2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology

Modular JADE Agents Design and Implementation using ASEME

Nikolaos Spanoudakis
Technical University of Crete, Greece
nikos@science.tuc.gr

Pavlos Moraitis
LIPADE, Paris Descartes University, France
pavlos@mi.parisdescartes.fr

Abstract

ASEME is an emerging Agent Oriented Software Engineering (AOSE) methodology. The Model-Driven Engineering (MDE) paradigm encourages software modelers to automate the transition of one type of software model to another and eventually the code generation process. This paper builds on previous work that describes the model-driven development of agent systems using ASEME and creating a Platform Independent Model (PIM) that adheres to the language of statecharts, the Intra-Agent Control Model. In this contribution we use the generated statecharts and show how to automatically transform them to Java programs using the Java Agent Development Framework (JADE). All agent and behaviour classes are automatically generated including the agent interaction protocols.

1. Introduction

A challenge in the Agent Oriented Software Engineering (AOSE) field is the automation of agent code generation from a design model. Most methodologies either automatically generate portions of the agent code or provide guidelines for the programmers to transform their design models to implementation models. The Java Agent Development Framework (JADE) has been used as a target agent implementation platform by many AOSE methodologies such as Ingenias [5], PASSI [3], Gaia [17], and, recently, by the Agent Systems Engineering Methodology (ASEME[1] [16], [13]). JADE has many qualities as it is open source; it is compliant with the FIPA (the Foundation for Intelligent Physical Agents standardization body) standards and can be compiled for devices with limited resources such as PDAs.

This paper extends our previous work [14], where we showed how to use Model-Driven Engineering (MDE) principles to define the Platform Independent Model (PIM) of a system specification as an Intra-Agent Control (IAC) model adhering to the language of statecharts [6]. Here-in, we show how the IAC can be transformed to a Java program using JADE and employing the agent's capabilities as reusable software components.

We define an automatic transformation of an Intra-Agent Control (IAC) model to JADE implementation (IAC2JADE) including the unique capability to derive the needed JADE behaviour types and also the possibility to automatically define all interactions needed by a complex agents' interaction protocol. This paper not only provides these theoretical results but also an implementation using the Xtext and Xpand languages [8] and the Eclipse Modeling[2] popular Integrated Development Environment (IDE).

2. Metamodels and Models Transformation

Model driven engineering relies heavily in model transformation [12]. Model transformation is the process of transforming a model to another model. The requirements for achieving the transformation are the existence of metamodels of the models in question and a transformation language in which to write the rules for transforming the elements of one metamodel to those of another metamodel.

In the software engineering domain a *model* is an abstraction of a software system (or part of it) and a *metamodel* is another abstraction, defining the properties of the model itself. However, even a metamodel is itself a model. Thus, there is yet another level of abstraction, the *metametamodel*, which is defined as a model that conforms to itself [7].

[1] From the ASEME web site the interested reader can download the tools and metamodels defined in this paper, URL: http://www.amcl.tuc.gr/aseme

[2] The Eclipse Modeling Project provides a unified set of modeling frameworks, tooling, and standards implementations, URL: http://www.eclipse.org/modeling/

978-0-7695-4191-4/10 $26.00 © 2010 IEEE
DOI 10.1109/WI-IAT.2010.136

A transformation that is used for transforming a graphical model to a textual representation (i.e. a computer program) is called a *Model to Text (M2T)* transformation. The graphical model must have a metamodel. Then, a transformation of the graphical model to text can be defined.

In the heart of the model transformation procedure is the Eclipse Modeling Framework (EMF, [2]). Ecore [2] is EMF's model of a model (metamodel). It functions as a metametamodel and it is used for constructing metamodels. It defines that a model is composed of instances of the *EClass* type, which can have attributes (instances of the *EAttribute* type) or reference other EClass instances (through the *EReference* type). Finally, EAttributes can be of various *EDataType* instances (such are integers, strings, real numbers, etc).

