NOVI SAD J. MATH.
VOL. 28, NO. 2, 1998, 187-205

AN APPROACH TO AGENT IMPLEMENTATION USING JAVA

Mihal Badjonski, Mirjana Ivanović, Zoran Budimac
Institute of Mathematics, University of Novi Sad
Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
email:{mihal, mira, zjb}@unsim.ns.ac.yu

Abstract

Autonomous agents represent a new field in computer science. Agent approach is suitable for building complex distributed software systems. The paper presents design of new agent-oriented programming language Lass and its implementation as a Java package aimed for agent programming. The package significantly reduces time needed for agent programming.

AMS Mathematics Subject Classification (1991): 68N20
Key words and phrases: software agents, Java, agent-oriented programming

1. Introduction

The concept of an agent has become nowadays important in artificial intelligence (AI). There are two general uses of the term 'agent': a weak notion of agency and a stronger notion of agency.

A weak notion of agency determines term agent as hardware or (more usually) software based computer system that enjoys the properties [11]:

- *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."

A stronger definition of an agent determines it as [11]: *"... 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.)."*

Multi-Agent systems (MAS) are a new, but extremely appealing field in computer science. They are usually classified as a field of distributed artificial intelligence. MAS is a system compound of at least two agents.

Agent-oriented programming languages are programming languages developed for the programming of agents. Agent-oriented programming (AOP) can also be seen as a post- object-oriented paradigm. AOP introduces new concepts such as mental categories, reactivity, pro-activeness, concurrent execution inside and between agents, communication, meta-level reasoning, etc.

This paper presents an approach to agent implementation using concepts of a new agent-oriented programming language *Lass*. Agent programmed with *Lass* possesses *intentions*, *beliefs* and *plans* for its *public* and *private* services. Besides deliberative properties, agent specified with *Lass* can behave reactively as well. *Lass* introduces the usage of *reflexes* - programming primitives enabling agent to react immediately when it is necessary. *Lass* enables communication between agents which is based on agent public services. Services are used similarly like remote procedure calls. They can be used for the implementation of speech-acts. Sending of a speech-act per formative can be implemented as a request for a service execution. A similar method is used in [10].

Basic concepts of *Lass* are implemented as classes in programming language Java (appropriate package) suitable for agents implementation. Classes

are organized in the package **LassMachine**. They are general enough to be used for implementation both weak and strong agents.

The paper is organized as follows. The architecture of agents programmed using presented package is given in the next section. The first part of third section is aimed to briefly introduce the language *Lass* and its main properties. The Java package **LASSMachine** is described in the rest of the third section. Possible applications of the package as well as some directions for further work are given in the fourth section. Related work and a conclusion are given in the last section.

2. The Main Parts of Agent

Since there are many notions of agents, it is impossible to make universal programming tool that will be useful in the programming of every agent. The *Lass* language and Java classes, implemented to simulate *Lass* properties, are suitable for the implementation of agents satisfying following requirements:

- static agents (they do not migrate over the network),
- software agents (as opposed to hardware agents) [1],
- persistent agents (this feature is common to most agents).

As it can be seen in the Fig. 1, an agent programmed using developed package possesses its beliefs about facts (which can be of various types,) intentions, public and private services, a special service named **webService**, plans, reflexes, and meta capabilities.

Agent is able to execute its services. The service instances that are currently executing are called intentions. Service execution can be invoked in three ways:

- an intention invokes the service execution,
- an active reflex may also invoke the execution of some service (see below for explanation of reflexes,)
- another agent in the system may ask the agent for the particular public service execution and this message can (but does not have to) invoke the execution of the required service.

[1] Hardware agents could also be programmed using this Java package provided that Java language is extended with constructs corresponding to robot actions.

