Scientific Journal of Riga Technical University
Computer Science. Applied Computer Systems
2010
Volume 43

Multi-Agent Based Intelligent Tutoring System Source Code Generation Using MASITS Tool

Egons Lavendelis, *Riga Technical University*, Janis Grundspenkis, *Riga Technical University*

***Abstract* - Several agent development tools have been proposed. At the same time, specific tools for agent based Intelligent Tutoring System (ITS) development do not exist. However, ITSs have some specific characteristics that must be taken into consideration during the development. General purpose Agent Oriented Software Engineering (AOSE) methodologies and therefore development tools do not sufficiently correspond to the characteristics of ITSs. Additionally, the general purpose AOSE methodologies and tools do not allow plugging in domain specific rules and diagrams. Thus, usage of knowledge gained in ITS research during the development process is limited. In the absence of general tools that allow plugging in knowledge from ITS research, a specific tool named MASITS has been developed. The tool supports a specific agent based ITS development methodology named MASITS which takes into consideration specific characteristics of ITSs and integrates knowledge from ITS research. The tool supports all phases of ITS development, starting from requirements analysis and ending with deployment. Requirements analysis and design phases are supported by appropriate diagram creation tools. Implementation is supported by source code generation from diagrams created during the design phase. JADE platform is used for agent implementation. Thus, Java classes for ontology, agents and behaviours are generated from the design diagrams. The paper includes a brief overview of diagrams used in the source code generation and detailed algorithms for source code generation from the diagrams.**

Keywords: **agent oriented software engineering, code generation, intelligent tutoring systems, multi-agent systems**

I. Introduction

Extensive research in the agent oriented software engineering (AOSE) is ongoing. A number of tools for agent oriented software development exist, for example: agentTool supporting MaSE methodology [1], Prometheus Design Tool (PDT) supporting Prometheus methodology [2], Ingenias Development Kit for INGENIAS methodology [3] and Goal Net Designer supporting Goal Net Methodology [4]. Additionally, some tools for specific development phases or tasks exist, for example, Conversation Design Tool CDT [5]. Agent implementation platforms like JADE [6], JACK [7], MADE [8] exist at their own.

Majority of agent oriented software development tools are developed to support certain general purpose AOSE methodology and can be successfully used only together with the corresponding methodology. Additionally, tools that support code generation are linked to a single agent implementation platform, whose code it is capable to generate, for example, PDT generates JACK code. So, the agent development tools are linked to specific methodologies and agent development platforms.

There are no general purpose AOSE methodologies that allow plugging in the knowledge about the type of the system under development. So, the development process can not be adapted to take advantage of knowledge gained in research of different types of agent oriented software. The development process can not be modified to fit the characteristics of some specific systems, too. As a consequence, agent development tools that support general purpose methodologies are less effective than specific tools for certain type of systems.

Moreover, many tools fail to support all phases of software development life cycle. Thus, these tools can not be used as the only tool to develop the system and some additional tools must be used making the development process more complicated.

During the last decade intelligent agents are widely used to implement Intelligent Tutoring Systems (ITS) [9]. Main characteristics of agents, such as reactivity, proactivity, cooperation, reasoning and planning capabilities match with the needs of ITSs. Additionally, usage of agents in ITS development has the following advantages: agents increase modularity of the system and facilitate implementation of intelligent mechanisms needed in ITSs. At the same time, specific tools for multi-agent based ITS development do not exist, despite the fact that ITSs have very active ongoing research and have some specific characteristics that must be taken into consideration. These characteristics are the following. Firstly, ITSs are hardly integrated into organisation and their structure does not correspond to the structure of the organization. Additionally, they have very few types of actors, namely, teachers, students and probably administrators. So, organisational and actor modelling can hardly be used and requirements come only from the goals of the system and its needed functionality. Secondly, ITSs are very complicated systems with a well established architecture [9] – [11], which should be taken into consideration during the design phase.

In the absence of general purpose methodologies and corresponding tools allowing to plug in domain specific knowledge and adjust the development process to fit the characteristics of specific systems, a specific agent based ITS development methodology and tool have been proposed. This paper contains description of a MASITS (Multi-Agent System based Intelligent Tutoring System) tool for ITS development. The main attention in this paper is paid to the code generation algorithms from diagrams created during the design. The remainder of the paper is organised in the following way. Section II contains a brief overview of the MASITS

methodology and tool. Section III includes the algorithm for diagram and code generation. Section IV gives conclusions and main directions of future work.

II. The MASITS Methodology and Tool

The MASITS methodology is created for agent based ITS development [12]. The methodology contains knowledge from two research fields, namely the AOSE and ITS. It is built by reusing the most appropriate techniques from the general purpose AOSE methodologies and adding knowledge from ITS research. The main characteristics of ITSs are taken into consideration during the choice of techniques used in the methodology. Appropriate requirements analysis techniques have been chosen. ITSs are hardly integrated into any organization and there are very few types of actors involved. Thus, requirements analysis can not be carried out using organisational or role analysis. Instead, it has to be done using techniques that concentrate on system's goals and functionality. Similarly, during the design phase the results of agent based ITS research are taken into consideration. Firstly, Grundspenkis and Anohina [9] have concluded that ITSs consist of four modules and these modules usually are implemented using similar agents. Therefore, a set of agents that build up an ITS is predefined. It can be only modified to meet specific needs of each particular system, and there is no need for any agent definition techniques. Usage of such approach in the MASITS methodology facilitates ITS development by replacing agent definition, which is one of the most challenging tasks of general purpose AOSE methodologies, with simple task allocation to the set of agents. Secondly, interactions among agents in ITS do not include complicated protocols. Therefore, interaction design in the MASITS methodology is done by specifying messages sent among agents. Thirdly, ITS development using holonic agent architecture is proposed in [11]. The holonic architecture has the following advantages: it increases the modularity and adds openness to ITSs. As a consequence it facilitates the development and change implementation into ITSs. Thus, a support for holons is included in the MASITS methodology.