2.1 Statechart Definition and Metamodel

Statecharts [6] are used for modeling systems. They are based on an activity-chart that is a hierarchical data-flow diagram, where the functional capabilities of the system are captured by activities and the data elements and signals that can flow between them. Statecharts define the behavioral aspects of a set of activities. The activities are represented as states that can be a) OR-states, b) AND-states, and c) basic states. OR-states have substates that are related to each other by "exclusive-or", and AND-states have orthogonal components that are executed concurrently, thus are "and" related. Basic states are those at the bottom of the state hierarchy, i.e., those that have no substates. A state with no parent state is called the root. Each transition from one state (source) to another (target) is labeled by an expression, whose general syntax is e[c]/a, where e is the event that triggers the transition; c is a condition that must be true in order for the transition to be taken when e occurs; and a is an action that takes place when the transition is taken. All elements of the transition expression are optional. Here we must note that if *SO* is the set of OR-states that are proper common ancestors of the source and target states of a transition, then the scope of the transition is defined as the element of *SO* that has no substates that are members of *SO*. CONDITION states are used when more than one transition have the same source state and other-different- states as targets, and the target state is determined by one or more conditions. Multiple concurrently active statecharts are considered to be orthogonal components at the highest level of a single statechart.

The fact that the statechart can capture together the functional and behavioral aspects of a system is its greatest advantage, as it completely defines a system. We use statecharts in a specific level of abstraction, that of an agent, in order to model the interactions between its components (or capabilities). The statechart, therefore, implements the *intra-agent control model* (IAC) of an agent. The formal model that is adopted here-in is part of the Agent Modeling Language (AMOLA) [13].

Before giving a formal representation of a statechart we need to present some useful definitions. Thus, an *ordered rooted tree* is a rooted tree where the children of each internal vertex are ordered [11]. To produce a total order of the vertices of an ordered rooted tree all the vertices must be labeled. This is achieved recursively as follows:

1. Label the root with the integer 0. Then label its k children (at level 1) from left to right with 0.1, 0.2, 0.3, …, 0.k.
2. For each vertex v at level n with label A, label its k_v children, as they are drawn from left to right, with $A.1, A.2, \ldots, A.k_v$.

Thus, A.1 means that A is the parent of A.1. A statechart can therefore be formally defined as follows (the definition is inspired by the one proposed by David et al. [4]).

Definition 1. A *statechart* is a tuple (L, δ) where:

- $L = (S, \lambda, Var, Name, Activity)$ is an ordered rooted tree structure representing the states of the statechart
  - $S \subseteq \mathbb{N}^*$ is the set of all nodes in the tree
  - λ: $S \rightarrow$ {AND, OR, BASIC, START, END, CONDITION} is a mapping from the set of nodes to labels giving the type of each node
  - *Var* is a mapping from nodes to sets of variables
  - *Name* is a mapping from nodes to their names
  - *Activity* is a mapping from nodes to their algorithms/functionality in text format
- $\delta \subseteq S \times TE \times S$ is the set of state transitions

The Intra-Agent Control (IAC) is defined as a statechart. IAC allows the modeling of interactions between the different capabilities of an agent. Its metamodel (see Figure 1) contains nodes and transitions according to Definition 1. The metamodel defines a *Model* concept that has *nodes*, *transitions* and *variables* EReferences. Note that it also has a *name* EAttribute. The latter is used to define the namespace of the statechart. A namespace is an abstract container conceived to hold a logical grouping of unique identifiers or symbols. The namespace should follow the Java or C# modern package namespace format.

The nodes contain the following attributes (followed by the relevant concept name in the statechart definition): a) *name* (Name), usually named after the Gaia liveness formula expression (see section 3 for more details), b) *type* (λ), c) *label* (label), and, d) *activity* (Activity). Nodes also refer to *variables*. The Variable EClass has the attributes *name* and *type* (e.g. the variable with *name* “count” has *type* “integer”). Finally the transitions have four attributes: a) *name*, b) *TE*, the transition expression, c) *source*, the source node label, and, d) *target*, the target node label.

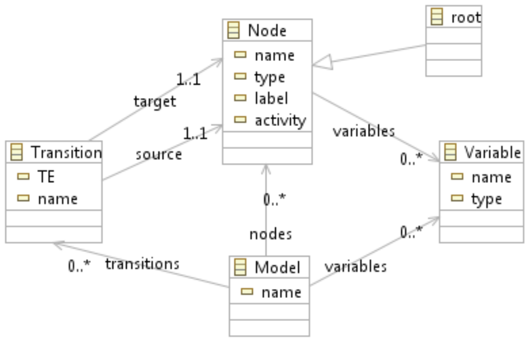


Figure 1. The statechart metamodel.