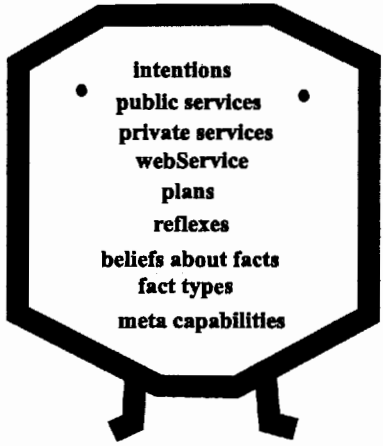

Figure 1: Parts of an agent.

`webService` is the special service that enables its agent to be accessible through the World Wide Web. Every agent has an Internet address. One can use an Internet browser to communicate with agent and to ask execution of offered tasks.

To every service exactly one plan is attached. Service execution corresponds to execution of its plan. Plan consists of actions to be performed.

Agent has its internal state consisting of its beliefs about some facts and of meta-level information about its intentions and active reflexes.

Reflexes are similar to services. While services represent proactive, goal-directed constructs, reflexes represent reactive ones. Reflexes have their priority and conditions for activation. Whenever there is at least one active reflex, the active reflex(es) with highest priority are executing.

3. LASS Language for Agent-Oriented Software Systems

3.1 LASS Language

Every program written in *Lass* is intended for the specification of exactly one agent. The main part of the *Lass* syntax and the description of the syntax categories are as follows.

```
program =
```

```
'AGENT' agent_name ';'
[known_agents_decl ';']
[fact_types_def ';']
[facts_decl ';']
[public_services_decl ';']
[private_services_decl ';']
[reflexes_decl ';']
[init_beliefs ';']
[init_intentions ';']
'END' agent_name
```

The agents that will communicate with the agent are specified in the first part. The agent can ask services from each of these agents and it can be asked for service by each one of them. The facts important to agent and their types have to be declared. Agent can perform public services and its private services as well. Agent can possess reflexes. They monitor the agent beliefs and use them to activate or deactivate themselves.

At the beginning of its existence, agent can have initial beliefs about the facts in its environment and it can also have initial intentions.

3.1.1. Other agents

- are specified (declared) using the following structure:

```
known_agents_decl =
  'KNOWN' 'AGENT' agent_decl {';' agent_decl}
agent_decl =
  agent_name ':' (internet_adr | 'LOCAL')
```

Example:

```
KNOWN AGENT
  Tom : alfa.bcd.com:2009 ;
  James : LOCAL;
  ...
```

The agent cooperates or competes with other agents in multi-agent system. Data about other agents have to be enlisted in the first part of the agent program. An agent in the system can be located on the remote machine, or it can be executed on the same machine where the agent is executing. In the former case the Internet address is used in the declaration of the remote agent (host name and port number), while in the latter case the keyword 'LOCAL' is used.

3.1.2. Facts

- have their types defined as follows:

```
fact_types_def = 'FACT' 'TYPE' ftype_def {';' ftype_def}
ftype_def = ftype_name '=' ftype
ftype = prim_type | record_type | array_type
```

Example:

```
FACT TYPE
  yearType = INTEGER;
  string = ARRAY [0..30] OF CHAR;
  person = RECORD
             firstName, lastName : string;
             yearOfBird : yearType;
             savings : real
           END;
  ...
```

Fact can be of a primitive type (standard type), a record, or an array. Fact types are used in facts declaration:

```
facts_decl = 'FACTS' fdecl {';' fdecl}
fdecl = fact_names ':' ftype_name
fact_names = fname {',' fname}
```

Example:

```
FACTS
  Bil, Lisa : person;
  isWindowOpened : BOOLEAN;
  ...
```

Facts are used as variables in traditional languages, and represent agent beliefs. Besides user-defined facts, meta-level facts `CURRINT` and `CURRREF` are also available. They contain information about current intentions and currently executing reflexes.

