AJA – Tool for Programming Adaptable Agents

Mihal Badjonski[1], Mirjana Ivanović[1], Zoran Budimac[2]
1 Datenknecht GmbH, Frankfurt am Main, Germany
mbadjonski@yahoo.com

2 Department of Mathematics and Informatics, Faculty of Science,
University of Novi Sad,
Trg D. Obradovića 4, 21000 Novi Sad, Serbia & Montenegro
{mira,zjb}@im.ns.ac.yu

Abstract. Agent-building tools have an important role in popularizing and application of agent technology. This paper describes a new agent-programming tool AJA. AJA consists of two programming languages: HADL for defining of higher-level agent constructs and Java+ for low-level programming of these constructs. Among other interesting features AJA presents an original approach of incorporating artificial neural nets, into a programming language.

1 Introduction

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 proc-ess. Beside the implementation issues, e.g. network communication among agents, there are many design and higher-level problems to be solved.

One way to facilitate agent transition from research laboratories to mainstream programming is to build and provide agent-oriented programming tools such as agent-oriented programming languages, agent-oriented integrated development environ-ments, (IDEs) and agent-oriented design tools.

This paper presents AJA – a tool for programming of multi-agent systems based on adaptable java agents (AJA). Using an original approach, AJA combines Java with higher-level agent-constructs and with AI-components and thus provides an effective environment for the programming of multi-agent systems.

Please use the following format when citing this chapter:
Badjonski, Mihal, Ivanovic, Mirjana, Budimac, Zoran, 2006, in IFIP International Federation for Information Processing, Volume 204, Artificial Intelligence Applications and Innovations, eds. Maglogiannis, I., Karpouzis, K., Bramer, M., (Boston: Springer), pp. 516–523

2 AJA features

The design of AJA is based upon the positive and negative experience obtained in de-signing and implementing the agent-oriented language LASS ([1], [2], [3], [4]) as well as upon the analyses of the existing agent building tools and environments. AJA tool is based on the following ideas.

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, PLACA, Concurrent MetateM, AgentSpeak, 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.

The goal of our work was to create an agent-development 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.

The existing programming languages, first of all Java, already support many 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 spec-ify high-level agent parts and Java can be used to implement these components.

2.1 AJA Agent Architecture

AJA agents can be classified as: software, static, middle-sized and big-sized agents, agents that can learn agents with hybrid architecture, and agents for both cooperative and competitive systems. An AJA agent consists of the following parts:

- Beliefs – primitive values, data structures, and the values generated by AI compo-nents.
- Actions – blocks of code 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.
- Negotiations – represent the inter-agent communication as automaton. There are three types of negotiations in AJA: requesting negotiation – used when agent initi-ates the communication; responding negotiation – used when agent responses to the request; WWW negotiation – used for user-agent communication via Internet.
- Initialization part – actions that execute after agent has been activated.

Two languages are used for the programming of AJA agents:

- HADL (Higher Agent Definition Language) – provides 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. Java+ makes the agent integration with the legacy Java software straightforward.

Agent program written in HADL and Java+ is translated into Java.

2.1.1 Beliefs

Beliefs of an AJA agent represent the information it has about the world. Beliefs define agent's internal state. There are three types of beliefs in AJA: Java values (both primitive and compound); Adaptable parameters; Dependant values.

Java Values - resembles global variables in traditional programming languages. They store primitive or compound values.

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. One possibility is to use adaptable parameters (real num-bers) instead of constants. Adaptable parameter adjusts its value at run-time according to the feedback received. Adaptable parameter in AJA has two optional attributes: lower bound, upper bound. The Listing 2. shows an example of adaptable parameter declaration.

Listing 1.

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

Dependant Values - are implemented using artificial neural network (ANN). A pro-grammer 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 de-pendant values (i.e. ANNs) can be, with no doubt, very useful in programming of in-telligent agents. Dependant values are declared as shown in the Listing 3.