The MASITS tool is intended for multi-agent based ITS development with the MASITS methodology. The tool supports full ITS development life cycle from requirements analysis to implementation.

Interface of the MASITS tool consists of the following main parts, as it is shown in Fig. 1. Menubar (1) contains all functions of the tool. The most frequently used features are included in the main toolbar, too (2). All diagrams developed during the development of the system are included in tabs (3). Each tab of the diagram contains a drawing toolbar (4) and a page for diagram drawing (5).

The development process with the MASITS methodology starts with the requirements analysis. It is the first phase supported by the MASITS tool, too. It is carried out using two well known techniques, namely, goal modelling and use case modelling. The goal modelling is done first, because goals are used in use case modelling. The second phase of ITS development is design. The MASITS methodology divides

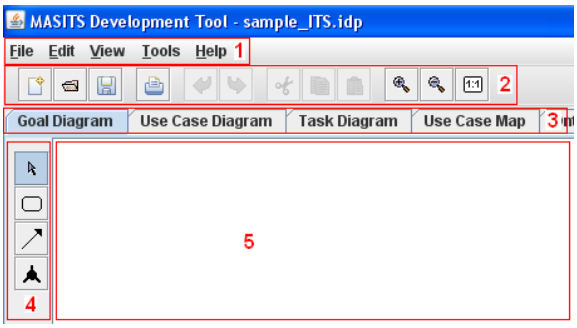

Fig. 1. Main parts of the MASITS tool's interface

design into two stages, namely, external and internal design of agents. During the external design agents functionality and interactions among agents is specified. Task diagram and interaction diagram are created during the external design. During the internal design the internal structure of agents is specified, i.e., it is defined how agents can achieve their functionality specified during the external design. Internal design of each agent is defined in its internal view. During the internal design a choice to implement agent as a simple agent or a holon is done. If an agent is implemented as a holon, design of the holon as a separate multi-agent system is performed in the corresponding interaction diagram.

Some design tools can be used to specify systems that are implemented on different platforms. Such an approach enables these tools to have wider usage. However, transformation from design concepts to agent implementation platform concepts is almost unique for each combination of design concept set and agent platform [13]. Such a transformation is extra effort during the development of the system if it is done by developers. Moreover, if the transformation is not built in the tool the code generation is not possible. Thus, we have chosen to use a set of implementation platform concepts already during the design phase. So, the MASITS tool can be used to implement multi-agent systems in JADE agent development framework [6]. JADE was chosen, because it provides simple way to create Java agents and organise their interactions. Moreover, Java classes of JADE agents can be easily generated from the design elements. Agent interactions in JADE are organised using predicates from the domain ontology. Agents are Java classes and their actions are defined as behaviours that are implemented as Java classes, too. Domain ontology is described in one ontology base class. Each concept and predicate included in the ontology is defined in corresponding Java class [14]. So, main implementation elements are ontology, agent and behaviour classes. Additionally to these elements a batch file to start the system has to be created. It includes deployment details of the system, which are specified in deployment diagram, and commands to start JADE containers. This file is generated automatically from the deployment diagram. So, the MASITS tool supports implementation and deployment phases, too. Generation of all these elements is described in Section III.

The ITS development using the MASITS methodology and tool consists of creation of a set of diagrams. All diagrams included in the MASITS tool and dependencies among them

are shown in Fig. 2. Sequential creation of diagrams is denoted with an arrowed link, crosscheck is denoted with a dashed line. Diagrams that are used during the code generation are denoted by gray rectangles and are described below in details. The following diagrams are included in the tool:

- *Goal diagram* is a system's goal hierarchy. It shows system's main goals and their subgoals that have to be achieved to reach the main goals. Decomposition is done until the lower level goals can be achieved by accomplishing simple actions (for details, see [15]).
- System level *use case diagram*. A well-known UML use case diagram [16] is used. Use cases are created to reach the lower level goals (for details, see [15]).
- *Task-agent diagram*. Task-agent diagram is a task hierarchy with agents specified for each task. Tasks are created according to the steps of use case scenarios (for details, see [17]).
- *Use case maps*. Use case maps show agents, tasks allocated to them and flow of control through these tasks. Use case maps are created for interactions that are too complicated to be designed directly in the interaction diagram. A use case map represents use case scenario in the form that interactions among agents are shown explicitly. After creating use case map only message content must be added to specify interactions (for details, see [17]).
- *Interaction diagram* depicts interactions among agents. Interactions are specified in the form of sent messages.
- *Ontology diagram* is a model of problem domain.
- *Agent's internal view*. Each agent's internal view is a design of agent's internal structure.
- *Holon hierarchy diagram* includes a hierarchy of all holons.

The interaction diagram consists of agents, user and links denoting messages sent among them. Predicates from the domain ontology are used to specify message contents. The main interaction diagram is created for the higher level agents. Additionally an interaction diagram is created for each holon. Interaction diagrams are used to include knowledge about other agents that an agent communicates with, into agent's beliefs. Fragment of the interaction diagram is shown in Fig. 3.

