Agent Systems Engineering Methodology: The Development Process

Nikolaos Spanoudakis[1,2], Pavlos Moraitis[2]

[1] Technical University of Crete, Department of Sciences,
University Campus, 73100, Kounoupidiana, Greece
nikos@science.tuc.gr
[2] Paris Descartes University, Department of Mathematics and Computer Science,
45, rue des Saints-Pères, 75270 Paris Cedex 06, France
{nikolaos.spanoudakis, pavlos}@mi. parisdescartes.fr

ABSTRACT

In this paper, we describe the process followed by the Agent Systems Engineering Methodology (ASEME), a new agent-oriented software engineering methodology (AOSE). This process integrates successful method fragments from existing methodologies, i.e. Gaia and Tropos. ASEME applies a model driven engineering approach to multi-agent systems development, thus the models of a previous development phase are transformed to models of the next phase. It includes the CIM, PIM and PSM models proposed by the model driven architecture specification of OMG and it introduces the system roles model (SRM) as the output of the analysis phase. We describe the development process using SPEM, the language that is proposed by FIPA for such processes specification. The ASEME Platform Independent Model (PIM) that is the output of the design phase is a statechart that can be instantiated in a number of platforms using existing CASE tools. In this paper, we also provide the model transformation process for the popular JADE platform. ASEME supports a modular agent design approach and introduces the concepts of intra-and inter-agent control. The first defines the agent's lifecycle by coordinating the different modules that implement his capabilities, while the latter defines the protocols that govern the coordination of the society of the agents. The modeling of the intra and inter-agent control is based on statecharts. The analysis phase builds on the concepts of capability and functionality. AMOLA deals with both the individual and societal aspect of the agents showing how protocols and capabilities can be integrated in agents designs.

1. INTRODUCTION

The Agent Systems Engineering Methodology (ASEME) is a recently emerging methodology for developing multi-agent systems. Its major advantages to existing methodologies are that it builds on existing languages such as statecharts [7] and UML [13] (which are familiar to engineers) in order to represent system analysis and design models. It provides three different levels of abstraction, thus catering for large-scale systems involving diverse technologies. It is agent architecture and agent mental model independent, allowing the designer to select the architecture type and the mental attributes of the agent that he prefers (e.g. procedural agents, belief-desire-intentions (BDI) agents, etc). Moreover, the ASEME process follows the modern model driven engineering style, thus the models of each phase are produced by applying transformation rules to the models of the previous phase. Each phase adds more detail and becomes more formal leading gradually to implementation. Thus, ASEME is a model-driven engineering (MDE, [2]) process that can be automated by using rules for models transformation and knowledge for adding detail in every development phase. We define a platform independent model at the end of the design phase that describes the system and allows its implementation with the use of different platforms or programming languages. In this paper we provide the model transformation process for implementing a multi-agent system using the popular Java Agent Development Environment (JADE). The process is formally presented using the Object Management Group's (OMG) Software Process Engineering Metamodel (SPEM) that has been used in the past for modeling such processes and is also used by the Foundation of Intelligent Physical Agents (FIPA) agent technology standardization body.

The main contribution of this paper is the presentation of the ASEME process showing the development steps and their products, as well as the models transformation between the different development phases. The latter allows for traceability from requirements to implementation level and facilitates iteration between the different software development phases. The models that are used by ASEME are defined by the Agent Modeling Language (AMOLA). This paper contains a working example that allows for the understanding of the ASEME process and the AMOLA models.

In what follows, section 2 provides an overview of the ASEME process, followed by the presentation of each development phase in sections 3 to 6. Section 7 provides some insight on agent systems verification and optimization, even though these issues are out of the scope of this paper. Finally, section 8 concludes.

2. ASEME PROCESS OVERVIEW

The software development phases of the Agent Systems Engineering Methodology (ASEME) are presented in Figure 1. There are six phases, the first four produce system models (development phases), while the last two (verification and optimization phases) evaluate and optimize these models. The process is iterative allowing for incremental development and provides the original possibility to jump backwards to any previous phase due to the utilized model driven engineering (MDE) approach.