2.2 The JADE Framework

In JADE, each agent is equipped with an incoming message box. Moreover, JADE provides methods for message filtering. The developer can apply advanced filters on the various fields of the incoming messages such as sender, performative or ontology.

Agent tasks or agent intentions are implemented through the use of behaviours. Each behaviour performs its designated operation by executing the core method *action()*. *Behaviour* is the root class of the behaviour hierarchy that defines several core methods. The children of this base class are *SimpleBehaviour* and *CompositeBehaviour*.

The classes that descend from *SimpleBehaviour* represent atomic simple tasks that can be executed a number of times specified by the developer. The class *CyclicBehaviour* models atomic behaviours that must be executed forever. So, its *done()* method (the method called at the end of each behaviour’s action execution) always returns false.

Classes descending from *CompositeBehaviour* support the handling of multiple behaviours according to a policy. The actual agent tasks that are executed through this behaviour are not defined in the behaviour itself, but inside its children behaviours. The class *SequentialBehaviour* executes its sub-behaviours sequentially and terminates when all sub-behaviours are done. The class *ParallelBehaviour* executes its sub-behaviours concurrently and terminates when a particular condition on its sub-behaviours is met.

The developer creates his agents by extending the JADE *Agent* class. He can add any number of behaviours along with defining the agent’s initialization and termination handling functionality.

3. The IAC2JADE Transformation

In [14] we showed how to automatically transform the Gaia liveness property [17] to a statechart compatible with the AMOLA Intra-agent Control (IAC) model. The analysis phase System Roles Model (SRM) of AMOLA has adopted the Gaia liveness formulas for defining the dynamic behaviour of the role. This section shows how this statechart is automatically transformed to Java code using the JADE API. The transformation process will be illustrated by an example regarding a meetings management system. For the better understanding of what happens in the design phase we will use an example starting from the analysis phase. An agent role participating in a meetings management system is the personal assistant of each user (see Figure 2). The last formula of this role’s liveness property shows how this agent implements his participation in an agent interaction protocol, i.e. the *negotiate meeting date* protocol. We suppose that the reader is familiar with the Gaia methodology.

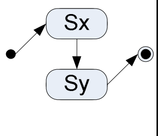
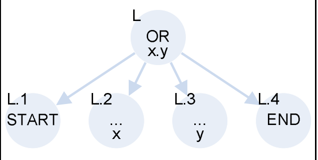
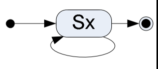
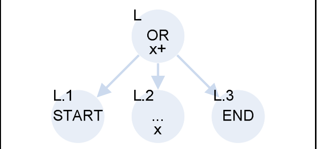
Role: Personal Assistant
Liveness:
personal assistant = (manage meetings. learn user habits)$^{\omega}$ || (negotiate meeting date)$^{\omega}$
manage meetings = get user request. (read schedule | request change meeting | request new meeting). show results
learn user habits = learn user preference. update user preferences
request change meeting = send change request. receive change results
request new meeting = send new request. receive new results
negotiate meeting date = *receive proposed date. (decide response. send results. receive outcome)+.* update schedule

Figure 2. The Personal Assistant Role

Starting from the role model an automated process applies transformation templates to the Gaia operators and recursively creates the IAC model. For our example the resulting IAC model for the Personal Assistant role is shown in Figure 4. The details for this transformation are presented in [14]. A rooted tree resembles this statechart. For demonstrating the recursive tree building process we show a branch of this tree in Figure 3 and the templates used for this branch in Table 1. Figure 3 presents a graphical representation of the statechart according to Definition

1 for the negotiation protocol. This protocol is one of the capabilities of the agent and it is intended to become a software module. In the figure, the reader can see for each node its label (top), type (middle) and name (bottom). If there is no name listed then the name is the same with the node's label. Grey nodes and arrows represent the statechart ordered routed tree, while the black arrows represent the state transitions.

