session, then the activation condition of the first declared responding negotiation is invoked. If it returns the value `true`, then the first declared responding negotiation is invoked. Otherwise, the second declared responding negotiation is examined etc. If the activation conditions of all responding negotiations return the value `false`, then the message received is ignored.

The negotiation body of a responding negotiation is similar to the negotiation body of a requesting negotiation. The only difference is that the return statement is not allowed in responding negotiation.

An example of the responding negotiation declaration is given in the section 6.5.1.

#### 4.1.7 Declaration of WWW Negotiation

AJA agent program can contain one WWW negotiation.

```
<wwwNegotiationDeclaration> = WWW_NEGOTIATION <negBody>
```

WWW negotiation can be seen as a special responding negotiation. WWW negotiation has not the activation condition part, because it is invoked whenever the first HTTP request from an Internet browser is received.

The next difference between an ordinary negotiation and a WWW negotiation is that in a WWW negotiation no messages are sent to and received from other agents. Instead, the Java+ constructs are used for the sending HTML forms to browser and for the obtaining the answer from the web user.

The section 6.6 contains an example of the WWW negotiation declaration.

#### 4.1.8 Declaration of Reflexes

```
<reflexesDeclaration> = <reflDecl> { <reflDecl> }
```

A reflex declaration follows the rule:

```
<reflDecl> =  
  REFLEX <reflName>  
  CHECKING PERIOD <numOfMs>  
  CONDITION '<<' <JAVA+> '>>'  
  EXEC <actionCall> ';' {<actionCall> ';' }
```

where:

```

<reflName> = identifier

<numOfMs> = non-negative integer number

<actionCall> = <actionName> '(' [ <realParams> ] ') '

<realParams> = <realPar> {',' <realPar>}

<realPar> = <expression>

<expression> =
  <belName> |
  <actionCall> |
  <reqNegCall> |
  <expressionJAVA+>

<reqNegCall> = <reqNegName> '(' [ <realParams> ] ') '

<expressionJAVA+> = '<<' expression in JAVA+ '>>'

```

AJA reflex has its name, a condition checking period, activation condition, and a sequence of action calls.

A Java+ block in the reflex condition part returns a boolean Java value. The activation condition is checked periodically using the specified checking period. If the condition returns the value **true**, then the sequence of action calls is executed.

The Java+ construct \$TRIGGER\_REFLEXES triggers all reflexes ignoring their condition checking periods. All activation conditions are evaluated and where the value true is returned, the action call sequence is executed.

Examples of the reflex declaration are given in the section 6.4.

#### 4.1.9 Initialization

```

<initialization> = INITIALIZATION '<<' <JAVA+> '>>'

```

The initialization part of an AJA agent program appears at the end of the program, after all agent components are declared. It consists of the keyword **INITIALIZATION** and a Java+ block.

The initialization block usually initializes agent beliefs and starts a GUI action. See the example in the section 6.7.

#### 4.1.10 All Grammar Production Rules

The grammar production rules of HADL given above are enlisted together in this section:

```

<program> =
  AGENT <name>
  LOCATED_ON <location> [RMI <port>] [HTTP <port>] [SSL <port>]

```

```

[ PICTURE <fileName> ]
KEYSTORE <fileName>
KEY_PAIR ALIAS <alias>
MAX_REPLY_TIME <hoursToWait>
[ <javaImport> ]
[ <beliefsDeclaration> ]
[ <actionsDeclaration> ]
[ <requestingNegotiationsDeclaration> ]
[ <respondingNegotiationsDeclaration> ]
[ <wwwNegotiationDeclaration> ]
[ <reflexesDeclaration> ]
[ <initialization> ]
END

<name> = identifier

<location> = string

<port> = positive integer number

<fileName> = string

<alias> = string

<hoursToWait> = positive integer number

<javaImport> = '<<' <JAVA+> '>>''

<beliefsDeclaration> = BELIEFS <belDecl> ';' { <belDecl> ';' }

<belDecl> =
  <belName> ':' (
    <JAVAType> [= <initValue> ] |
    ADAPTABLE [LBOUND <lowerBound>]
              [UBOUND <upperBound>]
              '=' <apInitValue> |
    DEPENDS_ON <neurParamList>
              [MIN_VAL <lowerBound>]
              [MAX_VAL <upperBound>]
              EXAMPLES_FILE <fileName>
              [ <netConf> ]
  )

<belName> = identifier

<initValue> = <expressionJAVA+>

<lowerBound> = <expressionJAVA+>

<upperBound> = <expressionJAVA+>

<apInitValue> = <expressionJAVA+>

<neurParamList> = <paramName>
                  [MIN <lowerBound>]
                  [MAX <upperBound>]
                  {',' <paramName>
                  [MIN <lowerBound>]
                  [MAX <upperBound>] }

<netConf> = HIDDEN_LAYERS <nodesNum> {',' <nodesNum>}

<nodesNum> = non-negative integer number

<actionsDeclaration> = <actionDecl> { <actionDecl> }

```



```

<actionDecl> =
    ACTION ( <returnType> | 'void' ) <actionName>
        '(' [ <formalParams> ] ')'
        [ INCOMPATIBLE <actionName> { ',' <actionName> } ]
        <body>

<returnType> = <JAVAType>

<actionName> = identifier

<formalParams> = <formalPar> { ',' <formalPar> }

<formalPar> = <JAVAType> <paramName>

<paramName> = identifier

<body> = '<<' <JAVA+> '>>'

<requestingNegotiationsDeclaration> = <reqNegDecl> { <reqNegDecl> }

<reqNegDecl> =
    REQUESTING_NEGOTIATION ( <returnType> | 'void' ) <negotiationName> '(' [
<formalParams> ] ')'
    <negBody>

<negotiationName> = identifier

<negBody> = <initPart> <startState> { <state> }

<initPart> = NEG_INIT '<<' <JAVA+> '>>'

<startState> = START ':' '<<' <JAVA+> '>>'

<state> = [ FINAL ] <stateName> ':' '<<' <JAVA+> '>>'

<stateName> = identifier

<respondingNegotiationsDeclaration> = <respNegDecl> { <respNegDecl> }

<respNegDecl> =
    RESPONDING_NEGOTIATION <negotiationName>
    ACTIVATION_CONDITION '(' 'MessageData' identifier ')'
    << <JAVA+> >>
    <negBody>

<wwwNegotiationDeclaration> = WWW_NEGOTIATION <negBody>

<reflexesDeclaration> = <reflDecl> { <reflDecl> }

<reflDecl> =
    REFLEX <reflName>
    CHECKING_PERIOD <numOfMs>
    CONDITION '<<' <JAVA+> '>>'
    EXEC <actionCall> ';' { <actionCall> ';' }

<reflName> = identifier

<numOfMs> = non-negative integer number

<actionCall> = <actionName> '(' [ <realParams> ] ')'

<realParams> = <realPar> { ',' <realPar> }

<realPar> = <expression>

<expression> =
    <belName> |
    <actionCall> |

```

```

    <reqNegCall> |
    <expressionJAVA+>

<reqNegCall> = <reqNegName> '(' [ <realParams> ] ') '
<expressionJAVA+> = '<<' expression in JAVA+ '>>'
<initialization> = INITIALIZATION '<<' <JAVA+> '>>'

```

## 4.2 Java+

The first part of this chapter presents the language HADL, which is used for the specification of the main agent parts. As can be seen above, many of HADL grammar production rules ends up with the nonterminal symbol <JAVA+>. This symbols stands for Java+ code.

Java+ code is similar to normal Java code, but in addition to Java, the special constructs starting with the symbol \$ also belong to Java+. Many of these constructs are already introduced in the chapter 3. This section documents systematically all Java+ constructs that do not belong to standard Java.

### 4.2.1 Java+ Constants

#### **\$AG\_NAME**

type: java.lang.String  
value: Name of the agent. For example: "Max".

#### **\$AG\_HOST**

type: java.lang.String  
value: Hostname of the computer where the agent is located.  
For example: "perun.im.ns.ac.yu".

#### **\$AG\_RMI\_PORT**

type: int  
value: Port on which agent listens for RMI requests.  
For example: 1099.

#### **\$AG\_HTTP\_PORT**

type: int  
value: Port on which agent listens for HTTP requests.  
For example: 1971.

#### **\$AG\_SSL\_PORT**

type: int  
value: Port on which agent listens for SSL connections.  
For example: 3456.

#### **\$AG\_RMI\_URL**

type: java.lang.String  
value: RMI URL of the agent (e.g. "rmi://perun.im.ns.ac.yu:1099/Max").

#### **\$AG\_HTTP\_URL**

type: java.lang.String  
value: HTTP URL of the agent  
(e.g. "http://perun.im.ns.ac.yu:1971/Max").

**\$AG\_SSL\_ADDRESS**  
type: java.lang.String  
value: SSL address of the agent  
(e.g. "perun.im.ns.ac.yu:3456").

**\$AG\_JFRAME**  
type: javax.swing.JFrame  
value: Agent window.

## 4.2.2 Java+ Constructs for Beliefs

**\$GET\_BEL(*belName*)**  
parameter: *belName* - name of the belief.  
type: Depends on the type of the belief.  
value: Belief value.

**\$GET\_BEL(*belName*(*param1*, ...))**  
parameter: *belName*(*param1*, ...) - name of the dependant belief  
(neural net) and real parameters.  
type: double  
value: Output of the neural net for given parameters.

**\$SET\_BEL(*belName*, *value*)**  
parameter: *belName* - name of the belief.  
parameter: *value* - value of the belief to set,  
type: depends on the belief type.  
type: void

**\$AP\_BAD(*belName*)**  
parameter: *belName* - name of the adaptable belief.  
type: void  
//negative reinforcement for an adaptable belief

**\$AP\_HIGHER(*belName*)**  
parameter: *belName* - name of the adaptable belief.  
type: void  
//negative reinforcement for an adaptable belief;  
//the value should be higher

**\$AP\_LOWER(*belName*)**  
parameter: *belName* - name of the adaptable belief.  
type: void  
//negative reinforcement for an adaptable belief;  
//the value should be lower

**\$AP\_TO\_FILE(*belName*, *fileName*)**  
parameter: *belName* - name of the adaptable belief.  
parameter: *fileName* - name of the file,  
type: java.lang.String.  
type: void  
throws java.io.IOException  
//saves adaptable belief to a file

**\$AP\_FROM\_FILE(*belName*, *fileName*)**

parameter: *belName* - name of the adaptable belief.  
parameter: *fileName* - name of the file,  
          type: *java.lang.String*.  
type: *void*  
throws *java.io.IOException*, *java.lang.ClassNotFoundException*  
//load adaptable belief from a file

***\$DV\_OFFLINE\_TRAINING(belName, maxCycles, maxAverageError)***

parameter: *belName* - name of the dependant belief.  
parameter: *maxCycles* - upper bound for the number of the  
          learning iterations,  
          type: *int*.  
parameter: *maxAverageError* - upper bound for the maximal  
          average error in percents allowed,  
          type: *double*.  
type: *double*  
value: Average error of the neural network after learning,  
      in percents.

***\$DV\_SHOULD\_BE(belName(param1, ...), value)***

parameter: *belName(param1, ...)* - name of the dependant belief  
          (neural net) and real parameters.  
parameter: *value* - the correct value of the network output,  
          type: *double*.  
type: *void*

***\$DV\_TO\_FILE(belName, fileName)***

parameter: *belName* - name of the dependant belief (neural net).  
parameter: *fileName* - name of the file,  
          type: *java.lang.String*.  
type: *void*  
throws *java.io.IOException*  
//saves dependent belief to a file

***\$DV\_FROM\_FILE(belName, fileName)***

parameter: *belName* - name of the dependant belief (neural net).  
parameter: *fileName* - name of the file,  
          type: *java.lang.String*.  
type: *void*  
throws *java.io.IOException*, *java.lang.ClassNotFoundException*  
//loads dependent belief from a file

### 4.2.3 Java+ Constructs for Actions

***\$EXEC(actionName, par1, par2, ...)***

parameter: *actionName* - name of the action to execute,  
          type: *java.lang.String*.  
optional parameters: *par1, par2, ...* - real parameters for  
          the action execution.  
type: depends on the action  
value: Action return value (if the type is not void).  
//Executes the action in current thread.

***\$EXEC\_PARALLEL(actionName, par1, par2, ...)***

parameter: *actionName* - name of the action to execute.  
optional parameters: *par1, par2, ...* - real parameter for  
          the action execution.  
type: *void*  
//Executes the action in a new thread.

**\$EXEC\_AT**(*date*, *actionName*, *par1*, *par2*, ...)  
parameter: *date* - time point when the action execution  
should be started,  
type: java.util.Date.  
parameter: *actionName* - name of the action to execute.  
optional parameters: *par1*, *par2*, ... - real parameter for  
the action execution.  
type: void  
//Executes the action in a new thread at specified time.

**\$IS\_EXECUTING\_ACTION**(*actionName*)  
parameter: *actionName* - name of an action.  
type: boolean.  
value: true if the specified action is being executed,  
false otherwise.

**\$IS\_WAITING\_ACTION**(*actionName*)  
parameter: *actionName* - name of an action.  
type: boolean.  
value: true if the specified action is waiting to be executed,  
false otherwise.

## 4.2.4 Java+ Constructs for Negotiations

**\$NEGOTIATE**(*negName*, *par1*, *par2*, ...)  
parameter: *negName* - name of the requesting negotiation  
to activate.  
optional parameters: *par1*, *par2*, ... - real parameter for  
the negotiation activation.  
type: depends on the requesting negotiation  
value: Requesting negotiation return value  
(if the type is not void)  
//Activates the requesting negotiation in current thread.

**\$NEGOTIATE\_PARALLEL**(*negName*, *par1*, *par2*, ...)  
parameter: *negName* - name of the requesting negotiation  
to activate.  
optional parameters: *par1*, *par2*, ... - real parameter for the  
negotiation activation.  
type: void  
//Activates the requesting negotiation in new thread.

**\$NEGOTIATE\_AT**(*date*, *negName*, *par1*, *par2*, ...)  
parameter: *date* - time point when the requesting  
negotiation activation should be started,  
type: java.util.Date.  
parameter: *negName* - name of the requesting  
negotiation to activate.  
optional parameters: *par1*, *par2*, ... - real parameter for  
the negotiation activation.  
type: void  
//Activates the requesting negotiation in new thread at  
//specified time.

**\$STATE**(*newState*)  
parameter: *newState* - name of the next state in negotiation.  
type: void

**\$IS\_ACTIVE\_NEGOTIATION(*negName*)**

parameter: *negName* - name of a negotiation.  
type: boolean.  
value: true if the specified negotiation is currently active,  
false otherwise.

**\$SEND\_FIRST(*toAgentRMIURL*, *speechAct*, *obj1*, *obj2*, ... )**

parameter: *toAgentRMIURL* - RMI URL of the message receiver agent,  
type: java.lang.String.  
parameter: *speechAct* - speech act of the message,  
type: java.lang.String.  
optional parameters: *obj1*, *obj2*, ... - serializable java objects,  
type: java.lang.Object.  
type: java.lang.String  
value: ID of the communication session  
throws *aja.framework.CommunicationException*  
//Sends the first message in a communication session  
//(without the digital signature).

**\$SEND\_FIRST\_SIGNED(*toAgentRMIURL*, *speechAct*, *obj1*, *obj2*, ... )**

parameter: *toAgentRMIURL* - RMI URL of the message receiver agent,  
type: java.lang.String.  
parameter: *speechAct* - speech act of the message,  
type: java.lang.String.  
optional parameters: *obj1*, *obj2*, ... - serializable java objects,  
type: java.lang.Object.  
type: java.lang.String  
value: ID of the communication session  
throws *aja.framework.CommunicationException*  
//Sends the first message in a communication session  
//with the digital signature.

**\$SEND\_FIRST\_ENCRYPTED(*toAgentSSLAddress*, *speechAct*,  
*obj1*, *obj2*, ... )**

parameter: *toAgentSSLAddress* - SSL address of the agent to whom the  
message is sent  
(e.g. "bambi.im.ns.ac.yu:3456").  
The address consists of a hostname,  
colon, and a SSL port number.  
type: java.lang.String.  
parameter: *speechAct* - speech act of the message,  
type: java.lang.String.  
optional parameters: *obj1*, *obj2*, ... - serializable java objects,  
type: java.lang.Object.  
type: java.lang.String  
value: ID of the communication session  
throws *aja.framework.CommunicationException*  
//Sends the first message in a communication session  
//using SSL protocol.

**\$REPLY(*toAgentURL*, *sessionID*, *speechAct*, *obj1*, *obj2*, ... )**

parameter: *toAgentURL* - RMI URL of the message receiver agent,  
type: java.lang.String.  
parameter: *sessionID* - ID of the communication session,  
type: java.lang.String.  
parameter: *speechAct* - speech act of the message,  
type: java.lang.String.  
optional parameters: *obj1*, *obj2*, ... - serializable java objects,  
type: java.lang.Object.  
type: void  
throws *aja.framework.CommunicationException*  
//Sends a reply message in a communication session  
//(without the digital signature).

**\$REPLY\_SIGNED(toAgentURL, sessionID, speechAct, obj1, obj2, ... )**  
parameter: toAgentURL - RMI URL of the message receiver agent,  
type: java.lang.String.  
parameter: sessionID - ID of the communication session,  
type: java.lang.String.  
parameter: speechAct - speech act of the message,  
type: java.lang.String.  
optional parameters: obj1, obj2, ... - serializable java objects,  
type: java.lang.Object.  
type: void  
throws aja.framework.CommunicationException  
//Sends a reply message in a communication session  
//(with the digital signature).

**\$REPLY\_ENCRYPTED(toAgentSSLAddress, sessionID, speechAct,  
obj1, obj2, ... )**  
parameter: toAgentSSLAddress - SSL address of the agent to whom the  
message is sent (e.g.  
"bambi.im.ns.ac.yu:3456").  
The address consists of a hostname,  
colon, and a SSL port number.  
type: java.lang.String.  
parameter: sessionID - ID of the communication session,  
type: java.lang.String.  
parameter: speechAct - speech act of the message,  
type: java.lang.String.  
optional parameters: obj1, obj2, ... - serializable java objects,  
type: java.lang.Object.  
type: void  
throws aja.framework.CommunicationException  
//Sends a reply message in a communication session  
//using SSL protocol.

**\$GET\_ANSW(sessionID, maxWaitMillis)**  
parameter: sessionID - ID of the communication session,  
type: java.lang.String.  
parameter: maxWaitMillis - milliseconds to wait for the answer.  
If the answer arrives before the specified time  
elapses, the answer is immediately returned and  
the current thread continues with the execution.  
type: long.  
type: aja.framework.MessageData  
value: the answer from the agent to whom a message had been  
previously sent, or null if the  
answer has not arrived in specified time  
throws aja.framework.CommunicationException

**\$GET\_ANSW\_SIGNED(sessionID, maxWaitMillis)**  
parameter: sessionID - ID of the communication session,  
type: java.lang.String.  
parameter: maxWaitMillis - milliseconds to wait for the answer.  
If the digitally signed answer arrives before  
the specified time elapses, the answer is  
immediately returned and the current thread  
continues with the execution.  
type: long.  
type: aja.framework.MessageData  
value: the answer from the agent to whom a message had been  
previously sent, or null if the  
answer has not arrived in specified time  
throws aja.framework.CommunicationException

**\$GET\_ANSW\_ENCRYPTED(sessionID, maxWaitMillis)**

```

parameter: sessionId - ID of the communication session,
           type: java.lang.String.
parameter: maxWaitMillis - milliseconds to wait for the answer.
           If the encrypted answer arrives before
           the specified time elapses, the answer is
           immediately returned and the current thread
           continues with the execution.
           type: long.
type: aja.framework.MessageData
value: the answer from the agent to whom a message had been
       previously sent, or null if the
       answer has not arrived in specified time
throws aja.framework.CommunicationException

```

## 4.2.5 Java+ Construct for Reflexes

```

$TRIGGER_REFLEXES
type: void
// forces all reflexes to fire.

```

## 4.2.6 Java+ Constructs for WWW Negotiation

```

$WWW_DISPLAY_TEXT(text)
parameter: text - text to display on the page,
           type: java.lang.String.
type: void
throws: aja.framework.WebException
//Sends a simple html page with text only.
//After the page sending, the connection
//with browser is closed. This is always the last html page
//sending in a web negotiation.
//This function must not be nested in a Java statement.

```

```

$WWW_DISPLAY_TEXT(text, bNames)
parameter: text - text to display,
           type: java.lang.String.
parameter: bNames - labels for buttons,
           type: java.lang.String[].
type: aja.framework.ButtonSelection
value: user response.
throws: aja.framework.WebException
//Sends a html page with text and buttons.
//Because the user answer is expected and because the answer
//comes as a new http request, this should not be the last html
//page sending in the web negotiation.
//This function must not be nested in a Java statement.

```

```

$WWW_GET_LONG_TEXT(desc, bNames, initText)
parameter: desc - text that tells user what he/she should do,
           type: java.lang.String.
parameter: bNames - labels for buttons,
           type: java.lang.String[].
parameter: initText - initial (default) text,
           type: java.lang.String.
type: aja.framework.TextInput
value: User response.
throws: aja.framework.WebException
//Obtains long text input from web user.
//This function must not be nested in a Java statement.

```



```

$WWW_GET_ONE_LINE_TEXT(desc, bNames, initText)
parameter: desc - text that tells user what he/she should do,
            type: java.lang.String.
parameter: bNames - labels for buttons,
            type: java.lang.String[].
parameter: initText - initial (default) text,
            type: java.lang.String.
type: aja.framework.TextInput
value: User response.
throws: aja.framework.WebException
//Obtains one-line text input from web user.
//This function must not be nested in a Java statement.

```

```

$WWW_GET_COMBO(desc, items, bNames, selected)
parameter: desc - text that tells user what he/she should do,
            type: java.lang.String.
parameter: items - items in the combo box,
            type: java.lang.String[].
parameter: bNames - labels for buttons,
            type: java.lang.String[].
parameter: selected - index of the initially selected
                    combo-box item,
                    type: int.
type: aja.framework.SingleSelection
value: User response.
throws: aja.framework.WebException
//Obtains combo-box selection from web user.
//This function must not be nested in a Java statement.

```

```

$WWW_GET_LIST_SINGLE(desc, items, bNames, selected)
parameter: desc - text that tells user what he/she should do,
            type: java.lang.String.
parameter: items - items in the list,
            type: java.lang.String[].
parameter: bNames - labels for buttons,
            type: java.lang.String[].
parameter: selected - index of the initially selected list item,
            type: int.
type: aja.framework.SingleSelection
value: User response.
throws: aja.framework.WebException
//Obtains single list selection from web user.
//This function must not be nested in a Java statement.

```

```

$WWW_GET_LIST_MULTIPLE(desc, items, bNames, selected)
parameter: desc - text that tells user what he/she should do,
            type: java.lang.String.
parameter: items - items in the list,
            type: java.lang.String[].
parameter: bNames - labels for buttons,
            type: java.lang.String[].
parameter: selected - initial selection,
            type: boolean[].
type: aja.framework.Selection
value: User response.
throws: aja.framework.WebException
//Obtains multiple list selection from web user.
//This function must not be nested in a Java statement.

```

```

$WWW_GET_CHECK_BOXES(desc, items, bNames, selected)
parameter: desc - text that tells user what he/she should do,
            type: java.lang.String.
parameter: items - check-boxes items,

```

```

        type: java.lang.String[].
parameter: bNames - labels for buttons,
        type: java.lang.String[].
parameter: selected - initial selection,
        type: boolean[].
type: aja.framework.MultipleSelection
value: User response.
throws: aja.framework.WebException
//Obtains check-box selection from web user.
//This function must not be nested in a Java statement.

$WWW_GET_RADIO(desc, items, bNames, selected)
parameter: desc - text that tells user what he/she should do,
        type: java.lang.String.
parameter: items - radio-buttons items,
        type: java.lang.String[].
parameter: bNames - labels for buttons,
        type: java.lang.String[].
parameter: selected - index of the initially selected item,
        type: int.
type: aja.framework.SingleSelection
value: User response.
throws: aja.framework.WebException
//Obtains radio-button selection from web user.
//This function must not be nested in a Java statement.

```

## 4.2.7 Java+ Constructs for GUI

```

$REMOVE_TEXT
type: void
//Removes input-output components from agent window.

```

```

$CLEAR_STATUS_BAR
type: void
//Clears the status bar of the agent window.

```

```

$WRITE_STATUS_BAR(text)
parameter: text - text to write in the status bar,
        type: java.lang.String.
type: void
//Writes text in the status bar of agent window.

```

```

$DISPLAY_TEXT(text)
parameter: text - text to display in agent window,
        type: java.lang.String.
type: void
//Displays text in the agent window. Appends text to the
//text on an existing text area, or makes a new text area
//with given text. In both cases the text will be visible
//on agent window. Method does not block the thread that
//has invoked it.

```

```

$DISPLAY_TEXT(text, bNames)
parameter: text - text to display,
        type: java.lang.String.
parameter: bNames - labels for buttons,
        type: java.lang.String [].
type: aja.framework.ButtonSelection
value: User response.
//Displays text with buttons and blocks the thread until

```

```
//one of the buttons is clicked.
```

```
$GET_LONG_TEXT(desc, bNames, initText)
```

```
parameter: desc - text that tells user what he/she should do,  
           type: java.lang.String.
```

```
parameter: bNames - labels for buttons,  
           type: java.lang.String[].
```

```
parameter: initText - initial (default) text,  
           type: java.lang.String.
```

```
type: aja.framework.TextInput
```

```
value: User response.
```

```
//Obtains long text from user. User has to enter a text end  
//to click one button. A current thread is blocked until one  
//of the buttons is clicked.
```

```
$GET_ONE_LINE_TEXT(desc, bNames, initText)
```

```
parameter: desc - text that tells user what he/she should do,  
           type: java.lang.String.
```

```
parameter: bNames - labels for buttons,  
           type: java.lang.String[].
```

```
parameter: initText - initial (default) text,  
           type: java.lang.String.
```

```
type: aja.framework.TextInput
```

```
value: User response.
```

```
//Obtains one-line text from user. User has to enter a text end  
//to click one button. A current thread is blocked until one  
//of the buttons is clicked.
```

```
$GET_COMBO(desc, items, bNames, selected)
```

```
parameter: desc - text that tells user what he/she should do,  
           type: java.lang.String.
```

```
parameter: items - items in the combo box,  
           type: java.lang.String[].
```

```
parameter: bNames - labels for buttons,  
           type: java.lang.String[].
```

```
parameter: selected - index of the initially selected  
                  combo-box item, type: int.  
           type: aja.framework.SingleSelection
```

```
value: User response.
```

```
//Obtains combo-box selection from user. User has to make  
//the selection and to click one button. A current thread is  
//blocked until one of the buttons is clicked.
```

```
$GET_LIST_SINGLE(desc, items, bNames, selected)
```

```
parameter: desc - text that tells user what he/she should do,  
           type: java.lang.String.
```

```
parameter: items - items in the list,  
           type: java.lang.String[].
```

```
parameter: bNames - labels for buttons,  
           type: java.lang.String[].
```

```
parameter: selected - index of the initially selected list item,  
           type: int.
```

```
type: aja.framework.SingleSelection
```

```
value: User response.
```

```
//Obtains a single list selection from user. User has to make  
//the selection and to click one button. A current thread is  
//blocked until one of the buttons is clicked.
```

```
$GET_LIST_MULTIPLE(desc, items, bNames, selected)
```

```
parameter: desc - text that tells user what he/she should do,  
           type: java.lang.String.
```

```
parameter: items - items in the list,  
           type: java.lang.String[].
```

```

parameter: bNames - labels for buttons,
           type: java.lang.String[].
parameter: selected - initial selection,
           type: boolean[].
type: aja.framework.MultipleSelection
value: User response.
//Obtains a multiple list selection from user.
//User has to make the selection and to click one button. A current
//thread is blocked until one of the buttons is clicked.

```

#### **`$GET_CHECK_BOXES(desc, items, bNames, selected)`**

```

parameter: desc - text that tells user what he/she should do,
           type: java.lang.String.
parameter: items - check-boxes items,
           type: java.lang.String[].
parameter: bNames - labels for buttons,
           type: java.lang.String[].
parameter: selected - initial selection,
           type: boolean[].
type: aja.framework.MultipleSelection
value: User response.
//Obtains a check-boxes selection from user. User has to make the
//selection and to click one button. A current thread is blocked
//until one of the buttons is clicked.

```

#### **`$GET_RADIO(desc, items, bNames, selected)`**

```

parameter: desc - text that tells user what he/she should do,
           type: java.lang.String.
parameter: items - radio-buttons items,
           type: java.lang.String[].
parameter: bNames - labels for buttons,
           type: java.lang.String[].
parameter: selected - index of the initially selected item,
           type: int.
type: aja.framework.SingleSelection
value: User response.
//Obtains a radio-button selection from user. User has to make the
//selection and to click one button. A current thread is blocked
//until one of the buttons is clicked.

```

## **4.2.8 Remaining Java+ Constructs**

### **`$NOW`**

```

type: java.util.Calendar
value: Current time.

```

### **`$WAIT(h, m, s, ms)`**

```

parameter: h - hours,
           type: int.
parameter: m - minutes,
           type: int.
parameter: s - seconds,
           type int.
parameter: ms - milliseconds,
           type int.
type: void
//Stops current thread for h hours, m minutes,
//s seconds, and ms milliseconds.

```

### **`$WAIT_UNTIL(date)`**

```

parameter: date - time point,

```

```
        type: java.util.Date.  
type: void  
//Stops current thread until specified time point.
```

## 5 AI Constructs in AJA

Adaptable Parameters .....	93
Dependant Values .....	97

The subjects of this chapter are AI constructs in AJA. These constructs support the adaptability and flexibility of AJA agents. There are two AI constructs in AJA: adaptable parameters and dependant values. They represent two belief types in AJA.

The adaptable parameters are the original contribution of this thesis, while the dependant values encapsulate artificial neural networks (ANNs) with RPROP backpropagation learning algorithm.

The first section of this chapter describes adaptable parameters in details, whereas the second section explains the implementation of dependant values.