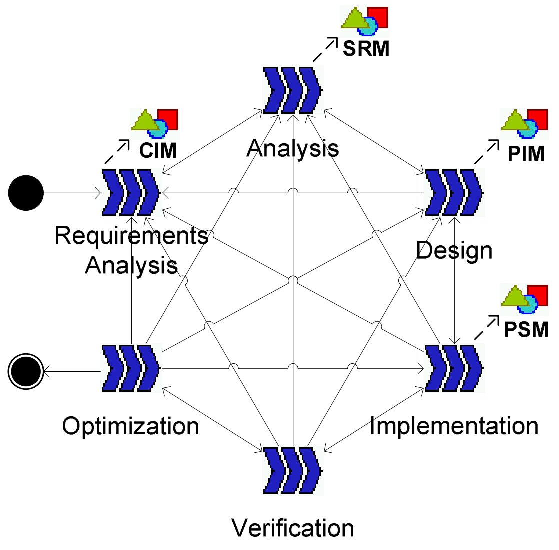


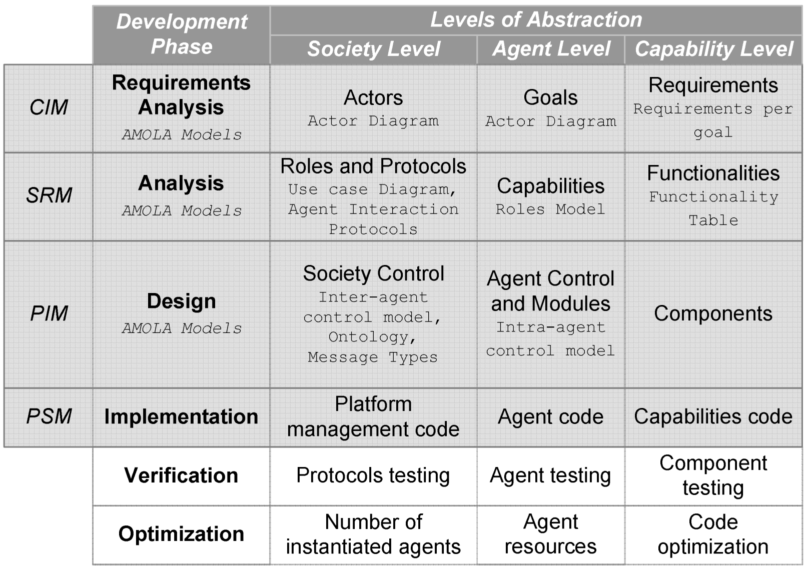
Figure 1: ASEME Process Overview

MDE is the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. It is compatible with the recently emerging Model Driven Architecture (MDA, [10]) paradigm of the Object Management Group (OMG). MDA's strong point is that it strives for portability, interoperability and reusability, three non-functional requirements that are very important for modern systems design. MDA defines three models:

- A computation independent model (CIM) is a view of a system that does not show details of the structure of systems. It uses a vocabulary that is familiar to the practitioners of the domain in question as it is used for system specification
- A platform independent model (PIM) is a view of a system that on one hand provides a specific technical specification of the system, but on the other hand exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms
- A platform specific model (PSM) is a view of a system combining the specifications in the PIM with the details that specify how that system uses a particular type of platform

In ASEME the CIM, PIM and PSM are the models outputted by the requirements analysis, design and implementation phases respectively. We have inserted another model as the output of the analysis phase, the System Roles Model (SRM). Each of these models is produced by applying simple transformation rules to the previous phase model and this transformation is traceable, that is it can be reverse engineered.