3.1.3. Public services

- are declared as follows:

```
public_services_decl =
  'PUBLIC' 'SERVICES' serv_decl {';' serv_decl}
  serv_decl = serv_name '(' [ par_decl {';' par_decl} ] ')'';'
  [incompatible_services]
  ('ALWAYS' | 'WHEN' bool_expr ';')
  [local_facts_decl]
  'PLAN'
  body
  'END' serv_name
```

Example:

```
PUBLIC SERVICES

  reserveTicket(request : reservationRequestType;
                VAR done : BOOLEAN);
  INCOMPATIBLE WITH reserveTicket, modifyTimeTable;
  ALWAYS;
  LOCAL FACTS
    r : reservationType;
  PLAN
    ...
    // making a reservation if possible
    ...
  END reserveTicket;

  cancelReservation(reservInfo : reservationInfoType);
  ALWAYS;
  PLAN
    ...
    // canceling the reservation
    ...
  END cancelReservation;

  ...
```

Service can contain parameters of input or input-output type (`VAR` parameters). `par_decl` represents the declaration of parameter(s). It consists of parameters' names optionally preceded with the word `VAR` and followed by the type of parameter(s). Service will not be performed if `bool_exp` in `WHEN` condition is not satisfied. Service execution is postponed until there are no incompatible intentions. Every service has its plan for its execution.

```
bool_expr =
```

```
  'TRUE' | 'FALSE' | '(' bool_expr ')' | test_service |
  term relation term | 'NOT' bool_expr |
  bool_expr ('AND' | 'OR') bool_expr
```

test_service is a special type of private service that returns logical value 'TRUE' or 'FALSE' after its execution.

```
body = action {';' action}
action =
  communicative_action |
  service_action |
  loop_action |
  cond_action |
  modify_fact_action |
  input_output_action
```

In addition to constructs from procedural programming languages *Lass* possesses special communicative primitives characteristic for the AOP languages.

```
communicative_action = ask_service_wait | ask_service
ask_service_wait =
  'SENDWAIT' serv_name '(' [params] ')'
  'TO' agent_name 'REPORT' 'IN' rep_fact_name
ask_service =
  'SEND' serv_name '(' [params] ')'
  'TO' agent_name 'STATUS' 'IN' stat_fact_name
```

Example:

```
productID := 'car439';
creditCard := 'Visa';
creditCardNo := '875648967853453';
deliveryAddress := '...';  //delivery address
SENDWAIT buyProduct( productId, creditCard, creditCardNo,
                     deliveryAddress, isOK )
  TO car_seller REPORT in servResult;
IF (servResult = DONE) AND isOK THEN
  ...
```

Communication is used when an agent requires service execution from another agent. Agent that asks the service may stop the execution of the action sequence while remote service is being performed or it may continue

to perform its actions. Agent can have several intentions and/or reflexes active simultaneously. If it uses remote service with wait, only one plan/reflex will be paused (the one that asks for remote service), while other will continue to execute. Report and status of the service can have the following values: `'DENIED'`, `'DONE'`, `'PRIVATE'`, `'BAD_TYPES'`, `'UNKNOWN_SERVICE'`, `'UNREACHABLE_AGENT'`, `'UNKNOWN_AGENT'`, `'BAD_RESPOND'`. Status can have two additional values as well: `'EXECUTING'`, and `'NOT_STARTED_YET'`.

```
service_action = service_wait | service
```

Agent can execute its services, both public and private, in two ways. The plan or reflex that invoked the service may continue to execute simultaneously with the new service or it may wait until the service finish its execution. Like with remote services, service is accompanied with report (`service_wait`) or status information (`service`).

3.1.4. Private services

- are declared similarly as public services:

```
private_services_decl =
  'PRIVATE' 'SERVICES' pri_serv_decl {';' pri_serv_decl}
pri_serv_decl = test_serv_decl | serv_decl
```

Unlike the public services, private services are visible only to the agent that executes them. Another difference between private and public services is that private service can return a `BOOLEAN` value (like function in Pascal).