## 5.1 Adaptable Parameters

As already described in 3.2.1.2, adaptable parameters can be used instead of constants, when the optimal values of constants are not known in advance or can be changed as the time goes by. Adaptable parameter adjusts its value at run-time according to the feedback received.

Adaptable parameters are declared with the following syntax:

```
<belName> ':' ADAPTABLE [LBOUND <lowerBound>]
                        {UBOUND <upperBound>}
                        '=' <apInitValue>
```

The production rule above allows two optional attributes in an adaptable parameter declaration:

- lower bound – the lowest possible value of the adaptable parameter,
- upper bound – the highest possible value of the adaptable parameter

The initial value of an adaptable parameter has to be specified.

### 5.1.1 Implementation

The adjusting of an adaptable parameter value, after the negative feedback has been received is done using the rules described in this subsection.

The following constants appear in the value adjustment algorithm:

- **incFac** = 1.2 (Factor increasing the step.)
- **decFac** = 0.5 (Factor decreasing the step.)
- **eps** = 0.1 (A small positive number.)
- **memLen** = 10 (Length of the memory storing the parameter recent values.)

In addition to the constants given above, the following variables play important roles in the value-adjusting algorithm:

- **step** – a number that determines the magnitude of the parameter value change. **step** is increased when the next change is in the same direction as the previous one, and decreased when the next change is in the opposite direction as the previous one.

- **badValues** – a queue-like list containing recent parameter values. At most **memLen** recent values are stored in the list.
- **lowerBound** – lower bound of the parameter value.
- **upperBound** – upper bound of the parameter value.
- **value** – current value of the parameter.

#### 5.1.1.1 Initialization

The first value of an adaptable parameter is the value specified as its initial value:

**value = initValue**

The value of the variable **step** is computed as follows:

- a) If there are both bounds specified, then

**step = (upperBound - lowerBound) \* eps**

- b) else if there is only one bound specified (no matter which one), then

**step = max{ |initValue - bound| \* 3 \* eps, eps }**

- c) else (no bound is specified):

1. if **initValue > 1**, then

**step = |initValue| \* eps**

2. else

**step = 2 \* eps**

#### 5.1.1.2 Negative Reinforcement: **\$AP\_HIGHER**

If the negative reinforcement **\$AP\_HIGHER(*be1Name*)** has been received then the value of the parameter should be increased. The following changes occur in the adaptable parameter:

- a) the current value of the parameter is added to the list **badValues**,
- b) if the previous change of the parameter value was also value increase, then

1. **step = step \* incFac**

2. **value = value + step**

else

1. **step = step \* decFac**

2. **value = value + step**

- c) if there is an upper bound for the parameter value, then



```
value = min{value, upperBound}
```

The change magnitude (**step**) depends on the current change direction and the previous change direction. That means if the previous change was increase (decrease) and the current change is increase (decrease), then the step becomes greater in order to move the parameter value faster to the area where the optimal value is located.

In the opposite situation, when the previous change and the current change differ in directions, then the step becomes smaller, because the current parameter value is relatively close to the optimal value.

The idea of modifying the step magnitude depending on the previous and the current change directions is adopted from the RPROP backpropagation ANN learning algorithm, which is described later in this chapter.

The Java method implementing the above rules is given bellow.

```
public synchronized void higher(){
    if (badValues.size() == memLen){
        badValues.removeFirst();
    }
    badValues.add(new Double(value));
    if (previousUp){
        step *= incFac;
        value += step;
    }
    else{
        step *= decFac;
        value += step;
        previousUp = true;
    }
    if (hasUpperBound){
        value = Math.min(value, upperBound);
    }
}
```

### 5.1.1.3 Negative Reinforcement: **\$AP\_LOWER**

If the negative reinforcement **\$AP\_LOWER** (*be1Name*) has been received then the value of the parameter should be decreased. This is done symmetrically to the above described procedure for the negative reinforcement **\$AP\_HIGHER**.

The source code of the corresponding Java method is given below.

```
public synchronized void lower(){
    if (badValues.size() == memLen){
        badValues.removeFirst();
    }
    badValues.add(new Double(value));
    if (previousUp){
        step *= decFac;
        value -= step;
        previousUp = false;
    }
}
```

```

}
else{
    step *= incFac;
    value -= step;
}
if (hasLowerBound){
    value = Math.max(value, lowerBound);
}
}
}

```

#### 5.1.1.4 Negative Reinforcement: \$AP\_BAD

In case when the negative reinforcement \$AP\_BAD has been received, it is not known whether the value of the parameter should be increased or decreased. It is only known that the current value should be changed. This is the most complicated case of all three cases.

To determine the direction of the parameter value change, i.e. the increase or the decrease, the recent parameter values in the **badValues** list are analyzed. They are compared with the two possible new parameter values. One possible new parameter value is the one obtained if the current parameter value decreases (**lowerPoint**) and the other one is the one obtained if the current parameter value increases (**upperPoint**).

The new parameter value will be the one of the two possible new values that has the greater minimal distance to one of the values in the **badValues** list. If the two minimal distances are the same or the list **badValues** is an empty list, then the random number generator determines which one of two possible values will be the new parameter value.

The Java method implementing these rules is given below.

```

public synchronized void bad(){
    // determination of two possible new values
    double lowerPoint = 0;
    double upperPoint = 0;
    if (previousUp){
        lowerPoint = value - decFac * step;
        upperPoint = value + incFac * step;
    }
    else{
        lowerPoint = value - incFac * step;
        upperPoint = value + decFac * step;
    }
    if (hasLowerBound){
        lowerPoint = Math.max(lowerPoint, lowerBound);
    }
    if (hasUpperBound){
        upperPoint = Math.min(upperPoint, upperBound);
    }

    // evaluation of both values
    double lowNearest = Double.MAX_VALUE;
    double upNearest = Double.MAX_VALUE;
    for (ListIterator li = badValues.listIterator(0); li.hasNext();){
        double badPoint = ((Double) li.next()).doubleValue();
        double difLow = Math.abs( lowerPoint - badPoint );
        if ( difLow < lowNearest ){
            lowNearest = difLow;
        }
    }
}

```

```

    }
    double difUp = Math.abs( upperPoint - badPoint );
    if ( difUp < upNearest ){
        upNearest = difUp;
    }

// selection of the new value
double newValue = 0;
if ( lowNearest == Double.MAX_VALUE ||
    upNearest == Double.MAX_VALUE ||
    lowNearest == upNearest ){
    if ( randGen.nextBoolean() ){
        newValue = lowerPoint;
    }
    else{
        newValue = upperPoint;
    }
}
else if (lowNearest < upNearest){
    newValue = lowerPoint;
}
else{
    newValue = upperPoint;
}

// badValues update
if (badValues.size() == memLen){
    badValues.removeFirst();
}
badValues.add(new Double(value));

// setting new values
if (newValue < value){
    if (previousUp){
        step *= decFac;
    }
    else{
        step *= incFac;
    }
    previousUp = false;
}
else{
    if (previousUp){
        step *= incFac;
    }
    else{
        step *= decFac;
    }
    previousUp = true;
}
value = newValue;
}
}

```

## 5.2 Dependant Values

Dependant values beliefs in AJA encapsulate multi-layer feedforward artificial neural networks. Thus, when an AJA programmer declares and uses a dependant value in his/her program, he/she actually implicitly uses a neural network.

A programmer does not have to be familiar with neural networks in order to be able to utilize AJA dependant values. The incorporation of ANNs into the language and

therefore the implicit use of ANNs is the main benefit of the dependant values in AJA.

Dependant values are declared with the following syntax:

```
<belName> ':' DEPENDS_ON <neurParamList>
                [MIN_VAL <lowerBound>]
                [MAX_VAL <upperBound>]
                EXAMPLES_FILE <fileName>
                [ <netConf> ]
```

where:

```
<neurParamList> = <paramName>
                  [MIN <lowerBound>]
                  [MAX <upperBound>]
                  (',' <paramName>
                  [MIN <lowerBound>]
                  [MAX <upperBound>] )

<netConf> = HIDDEN_LAYERS <nodesNum> {',' <nodesNum>}
```

An example of a dependant value belief declaration is already given in 3.2.1.3. Below is the same example given again:

```
consultationDuration :
    DEPENDS_ON
        numOfStudents MIN << 1 >> MAX << 30 >>,
        daysBefore MIN << 0 >> MAX << 50 >>
        MIN_VAL << 1 >>
        MAX_VAL << 240 >>
        EXAMPLES_FILE "nnsamples.txt"
        HIDDEN_LAYERS 5;
```

There are two input values for this dependant belief:

- **numOfStudents** with the lower bound 1 and the upper bound 30,
- **daysBefore** with the lower bound 0 and the upper bound 50.

The minimal output value is 1 and the maximal is 240. The examples for the ANN are in the file **nnsamples.txt**. The ANN has one hidden layer with five nodes.

As already said, AJA programmer does not have to be familiar with ANN, however he/she has to know some basic details about ANN, namely:

- An ANN has input nodes and one or more output nodes. The number of input nodes in AJA is equal to the number of input values of a dependant belief and there is always only one output node.
- A set of examples for the ANN learning has to be provided. Example set in AJA is provided as a text file containing one example in each line of the file. An example is a list of real numbers separated with spaces. The numbers before the last number represent the input values. The last number is the desired output value for the given input values.
- An ANN can have nodes in hidden layers. There are many rules of thumb for the determination of the optimal number of hidden layers and the optimal number of nodes in hidden layers [87]. However, it is easy to

construct counterexamples that disprove these rules of thumb [87]. Thus, the number of hidden layers and the number of nodes in each layer should be determined experimentally.

- Two parameters that determine the stopping conditions of the ANN off-line training have to be provided:
  - maximal generalization error allowed (e.g. 5%), and
  - maximal number of repeated iterations during the ANN training (e.g. 500).

### 5.2.1 RPROP

One of the best supervised learning algorithms, RPROP [85], [84], is chosen for ANN learning in AJA. RPROP stands for "Resilient backpropagation". It is a learning scheme performing supervised batch learning in multilayer feedforward ANNs.

RPROP is an improvement of the standard backprop. The main theoretical flow of the standard backprop (and many other backprops) is that the magnitude of the change in the weights (i.e. the step size) is a function of the magnitude of the error gradient. In some regions of the weight space, the gradient is small and a large step size is needed. In other regions of the weight space, the gradient is small and a small step size is needed if a local minimum is nearby. Likewise, a large gradient may call for either a small step or a large step. The great benefit of RPROP is that it does not have this unnecessary dependence on the magnitude of the gradient.

The basic principle of RPROP is to eliminate the harmful influence of the gradient magnitude on the weight step. As a consequence, only the sign of the derivative is considered to indicate the direction of the weight update.

The 'update-value'  $\Delta_j^{(t)}$  determines the size of the weight change:

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_j^{(t)} & \text{if } \frac{\partial E^{(t)}}{\partial w_{ij}} > 0 \\ +\Delta_j^{(t)} & \text{if } \frac{\partial E^{(t)}}{\partial w_{ij}} < 0 \\ 0 & \text{else} \end{cases}$$

Due to batch learning, the term  $\frac{\partial E^{(t)}}{\partial w_{ij}}$  in the above formula represents the summed gradient information over all examples in the example set.

In the next step the new update-values are determined:

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ \cdot \Delta_{ij}^{(t-1)} & \text{if } \frac{\partial E^{(t-1)}}{\partial w_{ij}} \cdot \frac{\partial E^{(t)}}{\partial w_{ij}} > 0 \\ \eta^- \cdot \Delta_{ij}^{(t-1)} & \text{if } \frac{\partial E^{(t-1)}}{\partial w_{ij}} \cdot \frac{\partial E^{(t)}}{\partial w_{ij}} < 0 \\ \Delta_{ij}^{(t-1)} & \text{else} \end{cases}$$

where  $0 < \eta^- < 1 < \eta^+$ .

In [84] the above formula is interpreted as follows:

"...Every time the partial derivative of the corresponding weight  $w_{ij}$  changes its sign, which indicates that the last update was too big and the algorithm has jumped over a local minimum, the update-value  $\Delta_{ij}^{(t)}$  is decreased by the factor  $\eta^-$ . If the derivative retains its sign, the update value is slightly increased in order to accelerate convergence in shallow regions. Additionally, in case of a change in sign, there should be no adaptation in the succeeding learning step. In practice, this can be achieved by setting  $\frac{\partial E^{(t-1)}}{\partial w_{ij}} := 0$  in the above adaptation rule."

The RPROP learning algorithm implemented in AJA is given below (from [84]):

$$\forall i, j: \Delta_{ij}(t) = \Delta_0$$

$$\forall i, j: \frac{\partial E}{\partial w_{ij}}(t-1) = 0$$

Repeat

$$\text{Compute Gradient } \frac{\partial E}{\partial w}$$

For all weights and biases {

$$\text{if } \left( \frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) > 0 \right) \text{ then } \{$$

$$\Delta_{ij}(t) = \min(\Delta_{ij}(t-1) * \eta^+, \Delta_{\max})$$

$$\Delta w_{ij}(t) = -\text{sign}\left(\frac{\partial E}{\partial w_{ij}}(t)\right) * \Delta_{ij}(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

$$\frac{\partial E}{\partial w_{ij}}(t-1) = \frac{\partial E}{\partial w_{ij}}(t)$$

}

else if  $(\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) < 0)$  then {

$$\Delta_{ij}(t) = \max(\Delta_{ij}(t-1) * \eta^-, \Delta_{\min})$$

$$\frac{\partial E}{\partial w_{ij}}(t-1) = 0$$

}

else if  $(\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) = 0)$  then {

$$\Delta w_{ij}(t) = -\text{sign}(\frac{\partial E}{\partial w_{ij}}(t)) * \Delta_{ij}(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

$$\frac{\partial E}{\partial w_{ij}}(t-1) = \frac{\partial E}{\partial w_{ij}}(t)$$

}

}

Until (converged)

Values of the parameters  $\Delta_0$ ,  $\Delta_{\min}$ ,  $\Delta_{\max}$ ,  $\eta^-$ , and  $\eta^+$  proposed in [84] and used in AJA are as follows:

$$\Delta_0 = 0.1$$

$$\Delta_{\min} = 10^{-6}$$

$$\Delta_{\max} = 50$$

$$\eta^- = 0.5$$

$$\eta^+ = 1.2$$

### 5.2.2 Offline Training: \$DV\_OFFLINE\_TRAINING

Offline training of a neural network encapsulated in an AJA dependant belief is invoked with the Java+ construct `$DV_OFFLINE_TRAINING(belName, maxCycles, maxAverageError)`.

The first parameter in the above construct, *belName*, is the name of a dependant value belief. The remaining two parameters, *maxCycles* and *maxAverageError*, determine the stopping condition in the offline training.

The number of training cycles is constrained with the parameter *maxCycles*. Nevertheless, the training can also be finished much sooner, namely if the average error has stopped to descend and it is lower than the parameter *maxAverageError*.

The construct `$DV_OFFLINE_TRAINING` returns the resulting average error as its value. For example:

```
double error = $DV_OFFLINE_TRAINING(consultationDuration, 500, 5);
```

Name of the dependant value belief is `consultationDuration`. The maximal number of cycles in training is 500. The desired maximal average error is 5%. After the training process has finished, the local variable `error` will get the average error in percents as its value.

### 5.2.3 Firing the Network: \$GET\_BEL

A neural network inside an AJA agent is fired, when the construct `$GET_BEL(belName(param1, param2, ...))` is called. *belName* is the name of a dependant value belief. The parameters *param1*, *param2*, ... are the input values. The network is fired using given input values, and the neural network output is returned as a result.

For example:

```
int expDurMin = (int) $GET_BEL(consultationDuration(10, 15));
```

Name of the dependant value belief is `consultationDuration`. There are ten students appointed for the consultation and there are fifteen days remaining before the next exam. The resulting value generated by the network is of type `double`, so it has to be casted into `int` in order to assign it to the `int` variable `expDurMin`.

### 5.2.4 Online Training: \$DV\_SHOULD\_BE

RPROP is an offline supervised training algorithm. However, it can be easily expanded with the online supervised training part, which takes place after the network has been trained offline. New training examples can emerge during the agent lifetime and they can be used for further training of the network and thus for the adaptation to a new situation.

The Java+ construct `$DV_SHOULD_BE(belName(param1, param2, ...), value)` is used for this purpose. *belName* is the name of a dependant belief. The input values part of the example is given as a list *param1*, *param2*, ..., and the target output is *value*.



### Example:

```
$DV_SHOULD_BE(consultationDuration(10, 15), 120);
```

A consultation took place. Ten students participated, it was fifteen days before the next exams and the consultation lasted exactly two hours. This empirical information can be used for the online training of the dependant belief `consultationDuration` as shown above.

## 6 A Case Study – Multi-Agent System Implemented in AJA

What does the MAS do? .....	105
Beliefs .....	105
Actions .....	108
Reflexes .....	111
Negotiations.....	112
WWW Negotiation .....	120
Initialization.....	122

This chapter describes a case study MAS, which has been implemented in AJA. The purpose of the implemented system was to demonstrate and to test the tool.

The MAS is a homogenous MAS, because all the agents in the system are very similar. Their programs differ only in the first few lines, where the agent name, location, ports, and keystore properties are defined.

The first section of this chapter describes what does MAS do. The remaining sections present the AJA program used.

The step-by-step use of the MAS is described in the appendix C.

## 6.1 What does the MAS do?

The implemented MAS consists of agents that act like personal digital assistants (PDAs). Each PDA belongs to one lecturer at University.

The main purpose of a PDA agent is:

- to maintain the timetable of its owner,
- to alert the owner to the approaching engagements registered in the timetable such as appointments with colleagues and consultations with students,
- to alert the owner when a colleague has a birthday,
- to be helpful in creating new engagements, especially the ones where other colleagues also participate, and
- to enable students to register themselves online for the consultations with the lecturer.

The implemented MAS is fully scalable, hence the number of agents in the system is irrelevant for the system performance.

*Although the purpose of the implemented MAS was to demonstrate and to test AJA, it is implemented thoroughly in a way the software is implemented for the commercial use.*

## 6.2 Beliefs

The AJA code defining PDA agent beliefs is given below.

```
BELIEFS
  //schedule of the lecturer's engagements
  timetable : TimeTable;

  //how many minutes before the next event
  //should the lecturer be reminded
  eventAlertTime : ADAPTABLE
    LBOUND << 0 >>
    = << 15 >>;
```

```

eventAlertTimeToBackup : boolean = << false >>;
                        //used in backupEventAlertTimeReflex

engToAlert : Engagement; // used in eventAlertReflex

birthdaysTomorrowToAlert : Vector ; //used in birthdayAlertReflex
birthdaysTodayToAlert : Vector ; //used in birthdayAlertReflex;

//expected duration of the consultation with students
consultationDuration : DEPENDS_ON
    numOfStudents MIN << 1 >> MAX << 30 >>,
    daysBefore MIN << 0 >> MAX << 50 >>
    MIN_VAL << 1 >>
    MAX_VAL << 240 >>
    EXAMPLES_FILE "nnsamples.txt"
    HIDDEN_LAYERS 5;
    //one hidden layer with five nodes

consultationDurationToBackup : boolean = << false >>;
                        //used in backupConsultationDurationReflex

```

### 6.2.1 timeTable

The first and the most important PDA agent belief is the belief **timeTable**.

```

//schedule of the lecturer's engagements
timeTable : TimeTable;

```

The type of this belief is the Java class **demo.Timetable**. The package name **demo** is omitted in the source code above, because it has been imported in the **import** part of the program. The class **demo.Timetable** is relatively complex one. One instance of this class stores and maintains all engagements, colleagues and available times of the agent owner. It is programmed in the thread-safe manner, because many concurrent threads in the agent program access this data structure.

Concurrent threads synchronization in AJA can be done at two levels. As first, in the declaration of agent actions it can be specified which actions are incompatible, i.e. which actions cannot execute concurrently. Secondly, Java threads can be synchronized by writing thread safe Java classes used as data structures for beliefs in AJA. In the implemented case study MAS the latter solution is used.

### 6.2.2 eventAlertTime

**eventAlertTime** is an adaptable parameter.

```

//how many minutes before the next event
//should the lecturer be reminded
eventAlertTime : ADAPTABLE
    LBOUND << 0 >>
    = << 15 >>;

```

The value of this belief determines the alert time for engagements in the timetable. The initial value of the belief is 15 (minutes). The value of the belief is adapted at

run-time according to reinforcements received. However, the lowest possible value of the belief is 0.

### 6.2.3 eventAlertTimeToBackup

**eventAlertTimeToBackup** belief has a boolean value. It is used in the activation condition of the reflex that stores the belief **eventAlertTime** into a file.

```
eventAlertTimeToBackup : boolean = << false >>;  
    //used in backupEventAlertTimeReflex
```

The initial value of the belief is **false**. Whenever the belief **eventAlertTime** changes, the value of the belief **eventAlertTimeToBackup** is set to **true** and the **backupEventAlertTimeReflex** will save the **eventAlertTime** belief into a file.

### 6.2.4 engToAlert

The belief **engToAlert** stores the first engagement to which the agent owner should be alerted.

```
engToAlert : Engagement; // used in eventAlertReflex
```

### 6.2.5 birthdaysTomorrowToAlert and birthdaysTodayToAlert

These two beliefs store the persons having birthdays tomorrow and today respectively. The type of both beliefs is **java.util.Vector**.

```
birthdaysTomorrowToAlert : Vector ; //used in birthdayAlertReflex  
birthdaysTodayToAlert : Vector ; //used in birthdayAlertReflex;
```

The beliefs are used in the reflex **birthayAlertReflex**.

### 6.2.6 consultationDuration

This belief is a dependant value. It is used for the estimation of the expected duration of consultations with students. It was empirically found out, that the duration of a consultation depends on the number of appointed students and the time remaining to the next exams. The analytical function however is not available; hence a dependant value belief (i.e. a neural net) is used.

```
//expected duration of the consultation with students  
consultationDuration : DEPENDS_ON  
    numOfStudents MIN << 1 >> MAX << 30 >>,  
    daysBefore MIN << 0 >> MAX << 50 >>  
    MIN_VAL << 1 >>  
    MAX_VAL << 240 >>
```

```
EXAMPLES_FILE "nnsamples.txt"
HIDDEN_LAYERS 5;
//one hidden layer with five nodes
```

The neural net nested in this belief has two input nodes. The first one represents the number of students appointed for the consultation (between one and thirty) and the second one specifies the number of days before the next exams (between zero and fifty).

The value of the belief is the expected duration of the consultation in minutes (between one and 240).

The examples for the supervised learning are stored in the file `nnsamples.txt`, which can be found in the current directory. The example file stores one example at a line. The first line in the file is:

```
1 24 8
```

If only one student comes to the consultations and there are 24 days before the next exams, then the consultation duration is eight minutes.

The neural net used has one hidden layer containing five nodes.

### 6.2.7 `consultationDurationToBackup`

The `consultationDurationToBackup` is a belief similar to the belief `eventAlertTimeToBackup`.

```
consultationDurationToBackup : boolean = << false >>;
//used in backupConsultationDurationReflex
```

It is used in the activation condition of the reflex `backupConsultationDurationReflex`, which stores the dependant belief `consultationDuration` to a file.

## 6.3 Actions

The actions declaration part in the PDA agent program contains thirty-eight actions. The actions declared can be logically grouped into the following five groups of actions:

- actions manipulating the timetable,
- actions alerting the user,
- actions performing backup of important beliefs,
- actions implementing GUI.

### 6.3.1 Timetable Manipulation

The actions that modify the `timeTable` belief are relatively simple ones. Their complexity is however hidden in the Java class `demo.Timetable`, to which the actual task is forwarded.

The actions in this group are very similar to each other. The action `removeOldEngagementsAct` given below is the typical action in this group.

```
ACTION void removeOldEngagementsAct() //removes old engagements
<<
    TimeTable tt = $GET_BEL(timeTable);
    tt.removeOldEngagements();
>>
```

This action removes old entries from the timetable.

### 6.3.2 Alerting the User

There are two actions that alert the user to the incoming events. The first one is the action `eventAlertAct`, which is given as example in 3.2.2. The second alerting action is the action `birthdayAlertAct`.

```
ACTION void birthdayAlertAct()
<<
    Vector tomorrow = $GET_BEL(birthdaysTomorrowToAlert);
    for (int i=0; i<tomorrow.size(); i++){
        Person p = (Person) tomorrow.get(i);
        BirthdayDialog bd = new BirthdayDialog($AG_JFRAME, p);
        bd.show();
    }
    Vector today = $GET_BEL(birthdaysTodayToAlert);
    for (int i=0; i<today.size(); i++){
        Person p = (Person) today.get(i);
        BirthdayDialog bd = new BirthdayDialog($AG_JFRAME, p);
        bd.show();
    }
>>
```

The action `birthdayAlertAct` does not have parameters and does not return a value. It pop-ups dialog windows that alert the user about birthdays of his/her colleagues taking place tomorrow and today.

The class `BirthdayDialog` is an ordinary Java class that extends `javax.swing.JDialog`. `$AG_JFRAME` is the reference of the agent window, which is a subclass of `javax.swing.JFrame`.

### 6.3.3 Backup

Theoretically, an AJA agent runs all the time, without stopping. However, in praxis this is not the case. Due to this reason, the important agent beliefs have to be saved in the files periodically. When the agent starts, it should check if the backup files exist and to initialize its beliefs from the files if they are found.

The beliefs of PDA agents in the implemented MAS that have to be stored in the secondary storage are the beliefs `timeTable`, `eventAlertTime`, and `consultationDuration`. Consequently, there are three actions storing the

three beliefs to files. One of them, `backupConsultationDurationAct` is given below.

```

ACTION void backupConsultationDurationAct()
    //stores a neural network into a file
    <<
    try{
        File toFile =
            new File($AG_NAME + "_consultationdurationbackup.dat");
        if (toFile.exists()){
            String name = toFile.getName();
            File oldFile = new File(name+".old");
            if (oldFile.exists()){
                oldFile.delete();
            }
            toFile.renameTo(oldFile);
        }
        $DV_TO_FILE(consultationDuration,
            $AG_NAME + "_consultationdurationbackup.dat");
    }
    catch (IOException ioe){
        System.err.println("IO Error while creating backup " +
            "of consultationDuration:");
        ioe.printStackTrace();
    }
    $SET_BEL(consultationDurationToBackup, false);
    >>

```

The action `backupConsultationDurationAct` does not have parameters and does not return a value. It checks first if the backup file for the `consultationDuration` already exists. If it exists, the file is renamed and a new file with new backup is created. Consequently, there are always two backups available: the last one and the one before the last.

At the end, the belief `consultationDurationToBackup` gets the value `false`.

### 6.3.4 GUI

At last, but not at least, there are many actions implementing GUI communication with the user. The action that implements the main menu is the action `doGUIAct`.

```

ACTION void doGUIAct() //main menu
    <<
    $EXEC_PARALLEL(updateStatusBarAct);
    String[] items ={"engagements",
        "colleagues",
        "your availability"};
    String[] bNames = {"OK"};
    int selected = 0;
    while (true){
        SingleSelection ss =
            $GET_RADIO("Select an option and press OK button.",
                items, bNames, selected);
        selected = ss.getSelItemIndex();
        if (selected == 0){ //engagements
            $EXEC(editEngagementsAct);
        }
        else if (selected == 1){ // colleagues
            $EXEC(editColleaguesAct);
        }
        else{ //your availability
            $EXEC(editAvailabilityAct);
        }
    }
    >>

```



When the action `doGUIAct` executes, the main agent window looks like the one in the Figure 6 on the page 68.

## 6.4 Reflexes

A PDA agent in the implemented MAS has six reflexes declared. They can be divided into three groups:

- reflexes invoking the actions that alert the user about incoming events,
- reflexes invoking actions performing backup of important beliefs,
- reflexes maintaining the timetable.

### 6.4.1 Alerting Reflexes

The two reflexes that invoke the actions alerting the user to the incoming events are `eventAlertReflex` and `birthdayAlertReflex`.

The source code and the description of the `eventAlertReflex` are given in 3.2.3.

The reflex `birthdayAlertReflex` invokes the action `birthdayAlertAct`, which is given above in 6.3.2.

### 6.4.2 Reflexes for Backup

Important beliefs are backed up using the three reflexes given below.

```
REFLEX backupTimeTableReflex
  CHECKING_PERIOD 4000
  CONDITION
    <<
      TimeTable tt = $GET_BEL(timeTable);
      return tt.isModified();
    >>
  EXEC
    backupTimeTableAct();

REFLEX backupEventAlertTimeReflex
  CHECKING_PERIOD 4000
  CONDITION
    <<
      return $GET_BEL(eventAlertTimeToBackup);
    >>
  EXEC
    backupEventAlertTimeAct();

REFLEX backupConsultationDurationReflex
  CHECKING_PERIOD 4000
  CONDITION
    <<
      return $GET_BEL(consultationDurationToBackup);
    >>
  EXEC
    backupConsultationDurationAct();
```

All three reflexes above have the checking period four seconds (4000 ms).

### 6.4.3 Reflexes for Timetable Maintenance

There is only one reflex that maintains the `timeTable` belief. This reflex removes past engagements from the timetable.

```
REFLEX removeOldEngagementsReflex
CHECKING_PERIOD 3600000
CONDITION
  <<
    TimeTable tt = $GET_BEL(timeTable);
    return tt.hasOldEngagements();
  >>
EXEC
  removeOldEngagementsAct();
```

The reflex is triggered every hour (3600000 ms). The action invoked when the condition is satisfied, is the action `removeOldEngagementsAct`. The source code of this action is shown in 6.3.1.