The MASITS methodology uses lightweight ontology [18] to specify messages sent among agents. The ontology is a formal hierarchy [19]. The ontology diagram is a model of problem domain that is depicted as a class hierarchy with two superclasses, namely, concept and predicate. Concepts are used as attribute types in other classes of the ontology. Predicates are used as message contents in agent interaction. Ontology diagram is used for ontology generation. An example of an ontology diagram is shown in Fig. 4.

The agent's internal view specifies design of agent's internal structure in terms of agent's actions, beliefs, and rules used to process received messages and other perceptions. It consists of the following elements:

- The agent diagram, specifying agent's perceptions, messages sent and received by the agent, agent's actions and links among these elements. All agents' actions except message receiving and sending are denoted by rectangles inside an agent. Perception (message and event) receiving and message sending actions are replaced with incoming and outgoing events and messages, as well as corresponding contacts (like in structural modelling [20]).
- Perception (message and event) processing IF-THEN rules that are built from a set of templates. IF pattern may contain the following templates: check, if particular predicate is received, compare, condition conjunction (AND), disjunction (OR), negation (NOT) and True (the condition that is always true). Templates Action, Belief Set, Action conjunction (AND) and IF-THEN rule can be used as THEN pattern.
- Start-up rules describing, what actions are done by agent during the start-up.

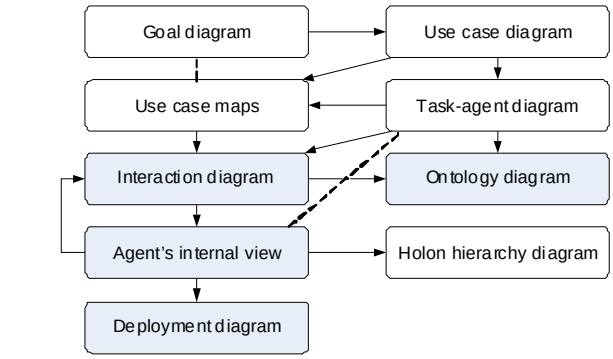

Fig. 2. Diagrams included in MASITS tool and dependencies among them

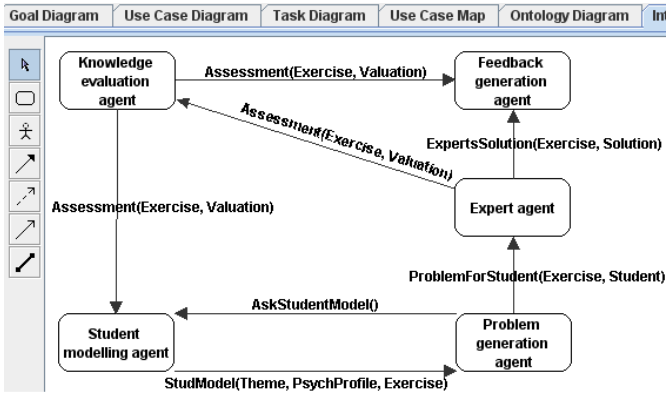

Fig. 3. Fragment of the interaction diagram

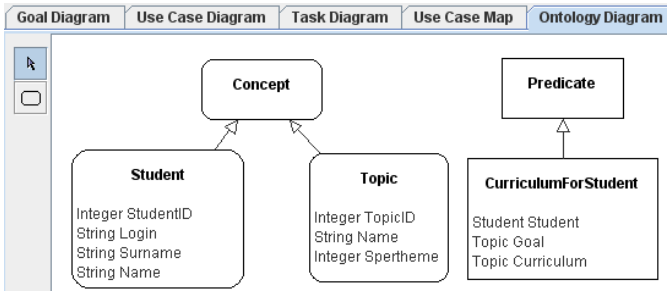

Fig. 4. Example of the ontology diagram

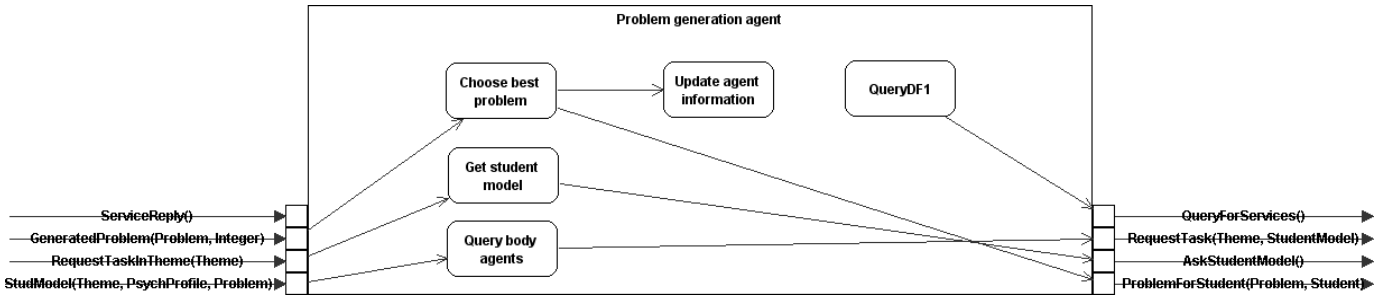

Fig. 5. Example of the agent diagram

- A list of agent's beliefs specified as pairs of belief's type and name: <Type>, <Name>.
- A list of agent's actions specifying implementation details. Action descriptions contain the following columns: name of action, name of corresponding behaviour class, type of action (one-shot, cyclic, timed, etc.), inner (implemented as inner class in agents class) or outer (implemented as a class in the same package as agent) behaviour class, and whether it is added when the agent is starting.