3.1.5. Reflexes

- are declared as follows:

```
reflexes_decl = 'REFLEXES' ref_decl {';' ref_decl}
ref_decl =
  ref_name ';' [ 'PRIORITY' integer ';' ]
  'ACTIVATE' 'WHEN' bool_expr
  [local_facts_decl]
  'BEGIN' body 'END' ref_name
```

Example:

```
REFLEXES
  correctPrices; PRIORITY 1;
```

```
    ACTIVATE WHEN currencyRateChanged
    LOCAL FACTS
      i : INTEGER;
    BEGIN
      FOR i := 1 TO itemNum DO
        item[i].price := item[i].index * rate
      END;
      currencyRateChanged := FALSE;
    END correctPrices;
  ...
```

Reflex activation depends on the truth-value of `bool_expr` defined in `'ACTIVATE' 'WHEN'` part of the declaration. When `bool_expr` is true, the reflex will be activated. There can be several reflexes active at the same time. However, only the active reflexes with the highest priority (lowest `INTEGER` number) are executed, while other active reflexes are paused. By default, reflex has the highest priority.

3.1.6. Initial beliefs and intentions

- are specified in the following structure:

```
init_beliefs = 'BELIEFS' 'INITIALIZATION' body
```

Example:

```
BELIEFS INITIALIZATION
  myStreet := 'Trg Dositeja Obradovica'
  numOfMovies := 100;
  movie[1].name := 'From Dusk till Dawn';
  ...
```

`body` should be used for the assignment of values to various facts. It is executed only once, at the beginning of agent life. Facts that are not initialized have an unspecified value.

```
init_intentions =
  'INITIAL''INTENTIONS' intention {';' intention}
intention = serv_name '(' [params] ')'
params = par {',' par}
```

Example:

```
INITIAL INTENTIONS
  EXECWAIT checkStockMarket();
  EXEC interfaceService();
```

Initial intentions are specified as a list of services that have to be performed at the beginning of agent existence.

3.2 LASSMachine - Java Package

Considering given objective to implement *LASS* language, a set of different Java classes has been developed and organized as the package `LASSMachine`. The most important classes of the package are given in Fig. 2. Some of them closely relate to the main parts of agent showed in Fig. 1.

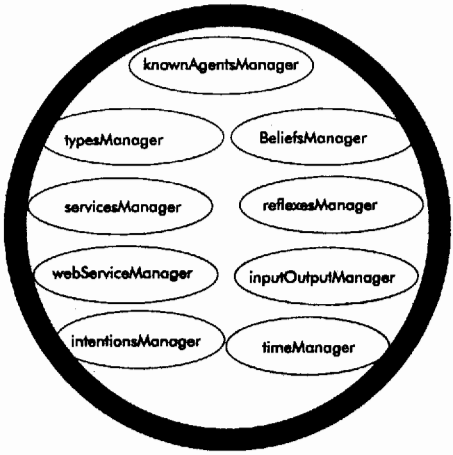

Figure 2: Some classes for *Lass* implementation

1. **agentRMI** - Java 1.1 API introduced remote method invocation, that provide the means for high-level communication between remote applications. This powerful tool was used for the communication between agents. Interface `agentRMI` declares the methods that can be invoked remotely.

   Messages between agents are requests for service executions. When agent **A** wants agent **B** to perform particular service on some data then agent **A** remotely invokes **B**'s `receiveSerivceCall` with appropriate input data. After receiving a request for service execution, agent **B**

asks for confirmation of the request, executing **A**'s `confirm` method. This step is necessary due to security of the system. After the **A**'s confirmation and **B**'s execution of the service (or its rejection), agent **B** remotely invokes **A**'s `receiveAnswer` method with the result of the service execution (or the explanation for the rejection of execution).

2. **agent** - The class that implements interface `agentRMI` is `agent` class. Besides three methods declared in the interface this class has many auxiliary methods. Most of them simply pass their arguments to the appropriate methods of appropriate manager object. `agent` has a constructor which creates instances for all managers and initializes static methods and variables in the class `typesManager`.