## 6.5 Negotiations

A PDA agent program contains the following requesting and responding negotiations (only the headers are given):

```
REQUESTING_NEGOTIATION void GetBirthdayReqNeg(Person p)
```

The negotiation obtains the birthday of the person `p` from its agent.

```
RESPONDING_NEGOTIATION GetBirthdayResNeg
```

Responds to a request made by `GetBirthdayReqNeg` of another agent. The birthday of the owner is sent to the requesting agent.

```
REQUESTING_NEGOTIATION void EngagementInitReqNeg( Interval[]
possibleStarts,
int expectedDuration, String subject, String comment, Vector
personsToInvite,
int priority)
```

This negotiation is used for the creation of joint engagements. The source code of the negotiation is enlisted later in this section.

```
RESPONDING_NEGOTIATION EngagementInitResNeg
```

The responding negotiation for the joint engagements creation. The first message sent from other agent's `EngagementInitReqNeg` activates this negotiation. The source code of this negotiation is also enlisted later in this section.

```
REQUESTING_NEGOTIATION void FindReplacementReqNeg(Engagement eng,
    Vector possibleReplacements)
```

This negotiation is used when the participant of an appointed joint engagement cannot participate any more and wants to find the replacement.

```
RESPONDING_NEGOTIATION ReplacementResNeg
```

The responding negotiation for the previous requesting negotiation.

```
REQUESTING_NEGOTIATION void InformAllReqNeg(Vector persons, String message,
    Object param)
```

This requesting negotiation sends a message containing the speech act `message` and the serializable object `param` to the agent of each person in the parameter `persons`. This is an auxiliary negotiation used in other negotiations.

```
REQUESTING_NEGOTIATION void RepeatedInformReqNeg(Person p, String message,
    Object param, long periodMS, int maxRepeat)
```

This requesting negotiation sends a message containing the speech act `message` and the serializable object `param` to the agent of person `p`. If the destination agent cannot be reached, the message sending is repeated with the period `periodMS` milliseconds `maxRepeat` times. This is also an auxiliary negotiation used in other negotiations.

```
RESPONDING_NEGOTIATION InformResNeg
```

This simple responding negotiation handles the receipt of several messages that inform the agent about the status of appointed negotiations.

### 6.5.1 EngagementInitReqNeg and EngagementInitResNeg

The requesting–responding negotiation pair used for the creation of joint engagements between two or among more persons is already described in 3.2.4. The negotiation states of `EngagementInitReqNeg` and `EngagementInitResNeg` are specified in pseudo-code and depicted in figures Figure 3 (on the page 60) and Figure 4 (on the page 62) respectively.

This subsection presents the source code of both engagements.

```
REQUESTING_NEGOTIATION void EngagementInitReqNeg(Interval[] possibleStarts,
    int expectedDuration, String subject, String comment, Vector
personsToInvite,
    int priority)
NEG_INIT
<<
    TimeTable tt = $GET_BEL(timeTable);
    String[] sessIds = new String[personsToInvite.size()];
    String failureMessage = null;
    Date engStart = null;
```

```

String engId = null;
>>

START: //send initial messages to all participants
<<
boolean ok = true;
engId = Engagement.generateNewId();
for (int i=0; ok && i<personsToInvite.size(); i++){
    Person p = (Person) personsToInvite.elementAt(i);
    try{
        sessIds[i] =
            $SEND_FIRST_ENCRYPTED(p.getAgentSSLAddress(),
                "new engagement request", possibleStarts,
                new Integer(expectedDuration), subject, comment,
                tt.getUser(), personsToInvite, new Integer(priority),
                engId);
    }
    catch (CommunicationException ce){
        System.err.println(ce);
        ok = false;
    }
}
if (ok){
    boolean allAnswered = false;
    int count = 0;
    while (!allAnswered && count<30){
        $WAIT(0,0,2,0);
        //0 hours, 0 minute, 2 seconds, 0 milliseconds
        allAnswered = $EXEC(AllHaveAnswered, sessIds);
        count++;
    }
    if (allAnswered){
        $STATE(DETERMINE_ENG_START);
    }
    else{
        $STATE(ERROR);
    }
}
else{
    $STATE(ERROR);
}
>>

DETERMINE_ENG_START: //find the earliest possible starting time
<<
try{
    IntervalsIntersectionFinder finder =
        new IntervalsIntersectionFinder();
    finder.addIntervals(possibleStarts);
    boolean ok = true;
    for (int i=0; i<personsToInvite.size(); i++){
        MessageData md = $GET_ANSW_ENCRYPTED(sessIds[i], 0);
        String speechAct = md.getSpeechAct();
        if (speechAct.equals("here are my intervals")){
            Object[] params = md.getParams();
            Interval[] intervals = (Interval[]) params[0];
            finder.addIntervals(intervals);
        }
        else{
            ok = false;
        }
    }
    if (ok){
        Interval[] intersection = finder.getIntersection();
        if (intersection != null){
            engStart =
                $EXEC(DetermineAndReserveTimeAct,
                    intersection,

```



```

try(
  boolean confirmed = true;
  for (int i=0; confirmed && i<personsToInvite.size(); i++){
    MessageData md = $GET_ANSW_ENCRYPTED(sessIds[i], 0);
    String speechAct = md.getSpeechAct();
    if (speechAct.equals("cannot take part any more")){
      confirmed = false;
    }
  }
  if (confirmed){
    $STATE(CONFIRM_ENGAGEMENT);
  }
  else{
    $STATE(REPEAT_ALL); //try again
  }
}
catch (CommunicationException ce){
  System.err.println(ce);
  $STATE(ERROR);
}
}
else{
  $STATE(ERROR);
}
}
else{
  $STATE(ERROR);
}
}
>>

```

REPEAT\_ALL:

```

<<
$EXEC(ReleaseIntervalAct, engStart, expectedDuration);
boolean ok = true;
for (int i=0; ok && i<personsToInvite.size(); i++){
  Person p = (Person) personsToInvite.elementAt(i);
  try{
    $REPLY_ENCRYPTED(p.getAgentSSLAddress(),
      sessIds[i],
      "let's try again");
  }
  catch (CommunicationException ce){
    System.err.println(ce);
    ok = false;
  }
}
if (ok){
  boolean allAnswered = false;
  int count = 0;
  while (!allAnswered && count<30){
    $WAIT(0,0,2,0);
    //0 hours, 0 minute, 2 seconds, 0 milliseconds
    allAnswered = $EXEC(AllHaveAnswered, sessIds);
    count++;
  }
  if (allAnswered){
    $STATE(DETERMINE_ENG_START);
  }
  else{
    $STATE(ERROR);
  }
}
else{
  $STATE(ERROR);
}
}
>>

```

```

ERROR: //network error or problems with keystore,
      //cancel the negotiation
<<
  if (engStart != null){
    $EXEC(ReleaseIntervalAct, engStart, expectedDuration);
  }
  for (int i=0; i<personsToInvite.size(); i++){
    Person p = (Person) personsToInvite.elementAt(i);
    try{
      $REPLY_ENCRYPTED(p.getAgentSSLAddress(),
                      sessIds[i],
                      "engagement cancelled");
    }
    catch (CommunicationException ce){
      System.err.println(ce);
    }
  }
  failureMessage = "An error has occurred.";
  $STATE(REPORT_FAILURE);
>>

FINAL REPORT_FAILURE: //inform the user about failure
<<
  JOptionPane.showMessageDialog($AG_JFRAME,
                                failureMessage,
                                "Failure",
                                JOptionPane.ERROR_MESSAGE);
>>

FINAL CONFIRM_ENGAGEMENT: //confirm the engagement
<<
  Date engEnd = new Date( engStart.getTime() +
                          expectedDuration * 60000 );
  Engagement eng =
    new Engagement(
      engId,
      engStart,
      engEnd,
      subject,
      comment,
      tt.getUser(),
      personsToInvite,
      Engagement.CONFIRMED_BY_USER_BUT_NOT_BY_ALL);
  $EXEC_PARALLEL( addNewEngagement, eng);
  for (int i=0; i<personsToInvite.size(); i++){
    Person p = (Person) personsToInvite.elementAt(i);
    try{
      $REPLY_ENCRYPTED(p.getAgentSSLAddress(),
                      sessIds[i],
                      "engagement created");
    }
    catch (CommunicationException ce){
      System.err.println(ce);
      $NEGOTIATE_PARALLEL(RepeatedInformReqNeg, p,
                          "engagement created", engId, 10 * 60000, 200);
    }
  }
>>

```

See the description of the negotiation **EngagementInitReqNeg** in 3.2.4.

RESPONDING\_NEGOTIATION EngagementInitResNeg

```

ACTIVATION_CONDITION(MessageData md)
<<

```

```

String spAct = md.getSpeechAct();
return spAct.equals("new engagement request");
>>

NEG_INIT
<<
String initiatorAgentRMIURL = md.getFromAgentRMIURL();
String initiatorAgentSSLAddress = md.getFromAgentSSLAddress();
String sessId = md.getSessId();
Object[] params = md.getParams();
Interval[] possibleStarts = (Interval[]) params[0];
int expectedDuration = ((Integer) params[1]).intValue();
String subject = (String) params[2];
String comment = (String) params[3];
Person initiator = (Person) params[4];
Vector invitedPersons = (Vector) params[5];
int priority = ((Integer) params[6]).intValue();
String engId = (String) params[7];
Engagement eng = null;
>>

START:
<<
Interval[] myPossibleStarts =
    $EXEC(FindMyPossibleStartsAct,
        possibleStarts,
        expectedDuration,
        priority);
try{
    if (myPossibleStarts == null){
        $REPLY_ENCRYPTED(initiatorAgentSSLAddress,
            sessId,
            "no time available");
        $STATE(END_STATE);
    }
    else{
        $REPLY_ENCRYPTED(initiatorAgentSSLAddress,
            sessId,
            "here are my intervals",
            myPossibleStarts);
        $STATE(SECOND_MESSAGE);
    }
}
catch (CommunicationException ce){
    System.err.println(ce);
    $STATE(ERROR);
}
>>

SECOND_MESSAGE:
<<
try{
    MessageData md2 = $GET_ANSW_ENCRYPTED(sessId, 10*60000);
    String spAct2 = md2.getSpeechAct();
    if ( spAct2.equals("engagement cancelled") ||
        spAct2.equals("abandon") ){
        $STATE(END_STATE);
    }
    else{ //"proposed start of the engagement"
        Date engStart = (Date) md2.getParams()[0];
        eng = $EXEC( CheckAddEngagementAct,
            engStart,
            expectedDuration,
            subject,
            comment,
            initiator,
            invitedPersons,
            priority,

```



```

        engId );
    if (eng == null){
        $REPLY_ENCRYPTED(initiatorAgentSSLAddress,
            sessId,
            "cannot take part any more");
        $STATE(WAIT_REPEAT);
    }
    else{
        $REPLY_ENCRYPTED(initiatorAgentSSLAddress,
            sessId,
            "ok");
        $STATE(THIRD_MESSAGE);
    }
}
}
catch(CommunicationException ce){
    System.err.println(ce);
    $STATE(ERROR);
}
>>

WAIT_REPEAT:
<<
    try{
        MessageData mdwr = $GET_ANSW_ENCRYPTED(sessId, 10*60000);
        String spActwr = mdwr.getSpeechAct();
        if ( spActwr.equals("let's try again") ){
            $STATE(START);
        }
        else{ //never happens
            $STATE(ERROR);
        }
    }
    catch(CommunicationException ce){
        System.err.println(ce);
        $STATE(ERROR);
    }
>>

THIRD_MESSAGE:
<<
    try{
        MessageData md3 = $GET_ANSW_ENCRYPTED(sessId, 10*60000);
        String spAct3 = md3.getSpeechAct();
        if ( spAct3.equals("engagement cancelled") ){
            $STATE(ERROR);
        }
        else if ( spAct3.equals("let's try again") ){
            $EXEC( RemoveEngagementAct, engId );
            eng = null;
            $STATE(START);
        }
        else{ //"engagement created"
            $EXEC_PARALLEL( RegisterCreatedEngagementAct,
                engId,
                initiatorAgentRMIURL );
            $STATE(END_STATE);
        }
    }
    catch(CommunicationException ce){
        System.err.println(ce);
        $STATE(ERROR);
    }
>>

FINAL ERROR:
<<
    if (eng != null){

```

```

        $EXEC_PARALLEL( RemoveEngagementAct, eng.getId() );
    }
    >>

FINAL END_STATE:
    <<
    >>

```

See the description of the negotiation `EngagementInitResNeg` in 3.2.4.

## 6.6 WWW Negotiation

The WWW negotiation of agents in the implemented MAS is given below. The states of the negotiation are described in 3.2.5 in pseudo-code and depicted in the

Figure 5 on the page 66.

```

WWW_NEGOTIATION
NEG_INIT
    <<
        TimeTable tt = $GET_BEL(timeTable);
        Consultation[] cons = null;
        TextInput purposeInput = null;
        boolean[] selected = null;
        String desc = null;
        String[] items = null;
        String[] bNames = null;
        String[] selItems = null;
    >>

START:
    <<
        cons = tt.getConsultations();
        if (cons.length == 0){
            $STATE(NO_CONSULTATIONS);
        }
        else{
            $STATE(WHEN);
        }
    >>
WHEN:
    <<
        desc = "When would you like to have consultation(s)?"
        items = new String[cons.length];
        for (int i=0; i<cons.length; i++){
            Date start = cons[i].getStart();
            SimpleDateFormat sdf =
                new SimpleDateFormat("dd.MM.yyyy HH:mm");
            items[i] = sdf.format(start);
        }
        bNames = new String[] {"choose", "cancel"};
        selected = new boolean[items.length];
        MultipleSelection consSel =
            $WWW_GET_CHECK_BOXES(desc, items, bNames, selected);
        if (consSel.getButtonIndex() == 1){
            $STATE(CANCEL);
        }
        else{
            selItems = consSel.getSelectedItems();
            if (selItems.length > 0){
                selected = consSel.getSelection();
                $STATE(PURPOSE);
            }
            else{

```

```

        $STATE(CANCEL);
    }
}
>>
PURPOSE:
<<
    desc = "Enter the short description of " +
           "the consultation purpose.";
    bNames = new String[] { "ok", "back", "cancel" };
    purposeInput = $WWW_GET_LONG_TEXT(desc, bNames, "");
    if ( purposeInput.getButtonIndex() == 2 ){
        $STATE(CANCEL);
    }
    else if ( purposeInput.getButtonIndex() == 1 ){
        $STATE(WHEN);
    }
    else{
        $STATE(WHO);
    }
>>
WHO:
<<
    desc = "Enter your name and your student Id.";
    bNames = new String[] { "ok", "back", "cancel" };
    TextInput whoInput =
        $WWW_GET_ONE_LINE_TEXT(desc, bNames, "");
    if ( whoInput.getButtonIndex() == 2 ){
        $STATE(CANCEL);
    }
    else if ( whoInput.getButtonIndex() == 1 ){
        $STATE(PURPOSE);
    }
    else{
        try{
            tt.addStudentToCons(cons,
                               selected,
                               purposeInput.getText().trim(),
                               whoInput.getText().trim());

            $STATE(DONE);
        }
        catch (TimeTableException tte){
            $STATE(FAILURE);
        }
    }
>>
FAILURE:
<<
    String text = "I'm sorry. In the meantime the consultation" +
                 " times have been changed. " +
                 "Would you like to try to appoint your " +
                 "consultation(s) again?";
    bNames = new String[] { "yes", "no" };
    ButtonSelection failureSel = $DISPLAY_TEXT(text, bNames);
    if (failureSel.getButtonIndex() == 1){
        $STATE(CANCEL);
    }
    else{
        $STATE(START);
    }
>>
FINAL DONE:
<<
    $WWW_DISPLAY_TEXT("OK. Have a nice day.");
>>
FINAL NO_CONSULTATIONS:
<<
    $WWW_DISPLAY_TEXT("Sorry, there will be no consultations " +
                     "in the near future. Try later.");

```

```

>>
FINAL CANCEL:
<<
  $WWW_DISPLAY_TEXT("Appointment has not been made. Bye.");
>>

```

See the description of this WWW negotiation in 3.2.5.

## 6.7 Initialization

The last part an AJA agent program is the initialization part. The initialization is executed only once, immediately after the agent program has been started.

The initialization part of the PDA agent in the implemented MAS does the following:

If the backup file of the `timeTable` belief exists, then the belief `timeTable` gets the value read from the file.

Else

the belief `timeTable` gets a new instance of the class `demo.TimeTable` as its value.

If the backup file of the `eventAlertTime` belief exists, then the belief `eventAlertTime` gets the value read from the file.

If the backup file of the `consultationDuration` belief exists, then the belief `consultationDuration` gets the value read from the file.

Else

off-line training of the neural network is called and the belief `consultationDurationToBackup` gets the value `true`.

The action `doGUIAct` is invoked in a new thread.

The initialization source code is given below:

```

INITIALIZATION
<<
  File timeTableBackupFile =
    new File( $AG_NAME + "_timetablebackup.dat");
  if (timeTableBackupFile.exists()){
    try{
      $SET_BEL(timeTable,
        TimeTable.restoreFrom(timeTableBackupFile));
    }
    catch (IOException e){
      System.err.println("INITIALIZATION: Cannot read " +
        "timeTable backup file: " + e);
      e.printStackTrace();
      System.exit(-1);
    }
  }
  else{
    $SET_BEL(timeTable, new TimeTable($AG_RMI_URL, $AG_JFRAME));
  }

  File eventAlertTimeBackupFile =
    new File( $AG_NAME + "_eventalerttimebackup.dat");
  if (eventAlertTimeBackupFile.exists()){
    try{
      $AP_FROM_FILE(eventAlertTime,
        $AG_NAME + "_eventalerttimebackup.dat");
    }
  }

```

```

    }
    catch (IOException e){
        System.err.println("INITIALIZATION: Cannot read " +
            "eventAlert backup file: " + e);
        e.printStackTrace();
        System.exit(-1);
    }
}

File consultationDurationBackupFile =
    new File( $AG_NAME + "_consultationdurationbackup.dat");
if (consultationDurationBackupFile.exists()){
    try{
        $DV_FROM_FILE(consultationDuration,
            $AG_NAME + "_consultationdurationbackup.dat");
    }
    catch (IOException e){
        System.err.println("INITIALIZATION: Cannot read " +
            "consultationDuration backup file: " + e);
        e.printStackTrace();
        System.exit(-1);
    }
}
else{
    System.out.println("consultationDuration: starting off-line " +
        "learning. Please wait...");
    double error = $DV_OFFLINE_TRAINING(consultationDuration,
        500,
        5);
    System.out.println("consultationDuration: off-line learning " +
        "finished with average error: " +
        error + "%.");
    $SET_BEL(consultationDurationToBackup, true);
}
$EXEC_PARALLEL(doGUIAct);
>>

```

## 7 Related Work

HOMAGE .....	125
JACK .....	125
LASSMachine and LASS.....	125
COOL .....	125
Subsumption Architecture .....	126
AI in Programming Language.....	126

This chapter describes the work related to this thesis.

## 7.1 HOMAGE

HOMAGE [81] (briefly described in 2.7) is a multi-language agent development environment. HOMAGE has two programming levels. The higher level is agent-oriented and the lower level is the object-oriented one consisting of C++ objects, Common Lisp parts, and/or Java objects.

The idea of having two programming levels instead of just one, originally used in HOMAGE, is also adopted in this thesis.

In AJA there are two similar programming levels just like in HOMAGE. The higher programming level in AJA consists of the programming language HADL, which is used for the description of the main agent parts. The lower level is object-oriented one, like in HOMAGE. It consists of the programming language Java extended with the constructs for the accessing agent parts defined in HADL. This extended Java is named Java+.

## 7.2 JACK

JACK [47], [23] (briefly described in 2.5) is an agent-oriented development environment, which extends Java with new, agent-oriented constructs.

In AJA, namely in Java+, Java is also extended with the new language constructs. However, JACK and AJA have nothing more in common. Jack agents are BDI (Belief-Desire-Intention) agents, whereas the architecture of an AJA agent is not a BDI architecture.

## 7.3 LASSMachine and LASS

The experience gained in the design and implementation of the Java package for agent programming LASSMachine [5] was very helpful in the creation of AJA. The purpose of LASSMachine Java package was to be used in the implementation of the object-oriented language LASS [10], [11]. However, the implementing a language LASS, which is completely independent of Java, was abandoned due to the reasons described in 3.1.1. Nevertheless, several classes in the package LASSMachine have been reused in the AJA implementation. Furthermore, the architecture of AJA agent has two parts that have been already used in the LASS agent architecture: beliefs and reflexes (reflexes are called behaviors in LASS).

## 7.4 COOL

COOL (COOrdination Language) [13] (briefly described in 2.8) is an agent-programming language that introduces the term conversation between agents. A conversation in COOL includes several message sending and message receiving. It is represented as an automaton.

This solution is also adopted in AJA. A negotiation construct in AJA corresponds to a conversation in COOL. AJA negotiation is also represented as automaton. In AJA however a state in automaton has slightly different semantic that it has in COOL. In AJA a state in the negotiation automaton is not strictly bound to message receiving and message sending like it is in COOL. AJA negotiation states are more flexible. They simply divide the negotiation into the logical parts that correspond to various states in the negotiation process.

## 7.5 Subsumption Architecture

Subsumption architecture [18], [19], [20] (see also 1.5.2.1) gave surprisingly good results in robotics. It consists of hierarchically organized behaviors, which are the only components controlling the robot.

AJA reflexes are similar to behaviors, but there are some important differences:

- Behaviors are triggered by sensors, while AJA reflexes are triggered by boolean conditions written in Java.
- In the subsumption architecture there is no internal representation of the world, while AJA agents have beliefs, which store their world models.
- Behaviors are the only control mechanism in the subsumption architecture, while AJA reflexes and AJA negotiations jointly control AJA agents.

## 7.6 AI in Programming Language

As already mentioned in 3.1.3, Steve Schoepke views intelligent agents as a vehicle for AI-related technologies in the mainstream programming. His papers [89] and [88] were the inspiration for including AI components into the AJA tool. At the time being, in the first version of AJA, which is implemented and described in this thesis, there are two AI components included:

- dependant values (i.e. neural networks), and
- adaptable parameters.

In the future work, however, other AI components can be supported as well.



## 8 Conclusion and Future Work

The Goal of the Thesis.....	128
The Work Done.....	128
Future Work .....	129

## 8.1 The Goal of the Thesis

The goal of this thesis was to create an agent development tool with the following features:

- The tool includes new agent-programming language for the higher-level agent specification, but Java is also used for the agent programming.
- The tool provides agent-programming infrastructure and thus it enables agent programmers to concentrate on the programming of agent business logic.
- The tool introduces AI components at the agent-programming language level and thus facilitates the application of AI technologies even if the programmer is not familiar with AI.
- The tool supports conversation-like inter-agent communication. It provides constructs that group several logically related message sending and message receiving together.
- The messages sent and received in the communication between agents can be digitally signed or encrypted via SSL.
- The tool provides reactive constructs similar to behaviors in the subsumption architecture.
- At last but not least: the tool does not only present an original approach to agent programming but it is also powerful enough to be used in commercial projects.

## 8.2 The Work Done

In the practical part of this thesis the agent-development tool AJA has been created. AJA consists of two programming languages:

- HADL – higher-level language used for the description of the main agent parts.
- Java+ – lower-level language that extends Java and is used for the programming of the agent parts defined in HADL.

AJA agents have the following features:

- Agent communicates with other agents using a construct called negotiation. The messages sent can be encrypted or digitally signed in order to ensure the security of the system.
- Agent possesses adaptable parameters and neural nets that adapt themselves when the environment changes.

- Agent has reflexes, which are the reactive component of the agent architecture.
- Agent can perform its actions parallel. Action executions are synchronized.
- Agent is accessible via Internet, because it acts as a simple HTTP server. People can use this way to communicate with an agent.
- Agent has Java Swing based graphical user interface. Its owner uses this interface to communicate with the agent.
- Because of the fact that Java+ language extends Java, it is possible to use all useful Java features in the implementation of AJA agents (e.g. JDBC for the database access).

AJA presents an original approach to integrating artificial intelligence technologies with a programming language.

This thesis starts with an overview of the field of agents and multi-agent systems in the first chapter. The first chapter as well as the second one, which surveys existing agent languages and tools, could be used as introduction to the field. The list of references from the first and the second chapter includes the most significant agent papers. The rest of the thesis describes AJA.

In order to test and to evaluate the tool, a multi-agent system consisting of four PDA agents has been implemented. Four personal digital assistants for appointment scheduling have been completely developed using AJA. To implement these four agents, all AJA constructs had to be used.

The agent implementation process and the performance of the implemented multi-agent system have fulfilled the expectations and shown that AJA is a suitable tool for agent programming.

### **8.3 Future Work**

Although the syntax grammar of HADL and Java+ are completely described in the thesis, the semantics of the language constructs is described only informally with the use of examples. The future work on AJA should include the formal specification of the semantics of all new language constructs.

This thesis presents the first version of AJA. In the future work AJA can be extended and improved first of all by including new AI constructs. For example, a case-based reasoning component can be added, a knowledge base component and an inference engine component, a fuzzy logic component, a planner component, etc. For each new component the corresponding language constructs in HADL and Java+ have to be defined and implemented. With the increase of the number of AI components supported, AJA agents will become more intelligent.

In addition, AJA tool can be implemented as an Integrated Development Environment (IDE). Such an IDE could support visual programming and rapid agent development, in the manner JBuilder, Visual Age, and Visual Cafe support the implementation of Java applications.

## Appendix A - The Implementation of AJA to Java Translator

AJA is implemented in Java 2 Standard Edition, Version 1.4.1 (J2SE v 1.4.1). In order to use AJA, one needs J2SE v 1.4 or newer version.

AJA implementation consists of three Java packages:

- `aja`
- `aja.framework`
- `aja.translator`

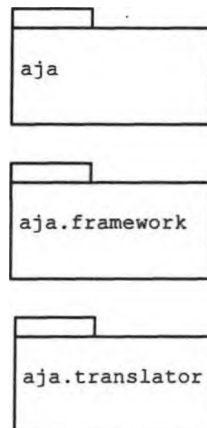


Figure 7 Java packages implementing AJA..  
Slika 7 Java paketi koji implementiraju AJA-u..

There are 97 classes and interfaces defined in the three above packages.

The package `aja` contains only one simple class, which defines the main method of the translator program.

The actual translation, i.e. scanning, parsing, Java code generation, etc., takes place in the package `aja.translator`. A successful translation of an AJA program generates source code files defining new Java classes. The generated classes implement the agent specified in AJA program.

Almost all of the generated classes extends and/or use the classes and interfaces defined in the package `aja.framework`. As it can be concluded from the package name, `aja.framework` consists of interrelated classes and interfaces organized as a framework. The framework implements a generic AJA agent.

This appendix describes the implementation details of AJA to Java translator. The first section of the appendix presents the package `aja.framework` and the generic AJA agent. After that, in the second section, the package `aja.translator` and its most important classes are briefly described. The third section explains the usage of the translator program and its command line arguments.

## aja.framework

Classes and interfaces in the package `aja.framework` implement the generic AJA agent.

An agent is represented as an instance of the class `Agent`, which is the central class in the package. Because of the fact, that the non-encrypted agent-to-agent communication is implemented using Remote Method Invocation (RMI) Java mechanism, the class `Agent` extends the class `java.rmi.server.UnicastRemoteObject` and implements a subinterface of the interface `java.rmi.Remote`.

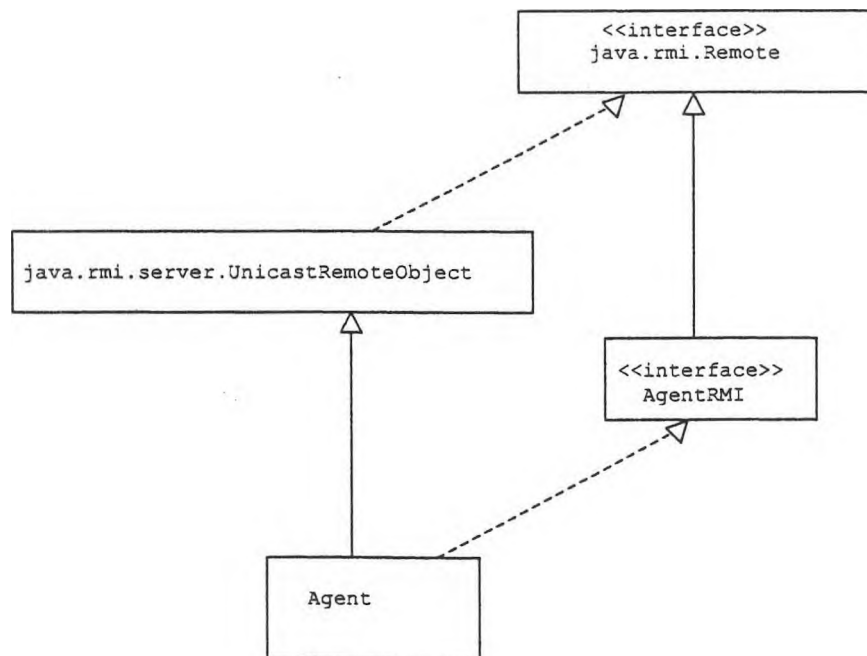


Figure 8 Agent and RMI.  
Slika 8 Agent i RMI.

An AJA agent is relatively complex object. It contains parts that manage agent beliefs, actions, reflexes, negotiations and web negotiation, a part that manages graphical user interface (GUI) and a part that manages secure and insecure agent-to-

agent communication. It would be a bad design decision to implement all these almost independent functionalities in a single class. Therefore, the class `Agent` delegates each task to a class implementing required functionality. There are seven such classes:

- `BeliefsManager` – manages agent beliefs,
- `ActionsManager` – manages agent actions,
- `ReflexesManager` – manages agent reflexes,
- `NegotiationsManager` – manages executions of requesting and responding negotiations,
- `CommunicationsManager` – manages message sending and receiving in agent-to-agent communication.
- `WebManager` – manages web negotiation and HTTP communication with browsers,
- `GUIManager` – manages GUI elements of agent window.