We define three levels of abstraction in each phase. The first is the *societal level*. There, the whole multi-agent system functionality is modeled. Then, in the *agent level*, we model (zoom in) each part of the society, the agent. Finally, we focus in the details that compose each of the agent's parts in the *capability level*. We define the concept of *capability* as the ability of an agent to achieve specific tasks that require the use of one or more *functionalities*. The latter refer to the technical solution(s) to a given class of tasks. Moreover, capabilities are decomposed to simple *activities*, each of which corresponds to exactly one functionality. Thus, an activity corresponds to the instantiation of a specific technique for dealing with a particular task. ASEME is mainly concerned with the first two abstraction levels assuming that development in the capability level can be achieved using classical (or even technology-specific) software engineering techniques. In Figure 2, we present the ASEME phases, the different levels of abstraction and the models related to each one of them.



	Development Phase	Levels of Abstraction		
		Society Level	Agent Level	Capability Level
CIM	**Requirements Analysis** *AMOLA Models*	Actors `Actor Diagram`	Goals `Actor Diagram`	Requirements `Requirements per goal`
SRM	**Analysis** *AMOLA Models*	Roles and Protocols `Use case Diagram, Agent Interaction Protocols`	Capabilities `Roles Model`	Functionalities `Functionality Table`
PIM	**Design** *AMOLA Models*	Society Control `Inter-agent control model, Ontology, Message Types`	Agent Control and Modules `Intra-agent control model`	Components
PSM	**Implementation**	Platform management code	Agent code	Capabilities code
	Verification	Protocols testing	Agent testing	Component testing
	Optimization	Number of instantiated agents	Agent resources	Code optimization

Figure 2: ASEME phases and their products

3. REQUIREMENTS ANALYSIS PHASE

In Figure 3 we present the requirements analysis phase in SPEM notation. The three levels of abstraction are represented by the three activities. In the society level we define the actors and their goals that depend on other actors; in the agent level we define the individual goals of each actor and in the capability level we assign specific requirements to each one of these goals that can be functional and non-functional. The output of the requirements analysis phase is the CIM model that is composed by the Actor diagram, which is similar to the TROPOS [3] actor diagram (thus, a Tropos requirements analysis method fragment could be combined with minimal effort with ASEME), containing the actors and their goals and the "Requirements per goal" table that associates requirements to each goal. Method fragments are reusable methodological parts that can be used by engineers in order to produce a new design process for a specific situation (see [4] for details).

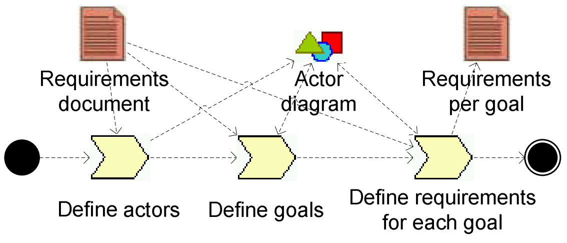


Figure 3: The ASEME Requirements Analysis Phase

For the ASEME process demonstration we show how to develop a meetings management system, as this paradigm has been widely used in the past for demonstrating the use of methodologies (e.g. for the Prometheus and MAS-CommonKADS methodologies in [8]). This system's requirements, in brief, are to support the meetings arrangement process. The user needs to be assisted in managing his meetings by a personal assistant. The latter manages the user's schedule and services the user. The meetings organization process is managed by the secretariat to which the users submit their requests to schedule a new meeting or change the date of an existing one. The secretariat contacts the users' assistants whenever she needs to negotiate a meeting date.

The actors involved are the user and the assistant that helps him to manage his meetings. The latter is adaptable to a specific user, thus is modeled as a personal assistant. Moreover, there is the secretariat role that is represented by the meetings manager actor. The reader can see the CIM model in Figure 4, where the actors are represented by circles and the goals by rounded rectangles. The goal of the user to manage his meetings is dependent on the personal assistant. In the agent level we define individual goals; one such is the adaptation to user needs for the personal assistant, named "learn user habits". In the capability level we define the functional and non-functional requirements for each goal in free text.

