NOVI SAD J. MATH.
VOL. 30, NO. 2, 2000, 1-10

Proceedings of the TARA 2000 Conference
Novi Sad, Yugoslavia, September 6-7, 2000

AJA - ADAPTABLE JAVA AGENTS

Mihal Badjonski[1], Mirjana Ivanović[1]

Abstract. Software agents usually operate in dynamic, unpredictable environments with many unforeseen features that change over time. In order to be effective an agent has to adapt itself to the changes in its environment. In this paper we present AJA, an agent development tool that can be used for the development of adaptable agents implemented in Java. In addition to adaptability, AJA agents possess other attractive features as well, such as negotiation capability realized using conversation scenarios, reactivity to events in the agent environment, meta-capability, accessibility via World Wide Web (WWW), etc.

AMS Mathematics Subject Classification (1991): 68N20

Key words and phrases: software agents, machine learning, artificial intelligence, Java.

1. Introduction

Currently, agents are the focus of intense interest on the part of many subfields of computer science [8]. Agent applications herald a fundamentally new paradigm for developing and implementing complex systems. The agent based system development paradigm involves building sophisticated, self-contained components, which can interact flexibly with a number of independently developed similar components [7].

In most cases, an agent is embedded in an unpredictable and dynamic environment. Therefore, agent needs a capability to adapt itself to new circumstances that emerge during its life. One way of achieving this adaptation is to use machine learning.

In this paper, a language-based agent development tool AJA is presented. AJA is aimed for the development of adaptable software agents in Java.

The remainder of this paper is structured as follows. A detailed description of AJA is given in the next section, which is also the most significant part of the paper. The third section is concerned with contributions related to our work and some conclusions.

[1]Institute of Mathematics, Faculty of Science, Trg Dositeja Obradovića 4, 21000, Novi Sad, Yugoslavia, e-mail: michal@eunet.yu, mira@unsim.ns.ac.yu

2. AJA - Adaptable Java Agent

AJA is a language-based development tool for adaptable software agents. The attribute language-based is used in the previous sentence, because the main parts of AJA are two programming languages:

- the programming language HADL (Higher Agent Definition Language), which is used for the definition of higher-level agent features, and
- the programming language Java+, which is used for the definition of lower-level agent parts.

Java+ is the standard Suns Java programming language extended with new, agent-specific statements and constants. Both HADL source code and Java+ source code are translated into Java.

In addition to HADL and Java+ translators to Java, AJA also contains Java classes that are used at run-time for agent execution. These classes implement the generic architecture of AJA agent.

2.1. The Generic Architecture of AJA agent

In this subsection we describe the generic architecture of AJA agents. These components are depicted in Figure 1.

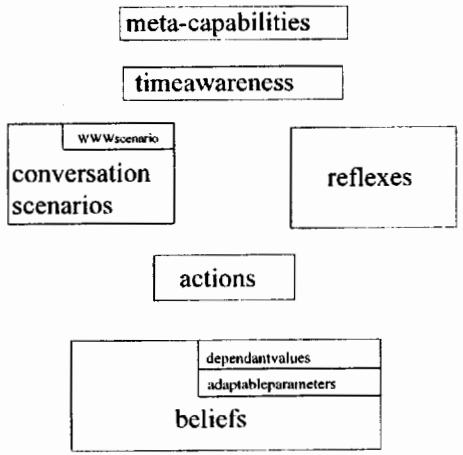

Figure 1: Components of AJA agents.

- *beliefs* - Beliefs are internal data structures in which agent stores the facts it believes that are true. These facts represent all what agent knows about itself, about other agents in the system, about its environment, and about the world. In AJA, a belief can be any cloneable and serializable Java object or Java primitive value, as well as a special value determined by one of the two available machine-learning components.