An example of agent diagram is shown in Fig. 5. Agents and their behaviours are generated from corresponding internal views of agents. For details of agent's internal view, see [17].

It is possible to generate agent and their behaviour classes from the previously described diagrams. However, deployment details are needed to generate agent instances. These details are specified in the deployment diagram. Deployment diagram consists of JADE containers used to deploy the system and agent instances that are deployed in these containers. Interactions among containers are specified, too. Labels on the links denote environment used for interactions. An example of deployment diagram is shown in Fig. 6. A batch file used to start the system is generated from the deployment diagram.

The advantage of the MASITS tool is the introduced concept of interdiagram links. An interdiagram link is a link among elements of two different diagrams. These links are introduced with two main purposes, namely, consistency check and crosscheck. Links created for consistency check are mainly used to link elements that have to be changed simultaneously. Such elements in the MASITS tool can be edited only in one diagram (source). Interdiagram links are created to all other elements (targets) that have to be changed if the source element is changed. When the source element is changed all target elements are affected. For example, name of an agent can be changed only in interaction diagram. If it is changed, the MASITS tool changes the name of this agent in agent diagram, holon hierarchy diagram, task-agent diagram and interaction diagrams of holons. Elements can be deleted in the same way as changed. However, there is one significant difference. If an element is deleted, major parts of other models can be affected. Two choices are available. A tool can either delete the affected parts or restrict deletion of the source elements until the dependant elements are not deleted. Both approaches are used in the MASITS tool. The MASITS tool just deletes dependant elements if the deleted element has the same semantics as the dependant elements. Deletion of the element is restricted if the dependant part is a significant part of other diagrams with different semantics.

The second purpose of interdiagram links is to link elements from different diagrams that support each other. This kind of interdiagram links are used to analyze if all elements of one diagram are supported by elements in another diagram. Such an approach is used in the MASITS tool four times. Firstly, links between use cases and goals are created. The links show, which goals have corresponding use cases defined. An unlinked goal indicates that corresponding use case has to be defined. Secondly, links between goals and tasks are defined to show which tasks support which goals. This kind of links is used during the crosscheck between goal and task diagrams. If after creating all links there is a goal that is not supported, then a set of tasks is insufficient and task diagram has to be refined. Thirdly, paths from use case maps are linked to messages in the main interaction diagram to ensure that every link from paths in use case maps has corresponding message in the main interaction diagram. Finally, each agent must have actions to realize all tasks assigned to it in the task diagram. Thus, interdiagram links are created among tasks and agents' actions.

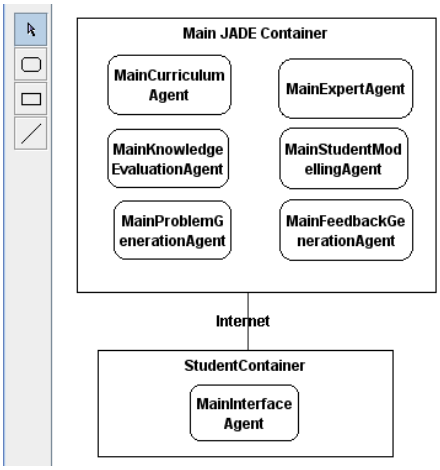

Fig. 6. Example of the deployment diagram

III. Diagram and Source Code Generation

One of the main advantages of the MASITS tool is diagram and code generation. Diagrams built up during the ITS

development process contain sufficient information to generate some parts of other diagrams automatically. For example, when a task-agent diagram is created, a set of higher level agents is specified. Thus, defined agents can be added to higher level interaction diagram and agent diagrams can be built for each agent. The MASITS tool supports automatic generation of all elements that can be created from previously developed diagrams. Generation of elements is done at the moment when user inserts an element that is used for generation. After element generation an interdiagram link between the source and target elements of generation is added. The interdiagram link is used to keep diagrams consistent in the way described in the previous section. Table 1 summarises generated elements.

Additionally, holon hierarchy diagram is generated from previously developed diagrams. This diagram does not contain any additional information, because superholon of any holon is specified at the moment of holon creation. The holon hierarchy diagram only explicitly shows holon hierarchy in the readable form.

The second advantage of the MASITS tool is the possibility to generate a source code of the system from the diagrams developed during the design. The following diagrams are used: ontology diagram, interaction diagrams, agent internal views, holon hierarchy diagram and deployment diagram. The code generation is simple diagram to code transformation, which is made possible by choosing to design the system using concepts of the JADE framework. During the code generation the source code for ontology classes, agent classes and behaviour classes is built up.

The code generation algorithm consists of the following steps:

1. Create the main package of the project. It is created in the user specified path with the name of the project.
2. Create package “Ontology” in the main package.
3. Create ontology base class in the package “Ontology”.
4. Create classes for each concept and predicate.
5. Create classes for all agents.
6. Create classes for external behaviours.
7. Create holons (including agents that build up the holon).
8. Generate the batch file to be used to start the system.

The first two steps is a simple folder creation in the path specified by the user. Other steps are described in the following subsections. Code generation is done using templates built in the tool. The following subsections consequently describe templates used to build up the code of each class.

A. Ontology Generation

Ontology generation is done during the steps 2-4. Ontology is formed in the package “Ontology”, thus corresponding folder is created at the second step.