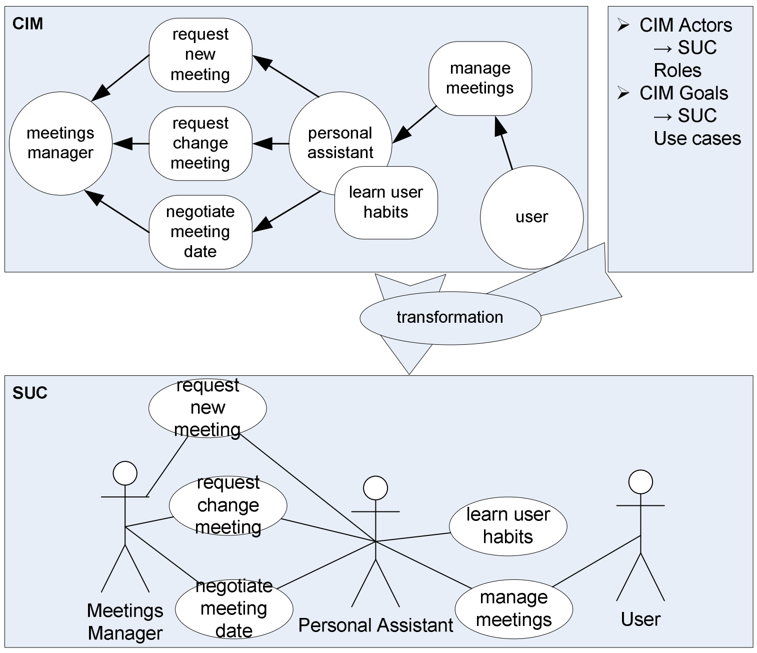


Figure 4: The CIM2SUC transformation

4. ANALYSIS PHASE

The ASEME analysis phase is presented in Figure 5. The first activity transforms the Actor diagram (CIM model) to an AMOLA use case diagram (SUC model). The model transformation process (CIM2SUC) is straightforward and is graphically presented in Figure 4. The actors are transformed to roles and the goals to use cases. The SUC model is an intermediate model, used only within the analysis phase, aiming to facilitate the creation of the SRM model, which is the output of the analysis phase.

The next activity refines the use cases in order to produce a rich SUC model. The SUC model is an extended UML use case diagram that allows actors to be defined within the System box. In this activity we define the human and artificial roles and decompose the general use cases to simple – task-specific use cases using the *include* relationship. UML suggests the use of sequence diagrams and activity diagrams for refining use cases. Another possible approach is the use of the MAS-CommonKADS task model method fragment [8] or any method for decomposing a general task to specific ones. Note that these models are optional, a modeler can work directly on the use case diagram. The SUC model in Figure 6 refines the "request new meeting" use case for the "personal assistant role" and assignes cardinality to the "negotiate meeting date" use case showing that two or more "personal assistant" roles are involved.

The next ASEME activity is about defining the *Agent Interaction Protocols* (or simply protocols). Protocols (in the society level) originate from use cases that connect two artificial agent roles (e.g. the "request new meeting" is a protocol use case). A protocol definition includes the protocol name (named after the use case), the participating roles (named after the involved roles), the rules for engaging and the outcomes for each party (both in free text based on the "Requirements per goal" product) and the process for each participant. For example, in Figure 4, the "request new meeting" use case is refined for the "Personal Assistant" actor to the simple tasks "send new request" and "receive new results". In the SUC model in Figure 6 we have five general use cases that correspond to the goals of the requirements analysis phase and the two aforementioned simple use cases. The included (simple) use cases, which represent simple tasks, are transformed to activities in the agent interaction protocol *Process* field. In Table 1 we present the "Negotiate meeting date" agent interaction protocol definition. It defines two roles, i.e. Personal Assistant and Meetings Manager, the rules for engaging (why would they participate in this protocol), the outcomes that they should expect in successful completion and the process that they would follow in the form of a liveness formula. The liveness formula is a process model that describes the dynamic behavior of the role inside the protocol. It connects all the role's activities using the Gaia operators (see Table 2, or [14] for more details). The liveness formula defines the dynamic aspect of the role, that is which activities execute sequentially, which concurrently and which are repeating.