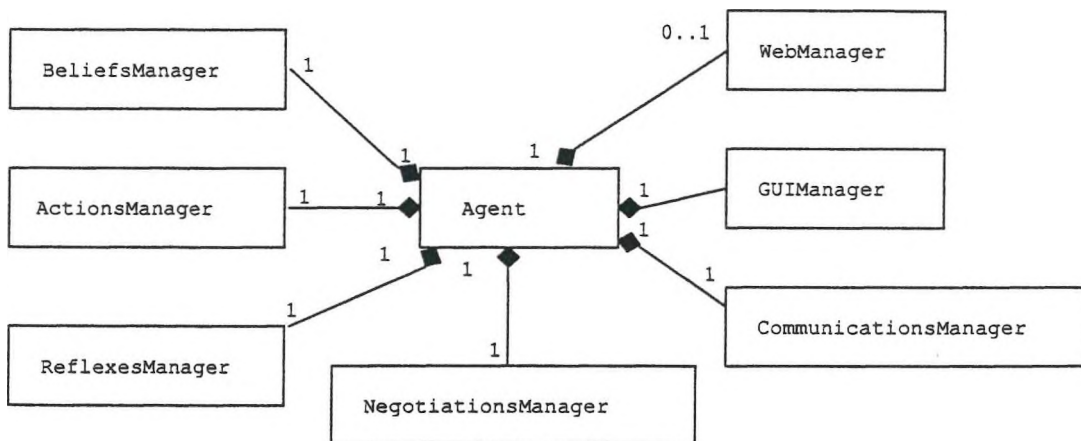


Figure 9 Agent and its managers.  
Slika 9 Agent i njegovi menadžeri.

## Beliefs

Agent beliefs are administered by one instance of the class `BeliefsManager`. This instance stores all agent beliefs in one instance of a `java.util.Hashtable` class.

An abstract class `Belief` is a superclass of four classes that implement particular beliefs types:

- `BeliefPrimitive` – used for primitive type beliefs,

- BeliefReference – used for reference type beliefs,
- BeliefAdaptable – used for adaptable parameter beliefs,
- BeliefDependant – used for dependant beliefs.

The class BeliefAdaptable is implemented using class AdaptableParameter, while the class BeliefDependant uses the class NeuralNetwork, which implements artificial neural network.

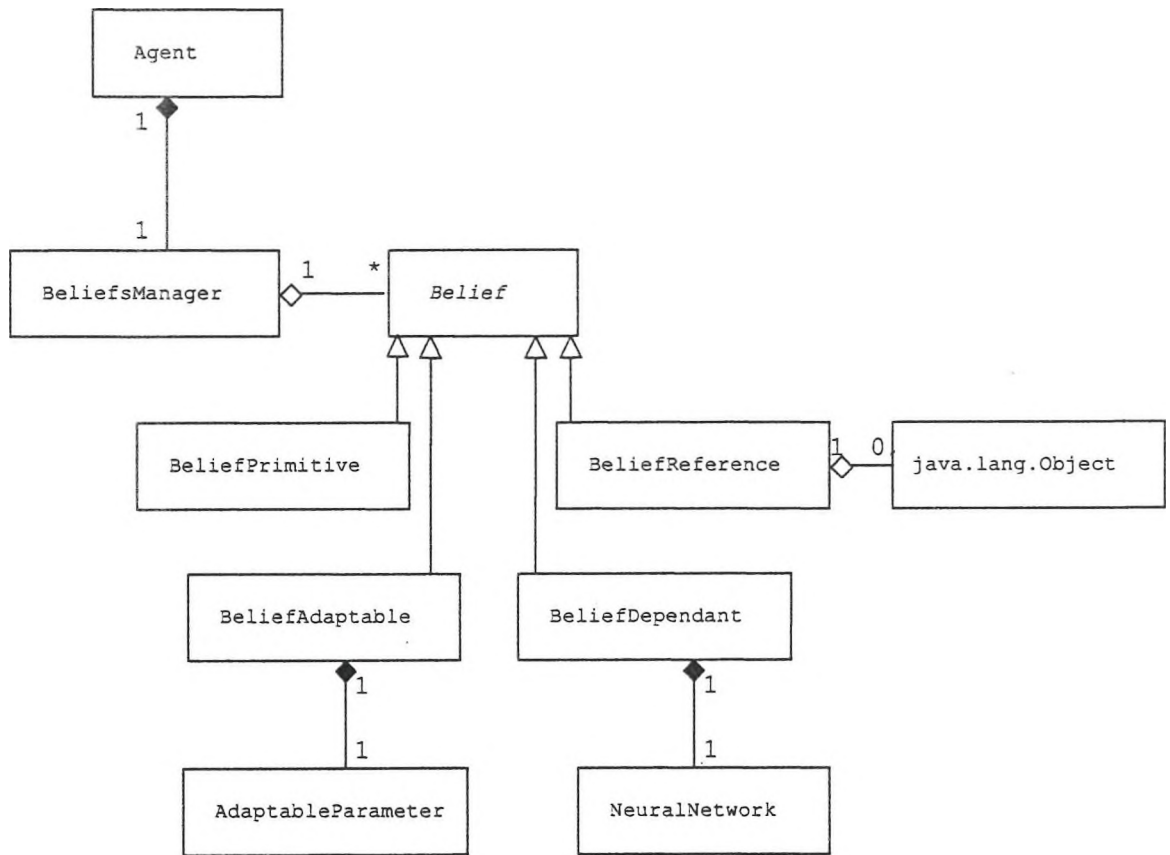


Figure 10 Implementation of agent beliefs.  
Slika 10 Implementacija agentovih verovanja.

## Actions

Actions execution, synchronization, and management are implemented in the class ActionsManager. Actions are represented with the abstract class Action. The subclasses of the class Action are generated by AJA translator.

If an action has to be executed in a new thread of execution, the class ActionExecThread is used.

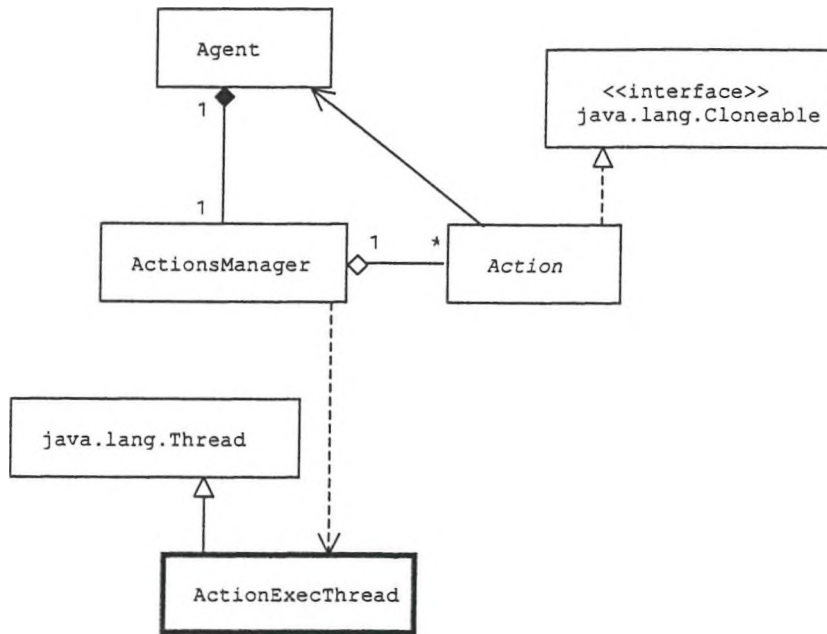


Figure 11 Implementation of agent actions.  
Slika 11 Implementacija akcija agenta.

## Reflexes

The management of agent reflexes is implemented similarly to the management of agent actions. An instance of the class `ReflexesManager` is responsible for reflexes triggering and control. Reflexes are represented with the abstract class `Reflex`. The subclasses of this class are generated by AJA translator.

A reflex execution occurs always in a new thread of execution. `ReflexExecThread` instances represent these threads.



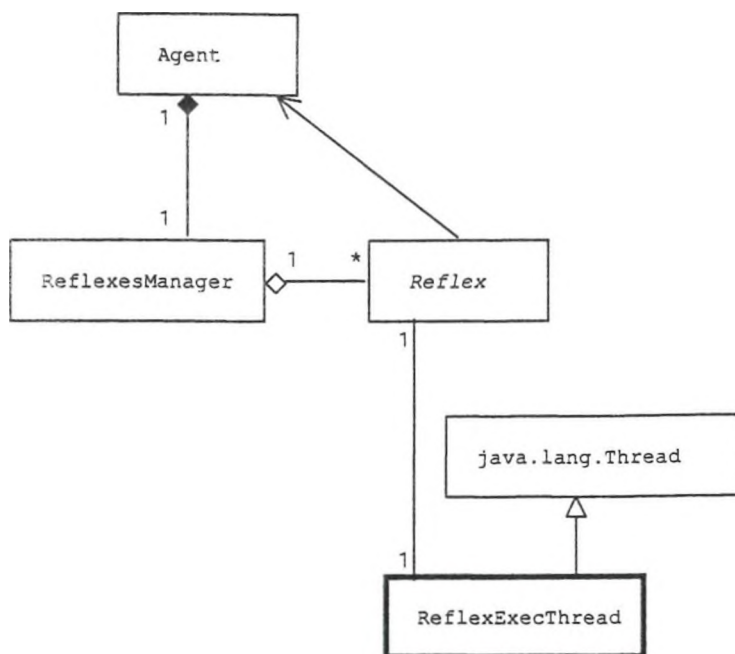


Figure 12 Implementation of agent reflexes.  
Slika 12 Implementacija agentovih refleksa.

## Requesting and Responding Negotiations

The management of requesting and responding negotiations of an agent is a responsibility of an instance of the class `NegotiationsManager`.

Requesting negotiations are subclasses of the abstract class `RequestingNegotiation`. Similarly, responding negotiations are subclasses of the abstract class `RespondingNegotiation`. Common parts of classes `RequestingNegotiation` and `RespondingNegotiation` are defined in their superclass, in the class `Negotiation`.

When a requesting negotiation is to be executed in a new thread of execution, an instance of the class `ReqNegExecThread` is used.

Responding negotiations are always executed in new thread of execution. The class `RespNegExecThread` is used for this purpose.

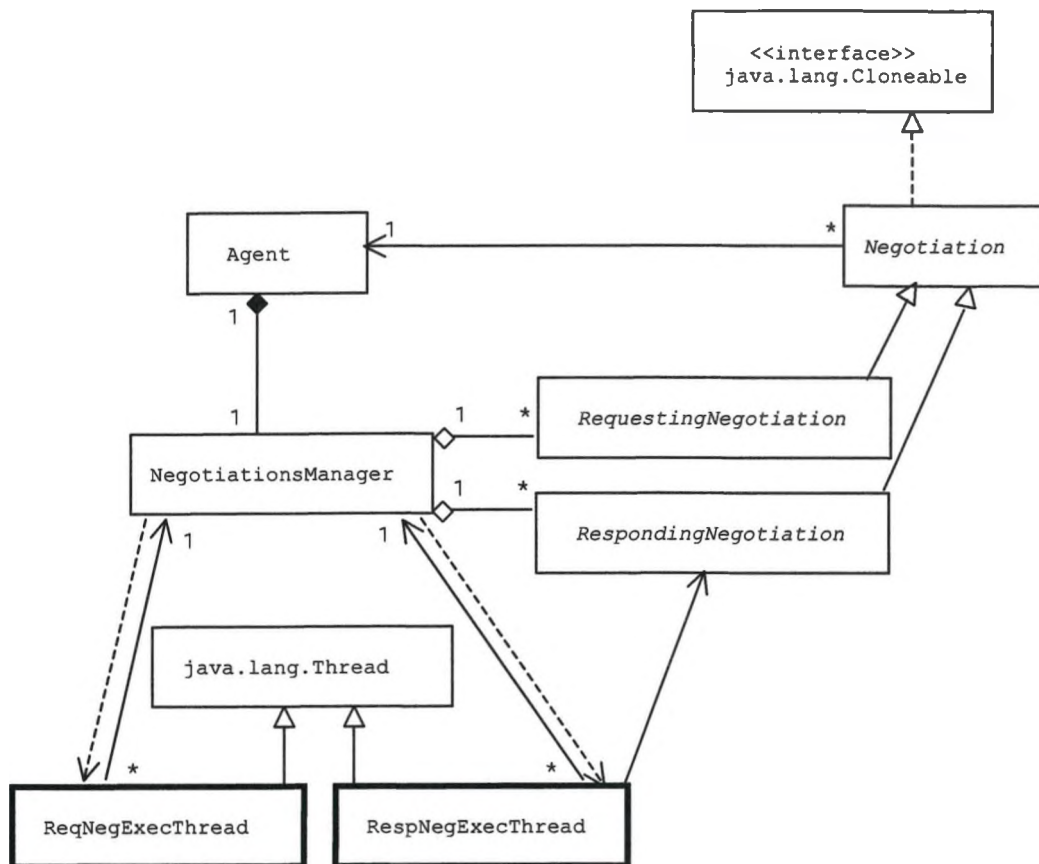


Figure 13 Implementation of requesting and responding negotiations.  
Slika 13 Implementacija zahtevnih i odgovarajućih pregovaranja.

## Agent-to-Agent Communication

Agents send messages to each other during the execution of their negotiations. There are three types of messaging implemented:

- sending and receiving of messages without any security measures,
- sending and receiving of messages that are digitally signed,
- sending and receiving of encrypted messages using SSL protocol.

Insecure and digitally signed messages are transported using RMI. Encrypted messages are sent using SSL sockets.

Both digital signing and SSL communication are supported in J2SE v 1.4.x. One needs no additional Java extensions or other software to work with certificates, key pairs, keystores, truststores, and other security concepts used in AJA.

An instance of the class `CommunicationsManager` manages agent-to-agent messaging in AJA. All messages being sent or received pass through the instance of `CommunicationsManager`.

Instances of the class `MessageData` represent messages and their meta-data. Both type of messages are represented with `MessageData`: messages being sent and messages being received.

An agent can be involved in more than one communication sessions with other agents. `CommunicationsManager` administers these sessions. Sometimes however, when e.g. the network goes down, a session ends abruptly without notifying `CommunicationsManager` that the session no longer exists. To avoid memory leak, one thread periodically removes old sessions data. This thread is an instance of the class `OldSessionsRemoverThread`.

The classes `SSLServer`, `SSLServerThread`, and `SSLCommunicationThread` enable receipt of encrypted messages. When such a message is received, `SSLCommunicationThread` instance hands it over to the `CommunicationsManager` instance.

The SSL communication used in AJA includes two-way authentication. Not only message receiver has to authenticate itself to a sender, but the sender has also authenticate itself to the receiver. Consequently, both agents have to contain the certificate of another agent in their truststores in order to communicate through SSL with two-way authentication.

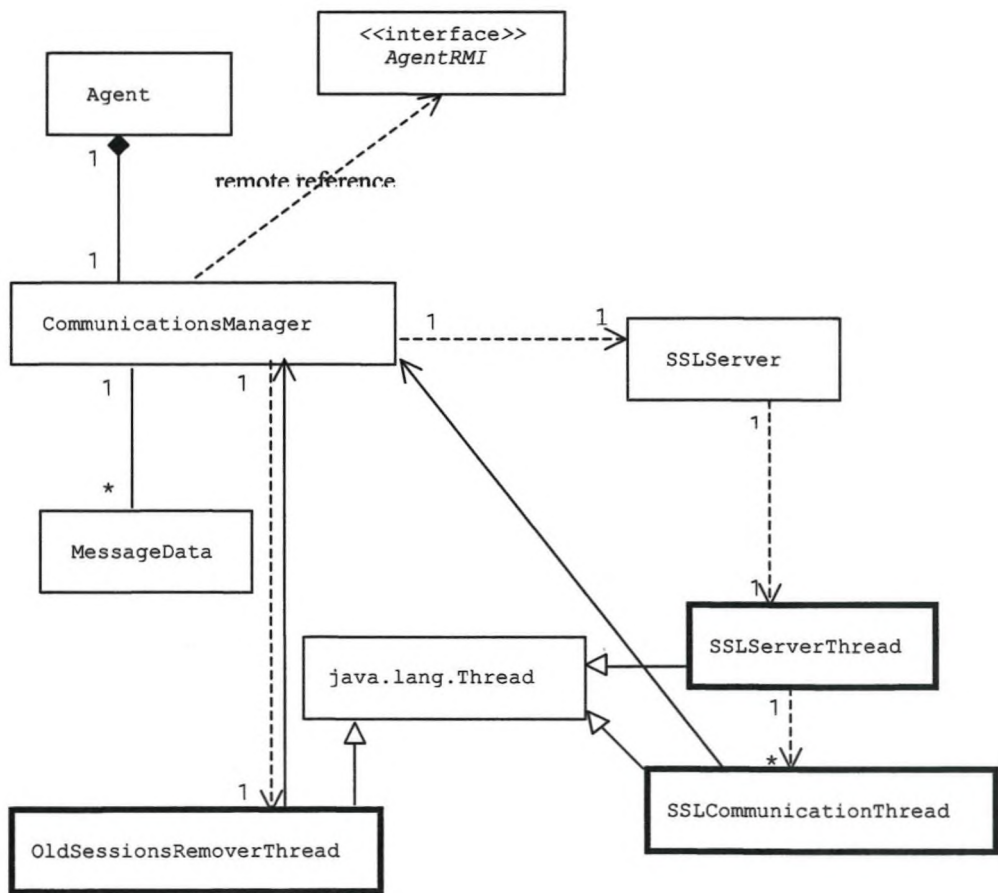


Figure 14 Implementation of agent-to-agent communication.

Slika 14 Implementacija međuagentske komunikacije.

## WebNegotiation

When an agent executes a web negotiation, it communicates with an Internet browser using HTTP 1.1 protocol [54]. If there are  $n$  web users that communicate with the agent at the same time, then there are  $n$  web negotiation instances executing concurrently.

The central class responsible for the executions of web negotiation is the class `WebManager`. An instance of this class starts two new threads:

- a thread of the type `WebServerThread` and
- a thread of the type `WebSweeperThread`.

The `WebServerThread` instance accepts connection requests from browsers and starts the instances of `WebNegThread`. Each `WebNegThread` instance communicates only with one Internet browser.

Low-level HTTP communication and HTML generation is implemented in the abstract class `WebNegotiation`. There is only one abstract method in this class, the method `execute`.

Due to request-respond nature of the HTTP protocol, there is a need to store the data about particular web negotiation execution after a respond has been sent to browser and to restore the stored data after a new request from the same browser is received. In order to differentiate the request belonging to particular web negotiation execution from the requests coming from other browsers, each web negotiation execution has a session id assigned. Session id is sent to browser as a hidden field in HTML form.

An instance of the class `WebSessionsStorage` stores the web session data of all currently executing web negotiations that are waiting for the next browser request.

In order to avoid a memory leak in cases when web user suddenly stops to communicate with the agent, a thread of the type `WebSweeperThread` is used to periodically clean up the web sessions storage.

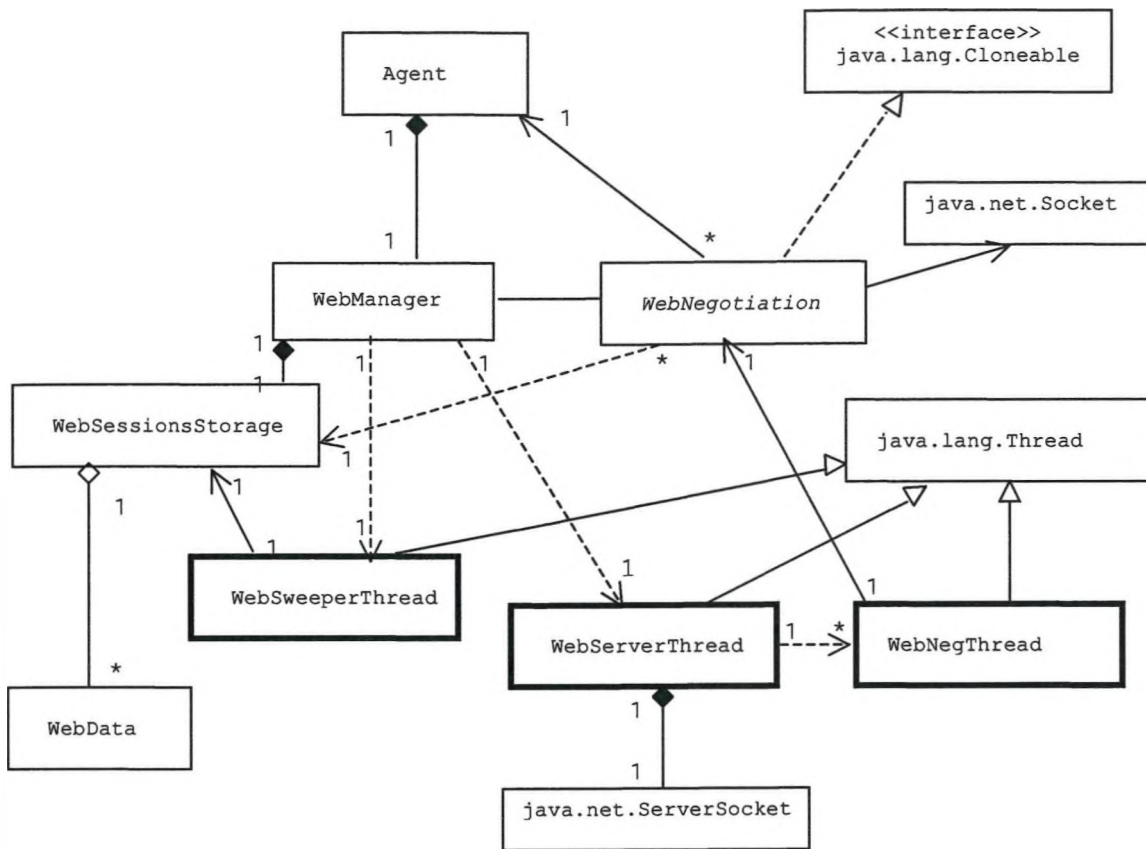


Figure 15 Implementation of web negotiation.  
 Slika 15 Implementacija web pregovaranja.

## Built-in GUI

AJA offers high-level Java+ constructs for GUI communication with agent user. The implementation of these constructs is located in the class `GUIManager`. The implementation is based on the standard Java package `javax.swing`.

The use of built-in AJA GUI is not obligatory. A reference of agent window can be obtained with the Java+ construct `$AG_JFRAME`. The type of the reference is `javax.swing.JFrame`. Using this reference, another swing-based GUI can be implemented.

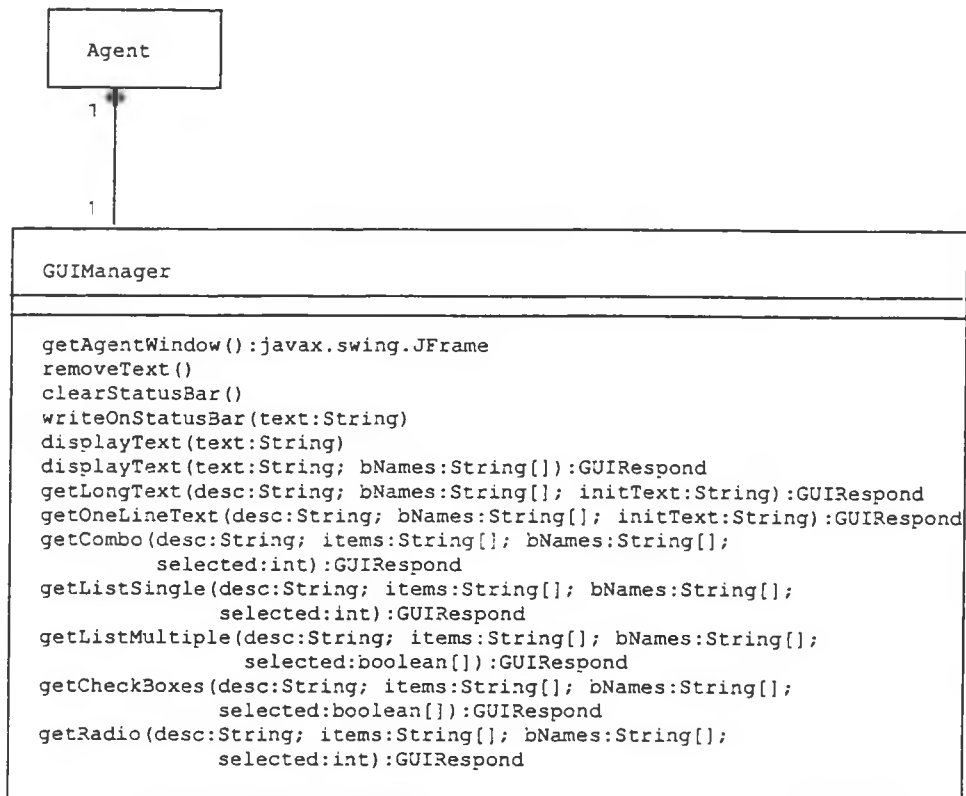


Figure 16 Implementation of built-in agent GUI.  
 Slika 16 Implementacija ugrađenog grafičkog korisničkog interfejsa agenta.

## aja.translator

The package `aja.translator` contains classes that scan and parse AJA agent program, create an internal representation of the program and generates the source code files of Java classes that together with the classes in the package `aja.framework` implement the given AJA agent program.

Figure 17 shows some of the classes in the package.



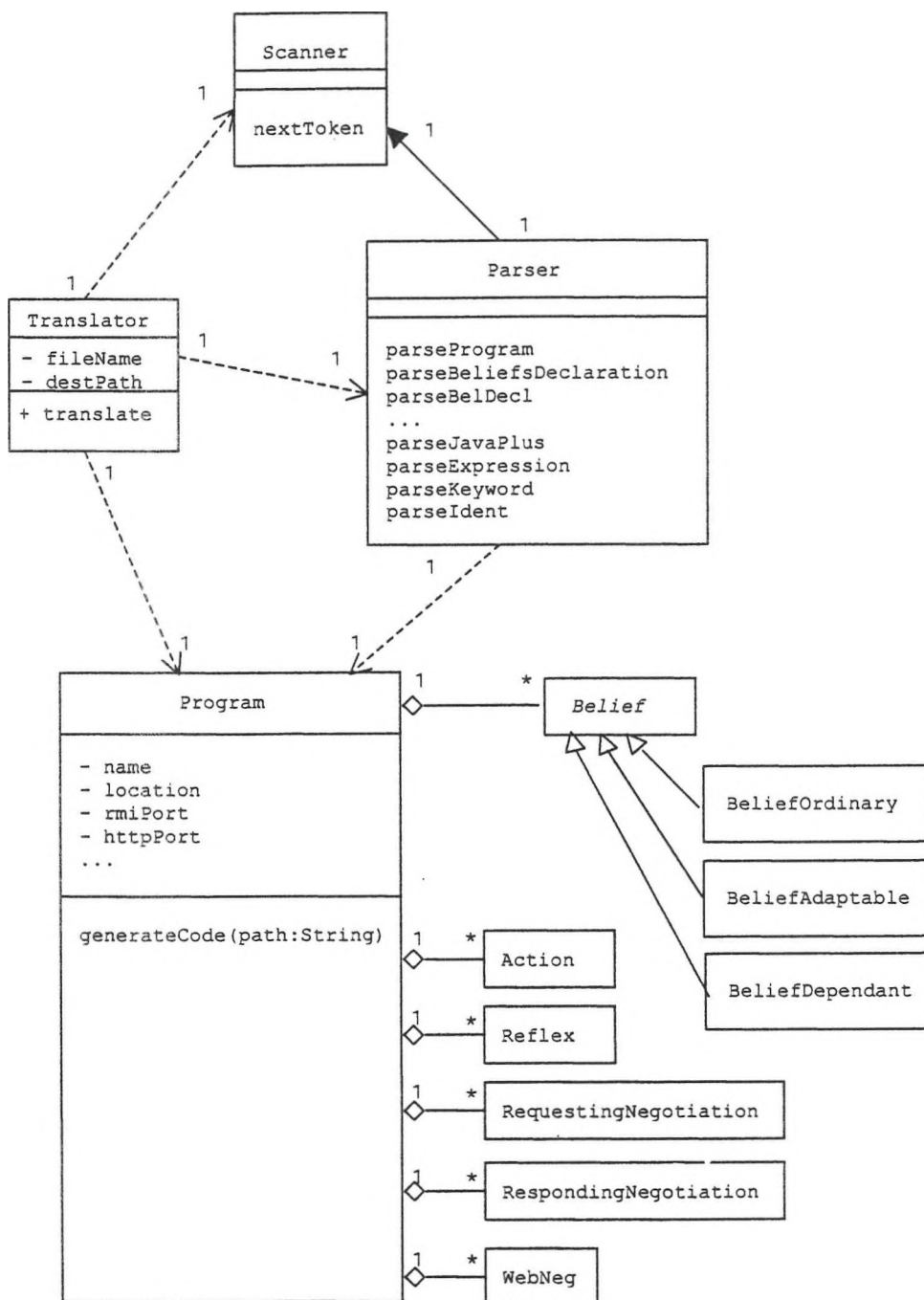


Figure 17 Some of the classes in the package aja.translator.

Slika 17 Neke od klasa u paketu aja.translator.

The translation starts in the method `translate` of the class `Translator`. The method `nextToken` in `Scanner` class returns tokens, which are then used in `Parser` class to parse the program using recursive-descent.

As a result of successful parsing, an internal representation of the program is generated. The internal representation is a tree-like structure with an instance of `Program` class as a root node. Remaining nodes are the instances of the classes

BeliefOrdinary, BeliefAdaptable, BeliefDependant, Action, Reflex, RequestingNegotiation, etc.

Most classes in the package `aja.translator` have the same names as their counterparts in the package `aja.framework`.

The method `generateCode` in `Program` class generates the java source code files.

## Translation

The main method of the translator program is located in the class `aja.translate`.

The following command line syntax is used to translate an AJA program:

```
java [VMoptions] aja.translate [-d destDir] file.aja
```

If the optional parameter destination directory is given, then the generated files will be stored in specified directory. Otherwise, they will be stored in current directory.



### Installation and the Directory Structure

To install AJA 1.0 one need only to extract the file **AJA\_1.0.zip**. If the JDK 1.4 or higher is installed on the computer, AJA 1.0 can be compiled and used.