Table 1. Templates for Statechart generation

Operator	*Template*	*Tree Branch*
x.y		 L OR x.y L.1 START L.2 ... x L.3 ... y L.4 END
x+		 L OR x+ L.1 START L.2 ... x L.3 END

Having transformed the liveness formula to a statechart, the designer can insert conditions for executing an activity (which may involve the use of variables) and the receipt or the dispatch of inter-agent messages as events. AMOLA makes no assumptions on ontology, communication means, reasoning processes, or the mental attitudes (e.g. belief-desire-intentions) of the agents, giving this freedom to the designer. However, it does define a grammar (see [13]) for the definition of the state transition expressions. The ontology can be defined in object-oriented format or in logic based format. The definition of the ontology using one format does not forbid the use of another during the development of the system. For example, in [15], we defined a way to encode an ontology that was developed using the Protégé ontology editor (http://protege.stanford.edu), in object-oriented format, to Prolog (logic programming) format. The ontology concepts and their properties can be used in defining conditions, events and actions in the IAC transition expressions.

3.1 The IAC2JADE Transformation Algorithm

Four types of JADE behaviours are automatically generated according to the transformation process. The transformation algorithm reads the statechart model (IAC) and creates Java source code files using templates (defined in the Xpand language).

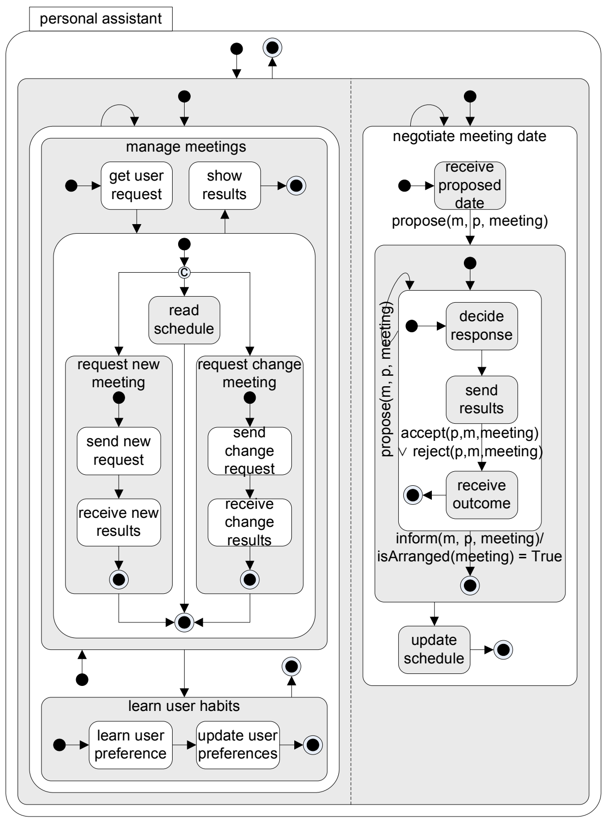


Figure 4. Personal Assistant Agent IAC Model

The transformation algorithm processes each node of the statechart (IAC). If it is the root, then it is transformed to a JADE *Agent* descendant class. All the

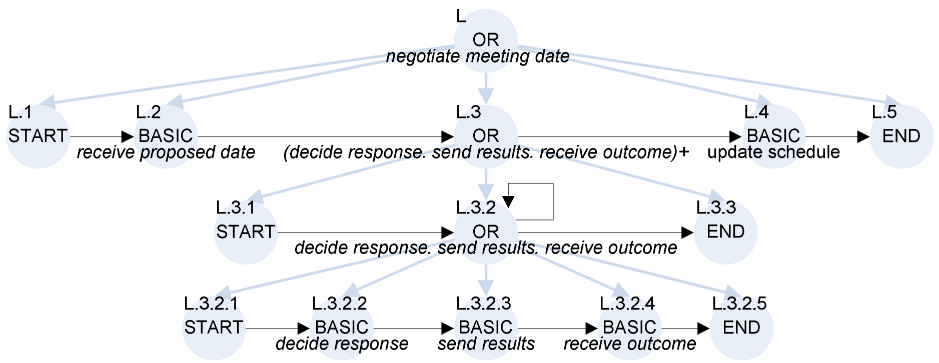


Figure 3. A graphical representation of a statechart branch starting at depth L.

nodes of the statechart that are of types OR, AND or BASIC (called the *eligible nodes* from now on) are transformed to some kind of JADE behaviour. Notice that the nodes of type START, END and CONDITION are not transformed to Behaviour classes.