During the third step the ontology base class containing descriptions of all schemas used in the ontology is generated. Ontology base class is used as metadata of all concept and predicate classes. Ontology base class is generated by consequently adding code templates according to diagrams and their elements.

The ontology base class begins with a template containing a package name, imports and ontology base class definition. The name of the ontology base class is defined by the user. Ontology base class extends class *jade.content.onto. Ontology:*

```
package <main package of the
    project>.ontology;
import jade.content.onto.*;
import jade.content.schema.*;
import jade.util.leap.HashMap;
import jade.content.lang.Codec;

public class <Ontology_base_class_name>
    extends jade.content.onto.Ontology {
```

Ontology’s name is defined in the second template. It is defined as a static string and a static instance is created for the ontology:

```
public static final String ONTOLOGY_NAME =
    "<ontology_name>”;
private static ReflectiveIntrospector
    introspect = new
    ReflectiveIntrospector();
private static Ontology theInstance = new
    <ontology_base_class_name>();
public static Ontology getInstance() {
  return theInstance;
  }
```

Vocabulary for concepts, predicates and their attributes (designed in ontology diagram) is created, i.e., a public static string is defined to denominate each element. Denomination for each predicate and concept is defined using the following template:

```
public static final String <NAME>="<Name>";
```

Denomination for each attribute is defined using the following template:

```
public static final String
    <CLASS>_<ATTRIBUTE>="Attribute";
```

Ontology constructor template consists of the call to superclass and element additions to the ontology schema:

```
private ItsOntology(){
  super(ONTOLOGY_NAME,
      BasicOntology.getInstance());
  try {
```

TABLE 1
SUMMARY OF ELEMENT GENERATION

No	Source		Target	
	Diagram	Element added	Diagram	Element generated
1.	Task diagram	Agent	Interaction diagram	Agent
2.	Task diagram	Agent	Agent diagram	Agent
3.	Interaction diagram	Link	Agent diagram	Incoming messages and events, outgoing messages, method calls and contacts

Concepts defined in the ontology diagram are added to the ontology schema using the following template:

```
ConceptSchema <Name>Schema = new
    ConceptSchema(<DENOMINATION>);
add(<Name>Schema, <Project
    name>.ontology.<Concept>.class);
```

Predicates are added in the same way (*PredicateSchema* is used instead of *ConceptSchema*). After adding all concepts and predicates from the ontology diagram, their attributes are added using the following template:

```
studentsSchema.add(<DENOMINATION>,
    <attribute's schema>,
    ObjectSchema.<CARDINALITY>);
```

Attribute's schema can be specified either by using schema variable (done if the attribute is concept), or by getting schema from BasicOntology in the following way *(TermSchema)getSchema(BasicOntology.<Type>)* (done if the type of the attribute is Java class or a primitive type). Cardinality can be OPTIONAL (meaning one or none), MANDATORY (meaning one and only one), or UNLIMITED (meaning an array).

Template for end of schema addition, constructor and ontology base class:

```
  }catch (java.lang.Exception e)
    {e.printStackTrace();}
  }
}
```

Concept and predicate classes for all concepts and predicates specified in the ontology diagram are generated after creating the ontology base class. Each concept or predicate class contains attribute definitions and corresponding *set* and *get* methods. Predicate and concept classes are generated by consequently adding templates corresponding to descriptions of all attributes.

Concept classes implement interface *Concept*. Predicate classes implement interface *Predicate*.

```
public class <Name> implements
    Concept/Predicate {
```

Attributes with single cardinality (OPTIONAL or MANDATORY) are defined using the following template:

```
//attribute <name>
private <type> <name>;
public void set<Name>(<type> value) {
  this.<name>=value;
}
public <type> get<Name>() {
  return this.<name>;
}
```

Attributes with cardinality MULTIPLE are specified as lists and list processing methods *add*, *remove*, *clearAll*, *getAll* are generated:

```
public void add<Name>(<Type> elem) {
  List oldList = this.<name>;
  <name>.add(elem);
}
public boolean remove<Name>(<Type> elem) {
  List oldList = this.<Name>;
  boolean result = <name>.remove(elem);
  return result;
}
public void clearAll<Name>() {
  List oldList = this.<name>;
  <name>.clear();
}
public Iterator getAll<Name>() {
  return <Name>.iterator();
}
public List get<Name>() {
  return <name>
}
```

B. Agent Generation

After finishing the ontology generation agent classes are generated (step 5). One JADE agent class is created for each agent internal view. Agent classes extend class *jade.core Agent*:

```
public class <Name> extends Agent {
```

Ontology and codec used in agent communication are private attributes of the ontology class:

```
private Codec slCodec = new SLCodec();
private Ontology onto =<Ontology base
    class>.getInstance();
```

Identifiers of agents that are needed in communication are added as private attributes, too:

```
private AID <ID>= new AID("<Agent name>",
    AID.ISLOCALNAME);
```

Agent's beliefs that are defined in the internal view are defined as private attributes:

```
private <type> <attribute's name>;
```

Setup method consists of the following templates:

```
public void setup () {
```

Codec and ontology registration:

```
getContentManager().registerLanguage(
    slCodec);
getContentManager().registerOntology(onto);
```

Addition of message receiving behaviour:

```
addBehavior(new <Name>ReceiveBehavior ());
```

Startup rules from agent's internal view:

```
    if (<condition>) <action>;
    else <action>;
}
```

Files of agent classes contain behaviour classes, too. Generation of these classes is described in the next subsection.