The file **AJA\_1.0.zip** contains all AJA 1.0 files. After extracting the archive, the following directory structure will be created:

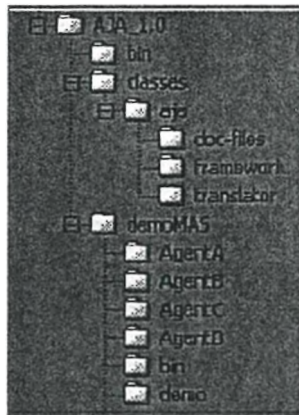


Figure 18 The directory structure after the extraction of **AJA\_1.0.zip**.  
Slika 18 Struktura direktorijuma nakon raspakovanja fajla **AJA\_1.0.zip**.

The directory **AJA\_1.0** contains two files:

- **AJA\_1.0\readme.txt** – briefly describes the files and directories in the archive.
- **AJA\_1.0\guide.pdf** – describes in details the usage of AJA 1.0 and implemented demo MAS.

In the directory **AJA\_1.0\bin** are MS Windows/DOS script files used to compile all AJA parts and to generate HTML javadoc documentation:

- **AJA\_1.0\bin\setJAVABIN.bat** – this batch file is called in all other AJA batch files in the archive. Edit this file and set the appropriate path of your JDK 1.4 bin directory as a value of the **JAVABIN** environment variable. The preset value is **C:\j2sdk1.4.1\bin**.

- **AJA\_1.0\bin\1\_comp.f.bat** – compiles the java package `aja.framework`.
- **AJA\_1.0\bin\2\_genstub.bat** – generates a RMI stub class used in the RMI communication between AJA Agents.
- **AJA\_1.0\bin\3\_compt.bat** – compiles the java package `aja.translator`.
- **AJA\_1.0\bin\4\_compmain.bat** – compiles the AJA translator program.
- **AJA\_1.0\bin\5\_gendoc.bat** – generates HTML javadoc documentation. The documentation files will be in the new directory **AJA\_1.0\doc**. The file **AJA\_1.0\doc\index.html** is the starting page of the documentation.

One can use these batch files as a template to write the appropriate script files for UNIX or some other operating system, if other operating system than MS Windows is used.

**AJA\_1.0\classes** directory contains subdirectories with all AJA source files, i.e. `.java` files. After the compilation of the AJA source files, the generated binary files, i.e. `.class` files, will also be located here.

- the directory **AJA\_1.0\classes\aja\framework** contains the files belonging to the package `aja.framework`.
- the directory **AJA\_1.0\classes\aja\translator** contains the files belonging to the package `aja.translator`.
- the directory **AJA\_1.0\classes\aja\doc-files** contains HADL grammar and the description of all the Java+ elements. These files are used in the generation of javadoc documentation.
- the directory **AJA\_1.0\classes\aja** contains the files belonging to the package `aja`. In this package is only the main program of the AJA translator.

**AJA\_1.0.zip** file contains an example MAS implemented in AJA, which demonstrates many AJA features. The files belonging to this example are located in the directory **AJA\_1.0\demoMAS**.

## Preparing AJA for the First Use

1. Edit the file **AJA\_1.0\bin\setJAVABIN.bat** and set the appropriate path of your JDK 1.4 bin directory as a value of the **JAVABIN** environment variable. The preset value is **C:\j2sdk1.4.1\bin**.

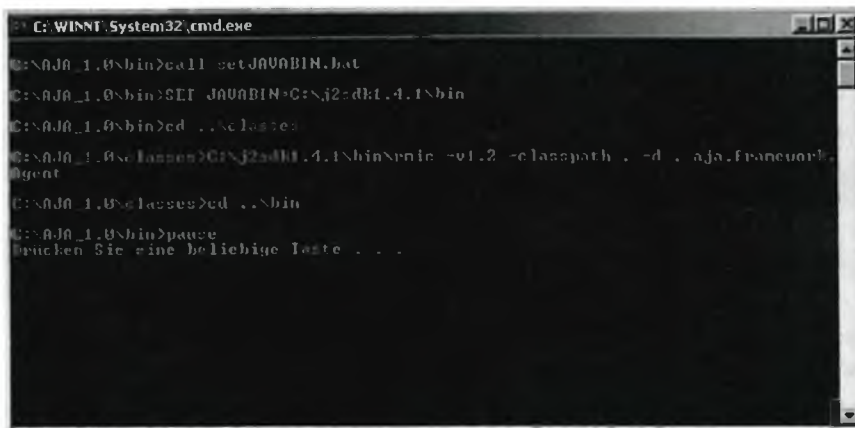
- Execute the batch file **AJA\_1.0\bin\1\_comp.f.bat** in order to compile the package `aja.framework`. The file can be executed e.g. by double-clicking its icon in the Windows-Explorer.



```
C:\WINNT\System32\cmd.exe
C:\AJA_1.0\bin>call setJAVABIN.bat
C:\AJA_1.0\bin>SET JAVABIN=C:\jdk1.4.1\bin
JAVABIN=C:\jdk1.4.1\bin
C:\AJA_1.0\bin>cd ..\classes\aja\framework
..\classes\aja\framework
C:\AJA_1.0\classes\aja\framework>C:\jdk1.4.1\bin\javac -d ..\..\sourcefiles\src
src
C:\AJA_1.0\classes\aja\framework>cd ..\..\bin
..\..\bin
C:\AJA_1.0\bin>pause
Drücken Sie eine beliebige Taste . . .
```

Figure 19 The file **AJA\_1.0\bin\1\_comp.f.bat** has been executed.  
Slika 19 Datoteka **AJA\_1.0\bin\1\_comp.f.bat** je izvršena.

- Execute the batch file **AJA\_1.0\bin\2\_genstub.bat** in order to generate a RMI stub class for the class `aja.framework.Agent`.



```
C:\WINNT\System32\cmd.exe
C:\AJA_1.0\bin>call setJAVABIN.bat
C:\AJA_1.0\bin>SET JAVABIN=C:\jdk1.4.1\bin
JAVABIN=C:\jdk1.4.1\bin
C:\AJA_1.0\bin>cd ..\classes
..\classes
C:\AJA_1.0\classes>C:\jdk1.4.1\bin\javac -cp ..\..\sourcefiles\src -d ..\..\bin
src
C:\AJA_1.0\classes>cd ..\bin
..\bin
C:\AJA_1.0\bin>pause
Drücken Sie eine beliebige Taste . . .
```

Figure 20 The file **AJA\_1.0\bin\2\_genstub.bat** has been executed.  
Slika 20 Datoteka **AJA\_1.0\bin\2\_genstub.bat** je izvršena.

- Execute the batch file **AJA\_1.0\bin\3\_compt.bat** in order to compile the package `aja.translator`.

```

C:\WINNT\System32\cmd.exe
C:\AJA_1.0\bin>call setJAUABIN.bat
C:\AJA_1.0\bin>SET JAUABIN=C:\j2sdk1.4.1\bin
C:\AJA_1.0\bin>cd ..\classes\aja\translator
C:\AJA_1.0\classes\aja\translator>C:\j2sdk1.4.1\bin\javac -d ..\.. \sourcefiles.txt
C:\AJA_1.0\classes\aja\translator>cd ..\..\bin
C:\AJA_1.0\bin>pause
Drücken Sie eine beliebige Taste . . .

```

Figure 21 The file **AJA\_1.0\bin\3\_compt.bat** has been executed.  
 Slika 21 Datoteka **AJA\_1.0\bin\3\_compt.bat** je izvršena.

- Execute the batch file **AJA\_1.0\bin\4\_compmain.bat** in order to compile the translator main program (i.e. the class `aja.translate`).

```

C:\WINNT\System32\cmd.exe
C:\AJA_1.0\bin>call setJAUABIN.bat
C:\AJA_1.0\bin>SET JAUABIN=C:\j2sdk1.4.1\bin
C:\AJA_1.0\bin>cd ..\classes\aja
C:\AJA_1.0\classes\aja>C:\j2sdk1.4.1\bin\javac -classpath .. -d .. translate.java
C:\AJA_1.0\classes\aja>cd ..\..\bin
C:\AJA_1.0\bin>pause
Drücken Sie eine beliebige Taste . . .

```

Figure 22 The file **AJA\_1.0\bin\4\_compmain.bat** has been executed.  
 Slika 22 Datoteka **AJA\_1.0\bin\4\_compmain.bat** je izvršena.

- To run the example MAS one need not the AJA documentation. However, the documentation will be useful later, when one start to develop his/her own MAS with AJA. Executing the batch file **AJA\_1.0\bin\5\_gendoc.bat** generates the AJA documentation.



Figure 23 The file `AJA_1.0\bin\5_gendoc.bat` has been executed.  
 Slika 23 Datoteka `AJA_1.0\bin\5_gendoc.bat` je izvršena.

A generated file `AJA_1.0\doc\index.html` is the starting page of the documentation.

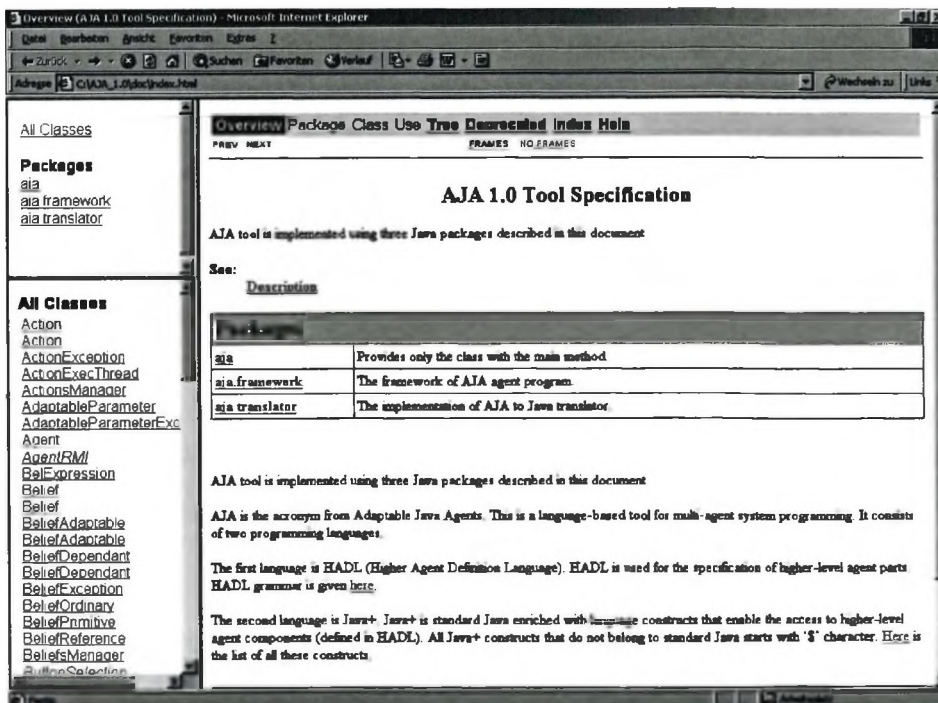


Figure 24 HTML documentation of AJA classes.  
 Slika 24 HTML dokumentacija AJA klasa.



## Appendix C - How to Translate, Compile, Run, and Use the Example Agents

### Agents in the System and the Locations of their Files

There are four personal digital assistant (PDA) agents in the example MAS. It is a homogenous MAS, i.e. all four PDA agents are equal. They only differ in their names and URLs.

The first agent is named **A** and the source code of its AJA program is in the directory `AJA_1.0\demoMAS\AgentA`.

Similarly, the source codes of the agents **B**, **C**, and **D** are in the directories `AJA_1.0\demoMAS\AgentB`, `AJA_1.0\demoMAS\AgentC`, and `AJA_1.0\demoMAS\AgentD` respectively.

Since AJA supports the use of java in the implementation of AJA agents, a java package `demo` is made and used in the implementation of the example agents. The files belonging to the package `demo` are placed in the directory `AJA_1.0\demoMAS\demo`.

The directory `AJA_1.0\demoMAS\bin` contains fifteen batch files used to translate, compile and run the agents. In order to do this, AJA classes has to be compiled first.

### Starting the Agents

There are fifteen batch files in the directory `AJA_1.0\demoMAS\bin`. The name of each batch file starts with a number between 01 and 15. These numbers determine the order of the batch files execution:

1. As first, execute the batch file `AJA_1.0\demoMAS\bin\01_compDemo.bat`. A batch file can be executed e.g. by double-clicking its icon in the Windows-Explorer. The file `01_compDemo.bat` compiles the java package `demo`, which is used in the implementation of the example agents.

```

C:\WINNT\System32\cmd.exe
C:\AJA_1.0\demoMAS\bin>call ..\..\bin\setJAVABIN.bat
C:\AJA_1.0\demoMAS\bin>SET JAVA_BIN=C:\j2sdk1.4.1\bin
C:\AJA_1.0\demoMAS\bin>cd ..\demo
C:\AJA_1.0\demoMAS\demo>C:\j2sdk1.4.1\bin\javac -d .. @resourcefiles.txt
C:\AJA_1.0\demoMAS\demo>cd ..\bin
C:\AJA_1.0\demoMAS\bin>pause
Drücken Sie eine beliebige Taste . . .

```

Figure 25 `AJA_1.0\demoMAS\bin\01_compDemo.bat` has been executed.  
 Slika 25 Datoteka `AJA_1.0\demoMAS\bin\01_compDemo.bat` je izvršena.

- Execute the batch file `AJA_1.0\demoMAS\bin\02_createKeys.bat` in order to create keystores for agents, to create public-private key pairs and to exchange certificates among agents. All messages that agent exchange among themselves are digitally signed.

```

C:\WINNT\System32\cmd.exe
C:\AJA_1.0\demoMAS\bin>C:\j2sdk1.4.1\bin\keytool -import -noprompt -alias local1
est_1099_D -keystore ..\AgentD\C.ke -storepass ckspass -file ..\AgentD\Ccertfile
.cer
Zertifikat wurde zu Keystore hinzugefügt.
C:\AJA_1.0\demoMAS\bin>REM import other certificates in ..\AgentD\D.ke
C:\AJA_1.0\demoMAS\bin>C:\j2sdk1.4.1\bin\keytool -import -noprompt -alias local1
est_1099_B -keystore ..\AgentD\D.ke -storepass dkspass -file ..\AgentD\Dcertfile
.cer
Zertifikat wurde zu Keystore hinzugefügt.
C:\AJA_1.0\demoMAS\bin>C:\j2sdk1.4.1\bin\keytool -import -noprompt -alias local1
est_1099_C -keystore ..\AgentD\D.ke -storepass dkspass -file ..\AgentD\Ccertfile
.cer
Zertifikat wurde zu Keystore hinzugefügt.
C:\AJA_1.0\demoMAS\bin>PAUSE
Drücken Sie eine beliebige Taste . . .

```

Figure 26 `AJA_1.0\demoMAS\bin\02_createKeys.bat` has been executed.  
 Slika 26 Datoteka `AJA_1.0\demoMAS\bin\02_createKeys.bat` je izvršena.

- Execute the batch file `AJA_1.0\demoMAS\bin\03_transA.bat` in order to translate the AJA source code of the agent A into Java.

```

C:\WINNT\System32\cmd.exe
... FindReplacementReqNeg
... EngagementInitReqNeg
... RequestedInfoReqNeg
processing responding negotiations:
... InfoReqNeg
... GetBirthDayReqNeg
... ReplacementReqNeg
... EngagementInitReqNeg
processing web negotiation
processing reflexes:
... eventAlertReflex
... birthdayAlertReflex
... removeOldEngagementReflex
... backupLineTableReflex
... backupEventAlertLineReflex
... backupConsultationDurationReflex
processing initial agent part
names of generated java files are listed in C:\AJA_1.0\demoMAS\agentA\files.txt
translation successfully completed in 0.922 sec.
C:\AJA_1.0\demoMAS\agentA>cd ..\bin
C:\AJA_1.0\demoMAS\bin>pause
Drücken Sie eine beliebige Taste . . .

```

Figure 27 **AJA\_1.0\demoMAS\bin\03\_transA.bat** has been executed.  
 Slika 27 Datoteka **AJA\_1.0\demoMAS\bin\03\_transA.bat** je izvršena.

- Execute the batch file **AJA\_1.0\demoMAS\bin\04\_compA.bat** in order to compile generated Java classes in the previous step.

```

C:\WINNT\System32\cmd.exe
C:\AJA_1.0\demoMAS\bin>call ..\..\bin\setJAVABIN.bat
C:\AJA_1.0\demoMAS\bin>SET JAVABIN=C:\jdk1.4.1\bin
C:\AJA_1.0\demoMAS\bin>cd ..\agentA
C:\AJA_1.0\demoMAS\agentA>C:\jdk1.4.1\bin\javac -classpath ..\..\classes -d
.. \files.txt
C:\AJA_1.0\demoMAS\agentA>cd ..\bin
C:\AJA_1.0\demoMAS\bin>pause
Drücken Sie eine beliebige Taste . . .

```

Figure 28 **AJA\_1.0\demoMAS\bin\04\_compA.bat** has been executed.  
 Slika 28 Datoteka **AJA\_1.0\demoMAS\bin\04\_compA.bat** je izvršena.

- Execute the batch file **AJA\_1.0\demoMAS\bin\05\_transB.bat** in order to translate the AJA source code of the agent B into Java.
- Execute the batch file **AJA\_1.0\demoMAS\bin\06\_compB.bat** in order to compile generated Java classes in the previous step.
- Execute the batch file **AJA\_1.0\demoMAS\bin\07\_transC.bat** in order to translate the AJA source code of the agent C into Java.
- Execute the batch file **AJA\_1.0\demoMAS\bin\08\_compC.bat** in order to compile generated Java classes in the previous step.
- Execute the batch file **AJA\_1.0\demoMAS\bin\09\_transD.bat** in order to translate the AJA source code of the agent D into Java.
- Execute the batch file **AJA\_1.0\demoMAS\bin\10\_compD.bat** in order to compile generated Java classes in the previous step.



11. Execute the batch file `AJA_1.0\demoMAS\bin\11_startRMireg.bat` in order to start the rmi registry. rmi registry will be started in a new minimized window.
12. Execute the batch file `AJA_1.0\demoMAS\bin\12_runA.bat` in order to start the agent A. Before the agent window appears, a small dialog window pops up:



Figure 29 Dialog window with password text fields.  
Slika 29 Dijalog prozor sa tekst poljima za unos lozinke.

Enter and confirm the keystore password for the agent A: **akspass**. This window appears by all AJA agents in order to avoid password storing in the program source code.

A new dialog window appears:

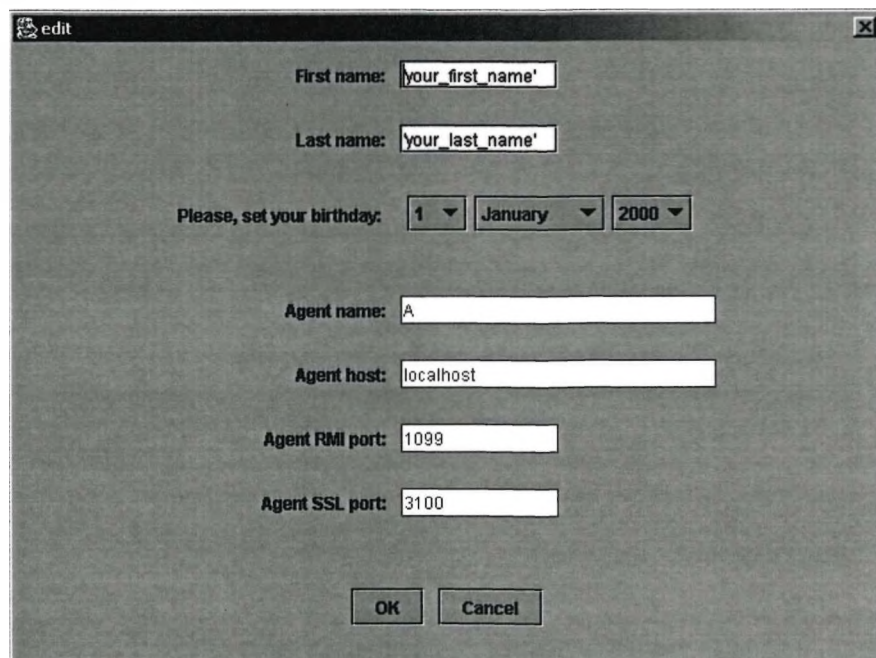


Figure 30 Entering owner data of the agent A.  
Slika 30 Unos podataka o vlasniku agenta A.

Enter the first name, the last name, and the birthday of the agent owner and click OK. For example:

edit

First name:

Last name:

Please, set your birthday:

Agent name:

Agent host:

Agent RMI port:

Agent SSL port:

Figure 31 Owner data for the agent A.  
Slika 31 Podaci o vlasniku agenta A.

After clicking OK button, the main agent window finally appears:

Agent A

engagements

colleagues

your availability

Select an option and press OK button.

Figure 32 The main window of the agent A.  
Slika 32 Glavni prozor agenta A.

Debug information can be seen in the console of the agent A:

```
C:\WINNT\System32\cmd.exe
C:\AJA_1.0\demoMAS\bin>call ..\..\bin\setJ0U0BIN.bat
C:\AJA_1.0\demoMAS\bin>SET J0U0BIN=C:\jdk1.4.1\bin
C:\AJA_1.0\demoMAS\bin>cd ..\AgentA
C:\AJA_1.0\demoMAS\AgentA>C:\jdk1.4.1\bin\java -cp .;..\..\classes Agent
consultationDuration: starting off-line learning. Please wait...
consultationDuration: off-line learning finished with average error: 4.192149096
660051%.
doGUIAct
updateStatusBarAct
backupLineTableAct
backupConsultationDurationAct
```

Figure 33 The console of the agent A.

Slika 33 Konzola agenta A.

13. Execute the batch file **AJA\_1.0\demoMAS\bin\13\_runB.bat** in order to start the agent B.

Keystore password of the agent B is **bkspass**.

Set the owner data for the agent B, e.g.:

The screenshot shows a dialog box titled 'edit' with the following fields and values:

- First name:
- Last name:
- Please, set your birthday:
- Agent name:
- Agent host:
- Agent RMI port:
- Agent SSL port:

At the bottom, there are two buttons:  and .

Figure 34 Owner data for the agent B.

Slika 34 Podaci o vlasniku agenta B.

14. Execute the batch file **AJA\_1.0\demoMAS\bin\14\_runC.bat** in order to start the agent C.

Keystore password of the agent C is **ckspass**.

Set the owner data for the agent C, e.g.:



edit

First name:

Last name:

Please, set your birthday:

Agent name:

Agent host:

Agent RMI port:

Agent SSL port:

Figure 35 Owner data for the agent C.  
Slika 35 Podaci o vlasniku agenta C.

- Execute the batch file `AJA_1.0\demoMAS\bin\15_runD.bat` in order to start the agent D.

Keystore password of the agent D is `dkspass`.

Set the owner data for the agent D, e.g.:

edit

First name:

Last name:

Please, set your birthday:

Agent name:

Agent host:

Agent RMI port:

Agent SSL port:

Figure 36 Owner data for the agent D.  
Slika 36 Podaci o vlasniku agenta D.

Now, all four PDA Agents run on localhost.

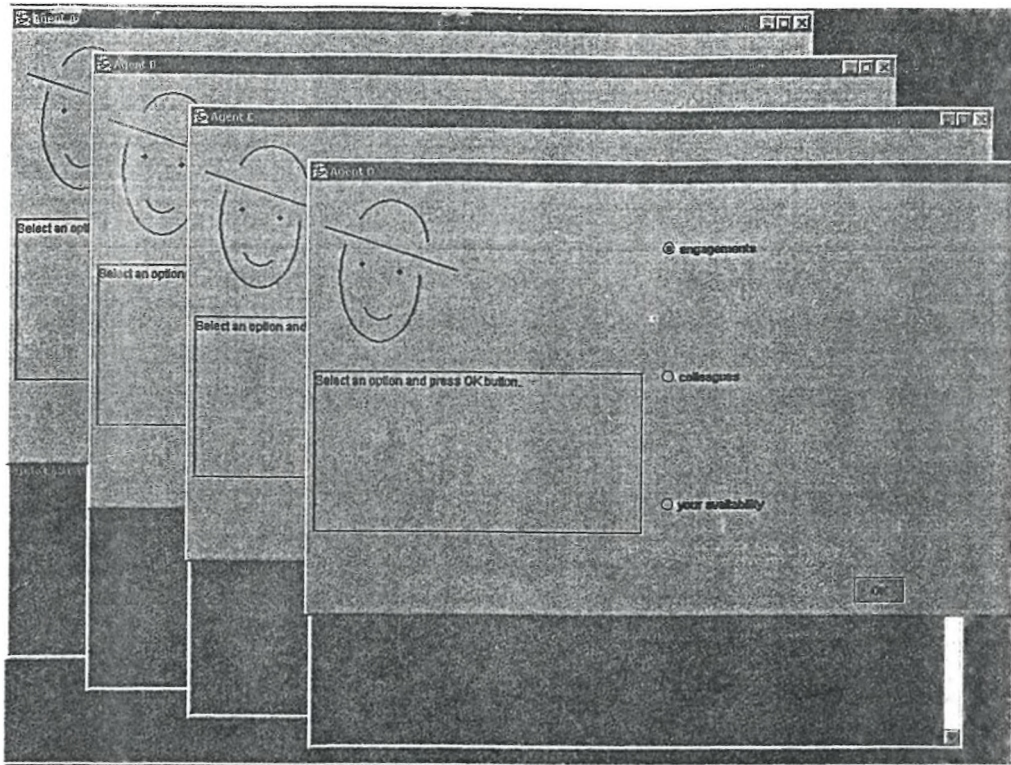


Figure 37 Four PDA Agents are executing on one computer.  
Slika 37 Četiri PDA agenta se izvršavaju na jednom računaru.

## Using Example Agents

The example MAS consists of four PDA agents. Each agent belongs to one assistant or to one professor at one university. An agent manages the available time and engagements of its owner. In most of the engagements there are two or more participants. Agents in the MAS are able to negotiate the time of the joint engagement. A participant in an engagement can also find the replacement using his/her agent.

An AJA agent can be accessed via Internet. Students communicate with PDA agents using their Internet browsers. Here they can find out when their assistant or professor conducts the consultations. If a student wants to visit the consultation, he has to register himself/herself by the agent. The agent can then estimate the consultation duration and inform its owner about the consultation participants and the consultation topics.

### Managing Colleagues

To add, remove, or edit a colleague the agent owner has to select **colleagues** radio button in the main agent window and then to click the **OK** button.



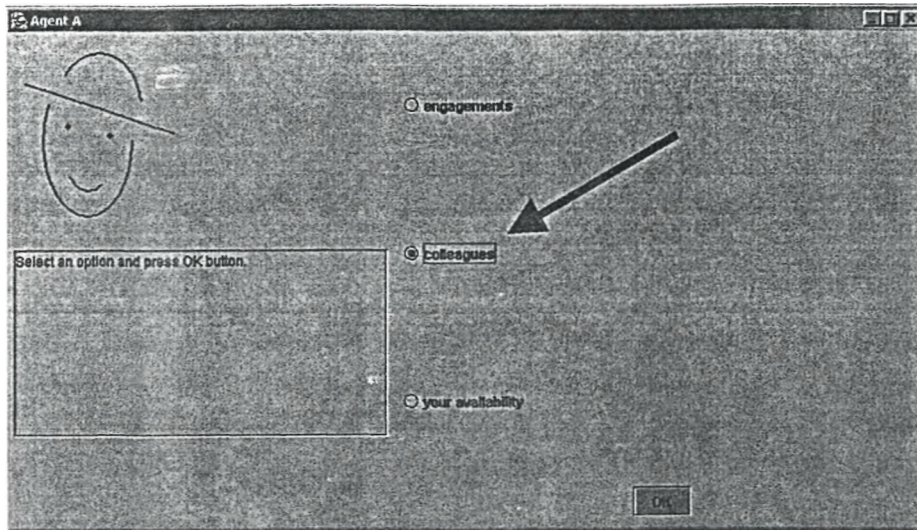


Figure 38 colleagues radio-button.  
 Slika 38 colleagues radio dugme.

After selecting **colleagues** and clicking **OK** in the window of the agent A, the window looks like the one on the following figure.

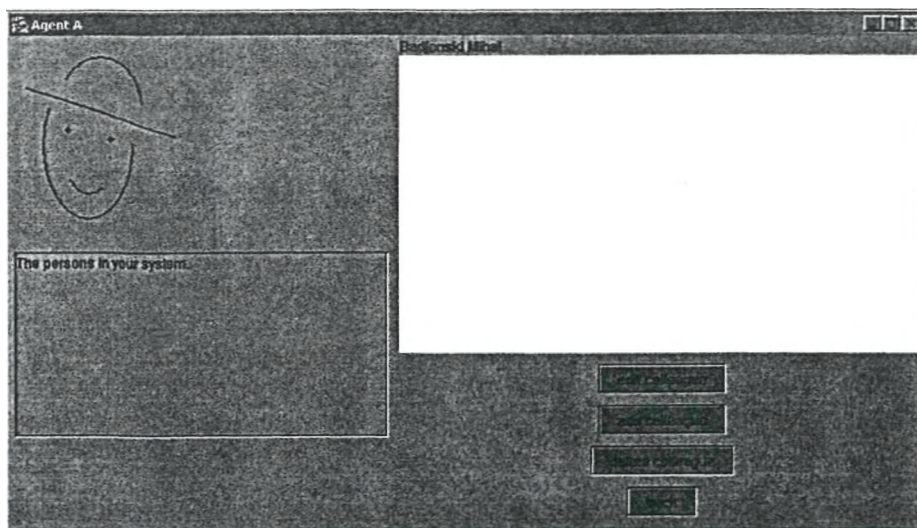


Figure 39 Initial colleagues list of the agent A.  
 Slika 39 Početna lista kolega agenta A.