For each of the other (than the root) eligible nodes one of the following holds:

- **If** the node's type is "BASIC" then it is transformed to a JADE *SimpleBehaviour* (corresponding to a Gaia activity).
    - **If** the node's name starts with "Send", then add a reference to the JADE *ACLMessage* class and write code for sending a message depending on the events of the transitions that have this node as their source. For example the "send results" activity in Figure 4 should send an ACL message with either the *accept* or *reject* performatives, the personal assistant as sender, the meetings manager as receiver and a *Meeting* instance as message body (from the transition expression exiting the state).
    - **Else**, **if** the node's name starts with "Receive", then add a reference to the JADE *ACLMessage* class and write code for receiving a message depending on the events of the transitions that have this node as their target. Also, add a reference to the *MessageTemplate* JADE class that is used for defining the type of message expected and instantiate it according to the events of the transitions that have this node as their target.
    - **Else**, add in the action method of the behaviour the contents of the node's *Activity* attribute.
- **Else**, **if** the node's type is "AND", then it is transformed to a JADE *ParallelBehaviour* (the "||" Gaia operator connects its sons). All the eligible sons of the node are added as threaded behaviours and the *ParallelBehaviour* ends when all its children have ended.
- **Else**, **if** the node has two sons, the second of which has a transition to itself then the latter is the case of a behavior that will execute forever (corresponding to the "*ω*" Gaia operator). Thus, this node must be transformed to a behavior that will continuously instantiate its second son (the first is a node of type START, thus is ignored). This is achieved by transforming it to a *CyclicBehaviour* that checks if the eligible son has finished and if this is true it restarts it.
- **Else**, **if** the node has three sons, the second of which has a transition to itself then the latter is the case of a behavior that will execute one or more times (corresponding to the "+" Gaia operator). Thus, this node must be transformed to a behavior that will continuously instantiate its second son (the first is a node of type START, thus is ignored), while a specific condition holds. This is achieved by transforming it to a *SimpleBehaviour* that checks if the eligible son has finished and then, if the condition of the transition that has it as target is true, it restarts it. If not the behavior terminates.
- **Else**, **if** the node has a son whose type is CONDITION then
    - **If** the node has four sons, then its third son is the case of a behavior that will execute zero or more times (corresponding to the "*" Gaia operator). Thus this node must be transformed to a behavior that will conditionally instantiate its third son (the first is a node of type START, the second the one of type CONDITION). This is achieved by transforming it to a *SimpleBehaviour* that conditionally adds the sub-behaviour in its constructor and that checks (in its action method) if the eligible son has finished and then if the condition of the transition that has it as target is true it restarts it. If not the behavior terminates.
    - **Else** this node has a number of eligible sons one of which must be instantiated (the "|" Gaia operator connects its sons). It is transformed to a *SequentialBehaviour* and at its constructor it conditionally instantiates one of its sons.
- **Else** this node has a number of eligible sons that must be executed sequentially (the "." Gaia operator connects its sons). This is achieved by transforming it to a *SequentialBehaviour* and adding all its eligible sons sub-behaviours.

Using the above algorithm we can deduce that the nodes L, L.3.2 (in Figure 3) are transformed to *SequentialBehaviour* descendants and L.2, L.3, L.4, L3.2.2, L3.2.3 and L3.2.4 to *SimpleBehaviour* descendants.

3.2 The Algorithm's Implementation

The transformation process is comprised of multiple steps and Eclipse allows to define this process (another advantage of using Xpand) using a workflow file. The latter can be used to define execution parameters, usually through property files, and file generating components. It initially loads some parameters through a property file, the most important of which are the name of the IAC model file and the directory for producing the source code.

An Xpand file defines the templates for creating the Java classes. The definition of the agent Xpand template is presented in Figure 5. It starts with the DEFINE statement. The FILE statement defines the name of the file that is outputted and its body is the file template. The used properties of the metamodel are inserted in the «» tags. If the text in the «» tags ends with parenthesis optionally including more text (e.g. the case *classFileName()*) then it implies the execution of an Xtend function. The *getAgentBehaviour* Xtend function for example searches the tree for the eligible sons of the root so that it finds the behaviours that need to be added to the *setup* method of the agent. The agent Xpand template file continues by defining relevant templates for the different behaviours. The template that will be used each time is selected according to the IAC2JADE algorithm that we showed earlier.

```
«DEFINE agentClass(String packageName, Model model)
FOR IAC::Node»
 «FILE classFileName()»
 package «packageName»;
 import jade.core.Agent;
 public class «className()» extends Agent{
  public void setup(){
   addBehaviour(«getAgentBehaviour(this,model)»);
  }
  protected void takeDown() {
   doDelete();
  }
 }
 «ENDFILE»
«ENDDEFINE»
```