   Methods `receiveSeriviceCall`, `confirm` and `receiveAnswer` invoke the methods with the same names in `serMan` (which is an instance of `serviceManager`).

3. **typeClass** - Every fact in agent beliefs has its type. Direct usage of Java types for facts would cause a limitation in agent communication, because remote method can only accept actual parameter of the same type as is the type of the corresponding formal parameter. This means that both agents involved in communication would have to have exactly the same definition of the parameter type. This drawback is overcome introducing a special class, `typeClass`, whose objects describe types. Actual parameter for remote method execution is compound of fact and its type (`typeClass` objects).

   There are eight standard types available: `integer`, `real`, `boolean`, `char`, `string`, `input-file`, `time`, and `output-file`. Beside them, `array` and `record` types can be defined.

   This class has three constructors. Depending on what kind of type the object represents (standard, `array`, or `record`), appropriate constructor will be used. Every type has its `name` and its `code`.

4. **typesManager** - `typesManager` class contains information of types which agent knows. This class has three methods. Method `init` initialize a hash table and puts into the table `typeClass` instances representing eight standard types. Two additional methods add and get types into/from the hash table. All the variables and methods of this class are static. No instance of this class should ever be created.

5. **fact** - Belief about some fact of an agent is represented as an object of the class `fact`. Every fact has its `type` and a value, which depends on a `type`.

   This class has several constructors. Which one will be used depends on `type` of the fact. Due to types there are also many methods for setting a new value and getting the current value.

   Method `clone` returns a new instance of the same `fact`. Method `write` stores the fact into given file, while static method `read` reads the `fact` from a file and returns it. If the `fact` is an array of character, method `toString` can be used for obtaining the string representation of that array. Method `fromString` does the opposite. It copies the characters from a given string into the `fact`, which is an array of characters.

6. **beliefsManager** - Instances of `beliefsManager` class maintain beliefs about facts. Beliefs can be global or local in services and reflexes. Instance of this class that maintains global beliefs has its variable `previous` set to `null`. Instances for local beliefs have as `previous` `beliefsManager` set the reference to the instance for global beliefs. This ensures a correct scope of names in agent. Local name can be the same as a global one and then only the local name is visible. In case that the name is not found in the local instance of `beliefsManager`, it will be searched in the `previous` instance (global names). Every instance has its hash table where the facts are placed. The key for putting fact in the hash table is the `name` of the fact.

7. **inputOutputManager** - Agent communication with its user is implemented through `inputOutputManager` class. Agent has its window where communication with its user occurs. Various types of input-output actions can be achieved using different methods.

8. **knownAgentsManager** - References to remote agents are maintained by the instance of `knownAgentsManager` class. They are placed in the hash table using their URLs as keys. Every agent at the beginning of its existence introduces all other agents in the system it will communicate with. It does that using `introduce` method of a `knownAgentsManager` instance. This introduction may fail due to later initialization of some remote agent. When existing agent tries to obtain the reference to an agent which does not exists, it will not succeed. In such case, the agent will postpone the introduction of the remote agent until the ref-

erence is really needed. When the agent wants to ask service execution from remote agent, it will use get method of its `knownAgentsManager` to obtain the reference to remote agent.

9. **servicesManager** - Every agent has exactly one **servicesManager** instance. This class has two hash tables as its variables. One for agent services definitions and one for data about current execution of services invoked by this agent (both local services and remote services). When agent wants some service to be performed it invokes `startService` method of its `servicesManager`. The data about service execution are hold in the `currentCalls` hash table. Method `startService` uses remote method invocation to invoke `receiveServiceCall` method of the agent that should execute the service. The executing agent can be some other agent (remote) or the same agent that asked the service. The `receiveServiceCall` of executing agent will invoke the method in its `servicesManager` instance with the same name - `receiveServiceCall`. After performing some security steps, this method will find the service definition in the `myServices` hash table and start its execution with received parameters in separate thread. After the execution of the service, `VAR` parameters will be returned to the first agent, using its `receiveAnswer` method. This method will invoke the method with the same name in the local `servicesManager` instance.