The only registered person at the moment is the owner of the agent A. Clicking on the **add colleague** button creates the following dialog window:

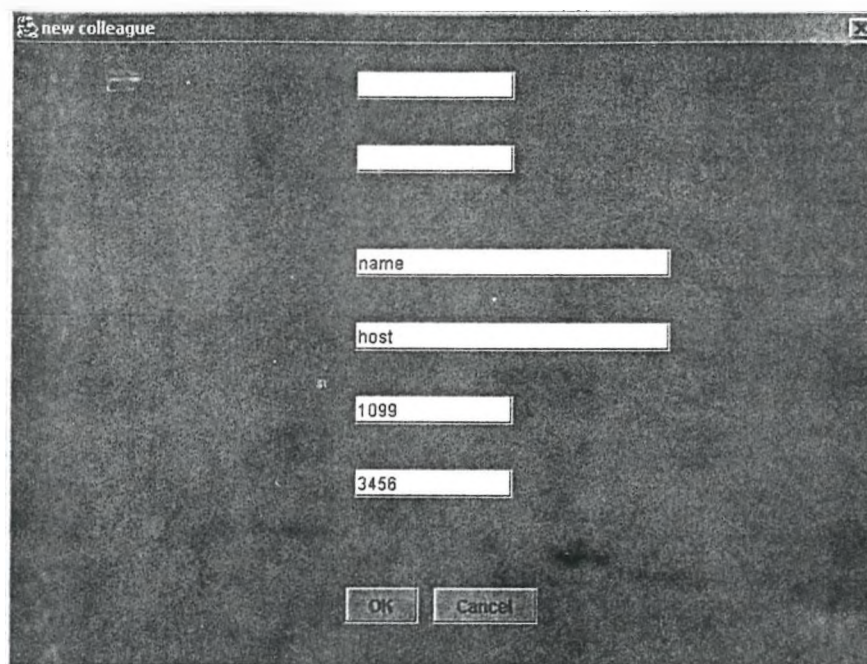


Figure 40 new colleague dialog window.  
Slika 40 new colleague dijalog prozor.

To add a colleague Nataša Ibrajter, which is the owner of the agent D, the text fields has to be filled out as on the following figure.

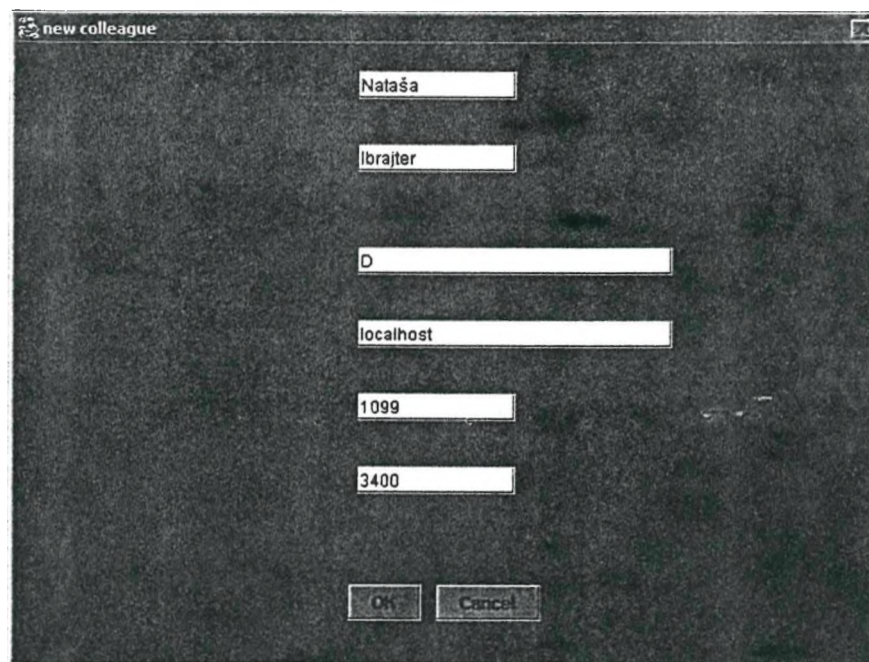


Figure 41 The owner of the agent A adds the owner of the agent D to its list of colleagues.  
Slika 41 Vlasnik agenta A dodaje vlasnika agenta D u listu svojih kolega.

Similarly can be added other two colleagues to the list of colleagues of A.



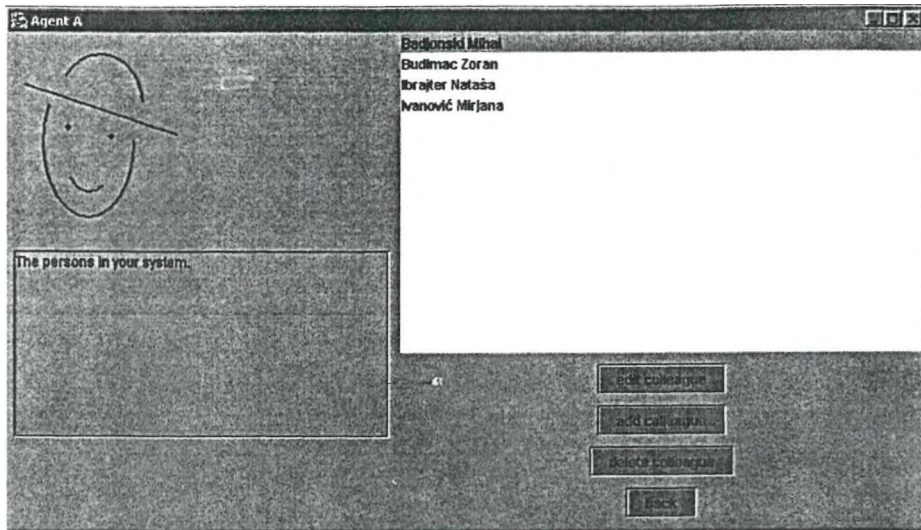


Figure 42 Colleagues list of the agent A.  
Slika 42 Lista kolega agenta A.

Owners of the agents B, C, and D add their colleagues on the same way.

### Managing Available Times

To specify the time intervals when he/she is available for appointments and other engagements, the agent owner has to select **your availability** radio button in the main agent window and then to click the **OK** button.

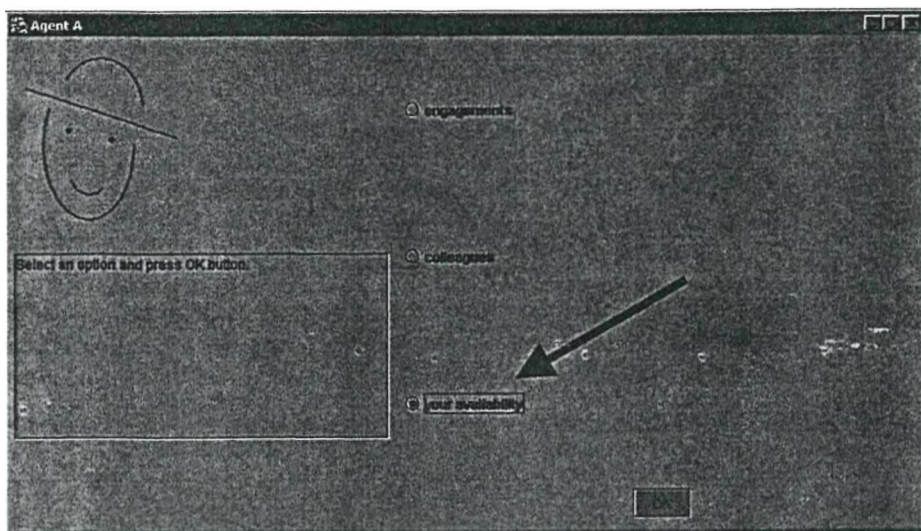


Figure 43 your availability radio-button.  
Slika 43 your availability radio dugme.

After clicking **OK** in the agent A window, the window gets a new look.



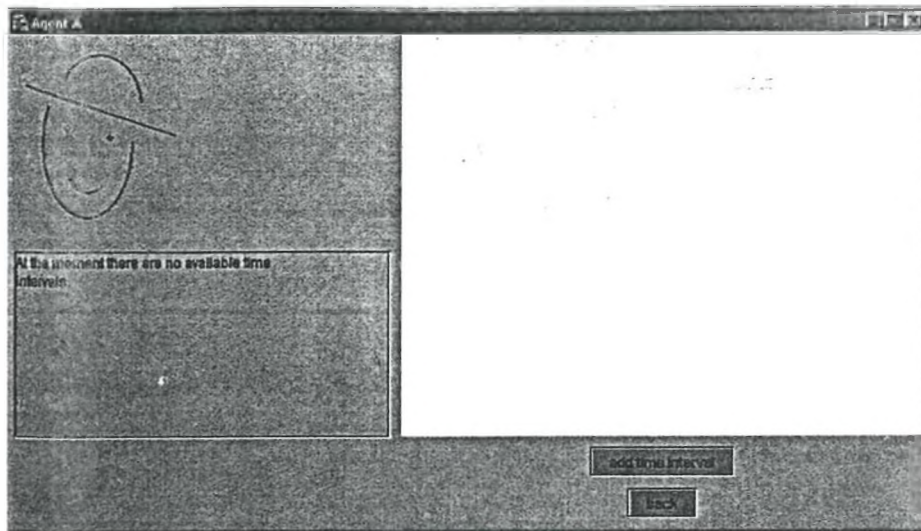


Figure 44 The agent A has at the beginning no registered available time intervals.  
Slika 44 Agent A na početku nema registrovanih slobodnih vremenskih intervala.

Click **add time interval** button in order to add a new available time interval. A new dialog window is created.

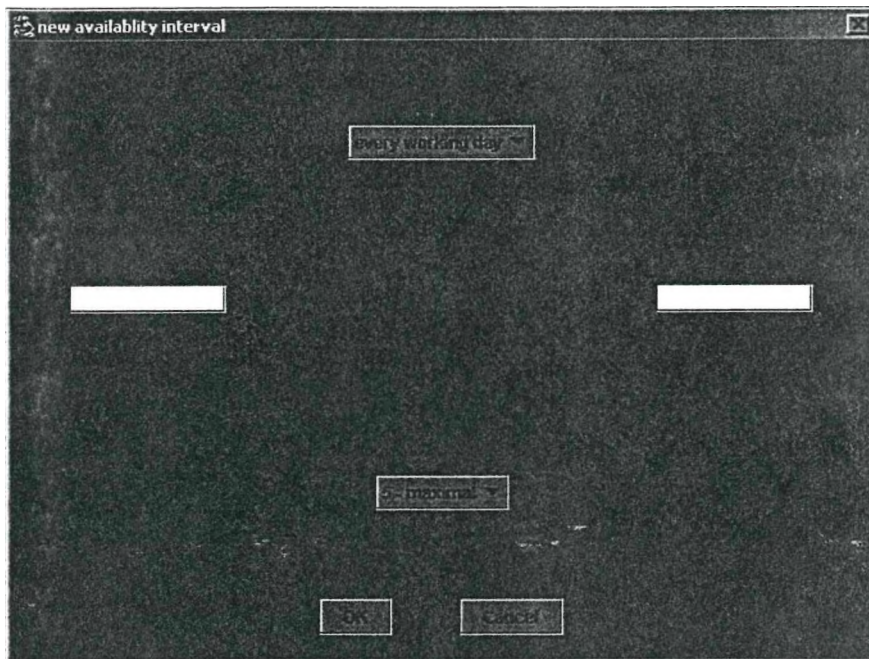


Figure 45 new availability interval dialog window.  
Slika 45 new availability interval dijalog prozor.

There are four types of availability intervals:

- repeating every working day
- repeating every day
- not periodical

- repeating every week

The availability level of an availability interval is a value between 1 (minimal availability) and 5 (maximal availability).

An availability interval with the availability 5, from 9:00 until 15:00 in every working day can be specified as below.

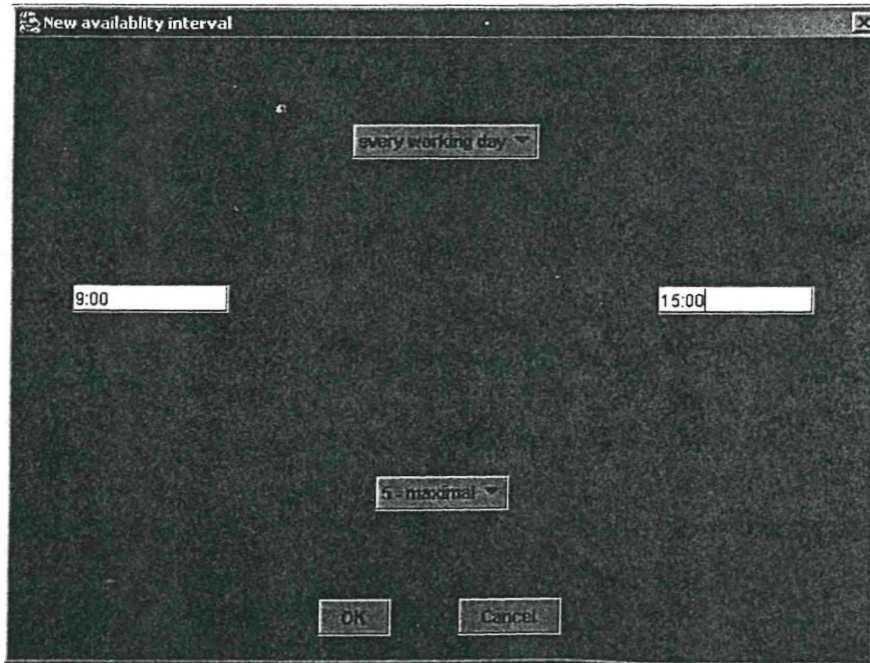


Figure 46 Defining an availability interval.  
Slika 46 Definisanje slobodnog intervala.

After clicking **OK** button, the list of the availability time intervals of the agent A has one element.

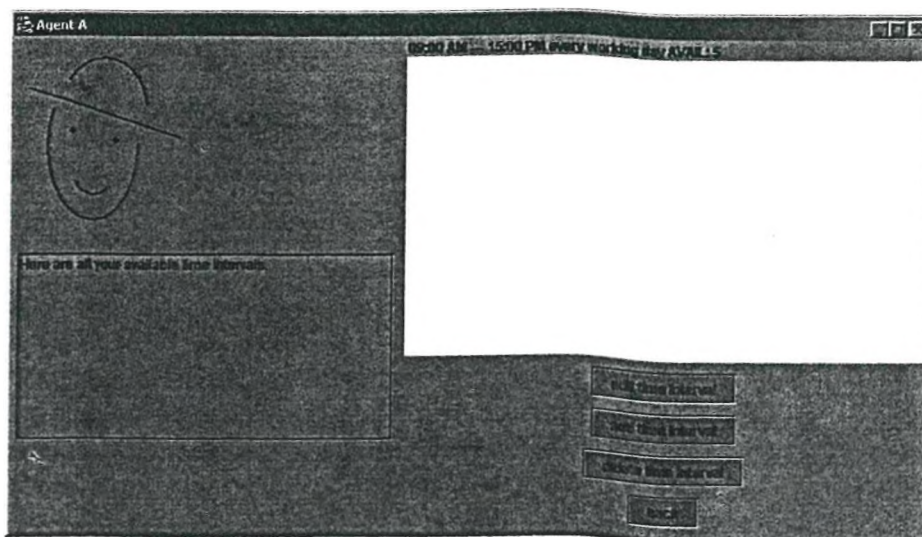


Figure 47 Available time intervals of the agent A.  
Slika 47 Slobodni vremenski intervala agenta A.



Repeating above steps, other time intervals can be defined for the agent A, as well as for the other three agents in the example MAS.

## Managing Engagements

A person can initiate an engagement involving other persons. For example, the owner of the agent A (Mihal Badjonski) can initiate an engagement with the owner of the agent B (Mirjana Ivanović) and the owner of the agent C (Zoran Budimac).

To create an engagement, one has to select **engagements** radio-button and to click **OK** button.

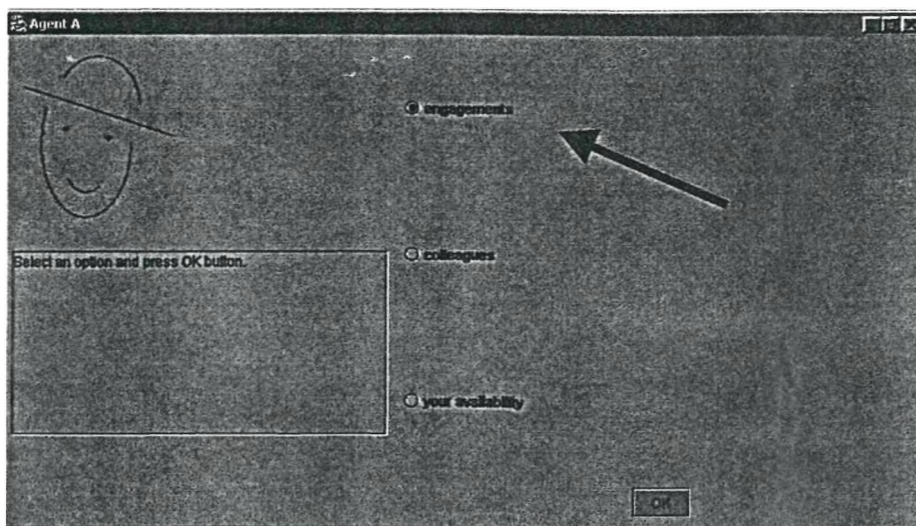


Figure 48 engagements radio-button.  
Slika 48 engagement radio dugme.

Afterwards, a new engagement button has to be clicked.

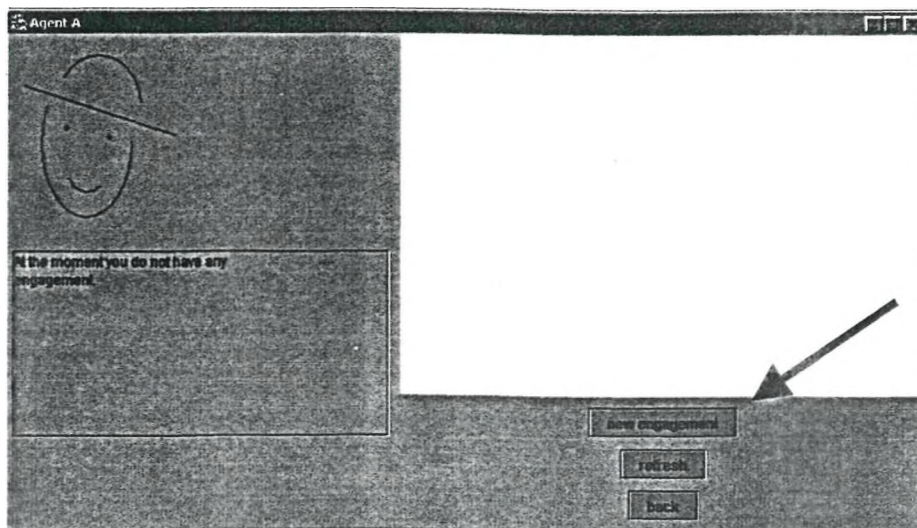


Figure 49 Currently there are no engagements.  
Slika 49 Momentalno nema registrovanih aktivnosti.

A small dialog window appears.

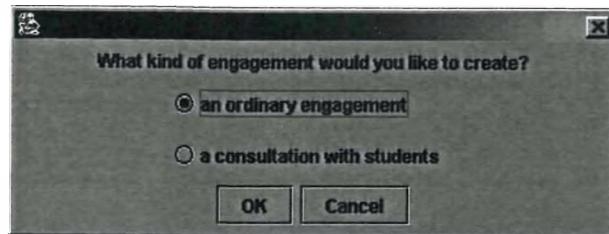


Figure 50 A window for the engagement type specification.  
Slika 50 Prozor za specifikaciju tipa aktivnosti.

After selecting an ordinary engagement radio-button, the following dialog window pops up.

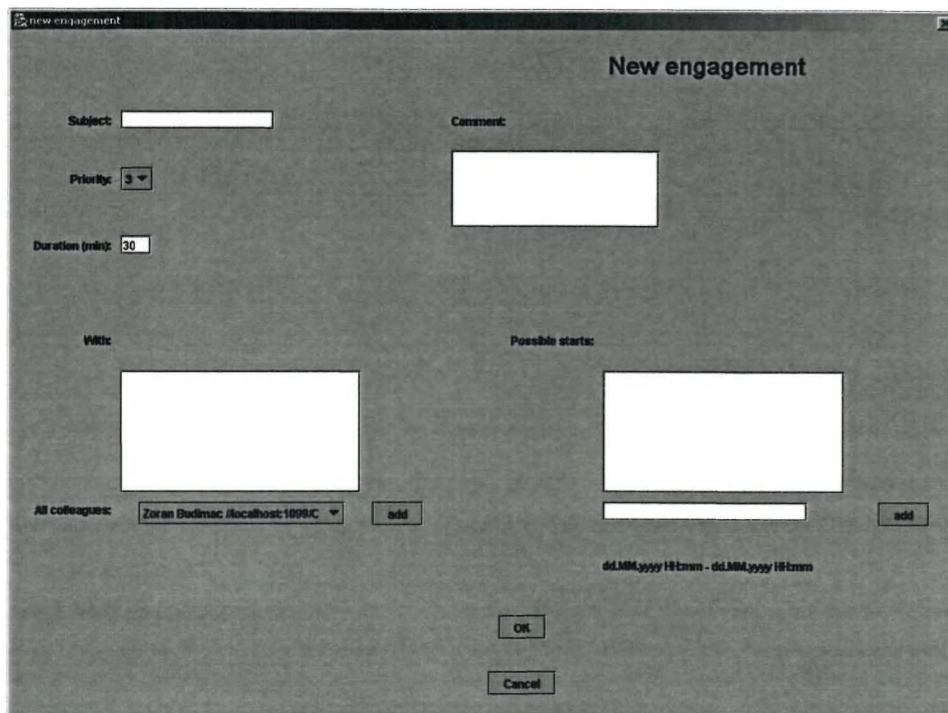


Figure 51 New engagement window.  
Slika 51 Prozor za dodavanje nove aktivnosti.

Input elements in the above window can be filled out as in the figure below.

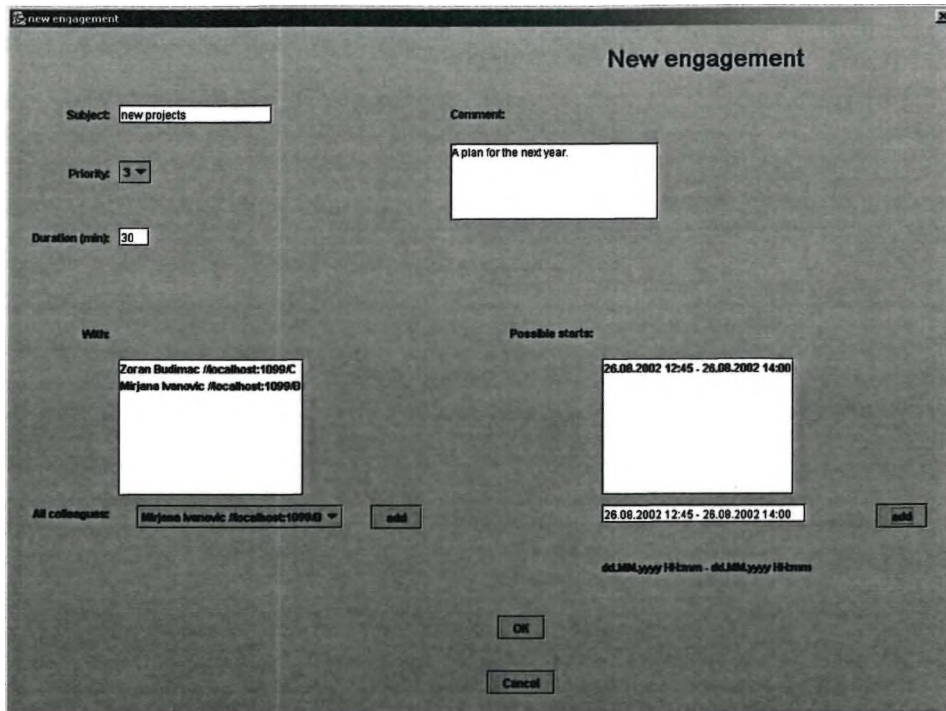


Figure 52 New engagement window after entering the engagement data.  
 Slika 52 Prozor za dodavanje nove aktivnosti nakon unosa podataka o aktivnosti.

If an intersection of available times of the invited persons in specified interval can be found, the engagement will be established. The owners of the agents A and B are notified using the status bar of their agent windows.

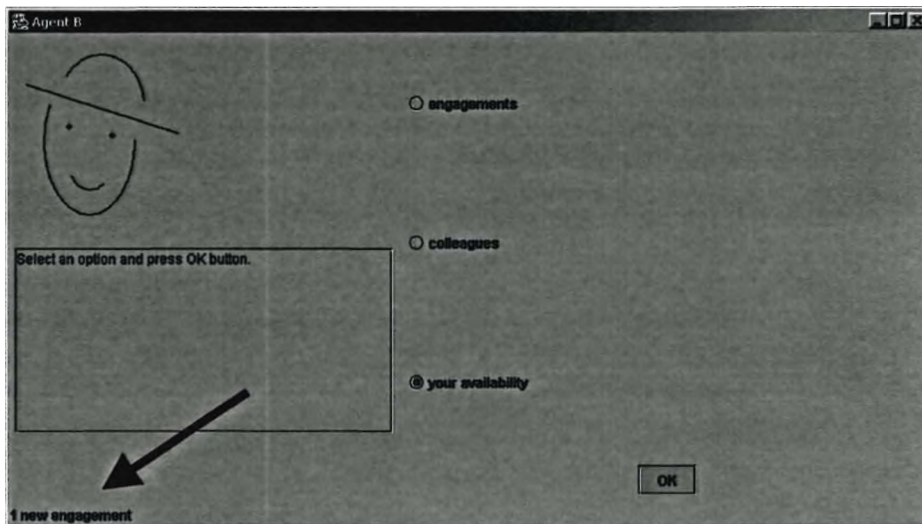


Figure 53 Agent B displays the new information in the status bar of its window.  
 Slika 53 Agent B ispisuje novu informaciju u donjem delu svog prozora.

The owner of the agent B can edit the engagement by selecting the **engagements** radio button, clicking on the **OK** button, and then clicking on the **edit engagement** button.



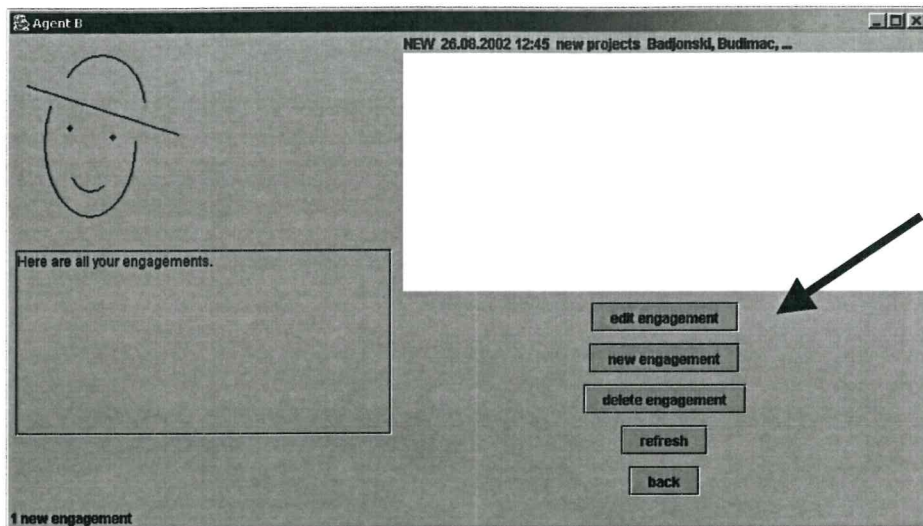


Figure 54 Engagements list of the Agent B.  
Slika 54 Lista aktivnosti agenta B.

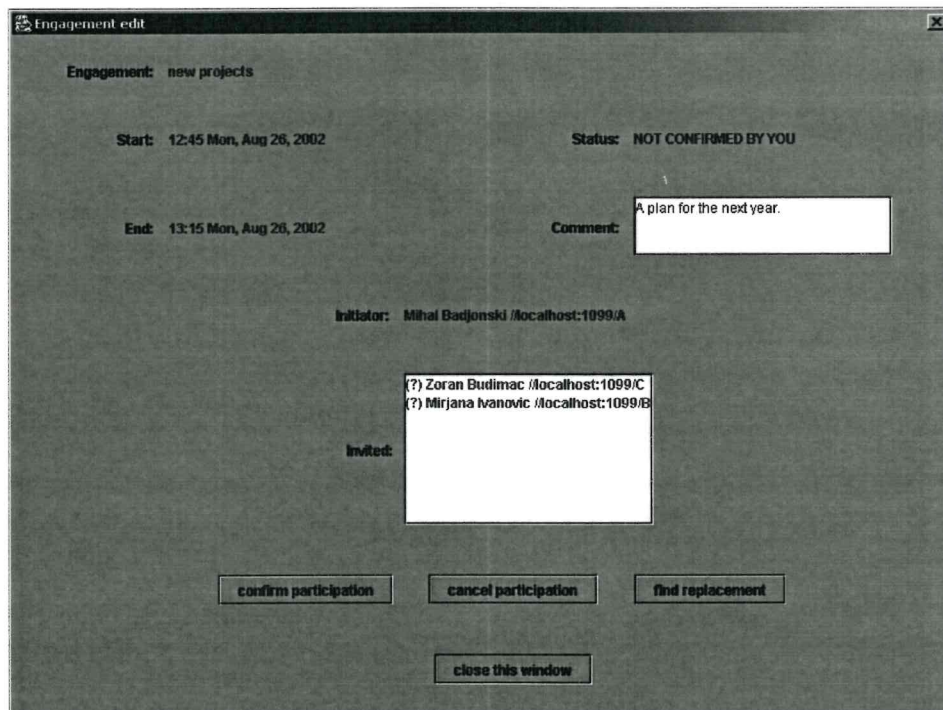


Figure 55 The owner of the agent B (Mirjana Ivanović) edits the new engagement.  
Slika 55 Vlasnik agenta B (Mirjana Ivanović) edituje novu aktivnost.

Because of the fact that Zoran Budimac and Mirjana Ivanović have not yet confirmed their participation, the question marks appears near their names in the above window.

An invited participant can:

- confirm the participation,
- cancel the participation,

- find the replacement.

In all cases the other involved agents are informed about the new status of the engagement. They inform their owners about the engagement update using their status bar.