Figure 5. The Agent class Xpand template

For each variable in the IAC model a java class will be created. If the variable type is that of an *ACLMessage* then the relevant class is imported from the jade framework. For all other variable types it is assumed that the ontology created for this project will contain them. In the case of the meetings management project, there are two variable types, the *Meeting* variable type refers to a class defined in the ontology of the project and the *ACLMessage* variable type. The class generated by the Xpand template is named after the type of the variable including the string "Holder". Thus, the class generated for the *Meeting* variable type is the *MeetingHolder* class. The latter has two attributes, the *owner*, which is a reference to a JADE Behaviour class (where the behavior that instantiates this variable is inserted through the class constructor) and the *meeting* attribute that references the *Meeting* class. This approach, which is transparent to the developer, allows a behaviour to change a variable value and this change to be visible to all behaviours that share this variable. The variables have the scope of the transition in the expression of which they are used; however, the developer may opt to widen the scope. Variables are used to coordinate the execution of the different agent capabilities (by being used in transition expressions) and to allow capabilities to exchange information (the scope of a variable defines which capabilities can access it).

If the user has inserted the activity related to each node in java code format he has to denote this by starting the activity description with the string "/*Java code*/". In this case the code is inserted as-is in the action method of the *SimpleBehaviour*. However, in normal projects it is expected that diverse technologies will be involved, in which case the programmers (each an expert in his own domain, e.g. logic programming, web service invocation, etc) will have to edit the action methods of the simple behaviours.

Thus, the ASEME developer can generate all the needed classes for his project just by executing the "workflow.oaw" transformation workflow file in the Eclipse IDE. The resulting files are the JADE *Agent* and *Behaviour* descendant classes along with the variable holder classes, 31 files total for the personal assistant agent. For this transformation the workflow execution time was 982 milliseconds in a normal workstation with the Intel Core2Duo processor running at 2.66GHz and 2GB of RAM.

The *ReceiveOutcomeBehaviour*, which corresponds to the BASIC state "receive outcome" of the IAC model, is depicted in Figure 6. The properties of the class are two holders for ACL messages and one holder for the *Meeting* class. These are initialized through its constructor. The *action* and *done* methods have been automatically produced including the definition of the *MessageTemplate* class that defines the characteristics of the expected message.

Finally, it is worth showing how a capability has been defined as a software module. The automatically generated *NegotiateMeetingDateBehaviour.java* file is presented in Figure 7. It is the implemented personal assistant's part of the "Negotiate Meeting Date" protocol. The reader can see that it defines the ACL message holders for the types of messages that it handles and which it then uses for adding its children behaviours to the agent scheduler. This behaviour along with its children behaviours and the used variables can be used by any future JADE agent that wants to participate as a personal assistant to the "Negotiate Meeting Date" protocol. He just has to add the *NegotiateMeetingDateBehaviour* to his agent's behaviour scheduler. The *MeetingHolder* variable is used to integrate this module to other agent modules (it is the variable passed to the behaviour constructor).

The reader should also note that the behaviour types that are not SimpleBehaviours are not meant to be edited by the programmers, they are complete. Thus, among the 31 automatically generated classes for the personal assistant agent of our example, the developer will need just to define the action methods of 14 of them, seven of which are message send and receive methods, which will just require a final touch. Therefore, the programmers will just have to write code for seven methods.

```
package fr.parisdescartes.mi.meetingsmanagement;
import jade.core.Agent;
import jade.core.behaviours.SimpleBehaviour;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

public class ReceiveOutcomeBehaviour extends
SimpleBehaviour {
 MeetingHolder e = null;
 ACLMessageHolder inform = null;
 ACLMessageHolder propose = null;
 protected MessageTemplate mt = null;
 boolean finished = false;
 public ReceiveOutcomeBehaviour(Agent a,
MeetingHolder e,  ACLMessageHolder inform,
ACLMessageHolder propose) {
  super(a);
  this.e = e;
  this.inform = inform;
  this.propose = propose;
 }
 public void action() {
  mt = MessageTemplate.MatchPerformative(ACLMessage.
PROPOSE);
  mt = MessageTemplate.or(mt, MessageTemplate.
MatchPerformative(ACLMessage.INFORM));
  ACLMessage msg = myAgent.receive(mt);
  if (msg != null) {
   //insert message handling code
   finished = true;
  } else {  block(); }
 }
 public boolean done() { return finished; }
}
```