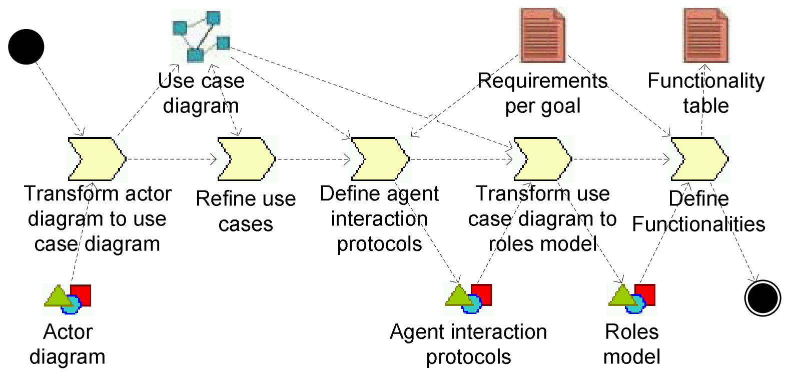

Use case diagram
Requirements per goal
Functionality table
Transform actor diagram to use case diagram
Refine use cases
Define agent interaction protocols
Transform use case diagram to roles model
Define Functionalities
Actor diagram
Agent interaction protocols
Roles model

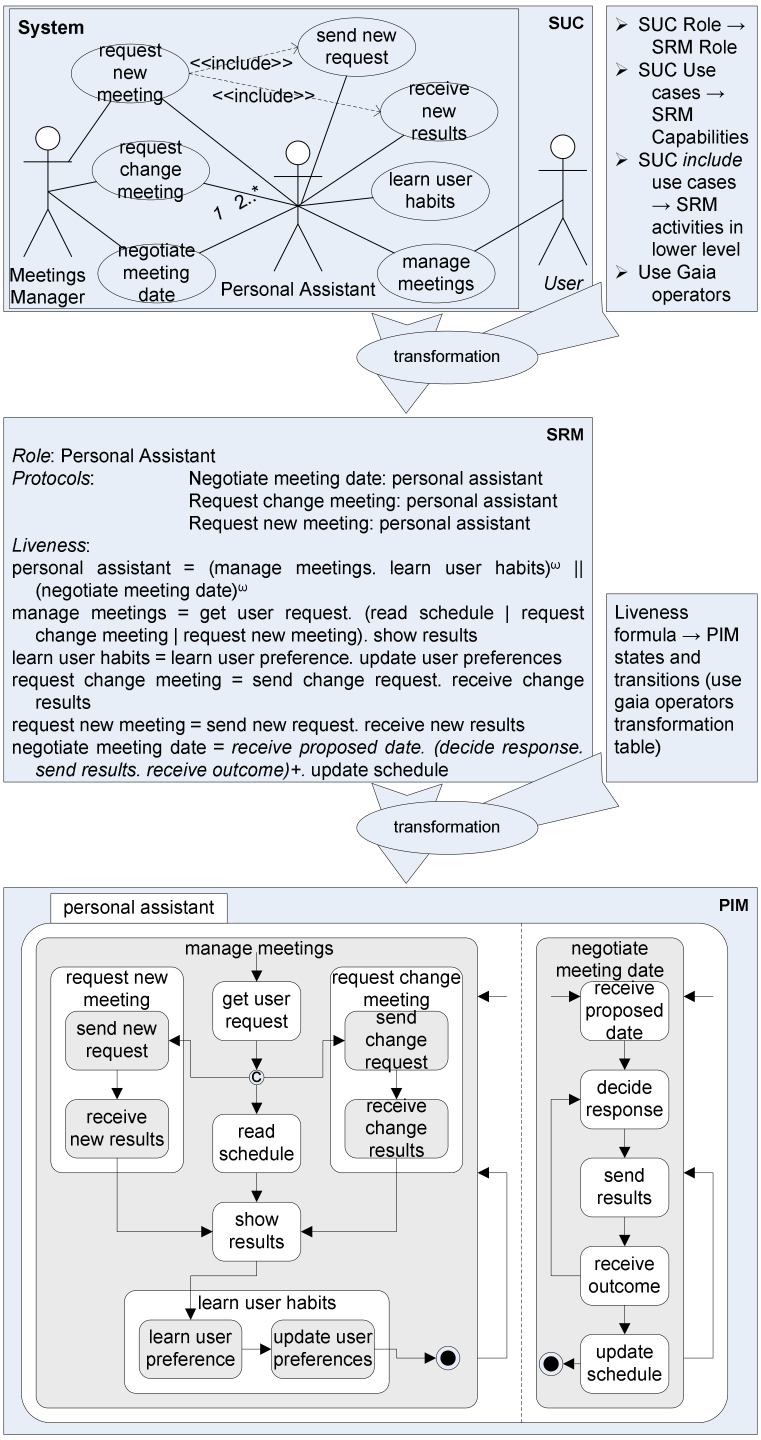
Figure 5: The ASEME Analysis Phase

Table 1. Agent interaction protocol

Negotiate meeting date		
Participants	*Personal Assistant*	*Meetings Manager*
Rules for engaging	He needs to create or participate to a meeting	He has a meeting with more than one participants that has no date assigned to it
Outcomes	He has scheduled participation to a meeting	He has arranged a meeting that met all the participants needs
Process	negotiate meeting date = receive proposed date. (decide response. send results. receive outcome)+	negotiate meeting date = (decide on date. send proposed date. receive results)+. send fixed date.

After refining the use cases we transform the system use case model to the system roles model (SRM), which is mainly inspired by the Gaia roles model. Figure 6 shows the transformation (SUC2SRM) process. We create a role model for each actor in the use case diagram. Each of the original use cases (the ones in Figure 4) is transformed to a capability. The capabilities are placed in the liveness model of the role in the form of liveness formulas. Actually, through this transformation, the analyst arranges the different capabilities in the liveness formulas in the right sequence and connects them with the appropriate Gaia operators, so that the