10. **timeManager** - It enables agent to be 'aware' of time. The class has two useful methods. The first one obtains current time and date (as a fact of type `time`). The second one is aimed for delayed service execution. Using this method, a service will execute at specify time and date, instead of immediately.

11. **intentionsManager** - All service instances that are currently executing in an agent are called the intentions of the agent. Data about intentions are hold in the hash table `intentions`. For every service that is executing the hash table contains its name and the number of its instances that are executing. Agent can use this meta information in its services and reflexes. Another important role of `intentionsManager` is the synchronization of services. Some services might be unable to execute while some other services are executing. In that case agent has to defer the start of new service until the incompatible services finish their execution. This waiting is performed in the `addIntention` method of this class.

12. **reflexesManager** - Every agent has one instance of the `reflexesManager` class. Reflex is a sequence of actions that is performed when its conditions for activation are satisfied and there is no active reflex with higher priority at the moment.

    The `activate` method of the instance of this class is executing (endless loop) in a separate thread. Every iteration of the main loop of this method corresponds to the following steps: a) selects reflex(es) for execution; b) execute them in the separate threads; c) wait for the end of all threads.

    Variable `state` is used for meta-capabilities of agent. It enables agent to know which reflexes are currently executing. It also blocks these queries when the `activate` method is at the beginning or at the end of its iteration, i.e. while the change of current reflexes is occurring.

13. **webServiceManager** - `webServiceManager` enables agent to be accessible via the World Wide Web. Whenever one accesses the agent through an Internet browser, one instance of the `webService` is performed. This service is similar to other agent services. It only differs in its input-output actions, because the output action results in a HTML page. When there is a need for user input, an HTML form is generated and sent.

The `LASSMachine` package is implemented in the Sun's Java 1.1.6 for the Microsoft's Windows 95 operating system. However, since Java is a platform independent language, the package can be used on other operating systems as well. The package consists of forty-eight classes and three interfaces. Nevertheless, only six classes from the package should be used for agent programming:

- `agent`,
- `typeClass`,
- `fact`,
- `service`,
- `webService`, and
- `reflex`

The remaining classes and interfaces are used for the creation of the above classes.

Classes `service`, `webService`, and `reflex` are abstract classes. Every agent service is implemented as a class that extends `service` class. Similarly, agent web-service is implemented as a class that extends `webService` class, and every agent reflex is implemented as a class that extends `reflex` class.

The package `LASSMachine` implements an agent architecture. When the six above classes from the package are used, a programmer concentrates only on domain dependant parts of the agent application, because the skeleton of the agent is already implemented in `LASSMachine`. Usage of `LASSMachine` does not prevent programmer from using another Java classes from both standard and user-defined Java packages. These classes can be used for programming of agent services plans, agent web-service plan, and for the programming of actions that reflexes perform.

4. Possible Applications

The Java package `LASSMachine` is suitable for the development of many multi-agent systems. Availability of Java API for most of the operating systems makes this package even more beneficial.

Advantages of the package can be especially evident when it is used for the development of complex distributed software systems. These advantages stem from the agent- oriented principles that are embedded into this software tool. Agent approach is based on decentralization of the system. Agents are autonomous components with the tractable complexity, while overall system complexity can be unmanageable if a centralized system organization is used. The complexity of the multi-agent system is overcome using agent communication.

For example, as described in [3], a tool such is this package can be used for the development of global software system for travel ticket reservations. It would consist of agents located at travel facilities such are airports, seaports, train stations, and bus stations. Besides these agents, every place where people live would have its own agent. These later agents would be organized into hierarchy depending on the size of their place and their geographical position. They would be superior to the agents placed at local travel facilities. As described in [3], using simple parallel algorithm, the sub-optimal journey satisfying given preferences can be found for any traveling between two places. Ticket reservations for all parts of the journey can also

be easily achieved.