C. Behaviour Generation

Behaviour classes are defined for all agent actions. Additionally, a message receiving and sending behaviours are defined. Message receiving behaviour is defined as inner class of all agent classes. This class extends *CyclicBehavior* and only method *action()* is redefined using the following template:

```
private class <Agent's name>ReceiveBehavior
    extends CyclicBehavior{
public void action(){
```

Firstly, the message is checked, if the correct ontology is used in it. It is done using message template:

```
  MessageTemplate mt =
      MessageTemplate.and(MessageTemplate.Ma
      tchLanguage(slCodec.getName()),
      MessageTemplate.MatchOntology
      (onto.getName()));
  ACLMessage msg =myAgent.receive(mt);
  if (msg!=null) {
     try
       {
      ContentElement ce = null;
      ce = getContentManager().
      extractContent(msg);
```

Message receiving rules are added using the following template:

```
if (<condition>) <action>;
    else <action>;
```

For example, check if a certain predicate is received is carried out in the following way:

```
  if (ce instanceof <Predicate class>) {
    <actions>
    }
    }
```

Exception handling can be added in such a way:

```
    catch (Codec.CodecException ce) {
     ce.printStackTrace();
     }
    catch (OntologyException oe) {
       oe.printStackTrace();
     }
     }
else
     {
       block();
     }
}
}
```

A single class *SendBehavior* is generated for message sending actions of all agents, because message sending behaviours are identical for all agents. This behaviour is a one-shot behaviour and it needs three parameters (message content, performative and receivers) for creation. Therefore, constructor and method *action()* are overridden:

```
private class SendBehavior extends
    OneShotBehaviour{
     private ArrayList receivers;
     private Object content;
     private int performative;
public SendBehaviour(ArrayList receivers,
    Object content, int performative) {
  super();
  this.receivers=receivers;
  this.content=content;
  this.performative=performative;
}
public void action(){
  try {
      ACLMessage msg = new
      ACLMessage(performative);
      for (int i=0; i<receivers.size(); i++)
       msg.addReceiver((Agent)
        (receivers.get(i)));
       msg.setOntology(onto.getName());
       msg.setLanguage(slCodec.getName());
       getContentManager().fillContent(msg,
        (Predicate)content);
       this.myAgent.send(msg);
  }
  catch (Exception ex) {
     ex.printStackTrace();
  }
}}
```

Other behaviour classes are generated in the following way. Internal behaviour classes are implemented as inner classes in agent class, external behaviour classes are defined in the same holon as agent. Classes are generated automatically, except the action method, which is written manually. Behaviour classes are subclasses of behaviour base class according to the behaviour's type (*OneShotBehavior*, *CyclicBehavior*, etc.). The following template is used:

```
private class <Behaviour's name> extends
    <Superclass>{
  public void action {
    //This part has to be written manually
    }
    }
```

If agents are implemented as holons then each agent from the holon is generated using the same algorithm as the higher level agents. The following steps are carried out to generate each holon designed in the holon diagram:

- Agents of each holon are generated in the package that has the name of the holon. Thus, the package of the holon is created in the package of it's superholon.
- Do steps 5-7 of the generation algorithm for the agents and behaviours of the holon.

D. Batch File Generation

Finally, a batch file used to start system is generated. This file contains commands to start JADE platform, containers and agents. The following templates of commands are included. Set classpath:

```
java -classpath .;<path to jade libraries>
```

Start JADE:

```
jade.Boot <parameters>
```

Start additional containers (specified in deployment diagram) if necessary:

```
jade.Boot - container <parameters>
```

Start agents specified in the deployment diagram:

```
<Agent's name>:<Agent's class>
```

TABLE 2

JAVA ELEMENT GENERATION FROM MASITS DIAGRAMS

No.	Diagram	Java element
1.	Ontology diagram	Ontology package Ontology base class Code fragments in agents' setup methods, message sending and receiving behaviour classes
2.	Agent diagram	Agent's class
3.	Deployment diagram	Batch file used to start the system

To summarise code generation from diagrams and their elements, pairs of generation sources and targets are included in Tables 2 and 3. Table 2 summarizes usage of whole diagrams in code generation; Table 3 summarizes code generation from separate elements.

After code generation by the tool, the programmer has to do the following additional jobs. Firstly, complete the code of behaviour classes by writing *action* methods. Secondly, develop a user interface for the system. Methods of an interface used by interface agents are defined in interaction diagrams. These methods can be partly generated from the interaction diagrams. Implementation of this generation is part of the future work.

IV. CONCLUSIONS

In the absence of a general purpose tool which allows domain-specific diagrams and rules to be plugged-into it and used, a specific tool for multi-agent based ITS development is proposed. The main advantages of the proposed tool are the following. The tool supports the whole life cycle of ITS development, thus, there is no need for any additional tools. A well known agent development environment (JADE) is used for implementation. The tool provides consistency checking

TABLE 3

JAVA ELEMENT GENERATION FROM MASITS DIAGRAM ELEMENTS

No.	Diagram element	Java element
1.	Concepts and predicates	Concept or predicate class Concept/predicate name definition in ontology class Schema elements in ontology base class
2.	Attributes of concepts and predicates	Attributes in concept and predicate classes and according set and get methods Attribute's name definition in ontology class Schema elements in ontology base class
3.	Agent's belief	Attribute in the agent's class
4	Agent's action	Behaviour class
5.	Message and event processing rules	Message receiving behaviour (class and instances in agent classes)
6.	Startup rules	Code fragment in the setup method of agent class
7.	Message link in interaction diagram	Agents' identifiers as attributes in agent classes
8.	Agent's instance	Agent startup code in the batch file
9.	Holon in the holon hierarchy diagram	Holon's package
10.	Platform	Platform startup code in the batch file

and important crosschecks during the development of system, so helping to find possible errors. Finally, Java classes are generated from diagrams created during the design.