formulas depict the process model of each role. The role name appears in the left hand side of the first formula (root formula). The use cases included by each capability are inserted as activities in a lower level formula (that has the capability on the left hand side).

The last activity of this ASEME phase defines the functionality table where the analyst associates each activity participating in the liveness formulas of the SRM to the technology or tool (functionality) that it will use (see Figure 7). The reader should note that a special capability not included in the use–case diagram named communicate appears. This capability includes the “send message” and “receive message” activities and is shared by all agents and is defined separately because its implementation is relative to the functionality provided by the agent development platform, e.g., in our example, JADE (the Java Agent Development Environment is an open source framework that adheres to the FIPA standards for agents development).



Role: Personal Assistant
Protocols: Negotiate meeting date: personal assistant
Request change meeting: personal assistant
Request new meeting: 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 6: The SUC2SRM and SRM2PIM transformations

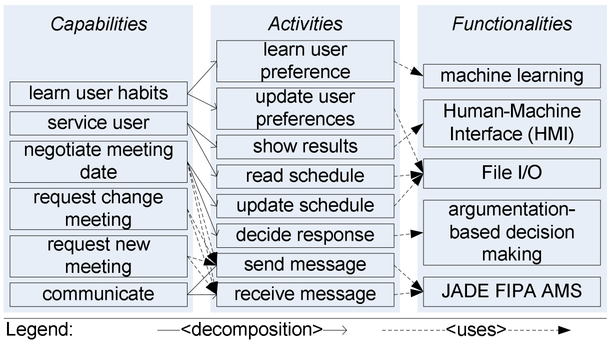
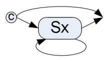
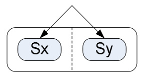
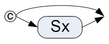
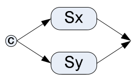
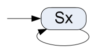
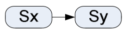
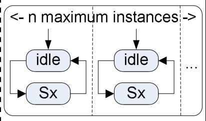
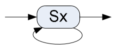