- *machine-learning components* - There are two machine learning components embedded into AJA agents. They enable the use of the following constructs:

  - *dependant values* - A dependant value depends on some other values. A dependant value can be used when the dependency cannot be analytically obtained. A dependant value is calculated using a neural net. For each dependant value inside an agent there exists one feed-forward back-propagation neural net. The agent programmer can specify parameters for each neural net, such as number of nodes and number of hidden layers. Otherwise, default values are used. A neural net can be trained at the beginning of the agent life if a file with training examples is provided. This learning is off-line supervised learning. During its life agent receives feedback for its actions. This feedback is then used for on-line reinforcement learning of the neural net used.

  - *adaptable parameters* - The second machine learning component controls adaptable parameters. An adaptable parameter is a more or less constant value, determined experimentally. The agent programmer defines the initial value of an adaptable parameter. During agents life, it receives feedback regarding the value of the adaptable parameter. Using this feedback, the machine-learning component slightly adjusts the value of the adaptable parameter.

- *actions* - Agent performs some actions. The body of an action in AJA consist of Java+ statements, i.e. standard Java statements, statements that access and modify agent beliefs, statements that invoke other actions execution, and statements that initiate conversation processes with other agents. Several actions can be executed simultaneously in separate Java threads. As there can be actions that cannot be executed simultaneously care is taken to synchronize their execution.

- *conversations* - Whenever two AJA agents communicate, they are involved in a conversation process. Every AJA agent has a predefined set of conversation scenarios. A conversation scenario can be seen as an automaton. When conversation starts, the automaton is in its starting state. Depending on the state of the automaton, agent executes appropriate Java+ code and changes the state of the automaton. Conversation process ends when a final state of the automaton is reached.

- *reflexes* - Reflexes are reactive components of AJA agent architecture. A reflex is a condition-actions pair. Every reflex has its priority. Condition parts of agent reflexes are evaluated periodically. Reflexes with highest priority from those whose conditions are satisfied are then selected for execution.

- *meta-capabilities* - At any time point an agent can find out the names of the actions that are being executed as well as the names of the actions that are waiting to be executed. An action execution can be postponed due to incompatibility with currently executing actions (synchronization).
- *web accessibility* - An agent has its HTTP address and it is accessible through the WWW. When one accesses the agent using an Internet browser, an instance of a special conversation scenario of the agent starts to execute.
- *awareness of time* - An agent is aware of time. It can delay an action execution until any specified time point in the future. It can also pause the execution of the current action for a given time period.

An AJA agent does not have to use all of the components provided by the generic AJA agent architecture.

2.2. HADL - Higher Agent Definition Language

Higher-level parts of an AJA agent are specified using HADL language. Terminal symbols of the grammar are written using capital letters, while non-terminal symbols are written using lowercase letters.

HADL specification is structured as follows:

```
specification =
  AGENT agentname
  LOCATED ON hostname [RMI portnumber][HTTP portnumber]
  [PICTURE filename]
  [javaimport]
  [beliefsDeclaration]
  [actionsDeclaration]
  [conversationScenariosDeclaration]
  [webConversationScenarioDeclaration]
  [reflexesDeclaration]
  [initialization]
  END agentname '.'
```

Every AJA agent has its name and it is located on a computer. The name of the computer where the agent is located is the Internet name of that computer (e.g. bambi.im.ns.ac.yu). Agent-to-agent communication in AJA is implemented using Java Remote Method Invocation (RMI). Agent specification can therefore contain the port number of the `rmiregistry`. Similarly, the HTTP port number can also be specified. Using this port the agent is accessible via WWW.

An AJA agent is a Java program that has its window. One element of this window is a picture of the agent.

The next part of HADL program, also optional one, is the Java import declaration. Here, the agent programmer declares Java packages, classes, and/or interfaces from various packages that are used in the agent implementation.

2.2.1 Agent Beliefs

Declaration of agent beliefs has the following syntax rule:

```
beliefsDeclaration = BELIEFS belDecl ; { belDecl ;}

belDecl =
  belName : javaType [ = initValue ] |
  belName : ADAPTABLE = initValue [LBOUND value] [UBOUND value] |
  belName : DEPENDS ON list [EXAMPLESFILE fileName] [netConf]

list = belNameOrdoubleExpression {, belNameOrdoubleExpression}

belNameOrdoubleExpression = belName | doubleExpression
```

Every belief has its name. Type of a belief can be any cloneable and serializable Java type, or the belief can be an adaptable parameter, or it is a dependant value that is calculated using a neural net. Lower and/or upper bounds for the value of this belief can be specified.