Figure 6. A JADE receiver behaviour

4. Related Work and Conclusion

This paper showed how engineers, who adopt the ASEME methodology, can design autonomous agents (and multi-agent systems) and implement them using the JADE framework. ASEME supports automatic code generation from the design phase model (the AMOLA Intra-Agent Control model), which is itself automatically derived from the analysis phase model and, specifically, the Gaia liveness formulas. The information added in the design phase is the IAC (which is a statechart) transition expressions and the variables accessed by the different nodes.

In the past, researchers have proposed some methods for generating JADE code from Gaia models. The Gaia2JADE process [9], did not automate the model transformation procedure from the Gaia roles model to JADE code, it just provided some rules of thumb for the selection of specific JADE behaviours based on the Gaia liveness formula. Another work [1] automated this selection using semantic web technologies but stopped at defining each Gaia activity as a JADE Behaviour (did not go in the detail of what type of behaviour and did not use JADE composite behaviours).

Other AOSE methodologies such as PASSI [3] and Ingenias [5] also provide some kind of automation for producing JADE code. PASSI employs the AgentFactory tool to help the developer edit a JADE-based implementation. This tool allows quick access to all FIPA specified communication protocols and the defined ontology. Ingenias is itself a development framework that the developer must study and learn in the same way that he learns the JADE framework. Moreover, it uses a specific mental model, while in ASEME the developers may use any mental model that suits their problem.

```
package fr.parisdescartes.mi.meetingsmanagement;
import jade.core.Agent;
import jade.core.behaviours.SequentialBehaviour;

public class NegotiateMeetingDateBehaviour extends
SequentialBehaviour {
 MeetingHolder e = null;
 ACLMessageHolder accept=new ACLMessageHolder(this);
 ACLMessageHolder inform=new ACLMessageHolder(this);
 ACLMessageHolder propose=new ACLMessageHolder(this);
 ACLMessageHolder reject=new ACLMessageHolder(this);
 public NegotiateMeetingDateBehaviour(Agent a,
MeetingHolder e) {
  super(a);
  this.e = e;
  addSubBehaviour(new ReceiveProposedDateBehaviour(
this.myAgent, e, propose));
  addSubBehaviour(new _open_group_DecideResponse_
sequence_SendResults_sequence_ReceiveOutcome_close_
group__one_or_more_times_Behaviour(this.myAgent, e,
accept, inform, propose, reject));
  addSubBehaviour(new UpdateScheduleBehaviour(
this.myAgent, e, inform));
 }
}
```

Figure 7. A JADE composite behaviour.

This work has some significant advantages compared to [1] and [9]. Using the IAC model it allows the designer to enrich his roles model by introducing variables, agent message types and adding conditions and events in the statechart transition expressions used in the Agent level to coordinate the execution of the agent capabilities. This allows for richer code generation using the JADE *CompositeBehaviour* descendant classes to orchestrate the implementation of the statechart logic, an advantage over [3] and [5] as well. The fact that the statechart logic is expressed through the use of 100% automatically generated behaviours addresses the "post-editing problem". The latter prevents the re-generation of code from models after doing some manual editing in the generated source files. In ASEME, the developer can manually edit the simple behaviors and re-generate the complex behaviors when the statechart logic is changed.

For evaluating our work we used two case studies on the development of two real-world systems. The first is a module of the ASK-IT project [10], where we built an agent-mediated service brokering system including a broker agent and several added value service provider agents. It included programming for semantic service matching and interfaces to other modules that were based on OSGi, a service oriented architecture. The ontology was developed using the Protégé ontology editor and its beangenerator add-on, which generates java files representing an ontology that can be used with the JADE environment.

The second is the Market-Miner project [15] where we developed an autonomous product pricing agent situated in a firm monitoring for changes of the prices of competitors along with changes in firm policies and deciding on the prices of the products on the self. In Market-Miner we used Prolog for implementing the decision making capability of the agent. Again, we used the Protégé editor for creating the ontology.