Figure 7. Capabilities, activities and functionalities

Table 2. Templates of extended Gaia operators (Op.) for Statechart generation

Op.	*Template*	*Op.*	*Template*
x^*		$x \parallel y$	
$[x]$		$x \mid y$	
x^ω			
$x.\ y$		$\lvert x^\omega \rvert^n$	
$x+$			

5. DESIGN PHASE

The ASEME design phase is presented in Figure 8. The three work definitions reflect the three different levels of abstraction in the software development. In the society level we have the inter-agent control model, in the agent level the intra-agent control model and in the capability level the models of the different components that will be used by the agent. Thus, each agent is considered to be part of a multi-agent system. The agents communicate using interaction protocols that are described by the *inter-agent control*, which defines the participating roles and their responsibilities in the form of tasks. The agents implement the roles that they can assume through their capabilities. The capabilities are the modules that are integrated using the *intra-agent control* concept.

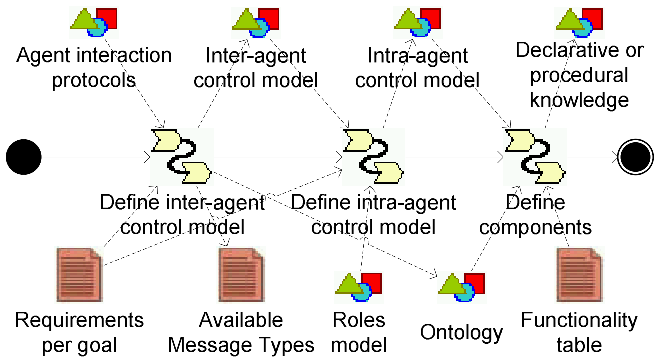

Agent interaction protocols
Inter-agent control model
Intra-agent control model
Declarative or procedural knowledge
Define inter-agent control model
Define intra-agent control model
Define components
Requirements per goal
Available Message Types
Roles model
Ontology
Functionality table

Figure 8: The ASEME Design Phase

AMOLA uses the language of statecharts [7] for defining the PIM model (the output of the design phase). 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. The *scope* of a transition is the lowest OR-state in the hierarchy of states that is a proper common ancestor of the source and target states of the transition. Multiple concurrently active statecharts are considered to be orthogonal components at the highest level of a single statechart.

The first work definition (define inter-agent control model) is detailed in Figure 9. The first activity uses the "Gaia operators transformation templates" for transforming the process part of the agent interaction protocol model to a statechart, namely the inter-agent control model. A state diagram is generated by an initial state named after the protocol. Then, all participating roles define AND sub-states. The right hand side of the liveness formula of each role is transformed to several OR-states within each AND-state by interpreting the Gaia operators in the way described in Table 2. The reader should note that we have defined a new operator, the $|x^{\omega}|^{n}$, with which we can define an activity that can be concurrently instantiated and executed more than one times (*n* times).

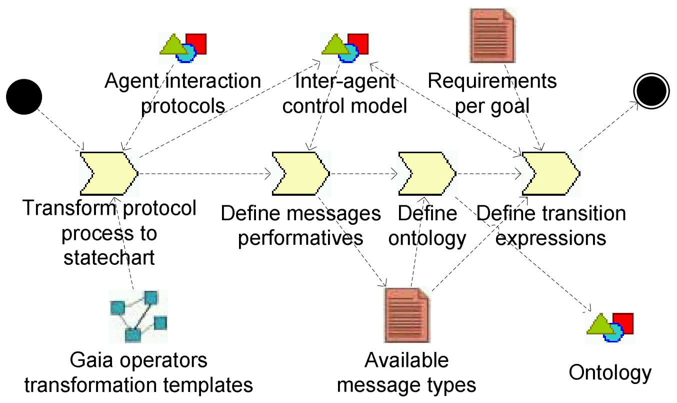

Agent interaction protocols
Inter-agent control model
Requirements per goal
Transform protocol process to statechart
Define messages performatives
Define ontology
Define transition expressions
Gaia operators transformation templates
Available message types
Ontology