The declaration of a belief that is a dependant value contains the keywords **DEPENDS ON** followed by the names of beliefs and/or Java+ **double** expressions it depends on. The value of the dependant belief is calculated using a neural net. The file containing training examples for the neural net can be specified as well as the neural net configuration (number of hidden layers and number of nodes in each layer). A default neural net configuration is used when a user-specified neural net configuration is omitted. During agent life, the reinforcement received will be used for further training of the neural net.

2.2.2 Agent Actions

Agent actions are declared using the following rules.

```
actionsDeclaration = actionDecl { actionDecl }

actionDecl =
  ACTION (returnType | void) actionName ( [parameters] )
  [INCOMPATIBLE actionName {, actionName }]
  body

body = { java+ }
```

The main part of an action declaration is the segment of Java+ code whose execution corresponds to the action execution. An action can have parameters and it can return a value, what makes actions similar to Java methods. However, the body of an AJA action is a Java+ code, i.e. agent-oriented constructs defined in AJA can be used in addition to standard Java statements.

Since AJA agent can have several actions executed at the same time there is a need for actions synchronization. Therefore, a declaration of each action contains part where the names of incompatible actions are listed.

2.2.3 Conversation Scenarios

Agents in AJA multi-agent system communicate by exchanging messages. A message contains an array of Java objects. The first element of this array has to be a `String` object. There are no restrictions on types and number of other objects in the array. The `String` object should be used as a speech-act performative (e.g. *request*, *ask*, *cancel*, *deny*, *confirm*) in the chosen agent communication language (ACL). The agent developer chooses the ACL it finds the most suitable for its system.

To jointly solve their problems, agents usually need to exchange several messages, i.e. they are involved in conversations. Therefore, instead of handling each received or sent message independently, AJA agent possesses conversation scenarios that group several logically related message sending and message receiving into appropriate structures.

Simultaneously, agent can be involved in several conversations.

The syntax rules for conversation scenarios are as follows.

```
conversationScenariosDeclaration = convSceDecl {convSceDecl}

convSceDecl = CONVERSATION SCENARIO (returnType | void)
              scenarioName ( [parameters] )
              [activCondition] convBody

activCondition = CONDITION { java+ }

convBody = { java+ } startState {state}

startState = START { java+ }

state = [FINAL] stateName { java+ }
```

A conversation scenario can be seen as an automaton. It contains a finite number of states. Exactly one state of the automaton is the starting state. In each state of the automaton agent executes a block of Java+ code and changes the state of the automaton. There can be zero, one or more final states of the automaton. When a final state of the automaton is reached and its code is executed, the conversation ends.

Each conversation scenario has its name, it can return a value and it can have parameters. At the beginning of the conversation scenario body a Java+ code can be used to declare local variables visible in all states of the automaton.

2.2.4 WWW Conversation Scenario

WWW conversation scenario is similar to the conversation scenarios described above. The only difference is in the type of conversation participants. In WWW conversation the agent communicates with a person through his/her web browser. In WWW conversation scenario special input-output Java+ statements have to be used. Moreover, due to client-server nature of HTTP protocol, there are some restrictions on usage of these special input-output statements.

The syntax rule for WWW conversation scenario declaration is:

```
webConversationScenarioDeclaration =
  WWW CONVERSATION SCENARIO convBody
```

Since there is at most one WWW conversation scenario in the agent program, it has no name. It does not possesseither return value or parameters. WWW conversation scenario activates whenever a new HTTP request is received.