The MASITS tool is intended for multi-agent based intelligent tutoring system development. However, it can be used to develop other purpose multi-agent systems, especially multi-agent systems that have similar characteristics to multi-agent based ITSs. The most important preconditions to successfully use the MASITS tool for other agent oriented software are the following: the requirements of the system must include goals and functionality, not organisational structure or agents. The set of agents used to implement the system must come from the domain specific research or has to be defined using some other tools. Additionally, the system must be implemented in JADE. Despite the proposed transformation rules are developed for ITS development, they can be reused in general purpose agent development tools to create the functionality for generation of JADE code from the design.

A simple case study of the MASITS methodology and tool has been developed (for details see [21]). The open ITS for teaching the course "Fundamentals of Artificial intelligence" is implemented. The MIPITS system adapts exercises to the preferences and characteristics of each learner. It is open for new types of exercises that can be introduced just by adding new agents. The case study proved that techniques included in the methodology are suitable for ITS development. Crosschecks carried out using interdiagram links proved to be useful to find design errors. The tool was capable to generate 20-60% of the code of agent classes and ontology classes were generated completely.

Our future work is to evaluate the complexity of the change implementation into the developed open ITS by adding more types of tasks into it. Additionally, development of more complicated ITSs with the methodology and tool is part of the future work.

REFERENCES

[1] S. DeLoach, "Analysis and design using MaSE and agentTool," in *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference*, Oxford OH, March 31 – April 1, 2001, pp. 1-7.

[2] L. Padgham, J. Thangarajah, and M. Winikoff, "Tool support for agent development using the Prometheus methodology," in *Proceedings of the Fifth International Conference on Quality Software (QSIC'05)*, pp. 383-388.

[3] J. Pavon, J. J. Gomes-Sanz, and R. Fuentes, "The INGENIAS methodology and tool," in *Agent-Oriented Methodologies*, B. Henderson-Sellers and P. Giorgini, Eds. London: Idea Group Publishing, 2005, pp. 236-276.

[4] H. Yu, Z. Shen, and C. Miao, "Intelligent software agent design tool using goal net methodology," in *Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pp. 43-46.

[5] M. T. Greaves, H. K. Holmback, and J. M. Bradshaw, "CDT: a tool for agent conversation design," in *AAAI-98 Workshop on Software Tools for Developing Agents*, AAAI 1998, pp 83-88.

[6] Java Agent DEvelopment Framework (JADE). [Online]. Available: http://jade.tilab.com. [Accessed: Oct. 10, 2008].

[7] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas, "JACK intelligent agents – summary of an agent infrastructure," in *Proceedings of the Fifth International Conference on Autonomous Agents*.

[8] Z. Q. Shen, C. Y. Miao, and R. Gay, "Goal-oriented methodology for agent-oriented software engineering," in *IEICE Transactions on Information and Systems*, Special Issue on Knowledge-based Software Engineering, vol. E89-D, no. 4.

[9] J. Grundspenkis and A. Anohina, "Agents in intelligent tutoring systems: state of the art," in *Scientific Proceedings of Riga Technical University*, Computer Science, Applied Computer Systems, 5th series, Vol. 22. Riga: RTU Publishing, pp.110-121.

[10] A. S. G. Smith, "Intelligent tutoring systems: personal notes," School of Computing Science at Middlesex University. [Online]. Available: http://www.eis.mdx.ac.uk/staffpages/serengul. [Accessed: Apr. 18, 2005].

[11] E. Lavendelis and J. Grundspenkis, "Open holonic multi-agent architecture for intelligent tutoring system development," in *Proceedings of IADIS International Conference "Intelligent Systems and Agents 2008,"* Amsterdam, The Netherlands, 22-24 July 2008, pp. 100-108.

[12] E. Lavendelis and J. Grundspenkis, "MASITS – a multi-agent based intelligent tutoring system development methodology," in *Proceedings of IADIS International Conference "Intelligent Systems and Agents 2009"*, 21-23 June 2009, Algarve, Portugal, pp. 116-124.

[13] P. Massonet, Y. Deville, and C. Neve, "From AOSE methodology to agent implementation" in *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, Bologna, Italy, pp. 27 – 34.

[14] G. Caire and D. Cabanillas, "JADE TUTORIAL: application-defined content languages and ontologies." [Online]. Available: http://jade.tilab.com/doc/tutorials/CLOntoSupport.pdf. [Accessed: Oct. 12, 2008].

[15] E. Lavendelis and J. Grundspenkis, "Requirements analysis of multi-agent based intelligent tutoring systems," in *Scientific Proceedings of Riga Technical University*, Computer Science, Applied Computer Systems, 5th series, Vol. 38. Riga: RTU Publishing, 2009, pp. 37-47.

[16] "OMG UML superstructure 2.1.2." Object Management Group. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/07-11-02.pdf. [Accessed: Oct. 03, 2008].

[17] E. Lavendelis and J. Grundspenkis, "Design of multi-agent based intelligent tutoring systems," in *Scientific Proceedings of Riga Technical University*, Computer Science, Applied Computer Systems, 5th series, Vol. 38. Riga: RTU Publishing, 2009, pp. 48-59.