## Access via Internet Browser

AJA Agents can be accessed via World Wide Web (WWW). An agent in the example MAS allows students to register themselves via WWW for the consultations by the agent owner.

For example, the owner of the agent A defines two consultations. The first one takes place on August 27<sup>th</sup>, 2002 at 10:00.

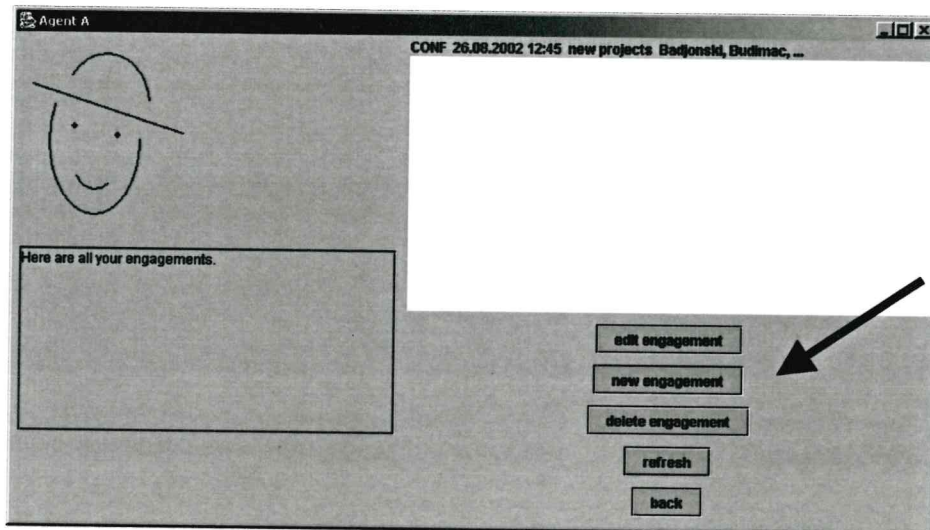


Figure 56 The first step in creating of a consultation.  
Slika 56 Prvi korak u kreiranju konsultacija.

After clicking the **new engagement** button, the radio button **a consultations with students** has to be selected.

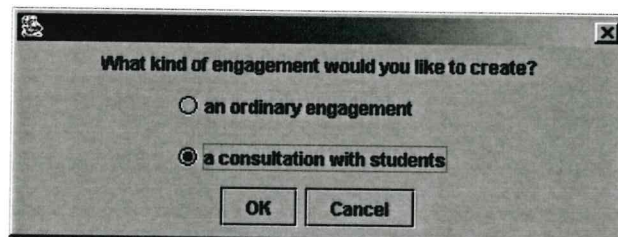


Figure 57 The second step in creating of a consultation.  
Slika 57 Drugi korak u kreiranju konsultacija.

A new window appears, where the consultation starting time has to be specified.

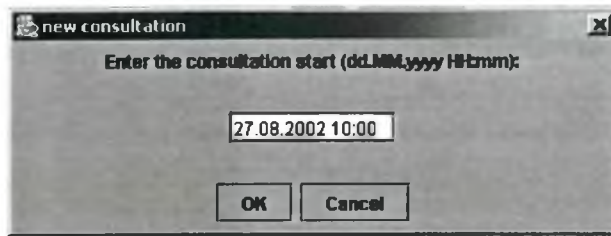


Figure 58 The third step in creating of a consultation.  
Slika 58 Treći korak u kreiranju konsultacija.

After clicking **OK**, the consultation is created.

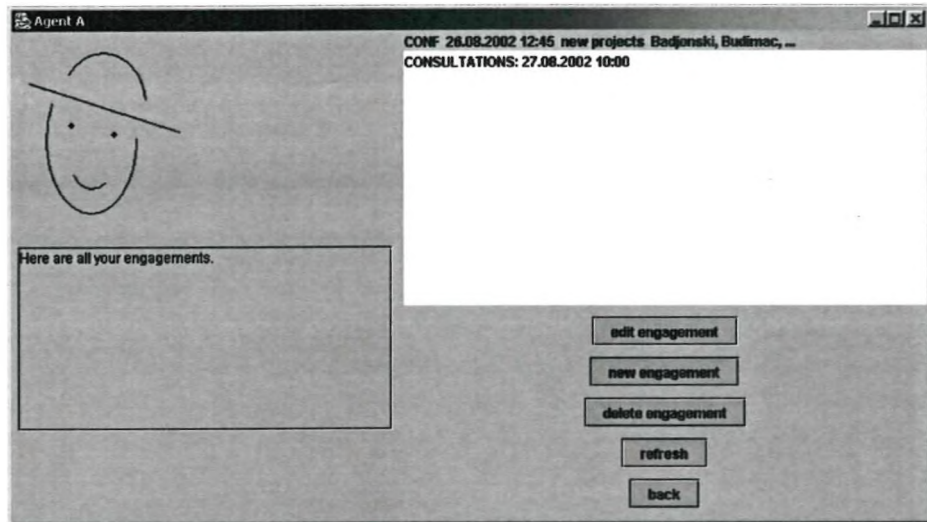


Figure 59 A consultation is created.  
Slika 59 Konsultacije su kreirane.

The second consultation date is, for example, August 30<sup>th</sup>, 2002 at 14:00. Repeating the above step, it can be added to the list of engagements.

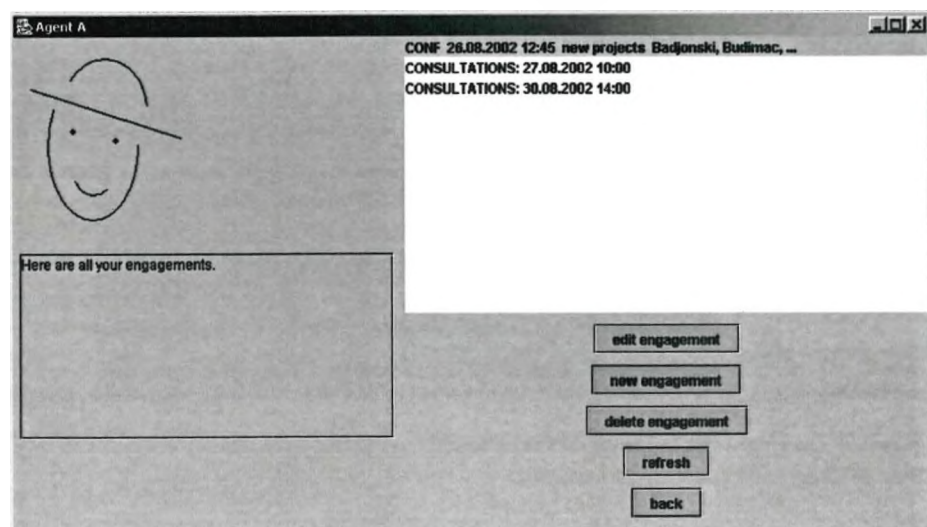


Figure 60 The second consultation is created.  
Slika 60 Druge konsultacije su kreirane.



The HTTP address of the agent A is <http://localhost:2100/A> . A student can use this address to access the agent over the Internet.

Of course, in the real life the computer name would not be localhost.

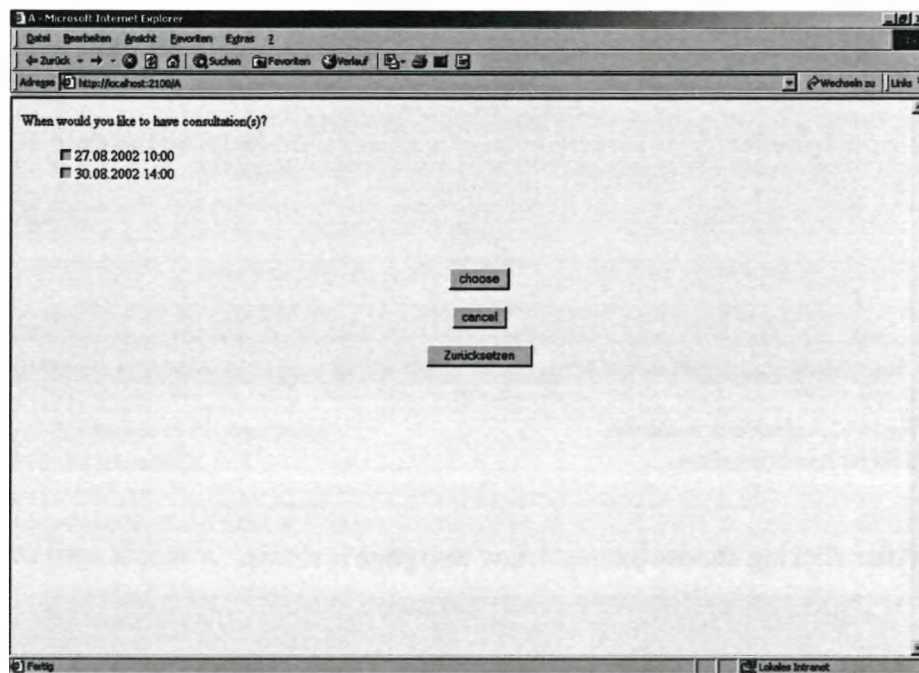


Figure 61 The first html page in the agent-student dialog.  
Slika 61 Prva html stranica u komunikaciji između agenta i studenta.

A student chooses e.g. the first consultation date.

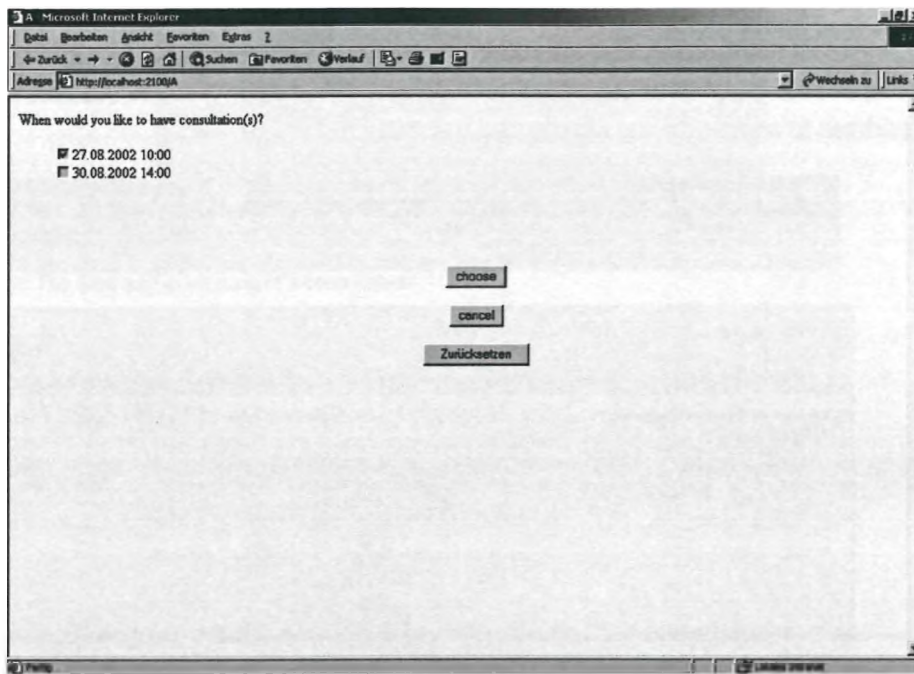


Figure 62 A check-box is selected.  
Slika 62 Termin je izabran.

After clicking **choose** button, a new web page is shown.

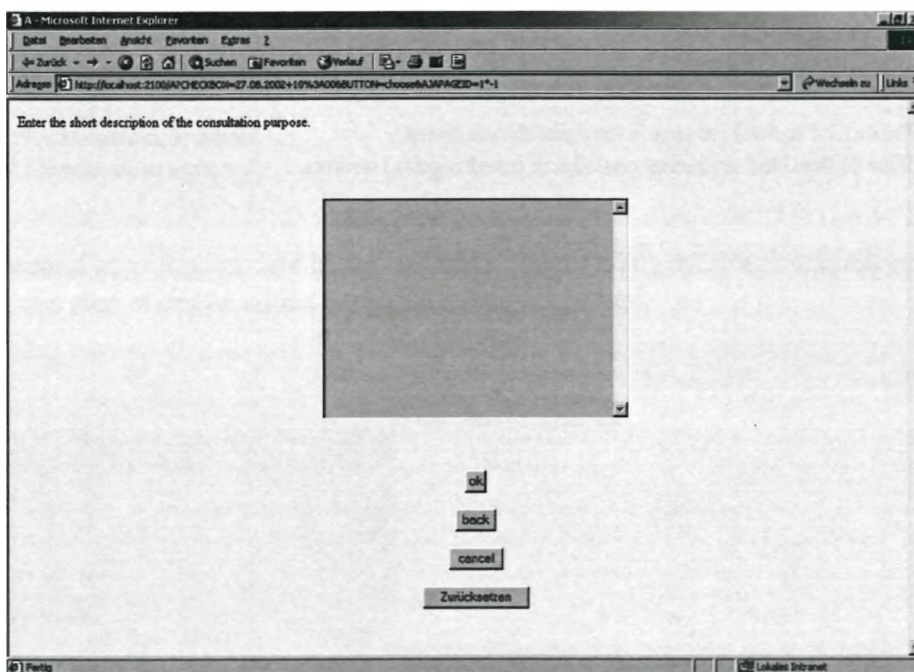


Figure 63 The second page.  
Slika 63 Druga stranica.

A student enters the short description of the consultation purpose and clicks **OK** button.

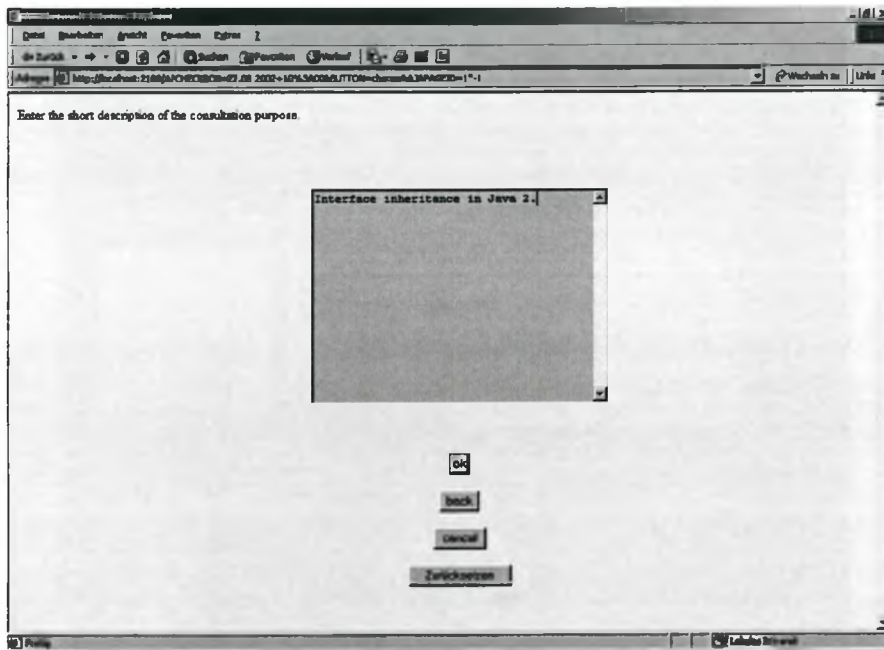


Figure 64 The purpose of the consultation.  
Slika 64 Svrha konsultacija.

A new page is shown.

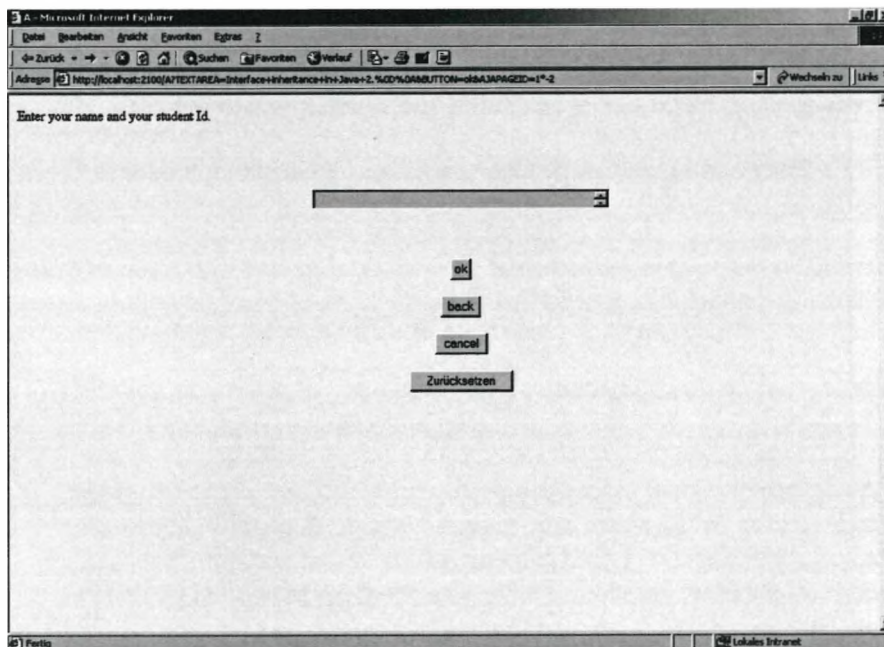


Figure 65 The third page.  
Slika 65 Treća stranica.

A student enters her/his name and student Id.





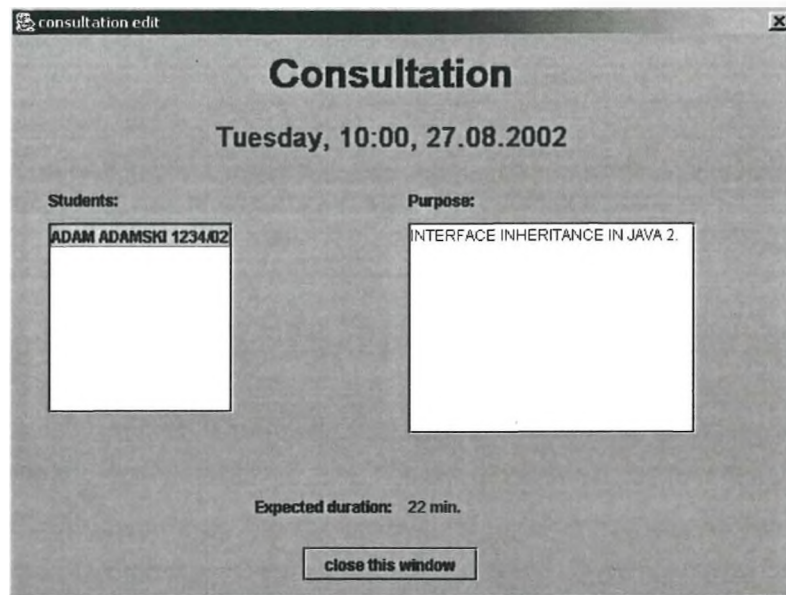


Figure 68 Editing the consultation.  
Slika 68 Editovanje konsultacija.

## Other features

Besides the features mentioned above, an example agent has some additional features. Some of these features are shortly described below:

- Agent owner is notified, when a colleague has a birthday. The notification occurs a day before the birthday as well as at the day of the birthday.
- Duration of a consultation with students depends on two values:
  - number of appointed students, and
  - number of days before the next exams (the sooner the exams, the longer the consultations).

Duration of a consultation is therefore a dependant belief. The value of this belief is computed using neural network.

- Agent owner is alerted before the engagement start. Nevertheless, some user would like to be alerted sooner and some other earlier. The exact number of minutes before the engagement start, when the alert window is created, is computed using an adaptable parameter. Initially, the parameter has a value fifteen minutes. During the agent life its owner gives the feedback to the agent regarding the alert timing. Using this feedback (i.e. 'sooner' or 'later'), the adaptable parameter adapts its value to its user preferences.
- Agent reflexes are used to store the agent beliefs into the file after they have been changed. Although an agent should never be stopped, eventually it will be stopped. After the new start of the agent, it reads its belief from the file and continues to work.

## References

- [1] Agent Oriented Software Pty. Ltd., "JACK Intelligent Agents™ User Guide", available at <http://www.agent-software.com.au/> .
- [2] Agents mailing list, [agents@cs.umbc.edu](mailto:agents@cs.umbc.edu).
- [3] J. L. Austin, "How to Do Things With Words", Oxford University Press, Oxford, England, 1962.
- [4] N. Azarmi, S. Thompson, "ZEUS: A Toolkit for Building Multi-Agent Systems", *Proceedings of fifth annual Embracing Complexity Conference*, Paris, 2000.
- [5] M. Badjonski, "*Implementation of Multi-Agent Systems using Java*", Master thesis, Institute of Mathematics, Faculty of Science, University of Novi Sad, Yugoslavia, 1998.
- [6] M. Badjonski, M. Ivanović, "Multi-agent System for Determination of Optimal Hybrid for Seeding", *Proceedings of EFITA '97 - First European Conference for Information Technology in Agriculture*, Copenhagen, Denmark, June 15-18, 1997, pp. 401-404.
- [7] M. Badjonski, M. Ivanović "An application of Multi-Agent Theory in Agriculture", *Proceedings of IEEE First International Conference on Intelligent Processing Systems (IEEE ICIPS)*, Beijing, China, 1997, pp. 866-870.
- [8] M. Badjonski, M. Ivanović "LASS - A language for Agent-Oriented Software Specification", *Proceedings of VIII Conference on Logic and Computer Science LIRA*, Novi Sad, September, 1997, pp. 9-18.
- [9] M. Badjonski, M. Ivanović, Z. Budimac, "Possibility of using Multi-Agent System in Education", *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics*, Orlando, Florida, USA, October 12-15, 1997, pp. 588-593.
- [10] M. Badjonski, M. Ivanović, Z. Budimac, "Software Specification Using LASS", *Proceedings of Asian'97*, Lecture Notes in Computer Science Vol 1345, Springer-Verlag, Kathmandu, Nepal, December, 1997, pp. 375-376.
- [11] M. Badjonski, M. Ivanović, Z., Budimac, "Agent Oriented Programming Language LASS", *Computer Science and Electronic Eng.*, Horwood Publishing Ltd., 1999.
- [12] M. Badjonski, K. Schröter, J. Wendler, H. D. Burkhard, "Learning of Kick in Artificial Soccer", *Proceedings of the fourth RoboCup Workshop*, Melbourne, Australia, 2000.
- [13] M. Barbuceanu, M. S. Fox, "The Design of a Coordination Language for Multi-Agent Systems", *Working Notes of the Third International Workshop*

on Agent Theories, Architectures and Languages, ECAI '96, Budapest, Hungary, pp. 263-278.

[14] J. Bates, "The Role of Emotion in Believable Agents", *Communication of the ACM*, 37(7):122-125, July 1994.

[15] J. Bell, "Changing Attitudes", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 40-55.

[16] M. Benerecetti, A. Cimatti, et. al., "Context-Based Formal Specification of Multi-Agent Systems", *Working Notes of the Third International Workshop on Agent Theories, Architectures and Languages*, ECAI '96, Budapest, Hungary, pp. 295-307.

[17] M. E. Bratman, D. J. Israel, M. E. Pollack, "Plans and Resource-Bounded Practical Reasoning", *Computational Intelligence*, 4:349-355, 1988.

[18] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot", *IEEE Journal of Robotics and Automation*, 2(1):14-23, 1986.

[19] R. A. Brooks, "Intelligence without Reason", *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, 1991, pp 569-595.

[20] R. A. Brooks, "Intelligence without Representation", *Artificial Intelligence*, 47:139-159, 1991.

[21] Z. Budimac, M. Ivanović, A. Popović, "Workflow Management System Using Mobile Agents", *Proceedings of ADBIS '99*, Lecture Notes in Computer Science, Maribor, Slovenia, 1999, pp. 169-178.

[22] H. D. Burkhard, "Agent-Oriented Programming for Open Systems", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 291-306.

[23] P. Busetta, R. Rönquist, A. Hodgson, A. Lucas, "Jack intelligent agents - Components for Intelligent Agents in Java", *AgentLink News Letter*, January 1999, pp. 2-5.

[24] C. Byrne, P. Edwards, "Refinement in Agent Groups", *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 22-39.

[25] A. Chavez, P. Maes, "Kasbah: An Agent Marketplace for Buying and Selling Goods", *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, London, UK, April, 1996.

[26] A. Cimatti, L. Serafini, "Multi-Agent Reasoning with Belief Contexts: the Approach and a Case Study", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 71-85.

[27] M. Costa, B. Feijo, "Agents with Emotions in Behavioral Animation", *Computers & Graphics*, Vol. 20, No 3, 1996, pp. 377-384.



- [28] S. A. DeLoach, "Multiagent Systems Engineering A Methodology and Language for Designing Agent Systems", *Proceedings of Agent Oriented Information Systems*, 1999, pp. 45–57.
- [29] O. Etzioni, D. Weld, "A Softbot - Based Interface to the Internet", *Communication of the ACM*, 37(7):72-76, July 1994.
- [30] P. Faratin, C. Sierra, N. R. Jennings, "Using similarity criteria to make trade-offs in automated negotiations", *Artificial Intelligence and Int Journal of Autonomous Agents and Multi-Agent Systems*, 2003, to appear.
- [31] S. S. Fatima, M. Wooldridge, N. R. Jennings, "Optimal negotiation strategies for agents with incomplete information", *Proceedings of the 8th Int. Workshop on Agent Theories, Architectures and Languages (ATAL)*, Seattle, USA, 2001, pp. 53-68.
- [32] S. S. Fatima, M. Wooldridge, N. R. Jennings, "The influence of information on negotiation equilibrium", *Proceedings of the 4th Int Workshop on Agent-Mediated Electronic Commerce*, Bologna, Italy, 2002, to appear.
- [33] T. Finin, J. Weber, et. al., "Draft Specification of the KQML Agent-Communication Language", *The Darpa Knowledge Sharing Initiative External Interfaces Working Group*, 1993, available as <http://www.cs.umbc.edu/kqml/kqmlspec.ps>.
- [34] M. Fisher, "A Survey of Concurrent MetateM - the language and its applications", *Temporal Logic - Proceedings of the First International Conference*, Vol 827, Springer-Verlag, July 1994, pp. 480-505.
- [35] M. Fisher, "Representing and Executing Agent-Based Systems", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 307-323.
- [36] Foundation for Intelligent Physical Agents, "FIPA ACL Message Structure Specification", available at <http://www.fipa.org/specs/fipa00061/>.
- [37] S. Franklin, A. Graesser, "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", *Working Notes of the Third International Workshop on Agent Theories, Architectures and Languages*, ECAI '96, Budapest, Hungary, pp. 193-206.
- [38] M. R. Genesereth, S. P. Ketchpel, "Software agents", *Communication of the ACM*, 37(7):48-53, July 1994.
- [39] M. P. Georgeff, A. L. Lansky, "Reactive Reasoning and Planning", *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, WA, 1987, pp. 677-682, 1987.
- [40] L. Glicoes, R. Staats, M. Huhns, "A Multi-Agent Environment for Department of Defense Distribution", *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 53-84.

- [41] P. Gu, A. B. Maddox, "A Framework for Distributed Reinforcement Learning", *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 97-112.
- [42] C. G. Harrison, D. M. Chess, A. Kershenbaum, "Mobile agents: Are they a good idea?", *Research Report, IBM Research Division*, available as <http://www.research.ibm.com/massdist/>.
- [43] T. Haynes, S. Sen, "Evolving Behavioral Strategies in Predators and Prey", *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 113-126.
- [44] <http://193.113.209.147/projects/agents/zeus/index.htm>
- [45] <http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm>
- [46] <http://agents.media.mit.edu/>
- [47] <http://www.agent-software.com.au/>
- [48] <http://www.ai.mit.edu/people/sodabot/slideshow/total/p001.html>
- [49] <http://www.robocup.org/>
- [50] J. Huang, N. R. Jennings, J. Fox, "An Agent Architecture for Distributed Medical Care", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 219-232.
- [51] P. Huang, K. Sycara, "A Computational Model For Online Agent Negotiation", *Proceedings of the 35th Hawaii International Conference on System Sciences HICSS'02*, 2002, to appear.
- [52] IBM, "Autonomic Computing: IBM's Perspective on the State of Information Technology", white paper, 2001, available at [http://www.research.ibm.com/autonomic/manifesto/autonomic\\_computing.pdf](http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf).
- [53] C. A. Iglesias, M. Garijo, J. C. Gonzalez, "A Survey of Agent-Oriented Methodologies", *Proceedings of Intelligent Agents V, Agent Theories, Architectures, and Languages*, 5th International Workshop, ATAL '98, Paris, France, 1998, pp. 317-330.
- [54] U.C. Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1", available as <http://www.w3.org/Protocols/rfc2616/rfc2616.html>, 1999.
- [55] N. R. Jennings, "Specification and Implementation of a Belief Desire Joint-Intention Architecture for Collaborative Problem Solving", *Journal of Intelligent and Cooperative Information Systems*, 2(3):289-318 1993.
- [56] N. R. Jennings, "On Agent-Based Software Engineering", *Artificial Intelligence*, 117 (2): 277-296, 2000.