These projects used different implementation platforms, the first the JADE platform, while the second a Java CASE tool, IBM Rational Rhapsody (URL: *http://www.ibm.com*). For the latter it was needed to transform the SRM model to an IAC model manually (as Rhapsody does not offer an import tool for statechart models) using the process defined in [14]. Table 1 shows the percentages of the total code (code is measured in bytes) that was generated automatically for each project (26% and 54% respectively) as a benchmark.

Table 2. ASEME case studies

Project – Case study	**ASK-IT**	**Market-Miner**
Implementation Platform	*JADE*	*Rhapsody*
Total project code (bytes)	160,241	179,750
Automatically generated code	41,759	97,631
Written Java code	118,482	60,365
Other written code (Prolog)	-	21,754
Automatically generated	**26.06%**	**54.31%**

References

[1] I.I. Bittencourt, P. Bispo, E. de Barros Costa, J. Pedro, D. Véras, D. Dermeval, H.P.L. Luna, "Modeling JADE Agents from GAIA Methodology under the Perspective of Semantic Web", *Lecture Notes in Business Information Processing 24*, Springer-Verlag, 2009, pp. 780-789.

[2] F. Budinsky, D. Steinberg, R. Ellersick, E. Merks, S.A. Brodsky, T.J. Grose, *Eclipse Modeling Framework*, Addison Wesley, 2003.

[3] A. Chella, M. Cossentino, L. Sabatucci, "Tools and patterns in designing multi-agent systems with PASSI", *WSEAS Transactions on Communications 3(1)*, 2004.

[4] A. David, J. Deneux, J. d'Orso, "A Formal Semantics for UML Statecharts", Technical Report 2003-010, Uppsala University, 2003.

[5] J.J. Gomez-Sanz, J. Pavon, "Implementing Multi-agent Systems Organizations with INGENIAS", *Lecture Notes in Computer Science 3862*, Springer, 2006, pp. 236-251.

[6] D. Harel, H. Kugler, "The RHAPSODY Semantics of Statecharts (Or on the Executable Core of the UML)", *Lecture Notes in Computer Science 3147*, 2004, pp. 325-354.

[7] F. Jouault, J. Bézivin, "KM3: A DSL for Metamodel Specification", *Lecture Notes in Computer Science 4037*, Springer, Heidelberg, 2006, pp. 171-185.

[8] B. Klatt, "Xpand : A Closer Look at the Model2Text Transformation Language", 2007, URL: http://bar54.de/ benjamin.klatt-Xpand.pdf

[9] P. Moraitis and N. Spanoudakis, "The Gaia2JADE Process for Multi-Agent Systems Development", *Applied Artificial Intelligence Journal 20(4-5)*, 2006, pp. 251-273.

[10] P. Moraitis and N. Spanoudakis, "Argumentation-based Agent Interaction in an Ambient Intelligence Context", *IEEE Intelligent Systems 22(6)*, 2007, pp. 84-93.

[11] H.K. Rosen, *Discreet Mathematics and its Applications*, Forth edition, McGraw Hill, 1999.

[12] S. Sendall, W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development", *IEEE Software 20(5)*, 2003, pp. 42-45.

[13] N. Spanoudakis, *The Agent Systems Engineering Methodology (ASEME)*, Philosophy Dissertation, Paris Descartes University, Paris, France, 2009, URL: http://users.isc.tuc.gr/~nispanoudakis/SpanoudakisThesis.pdf

[14] N. Spanoudakis, P. Moraitis, "Gaia Agents Implementation through Models Transformation". *Lecture Notes in Artificial Intelligence 5925*, 2009, pp. 127-142.

[15] N. Spanoudakis, P. Moraitis, "Automated Product Pricing Using Argumentation", *IFIP Advances in Information and Communication Technology Book series, Vol. 296*, Springer, 2009, pp. 321-330.

[16] N. Spanoudakis, P. Moraitis, "Model-Driven Agents Development with ASEME", *11th International Workshop on Agent Oriented Software Engineering (AOSE 2010)*, Toronto, Canada, May 10-11, 2010, pp. 49-60.

[17] F. Zambonelli, N.R. Jennings, M. Wooldridge, "Developing multiagent systems: the Gaia Methodology", *ACM Transactions on Software Engineering 12(3)*, 2003, pp. 317-370.