[18] O. Corcho, M. Frenandez–Lopez, and A. Gomez–Perez, "Methodologies, tools and languages for building ontologies. Where is their meeting point?" *Data and Knowledge Engineering*, vol. 46, 2003, pp. 41-64.

[19] J. de Bruijn, "Using ontologies enabling knowledge sharing and reuse on the semantic web," DERI Technical Report DERI–2003-10-29, 2003.

[20] J. Grundspenkis, "Structural modelling of complex technical systems in conditions of incomplete information: a review," in *Modern Aspects of Management Science*, no. 1, Riga, Latvia, pp. 111-135.

[21] E. Lavendelis and J. Grundspenkis, "MIPITS – an agent based intelligent tutoring system," in *Proceedings of the 2nd International Conference on Agents and Artificial Intelligence (ICAART 2010)*, Valencia, Spain, January 22-24, 2010, pp. 5-13.

Egons Lavendelis received Dr.sc.ing. from Riga Technical University (Riga, Latvia) in 2009.

Since 2006 he is a Researcher at Riga Technical University, Department of Systems Theory and Design. His research interests are agent technologies, agent oriented software engineering and intelligent tutoring systems.

Janis Grundspenkis received Dr.habil.sc.ing. from Riga Technical University (Riga, Latvia) in 1993.

He is a Professor at Riga Technical University Department of Systems Theory and Design. His research interests are agent technologies, knowledge engineering and management, and structural modelling.

He is a member of Institute of Electrical and Electronics Engineers (IEEE), Association for Computing Machinery (ACM). He is a full member of Latvia Academy of Science.

Egons Lavendelis, Jānis Grundspeņķis. Daudzaģentu sistēmās sakņotu intelektuālu mācību sistēmu koda ģenerēšana ar MASITS rīku

Lai arī eksistē vairāki aģentu izstrādes rīki, specifisku aģentos sakņotu Intelektuālu Mācību Sistēmu (IMS) izstrādes rīku nav. Taču IMS-ām ir specifiskas īpašības, kas ir jāņem vērā, tās izstrādājot. Vispārīgās aģentorientētas programmatūras izstrādes metodoloģijas un līdz ar to tās atbalstošie rīki nepietiekami atbalsta sistēmu ar tādām īpašībām, kā IMS-as, izstrādi. Bez tam, vispārīgās metodoloģijas neatbalsta problēmsfēras specifisku likumu un diagrammu pievienošanu un līdz ar to arī IMS-u pētījumos iegūto zināšanu izmantošanu izstrādes gaitā. Neeksistējot vispārīgiem rīkiem, kas ļauj tos papildināt ar problēmspecifiskām zināšanām, likumiem un diagrammām, rakstā ir piedāvāts MASITS rīks, kas atbalsta specifisku aģentos sakņotu IMS-u izstrādes metodoloģiju MASITS. MASITS metodoloģija ir veidota atbilstoši IMS-u īpašībām un iekļauj izstrādes procesa kontekstā nozīmīgākās zināšanas no IMS-u pētījumiem. Aprakstītais rīks atbalsta visas IMS-u izstrādes fāzes, sākot no prasību analīzes un beidzot ar realizāciju. Prasību analīzes un projektēšanas fāzēm rīks iekļauj atbilstošu diagrammu zīmēšanas funkcionalitāti. Realizācijas fāzē rīks ļauj ģenerēt sistēmas kodu, izmantojot projektēšanas laikā izveidotās diagrammas. Ar MASITS rīku izstrādātās sistēmas tiek realizētas JADE platformā. Līdz ar to, rīks ģenerē ontoloģijas, aģentu un to uzvedību Java klases. Raksts satur īsu koda ģenerēšanas laikā izmantoto diagrammu aprakstu un detalizētus algoritmus JADE platformai atbilstoša Java koda ģenerēšanai no tām.

Эгонс Лавенделис, Янис Грундспенькис. Генерация кода для интеллектуальной обучающей системы, основанной на интеллектуальных агентах, с использованием программы MASITS

Несмотря на т, что на данный момент существуют различные программные продукты для создания систем на основе интеллектуальных агентов, ни одна из существующих программ не специализирована на разработку интеллектуальных обучающих систем (ИОС). ИОС обладает рядом характеристик, которые необходимо брать во внимание, разрабатывая данные системы. Общие методологии разработки агентно-ориентированных систем, и им соответствующие среды разработки, в достаточной степени не поддерживают в особенности ИОС. Общие методологии не позволяют присоединять специализированные правила и диаграммы, используемые при создании ИОС. Таким образом, учитывая недостатки общих методов и инструментов, была разработана методология для создания агентно-ориентированных ИОС MASITS. Методология учитывает особенности разработки ИОС, выделенные в ходе исследований, проведенных в данной области. На основе предложенной методологии был разработан программный продукт MASITS. Система MASITS поддерживает все этапы разработки ИОС, начиная с анализа требований и заканчивая внедрением и запуском системы. Для сбора и анализа требований предлагается использовать соответствующие MASITS диаграммы, на основе которых в дальнейшем происходит автоматическая генерация кода. Для разработки и внедрения интеллектуальных агентов используется платформа JADE. Таким образом JAVA классы для отображения онтологий, агентов и их поведения генерируются на основе диаграмм, подготовленных на этапе проектирования. Данная статья включает краткий обзор диаграмм, а также детальное описание алгоритмов, используемых для генерации кода.