2.2.5 Reflexes

A reflex consists of a condition that has to be satisfied in order to activate the reflex and actions that sequentially execute when the reflex is activated and there are no other activated reflexes with higher priority. The syntax rule for reflexes is:

```
reflexesDeclaration =
  [REFLEX CHECKING PERIOD numOfms] reflDec { reflDec }

reflDec = REFLEX reflexName [PRIORITY number]
  CONDITION { java+ }
  EXEC actionName ( realParameters )
  { ; actionName ( realParameters ) } [;]
```

Conditions of reflexes are checked periodically. The period for reflex condition checking can be specified (in milliseconds). Besides periodical reflex checking, a Java+ statement `$CHECK_REFLEXES` can be used in any agent part to force immediate reflex checking.

2.2.6 Agent Initialization

At the beginning of the agent life its beliefs can be initialized and initial actions can be executed.

```
initialization = INITIALIZATION { java+ }
```

2.3 Java+

Java+ is Java language extended with necessary constructs that are needed to access and control higher-level components of AJA agent.

- Statements that access agent beliefs: `$GET_BEL(`*`belName`*`)`, `$GET_BEL_COPY(`*`belName`*`)`, and `$SET_BEL(`*`belName, newValue`*`)`.
- Statements that give reinforcements to adaptable parameters and dependant values: `$AP_BAD(`*`belName`*`)`, *belName*, `$AP_HIGHER(`*`belName`*`)`, `$AP_LOWER(`*`belName`*`)`, `DV_BAD(`*`belName`*`)`, `DV_SHOULD_BE(`*`belName, value`*`)`.
- Constants such as `$AGENT_NAME`, `$AGENT_HOST`, `$AGENT_RMI_PORT`, `$AGENT_HTTP_PORT`, and `$AGENT_URL` whose values describe various agent properties.

- Statements that are used for the communication with other agents: `$SEND(agentURL, message, obj, obj,...)`, `$REPLY(agentURL, messageID , message, obj, obj,...)`, `$ANSW_TO_MESSAGE(messageId)`, and `$ANSW_TO_MESSAGE_WAIT(messageId)`.

- Statements that invoke action executions: `$INVOKE(actionName(`par_1, par_2, ..., par_n`))`, `$INVOKE_WAIT(actionName(` par_1, par_2, ..., par_n`))`, and `$INVOKE_AT(time, actionName(`par_1, par_2, ..., par_n`))`.

- Constructs related to conversation scenarios are: `$START_CONVERSATION(convName (`par_1, par_2, ..., par_n`))`, `$START_CONVERSATION_WAIT (convName(`par_1, par_2, ..., par_n`))`, `$START_CONVERSATION_AT(time, convName(`par_1, par_2, ..., par_n`))`.

- Statements related to time: `$NOW`, `$WAIT(h, m, s, ms)`, and `$WAIT_UNTIL(dateTime)`.

- Meta-constructs: used for querying which actions are currently executed, which actions are waiting to be executed, which conversation scenarios are active, and meta-constructs for canceling the executing and waiting actions and active conversations:

- Other statements: `$CHECK_REFLEXES` for immediate reflex checking and triggering and various input-output statements that enables user-friendly agent-user and agent-web user communication.

Some of the above mentioned elements cannot be used in some contexts. For example, the agent belief modification can occur only as part of the action execution or in the initialization part of agent program. For that reason statements `$SET_BEL` and `$GET_BEL` can be used only in this parts of agent program. Nevertheless, all beliefs can be read in any part of agent program using `$GET_BEL_COPY` statement. This restriction to belief modification is introduced in order to avoid simultaneous modification of agent beliefs. Actions are allowed to modify beliefs, because their execution is synchronized.

3. Related work and conclusion

Many of the concepts and ideas used for the creation of AJA are not new. However, the way these ideas are compounded together and implemented in AJA is unique. In this section we briefly mention the works that are related to our undertaking.

A significant influence on our current work has our previous work. AJA is a direct descendant of the agent-oriented language LASS [2], [3]. To implement LASS in Java, we have implemented a Java package LASSMachine [1]. Nevertheless, soon we realized that instead of making a new independent agent-oriented programming language, the better idea is to enhance Java with suitable agent-oriented constructs. For that reason we have developed AJA as an enrichment of Java language.