Listing 2.

```
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.

2.1.2 Actions

An AJA action consists of a block of Java+ code, return value type, and parameters. 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. In the Listing 4. declarations of an AJA action are given.

Listing 3.

```
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 pop-ups a dialog window that alerts user about the incoming engagement. The class AlertDialog extends javax.swing.JDialog. $AG_JFRAME is the reference of the agent window (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 ac-tions.

AJA action can be started explicitly using one of the three Java+ constructs. Further-more, an AJA action can also be started as a result of reflex triggering.

2.1.3 Reflexes

The first part of AJA reflex is activation condition, whose value determines whether the second part of the reflex should be executed or not. Reflex activation condition is checked periodically. Each reflex executes in separate thread. If there are more than one reflexes being executed at the same time, then they execute concurrently in paral-lel threads of execution. There is only one Java+ construct used with reflexes.

2.1.4 Negotiations

Requesting and responding negotiations are the construct an agent uses in order to communicate with other agents. Agent initiates the communication with a requesting negotiation. In the opposite situation, when other agent has initiated the communica-tion, the agent uses a responding negotiation to respond. There can be more than one instances of the same negotiation executed at the same time. Both types of negotia-tions 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 dif-ferent 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 (`FINAL`) is reached.

3 A PDA implemented in AJA

Personal digital assistant (PDA) agent is a computer program aimed at performing simple and tasks on behalf of its user: handle e-mail of its user or it may monitor or find interesting newsgroups or web sites on the Internet and filter 'interesting' information [12]. A PDA may maintain the appointment schedule of its user and independently make or cancel appointments [12], [13]. A PDA communicates with other PDAs and performs some tasks that would otherwise have to be performed by the user. The implemented MAS using AJA tool 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 (where other colleagues participate),
- to enable students to register themselves online for the consultations.

The implemented MAS is fully scalable, hence the number of agents in the system is irrelevant for the system performance. For implementation of the PDA new concepts introduced in AJA (dependable values and adaptable parameters.) are used.

3.1 Beliefs

The AJA code defining PDA agent beliefs is given in Listing 7.

timeTable - one instance of the class **TimeTable** stores and maintains all engage-ments, colleagues-data and available times of the agent owner. **eventAlertTime** - is an adaptable parameter and represents how many minutes before the next event should be the lecturer reminded. The value of this belief determines the alert time for engagements in the timetable. **eventAlertTimeToBackup** - belief has a boolean value. It is used in the activation condition of the reflex that stores the belief **eventAlertTime** into a file. **engToAlert** - belief stores the first engagement to which the agent owner should be alerted. **birthdaysTomorrowToAlert** and **birthdaysTodayToAlert** - These two beliefs store the persons having birthdays tomorrow and today respectively. **consultationDuration** - this belief is a dependant value. It is used for the estimation of the expected duration of consultations with students. It was empiri-cally found out, that the duration of a consultation depends on the number of ap-pointed 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.

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. 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', meaning that if only one

student comes to the consultations, 24 days before the exams, the consultation duration is 8 minutes.

Listing 4.

```
BELIEFS
   //schedule of the lecturer's engagements
  timeTable :  TimeTable;
   //how many minutes before the next event
   //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
```

3.2 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: actions manipulating the timetable; actions alerting the user; actions performing backup of important beliefs; actions implementing GUI.

Timetable Manipulation - The actions that modify the **timeTable** belief are rela-tively simple ones. For example the action **removeOldEngagementsAct** removes old entries from the timetable.

Alerting the User - There are two actions that alert the user to the incoming events. The first one is the action **eventAlertAct**, and the second alerting action is the action **birthdayAlertAct**. Backup - Theoretically, an AJA agent runs all the time. However, in praxis this is not the case, so, the important agent be-liefs 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.

GUI - At last, but not at least, there are several actions implementing GUI communi-cation with the user. The action that implements the main menu is the action **doGUIAct**.

3.3 Reflexes

A PDA agent in MAS has six reflexes divided into three groups: a) reflexes invoking the actions that alert the user about incoming events; b) reflexes invoking actions per-forming backup of important beliefs; c) reflexes maintaining the timetable.