Another example is given in [4], where multi-agent system consisting of personal digital assistants for appointment schedule is described. Each agent (personal digital assistant) is devoted to one person in some organization. When agent user wants to make an appointment with someone who also has his/her agent, the negotiation is left to its agent(s). Agent beliefs includes information about its user available time (with various degrees of availability) and the individual importance of other people to the user.

The package `LASSMachine` can also be used for the programming of complex systems that are placed on single computer. For example an expert system can be modeled and specified as a multi-agent system [1]. After the identification of the main components (agents) of an expert system, the package can be used for the programming of individual agents.

5. Related Work and Conclusion

Many agent-oriented programming languages have influenced the creation of the package `LASSMachine`. However, we believe that described package and corresponding high-level language (*Lass*) possess a unique combination of agent features.

The agent-oriented language [9] AGENT0 has introduced agent-oriented concepts in programming and therefore it influences this package as well as it influences any other software tool used for programming of agents based on stronger notion of agency.

Some ideas used in the language AgentSpeak [10] has been used in the development of the package `LASSMachine`. However there are also some differences in concepts used. For example in AgentSpeak there can be many plans for service execution, while there is only one plan for a service in our approach. AgentSpeak allows non-deterministic actions to be performed, while this non-determinism is not present in `LASSMachine` agents. We believe that this modifications increases the readability and the reliability of agent program.

None of the other known software agent tools possesses constructs for agent reactivity such are reflexes. The use of reflexes is improved by behavioral approach to artificial intelligence which is developed at MIT. Its creator, R. Brooks [5], [6], [7] has developed many simple robots that are able to perform complex tasks. Brooks proposes Subsumption Architecture for the organization of reflexes. Reflexes in `LASSMachine` agent are organized

in the similar manner.

The combination of deliberative agent architecture and reflexes is proposed in [8], for the programming of animated agents in computer animations.

This paper presents a Java package which is aimed for the implementation of multi- agent systems. This software tool implements some ideas from agent theory and thus enables high level agent programming instead of building agents from scratch.

The classes in the package are general enough to be used in various agent applications, but they are still enough effective to release the programmer from many hours of low-level programming.

References

[1] Badjonski, M., Ivanović, M., 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, 401-404.

[2] Badjonski, M., Ivanović, M., Budimac, Z., Agent-oriented programming language Lass, Proceedings of COTSR, United Kingdom , 1998.

[3] Badjonski, M., LASS in action, Proceedings of PRIM'97, Palić, Yugoslavia, September, 1997, 1-10.

[4] Badjonski, M., Ivanović, M., Budimac, Z., Software specification using LASS, Proceedings of Asian'97, Lecture Notes in Computer Science Vol-1345, Springer-Verlag, Kathmandu, Nepal, December, 1997, 375-376.

[5] Brooks, R.A., A robust layered control system for a mobile robot", IEEE Journal of Robotics and Automation, 2(1) (1986), 14-23.

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

[7] Brooks, R.A., Intelligence without representation, Artificial Intelligence, 47 (1991), 139-159.

[8] Costa, M., Feijo, B., Agents with emotions in behavioral animation, Computers & Graphics, Vol. 20, No 3, 1996, 377-384.

[9] Shoham, Y., Agent-oriented programming, Artificial Intelligence, 60(1):51-92, 1993.

[10] Weerasooriga, D., Rao, A., Ramamohanarao, K., Design of a concurrent agent-oriented language, Intelligent Agents, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, 386-401.

[11] Wooldridge, M., Jennings, N.R., Agent theories, architectures, and languages: A survey, Intelligent Agents, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, 1-39.

Received by the editors December 16, 1998.