- [57] N. R. Jennings, "An agent-based approach for building complex software systems", *Communications of the ACM*, 44(4): 35-41, April 2001.
- [58] N. R. Jennings, "Agent-based computing", *Proceedings of the 17th IFIP World Congress on Computing*, Montreal, Canada, 2002, to appear.
- [59] N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, C. Sierra, M. Wooldridge, "Automated negotiation: prospects, methods and challenges", *Int. J. of Group Decision and Negotiation*, 10 (2): 199-215, 2001.
- [60] N. R. Jennings, P. Faratin, T. J. Norman, P. O'Brien, B. Odgers, "Autonomous Agents for Business Process Management", *Int. Journal of Applied Artificial Intelligence*, 14 (2): 145-189, 2000.
- [61] N. R. Jennings, P. Faratin, T. J. Norman, P. O'Brien, B. Odgers, J. L. Alty, "Implementing a Business Process Management System using ADEPT: A Real-World Case Study", *Int. Journal of Applied Artificial Intelligence*, 14 (5): 421-465, 2000.
- [62] N. R. Jennings, M. Wooldridge, "Software Agents", *IEE Review*, January, 1996, pp. 17- 20.
- [63] N. R. Jennings, M. Wooldridge, "Applications of Intelligent Agents", *Agent Technology: Foundations, Applications, and Markets*, 1998, pp. 3-28
- [64] L. P. Kaelbling, "A Situated Automata Approach to the Design of Embedded Agents", *SIGART Bulletin*, 2(4):85-88, 1991.
- [65] D. B. Lang, "Java Aglet Application Programming Interface (J-AAPI) White Paper - Draft 2", *IBM Tokyo Research Laboratory*, available as <http://www.trl.ibm.co.jp/aglets/JAAPI-whitepaper.html>, February 1997.
- [66] B. Lenzmann, I. Wachsmuth, "A User-Adaptive Interface Agency for Interaction with a Virtual Environment", *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 140-151.
- [67] R. Li, L. M. Pereira, "Knowledge-Based Situated Agents among Us A Preliminary Report", *Working Notes of the Third International Workshop on Agent Theories, Architectures and Languages*, ECAI '96, Budapest, Hungary, pp. 309-322.
- [68] A. R. Lomuscio, M. Wooldridge, N. R. Jennings, "A classification scheme for negotiation in electronic commerce", *Int Journal of Group Decision and Negotiation*, 2002, to appear.
- [69] P. Maes, "The Agent Network Architecture (ANA)", *SIGART Bulletin*, 2(4):115-120, 1991.
- [70] P. Maes, "Modelling Adaptive Autonomous Agents", *Artificial Life Journal*, Ed. C. Langton, Vol 1, No. 1&2, MIT Press, 1994, pp. 135-162.
- [71] P. Maes, "Agent that Reduce Work and Information Overload", *Communications of the ACM*, 37(7):31-40, July 1994.

- [72] M. J. Mataric, "Learning in Multi-Robot Systems", *Adaptation and Learning in Multi-Agent Systems*. Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 152-163.
- [73] T. M. Michell, R. Caruana, D. Freitag, J. McDermott, D. Zabowski, "Experience with a Learning Personal Assistant", *Communication of the ACM*, 37(7):80-91, July 1994.
- [74] M. Minsky, "Steps Towards Artificial Intelligence", *Proceedings of the IRE*, 1961, pp. 8-30. (Reprinted in E.A. Feigenbaum and J. Feldman (Eds.), "Computers and Thought", McGraw-Hill, 1963, pp. 406-450.)
- [75] M. Minsky, *"The Society of Mind"*, Simon and Schuster, New York, 1986.
- [76] H. S. Nwana, "ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems", *Proceedings of PAAM'98*, London 1998, pp. 377-392.
- [77] H. S. Nwana, D. T. Ndumu, L. C. Lee, J. C. Collis, " A Toolkit and Approach for Building Distributed Multi-Agent Systems ", *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, Seattle, WA, USA, 1999, pp. 360-361.
- [78] T. Ohko, K. Hiraki, Y. Anzai, "Learning to Reduce Communication Cost on Task Negotiation among Multiple Autonomous Robots", *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 177-190.
- [79] H. V. D. Parunak, J. Odell, "Representing Social Structures in UML", *Agent-Oriented Software Engineering II*, Lecture Notes in Computer Science, Vol 2222, Springer-Verlag, 2002. pp. 1-16.
- [80] C. J. Petrie, "Agent-Based Engineering, the Web, and Intelligence", *IEEE Expert*, December 1996, available as <http://cdr.stanford.edu/NextLink/Expert.html>.
- [81] A. Poggi, G. Adorni, "A Multi Language Environment to Develop Multi Agent Applications", *Working Notes of the Third International Workshop on Agent Theories, Architectures and Languages*, ECAI '96, Budapest, Hungary, pp. 249-261.
- [82] E. Rich, *"Artificial Intelligence"*, McGraw-Hill Book Company, 1983.
- [83] D. Riecken, "M: An Architecture of Integrated Agents", *Communication of the ACM*, 37(7):107-116, July 1994.
- [84] M. Riedmiller, "Rprop – Description and Implementation Details", Technical Report, University of Karlsruhe, Germany, 1994.
- [85] M. Riedmiller, H. Braun, "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm", *Proceedings of the IEEE International Conference on Neural Network (ICNN)*, San Francisco, USA, 1993, pp. 586-591.

- [86] T. W. Sandholm, R. H. Crites, "On Multiagent Q-Learning in a Semi-Competitive Domain", *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 191-205.
- [87] W. S. Sarle ed., "Neural Network FAQ, part 1 of 7: Introduction, periodic posting to the Usenet newsgroup comp.ai.neural-nets", URL: <ftp://ftp.sas.com/pub/neural/FAQ.html>.
- [88] S. Schoepke, "Facilitating the Deployment of Intelligent Agents in the Application Development Mainstream", *AgentLink News Letter*, July 1999, pp. 10-12.
- [89] S. Schoepke, "Intelligent Agents will be a Vehicle for other AI-related Technologies", Position Paper, International Workshop on Agent-Oriented Information Systems (AOIS'99), 1999, available at <http://www.aois.org/99/schoepke.html>.
- [90] S. Sen, M. Sekaran, "Using Reciprocity to Adapt to Others", *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 206-217.
- [91] Y. Shoham, "Agent-Oriented Programming", *Artificial Intelligence*, 60(1):51-92, 1993.
- [92] D. C. Smith, A. Cypher, J. Spohrer, "KIDSIM: Programming Agents without a Programming Language", *Communication of the ACM*, 37(7):55-67, July 1994.
- [93] V. Tamma, M. Wooldridge, I. Dickinson, "An Ontology for Automated Negotiation", *Proceedings of the Workshop on Ontologies in Agent Systems*, Bologna, Italy, July 2002, to appear.
- [94] R. S. Thomas, "PLACA, an Agent Oriented Programming Language", PhD thesis, Computer Science Department, Stanford University, Stanford, CA 94305, August 1993. (Available as technical report STAN-CS-93-1487).
- [95] S. R. Thomas, "The PLACA Agent Programming Language", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 356-370.
- [96] A. Tveit, "A Survey of Agent-Oriented Software Engineering", NTNU Computer Science Graduate Student Conference, Norwegian University of Science and Technology, May 2001.
- [97] D. A. Waterman, "A Guide to Expert Systems", Addison-Wesley, 1986.
- [98] D. Weerasooriga, A. Rao, K. Ramamohanarao, "Design of a Concurrent Agent-Oriented Language", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 386-401.
- [99] G. Weiss, "Distributed Machine Learning", Sankt Augustin: Infix Verlag, (ISBN 3-929037-75-0).

- [100] G. Weiss, "Adaptation and Learning in Multi-Agent Systems: Some Remarks and a Bibliography", *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 1-21.
- [101] G. Weiss, P. Dillenbourg, "What is 'multi' in multiagent learning?", P. Dillenbourg (Ed.), *Collaborative learning. Cognitive and computational approaches.*, Pergamon Press, 1999, pp. 64-80.
- [102] M. F. Wood, S. A. DeLoach, "An Overview of the Multiagent Systems Engineering Methodology", *Proceedings of The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000)*, 2000, pp. 207-222.
- [103] M. Wooldridge, "*The Logical Modeling of Computational Multi-Agent Systems*", PhD thesis, Department of Computation, UMIST, Manchester, UK, October 1992. (Also available as Technical Report MMU-DOC-94-01, Department of Computing, Manchester Metropolitan University, Chester St., Manchester, UK).
- [104] M. Wooldridge, "A Knowledge-Theoretic Semantics for Concurrent MetateM", *Working Notes of the Third International Workshop on Agent Theories, Architectures and Languages*, ECAI '96, Budapest, Hungary, pp. 279-293.
- [105] M. Wooldridge, N. R. Jennings, "Agent Theories, Architectures, and Languages: A Survey", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 1-39.
- [106] M. Wooldridge, N. R. Jennings, "Intelligent Agents: Theory and Practice", available as <http://www.doc.mmu.ac.uk:80/STAFF/mike/ker95/ker95-html.html>, 1994.
- [107] M. Wooldridge, N. R. Jennings, "Software Engineering with Agents: Pitfalls and Pratfalls", *IEEE Internet Computing* 3 (3): 20-27, 1999.
- [108] M. Wooldridge, N. R. Jennings, D. Kinny, "A Methodology for Agent-Oriented Analysis and Design", *Proceedings of the 3rd Int Conference on Autonomous Agents (Agents-99)*, Seattle, WA, 1999, pp. 69-76.
- [109] M. Wooldridge, N. R. Jennings, D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design", *Journal of Autonomous Agents and Multi-Agent Systems*, 3 (3): 285-312, 2000.
- [110] H. Yim, K. Cho, K. Jongwoo, S. Park, "Architecture-Centric Object-Oriented Design Method for Multi-Agent Systems", *Proceedings of the Fourth International Conference on MultiAgent Systems (ICMAS-2000)*, 2000.
- [111] J. Youll, J. Morris, R. C. Krikorian, P Maes, "Impulse: Location-based Agent Assistance", *Proceedings of the Fourth International Conference on Autonomous Agents (Agents 2000)*, Barcelona, Spain, 2000.

[112] F. Zambonelli, N. R. Jennings, A. Omicini, M. Wooldridge, "Agent-Oriented Software Engineering for Internet Applications", *Coordination of Internet Agents*, Springer Verlag, 2001. pp. 326-346.



## Figures

Figure 1 The architecture of a knowledge-based situated agent.....	16
Figure 2 Generic Agent Interpreter.....	31
Figure 3 States and possible transitions in the requesting negotiation EngagementInitReqNeg.....	60
Figure 4 States and possible transitions in the responding negotiation EngagementInitResNeg.....	62
Figure 5 States and possible transitions in the WWW negotiation.....	66
Figure 6 The appearance of the main window of an AJA agent.....	68
Figure 7 Java packages implementing AJA.....	130
Figure 8 Agent and RMI.....	131
Figure 9 Agent and its managers.....	132
Figure 10 Implementation of agent beliefs.....	133
Figure 11 Implementation of agent actions.....	134
Figure 12 Implementation of agent reflexes.....	135
Figure 13 Implementation of requesting and responding negotiations.....	136
Figure 14 Implementation of agent-to-agent communication.....	137
Figure 15 Implementation of web negotiation.....	139
Figure 16 Implementation of built-in agent GUI.....	140
Figure 17 Some of the classes in the package aja.translator.....	141
Figure 18 The directory structure after the extraction of AJA_1.0.zip.....	143
Figure 19 The file AJA_1.0\bin\1_comp.f.bat has been executed.....	145
Figure 20 The file AJA_1.0\bin\2_genstub.bat has been executed.....	145
Figure 21 The file AJA_1.0\bin\3_compt.bat has been executed.....	146
Figure 22 The file AJA_1.0\bin\4_compmain.bat has been executed.....	146
Figure 23 The file AJA_1.0\bin\5_gendoc.bat has been executed.....	147
Figure 24 HTML documentation of AJA classes.....	147
Figure 25 AJA_1.0\demo\MAS\bin\01_compDemo.bat has been executed.....	149
Figure 26 AJA_1.0\demo\MAS\bin\02_createKeys.bat has been executed.....	149
Figure 27 AJA_1.0\demo\MAS\bin\03_transA.bat has been executed.....	150
Figure 28 AJA_1.0\demo\MAS\bin\04_compA.bat has been executed.....	150
Figure 29 Dialog window with password text fields.....	151
Figure 30 Entering owner data of the agent A.....	151
Figure 31 Owner data for the agent A.....	152
Figure 32 The main window of the agent A.....	152
Figure 33 The console of the agent A.....	153
Figure 34 Owner data for the agent B.....	153
Figure 35 Owner data for the agent C.....	154
Figure 36 Owner data for the agent D.....	154
Figure 37 Four PDA Agents are executing on one computer.....	155
Figure 38 colleagues radio-button.....	156
Figure 39 Initial colleagues list of the agent A.....	156
Figure 40 new colleague dialog window.....	157
Figure 41 The owner of the agent A adds the owner of the agent D to its list of colleagues.....	157
Figure 42 Colleagues list of the agent A.....	158
Figure 43 your availability radio-button.....	158
Figure 44 The agent A has at the beginning no registered available time intervals.....	159
Figure 45 new availability interval dialog window.....	159
Figure 46 Defining an availability interval.....	160
Figure 47 Available time intervals of the agent A.....	160
Figure 48 engagements radio-button.....	161
Figure 49 Currently there are no engagements.....	161
Figure 50 A window for the engagement type specification.....	162
Figure 51 New engagement window.....	162
Figure 52 New engagement window after entering the engagement data.....	163
Figure 53 Agent B displays the new information in the status bar of its window.....	163
Figure 54 Engagements list of the Agent B.....	164
Figure 55 The owner of the agent B (Mirjana Ivanović) edits the new engagement.....	164

Figure 56 The first step in creating of a consultation.....	165
Figure 57 The second step in creating of a consultation.....	165
Figure 58 The third step in creating of a consultation.....	166
Figure 59 A consultation is created. ....	166
Figure 60 The second consultation is created. ....	166
Figure 61 The first html page in the agent-student dialog. ....	167
Figure 62 A check-box is selected. ....	168
Figure 63 The second page. ....	168
Figure 64 The purpose of the consultation. ....	169
Figure 65 The third page. ....	169
Figure 66 Student name and Id.....	170
Figure 67 The last page. ....	170
Figure 68 Editing the consultation. ....	171

## Tables

Table 1 Rewards in Iterated Prisoner Dilemma .....	27
Table 2 OOP and AOP .....	30
Table 3 Connectives in PML.....	34
Table 4 Mobile agent programming tools. ....	45

## Index

- Adaptable Parameters .....60, 101  
AGENT0 .....23, 38, 39, 40, 41, 55  
AgentSpeak .....43, 44, 50, 55  
Architecture..15, 23, 24, 25, 27, 28, 58,  
134, 184, 185, 186, 188  
Believable Agents .....18, 182  
Business Process Management .19, 185  
Contract Net Protocol .....27, 33, 34  
COOL.....48, 49, 51, 57, 133, 134  
Dependant Values .....61, 105  
Electronic Commerce.....18, 183  
FIPA ACL .....46, 51, 52, 183  
Gaia .....28, 29, 188  
Genetic Algorithms .....33  
HADL.... vi, vii, 58, 59, 78, 79, 80, 85,  
88, 133, 136, 137, 152, 194, 195  
HOMAGE .....48, 133  
Iterated Prisoner Dilemma ...16, 26, 34,  
35  
JACK.....44, 45, 133, 181  
Java+vi, vii, 59, 60, 61, 62, 63, 64, 70,  
71, 72, 74, 75, 76, 77, 78, 79, 80, 82,  
84, 85, 88, 89, 90, 91, 94, 96, 98,  
110, 133, 136, 137, 147, 152, 194  
KQML.....37, 49, 50, 51, 52, 183  
Learning...9, 14, 29, 30, 31, 32, 33, 34,  
181, 182, 183, 184, 185, 186, 187,  
188  
MaSE .....28, 29  
mobile agent.....12, 13, 33, 52  
PDA ...15, 17, 18, 60, 65, 72, 113, 114,  
116, 117, 119, 120, 130, 137, 156,  
163  
PLACA .....15, 23, 40, 41, 55, 187  
Possible Worlds Semantics.....21  
RoboCup .....35, 36, 181  
RPROP....101, 103, 107, 108, 110, 186  
Society of Mind .....14, 186  
Software Engineering .....28, 184, 186,  
187, 188, 189  
Speech Act Theory .....21  
SSL ..57, 71, 72, 79, 80, 85, 88, 89, 92,  
93, 136, 144, 145  
Stronger Notion of Agency.....11  
Weak Notion of Agency .....10  
ZEUS .....45, 46, 47, 181, 186

Glavni doprinos doktorske teze je napravljeni alat za programiranje agenata AJA. AJA – Adaptabilni Java Agenti je jezički alat za programsku implementaciju agenata. Sastoji se od dva programska jezika:

- Jezik višeg nivoa kojim se opisuju glavne komponente agenta. Ovaj jezik se naziva HADL – Higher Agent Definition Language.
- Jezik nižeg nivoa koji služi za implementaciju pojedinih komponenti agenta specificiranih HADL jezikom. Ovaj jezik se naziva Java+, jer je on zapravo programski jezik Java obogaćen konstrukcijama pomoću kojih je moguće pristupati komponentama agenta, definisanim u jeziku HADL.

AJA agent poseduje sledeće osobine:

- Sigurna komunikacija sa drugim AJA agentima koristeći pregovaranja, šifrovanje i digitalno potpisivanje poruka.
- Mogućnost adaptiranja na promene u okruženju u kom se nalazi, koristeći neuralne mreže i adaptabilne parametre.
- Reaktivnost zasnovana na komponenti zvanj refleks.
- Paralelno izvršavanje akcija agenta uz njihovu internu sinhronizaciju.
- Dostupnost agenta preko Interneta. Agent se ponaša kao jednostavan HTTP server. Na ovaj način se drugim osobama omogućuje da komuniciraju sa agentom.
- Grafički korisnički interfejs zasnovan na Java Swing tehnologiji.
- Pošto se u programiranju agenta koristi Java+, moguće je uposliti sve pogodnosti Jave, kao što su na primer pristup bazama podataka koristeći JDBC, rad sa multi-medijalnim sadržajem, itd.

U tezi je predstavljen i originalni pristup integrisanja tehnika veštačke inteligencije sa programskim jezikom. Ugrađujući komponente veštačke inteligencije u izvršnu okolinu jezika čini njihovo korišćenje veoma jednostavnim. Programer ne mora da bude ekspert iz veštačke inteligencije a da pri tome koristi konstrukcije jezika koje su implementirane pomoću veštačke inteligencije.

AJA specifikacija agenta se sastoji od HADL i Java+ delova. U tezi je implementiran prevodioc kojim se AJA specifikacija prevodi u skup klasa programskog jezika Java. Implementiran je i jedan multi-agentski sistem kojim se praktično pokazuje korišćenje i mogućnosti napravljenog alata.

Doktorska teza sadrži i detaljan pregled oblasti o agentskoj metodologiji. Ona kruniše višegodišnji rad kandidata i njegovog mentora u ovoj sve značajnijoj oblasti računarstva.

Teza sadrži osam glava i tri dodatka. U prvoj glavi se opisuje oblast agenata i multi-agentskih sistema. Pregled postojećih agentskih programskih jezika i alata se daje u drugoj glavi. Opis AJA agenata i njihove arhitekture je dat u trećoj glavi teze. Četvrta glava se bavi sintaksom i semantikom oba AJA jezika: HADL-a i Jave+. Elementi veštačke inteligencije AJA agenata se opisuju u petoj glavi. U šestoj glavi je opisan multi-agentski sistem koji je ujedno i primer primene AJA alata. AJA se sa drugim postojećim agentskim alatima upoređuje u sedmoj glavi. Osmo glava sadrži zaključak. Na kraju se u tri dodatka detaljno opisuju implementacija prevodioca AJA-e u Javu, instalacija prevodioca i korišćenje napravljenog multi-agentskog sistema respektivno. U doktorskom radu su korišćene i navedene brojne reference kojima su obuhvaćeni gotovo svi najznačajniji i najaktuelniji radovi iz oblasti multi-agentskih sistema. Lista referenci je navedena na kraju teze.

## Kratka biografija



Mihal Badjonski je rođen 16.08.1971. u Novom Sadu. Osnovnu i srednju školu je završio u Bačkom Petrovcu sa odličnim uspehom. Posle završene srednje škole, 1990-te godine, upisuje studije informatike na Prirodno-matematičkom fakultetu u Novom Sadu. Redovne studije završava u roku, 7.7.1995. sa prosečnom ocenom 9.86. Nakon diplomiranja zapošljava se na Institutu za matematiku Prirodno-matematičkog fakulteta u Novom Sadu kao asistent pripravnik, gde i upisuje postdiplomske studije. Postdiplomske studije završava 6.7.1998. odbranom magistarskog rada "Implementacija multiagentskih sistema na jeziku Java" sa prosečnom ocenom 10. Dva puta je nagrađivan nagradom Mileva Marić-Ajnštajn: za uspešne redovne studije i za napisani magistarski rad.

Kao asistent pripravnik Mihal Badjonski je držao vežbe iz više informatičkih i matematičkih predmeta, između ostalih i uže stručne predmete na smeru "Diplomirani informatičar": Strukture podataka i algoritmi, Programski jezici i Uvod u programiranje. 16.09.1999. godine izabran je u zvanje asistenta. Od 15.01.2001 se nalazi na privremenom radu u Nemačkoj.

Mihal Badjonski je do sada objavio 30 naučnih radova iz oblasti agentno-orijentisanog programiranja, multi-agentnih sistema, ekspertnih sistema i informacionih sistema. Od toga je 5 radova objavljeno u međunarodnim naučnim časopisima a 16 radova je objavljeno u zbornicima radova sa međunarodnih konferencija. Autor je i jednog udžbenika iz programskih jezika.

Novi Sad, 12 June, 2003.

*Mihal Badjonski*

*Mihal Badjonski*



UNIVERZITET U NOVOM SADU  
PRIRODNO - MATEMATIČKI FAKULTET  
KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj:

RBR

Identifikacioni broj:

IBR

Tip dokumentacije: Monografska dokumentacija

TD

Tip zapisa: Tekstualni štampani materijal

TZ

Vrsta rada: Doktorska disertacija

VR

Autor: Mihal Badjonski

AU

Mentor: Mirjana Ivanović

MN

Naslov rada: Adaptable Java Agents (AJA) – a Tool for Programming of Multi-Agent Systems

MR

Jezik publikacije: *Engleski*

JP

Jezik izvoda: *s / e*

JI

Zemlja publikovanja: Srbija i Crna Gora

ZP

Uže geografsko područje: Vojvodina

UGP

Godina: 2003

GO

Izdavač: Autorski reprint

IZ

Mesto i adresa: Novi Sad, Trg Dositeja Obradovića 4

MA

Fizički opis rada: (8/195/112/4/68/0/0)

FO

Naučna oblast: Informatika

NO

Naučna disciplina: Multi-agentski sistemi

ND

Ključne reči: agent, multi-agentski sistem, programski jezici, Java, distribuirana veštačka inteligencija

PO

UDK:

Čuva se:

ČU

Važna napomena:

VN

Izvod:

Glavni doprinos doktorske teze je napravljeni alat za programiranje agenata AJA. AJA – Adaptabilni Java Agenti je jezički alat za programsku implementaciju agenata. Sastoji se od dva programska jezika:

- Jezik višeg nivoa kojim se opisuju glavne komponente agenta. Ovaj jezik se naziva HADL – Higher Agent Definition Language.

- Jezik nižeg nivoa koji služi za implementaciju pojedinih komponenti agenta specificiranih HADL jezikom. Ovaj jezik se naziva Java+, jer je on zapravo programski jezik Java obogaćen konstrukcijama pomoću kojih je moguće pristupiti komponentama agenta, definisanim u jeziku HADL.

AJA agent poseduje sledeće osobine:

- Sigurna komunikacija sa drugim AJA agentima koristeći mehanizam pregovaranja, šifrovanje i digitalno potpisivanje poruka.

- Mogućnost adaptiranja na promene u okruženju u kom se nalazi, koristeći neuralne mreže i adaptabilne parametre.

- Reaktivnost zasnovana na komponenti zvanj refleksi.

- Paralelno izvršavanje akcija agenta uz njihovu internu sinhronizaciju.

- Dostupnost agenta preko Interneta. Agent se ponaša kao jednostavan HTTP server. Na ovaj način se drugim osobama omogućuje da komuniciraju sa agentom.

- Grafički korisnički interfejs zasnovan na Java Swing tehnologiji.

- Pošto se u programiranju agenta koristi Java+, moguće je uposliti sve pogodnosti Jave, kao što su na primer pristup bazama podataka koristeći JDBC, rad sa multi-medijalnim sadržajem, itd.

U tezi je predstavljen i originalni pristup integrisanja tehnika veštačke inteligencije sa programskim jezikom. Ugrađujući komponente veštačke inteligencije u izvršnu okolinu jezika čini njihovo korišćenje veoma jednostavnim. Programer ne mora da bude ekspert iz veštačke inteligencije a da pri tome koristi konstrukcije jezika koje su implementirane pomoću veštačke inteligencije.

AJA specifikacija agenta se sastoji od HADL i Java+ delova. U tezi je implementiran prevodioc kojim se AJA specifikacija prevodi u skup klasa programskog jezika Java. Implementiran je i jedan multi-agentski sistem kojim se praktično pokazuje korišćenje i mogućnosti napravljenog alata.

Doktorska teza sadrži i detaljan pregled oblasti o agentskoj metodologiji. Ona kruniše višegodišnji rad kandidata i njegovog mentora u ovoj sve značajnijoj oblasti računarstva.

Teza sadrži osam glava i tri dodatka. U prvoj glavi se opisuje oblast agenata i multi-agentskih sistema. Pregled postojećih agentskih programskih jezika i alata se daje u drugoj glavi. Opis AJA agenata i njihove arhitekture je dat u trećoj glavi teze.

Četvrta glava se bavi sintaksom i semantikom oba AJA jezika: HADL-a i Java+. Adaptabilni elementi AJA agenata se opisuju u petoj glavi. U šestoj glavi je opisan multi-agentski sistem koji je ujedno i primer primene AJA alata. AJA se sa drugim postojećim agentskim alatima upoređuje u sedmoj glavi. Osmu glavu sadrži zaključak. Na kraju se u tri dodatka detaljno opisuju implementacija prevodioca AJA-e u Javu, instalacija prevodioca i korišćenje napravljenog multi-agentskog sistema respektivno. U doktorskom radu su korišćene i navedene brojne reference kojima su obuhvaćeni gotovo svi najznačajniji i najaktuelniji radovi iz oblasti multi-agentskih sistema. Lista referenci je navedena na kraju teze.

IZ

Datum prihvatanja teme od strane NN veća:

DP

Datum odbrane:

DO

Članovi komisije:

KO

Predsednik: Dr Zoran Budimac, vanredni profesor, Prirodno-matematički fakultet Novi Sad

Član: Dr Mirjana Ivanović, vanredni profesor, Prirodno-matematički fakultet Novi Sad

Član: Dr Dušan Tošić, vanredni profesor, Matematički fakultet Beograd

Član: Dr Živko Tošić, redovni profesor, Elektronski fakultet Niš

UNIVERSITY OF NOVI SAD  
FACULTY OF SCIENCE KEY  
WORDS DOCUMENTATION

Accession number:

ANO

Identification number:

INO

Document type: Monograph type

DT

Type of record: Printed text

TR

Contents Code: PhD thesis

CC

Author: Mihal Badjonski

AU

Mentor: Mirjana Ivanović

MN

Title: Adaptable Java Agents (AJA) – a Tool for Programming of Multi-Agent Systems

XI

Language of text: *English*

LT

Language of abstract: *s/e*

LA

Country of publication: Serbia and Montenegro

CP

Locality of publication: Vojvodina

LP

Publication year: 2003

PY

Publisher: Author's reprint

PU

Publ. place: Novi Sad, Trg Dositeja Obradovića 4

PP

Physical description: (8/195/112/4/68/0/0)

PD

Scientific field: Computer Science

SF

Scientific discipline: Multi-Agent Systems

Key words: agent, multi-agent system, programming languages, Java, distributed artificial intelligence

UC:

Holding data:

HD Note:

Abstract:

The main goal of this thesis is the creation of the tool agent-oriented programming tool AJA. AJA is the acronym for Adaptable Java Agents. AJA consists of two programming languages:

- A higher-level language used for the description of the main agent parts. This language is called HADL, which is the acronym for Higher Agent Definition Language.
- A lower-level language used for the programming of the agent parts defined in HADL. This language is called Java+. It is actually Java enriched with the constructs for accessing higher-level agent parts defined in HADL.

A translator from AJA to Java is implemented in the practical part of the thesis.

AJA agents have the following features:

- Agent communicates with other agents using a construct called negotiation. The messages sent can be encrypted or digitally signed in order to ensure the security of the system.
- Agent possesses adaptable parameters and neural nets that adapt themselves when the environment changes.
- Agent has reflexes, which are the reactive component of the agent architecture.
- Agent can perform its actions parallel. Actions execution is synchronized.
- Agent is accessible via Internet, because it acts as a simple HTTP server. People can use this way to communicate with an agent.
- Agent has Java Swing based graphical user interface. Its owner uses this interface to communicate with the agent.
- Because of the fact that Java+ language extends Java, it is possible to use all useful Java features in the implementation of AJA agents (e.g. JDBC for the database access).

The thesis also presents an original approach of integrating artificial intelligence techniques, such as neural nets, with a programming language. Having the artificial intelligence components as a part of the programming language runtime environment makes their use straightforward. A programmer uses the language constructs that are implemented using the artificial intelligence without the need for understanding their background and theory.

The thesis contains eight chapters and three appendixes. In the first chapter, an overview of agents and multi-agent systems is given. The second chapter surveys existing agent-oriented programming languages and tools. The third chapter introduces AJA and describes the architecture of AJA agents. The syntax and semantics of AJA languages HADL and Java+ is described in the fourth chapter. The fifth chapter presents adaptable AJA constructs in more details. To demonstrate and test the created tool, a case-study multi-agent system has been implemented in AJA. There are four personal digital assistant agents in the system. The sixth chapter describes the example agents and positively evaluates the tool. In the seventh chapter the related work and tools are analyzed and compared to AJA. The last chapter concludes the thesis. The first appendix describes the implementation details of the AJA to Java translator. The second appendix is a guide for the installation and usage of the implemented AJA to Java translator. Finally, the third appendix describes step by step how to translate, compile, run, and use the example agents. The thesis contains many references, which include almost all the most important and the most actual papers in the field. The reference list can be found at the end of the thesis.

AB

Accepted by the Scientific Board on:

Defended:

Thesis defend board:

President: Dr. Zoran Ćudimac, associate professor, Faculty of Science, Novi Sad

Member: Dr. Mirjana Ivanović, associate professor, Faculty of Science, Novi Sad

Member: Dr Dušan Tošić, associate professor, Faculty of Mathematics, Belgrade

Member: Dr Živko Tošić, full professor, Faculty of Electronics, Niš