4 Related work

AJA presents an original approach to integrating artificial intelligence technologies with a programming language. Our research was influenced by different sources during several last years.

In AJA there are two programming levels. This concept of two languages is adopted from HOMAGE [14]. In HOMAGE the lower level is object oriented and it is used for the definition of objects in languages C++, Common Lisp and/or Java. Ob-jects defined at lower level are used in a higher, agent-oriented level. The higher pro-gramming level in AJA consists of the programming language HADL, which is used for the description of the main agent parts. The lower level Java+ consists of the pro-gramming language Java extended with the constructs for the accessing agent parts defined in HADL, like in HOMAGE. 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).

JACK [10], [11] 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.

The experience gained in the design and implementation of the Java package for agent programming LASSMachine [1] was very helpful in the creation of AJA.

A negotiation construct in AJA corresponds to a conversation in COOL [4]. AJA ne-gotiation 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.

Subsumption architecture [6], [7], [8] gave surprisingly good results in robotics. It consists of hierarchically organized behaviors, which are the only components con-trolling the robot. AJA reflexes are similar to behaviors, but there are some important differences: a) AJA reflexes are triggered by boolean conditions; b) AJA agents have beliefs, which store their world models; c) AJA reflexes and AJA negotiations jointly control AJA agents.

In papers [15] and [16] intelligent agents were seen as a vehicle for AI-related tech-nologies in the mainstream programming. It also was the inspiration for including AI components into the AJA tool: dependant values (i.e. neural networks), and adaptable parameters.

5 Conclusions

The paper presented a tool for programming of multi-agent systems named AJA. AJA consists of two programming languages: HADL and Java+. Higher-level components of an agent are specified using HADL. The low-level implementation of HADL com-ponents is done using Java+.

One of the most interesting and original features of AJA is the integration of artificial intelligence techniques, such as artificial neural nets, with a programming language. Other advantages of AJA include: grouping of individual communicative acts into negotiations, proactive and/or reactive nature of AJA agents, secure inter-agent communication using SSL and digital signatures, accessibility via Internet, unrestricted use of Java platform in AJA programs, etc.

References

1. M. Badjonski, "Implementation of Multi-Agent Systems using Java", Master thesis, Institute of Mathematics, Faculty of Science, University of Novi Sad, Yugoslavia, 1998.
2. M. Badjonski, M. Ivanović "LASS - A language for Agent-Oriented Software Specification", Proceedings of VIII Conference on Logic and Computer Science LIRA, Novi Sad, Septem-ber, 1997, pp. 9-18.
3. 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.
4. M. Badjonski, M. Ivanović, Z., Budimac, "Agent Oriented Programming Language LASS", Computer Science and Electronic Eng., Horwood Publishing Ltd., 1999.
6. R. A. Brooks, "A Robust Layered Control System for a Mobile Robot", IEEE Journal of Ro-botics and Automation, 2(1):14-23, 1986.
7. R. A. Brooks, "Intelligence without Reason", Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, Australia, 1991, pp 569-595.
8. R. A. Brooks, "Intelligence without Representation", Artificial Intelligence, 47:139-159, 1991.
10. P. Busetta, R. Rönnquist, A. Hodgson, A. Lucas, "Jack intelligent agents - Components for Intelligent Agents in Java", AgentLink News Letter, January 1999, pp. 2-5.
11. http://www.agent-software.com.au/
12. P. Maes, "Agent that Reduce Work and Information Overload", Communications of the ACM, 37(7):31-40, July 1994.
13. 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.
14. 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.
15. S. Schoepke, "Facilitating the Deployment of Intelligent Agents in the Application Development Mainstream", AgentLink News Letter, July 1999, pp. 10-12.
16. S. Schoepke, "Intelligent Agents will be a Vehicle for other AI-related Technologies", Posi-tion Paper, International Workshop on Agent-Oriented Information Systems (AOIS'99), 1999, available at http://www.aois.org/99/schoepke.html .