The idea of enhancing Java by adding agent constructs that we have adopted for the creation of AJA has also been adopted by researchers in the Australian company Agent Oriented Software Pty. Ltd. They have developed agent development tool Jack [6]. Besides the similarity in the approach, there are no other similarities between Jack and AJA.

Conversation scenario in AJA contains automaton that controls the conversation. The idea of using automaton as agent coordination technique is adopted from the work of Barbuceanu and Fox. In their language COOL [4] they use automata to program inter-agent coordination.

Reflexes of AJA agents are similar to behaviors in Subsumption Architecture [5]. Behaviors are mostly used for robot programming, whereas reflexes enable the software agent to react to the events in its environment.

At last but certainly not least significant, we find the idea of Schoepke expressed in [9] very inspiring for our work. Here Schoepke suggests that intelligent agents can be a vehicle that introduces other AI-related technologies into mainstream computer science. Following his suggestion, we have included two machine-learning components into AJA agent generic architecture. An application programmer that uses AJA does not need to know anything about machine learning. However, using adaptable parameters or dependent values he/she will actually use machine learning in its product.

There is room for further improvements of AJA in the future work.

Although AJA agent can communicate only with other AJA agents care has been taken during AJA development to leave an opportunity of making AJA agents FIPA compliant in the future.

Another way to improve AJA is to make AJA agents more intelligent. Other AI-related components can be included into generic AJA agent architecture.

References

[1] Badjonski, M., Implementation of Multi-Agent Systems using Java, M.Sc. thesis, University of Novi Sad, 1998.

[2] Badjonski, M., Ivanović, M., LASS - A Language for Agent-Oriented Software Specification. In Proceedings of VIII Conference on Logic and Computer Science Lira 97. pp. 1-4. Novi Sad, Yugoslavia, 1997.

[3] Badjonski, M., Ivanović, M., Budimac, Z., Software Specification using LASS. In: Advances in Computing Science, Lecture Notes in Computer Science, Vol. 1345. (R. K., Shyamasundar, K., Ueda, eds.), pp. 375-376. Springer-Verlag, Berlin Heidelberg New York, 1997.

[4] Barbuceanu, M., Fox, M., S., The Design of a Coordination Language for Multi-Agent Systems. In: Intelligent Agents III: Agent Theories, Architectures and Languages, Lecture Notes in Artificial Intelligence 1193. (J.P.Muller, M.J.Wooldridge, N.R.Jennings, eds.), pp. 341-357. Springer Verlag, 1997.

[5] Brooks, R. A., Intelligence without Reason. In: Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91). pp. 569-595. Sydney, Australia, 1991.

[6] Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A., JACK Intelligent Agents - Components for Intelligent Agents in Java. In: AgentLink News 2. (P. Davidsson, ed.) pp. 2-5. ISSN 1465-3842, http://www.agentlink.org, 1999.

[7] Jennings, N. R., Agent Software. In: Proceedings of UNICOM Seminar on Agent Software. pp. 12-27. London, UK, 1995.

[8] Jennings, N. R., Wooldridge, M., Application of Intelligent Agents. In: Agent Technology: Foundations, Applications, and Markets. (N. R. Jennings, M. Wooldridge, eds.), pp. 3-28, 1998.

[9] Schoepke, S. H., Intelligent agents will be a vehicle for other AI-related technologies. In: Proceedings of International Workshop on Agent-Oriented Information Systems (AOIS'99). To appear (available as http://www.aois.org/99/schoepke.html).

[10] Weerasooriga, D., Rao, A., Ramamohanarao, K., Design of a Concurrent Agent-Oriented Language. In: Intelligent Agents, Lecture Notes in Artificial Intelligence, Vol. 890. (M. Wooldrigde, N.R. Jennings, eds.), pp. 386-401. Springer-Verlag, 1994.

[11] Wooldridge, M., Jennings, N. R., Agent Theories, Architectures, and Languages: A Survey. In: Intelligent Agents, Lecture Notes in Artificial Intelligence, Vol. 890. (M. Wooldrigde, N.R. Jennings, eds.), pp. 1-39. Springer-Verlag, 1994.

[12] http://www.nortelnetworks.com/products/announcements/fipa/info.html.