Figure 9: The "Define Inter-agent Control Model" work definition

Then, in the next activity, we define the message performatives allowed within the protocol. A message is expressed by *P(x,y,c)* where *P* is the *performative*, *x* is the sender, *y* the receiver and *c* the message body. In our example, we define that $P \in \{ accept, propose, reject, inform\}$. The items that the designer must define at the next activity are the data structures used for defining the protocol parameters (also referred to as the ontology), the timers (defined as in [6]) and the message contents (also part of the ontology). Finally, in the last activity the transition expressions are defined. The resulting statechart for the *negotiate meeting date protocol* (starting from the analysis model presented in Table 1) is depicted in Figure 10. At this point the reader should notice the differences with Moore's proposal ([11]). All states represent activities, while in his work they just represent a point in time where a condition is true (like in finite state machines).

The preconditions of the agent interaction protocol become the conditions of a source-less transition that targets the first state of the protocol for each role. The transition will be connected to a source state in the intra-agent control model. The preconditions for the meetings manager role, which have been described in free text in the agent interaction protocol model (see *rules for engaging* in Table 1), concern the arrangement of a meeting with a list of more than one participants. They become the conditions for the *decide on date* state and can be formally expressed by the atoms hasToArrange(m, meeting), participantsListOf(meeting, list) and hasLength(list,n). The variables m, meeting, list and d that appear in these atoms refer to the *meetings manager* role, to the meeting that needs to be arranged, to the list of participants in that meeting and to the date of the meeting respectively. Variables p_i, i=1,…,n refer to the *n* personal assistants that form the *list* of participants to the protocol instance. The only variable that is assigned a value while the protocol is executing is *d* and that is formally expressed by the [assignedValue(d)] condition of the transition with source the *decide on date* activity. These variables can be accessed by all the protocol's states, since their scope is the same with the scope of the transition in which they appear, i.e. the OR-state that contains both the source and target of the transition, in this case the *negotiate meeting protocol* state. The goal for both (i.e. manager and participant) roles is that a date has been assigned to the meeting (from *outcomes* field in Table 1) and it is formally expressed by the atom arrangedMeetingDate(meeting, d).

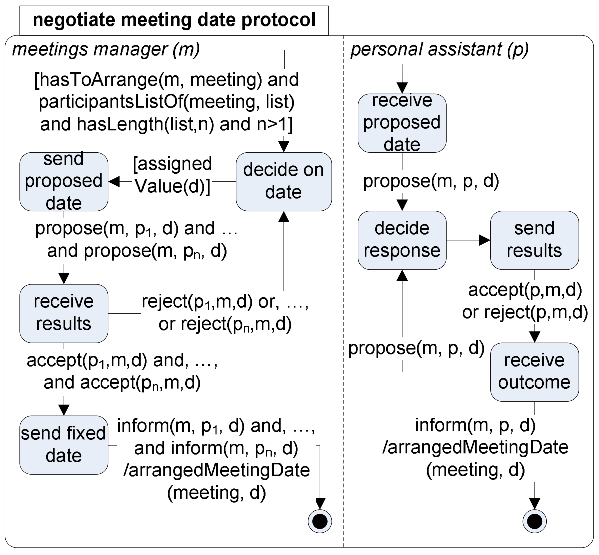

negotiate meeting date protocol
meetings manager (m)
personal assistant (p)
[hasToArrange(m, meeting) and participantsListOf(meeting, list) and hasLength(list,n) and n>1]
send proposed date
[assigned Value(d)]
decide on date
propose(m, p_1, d) and … and propose(m, p_n, d)
receive results
reject(p_1,m,d) or, …, or reject(p_n,m,d)
accept(p_1,m,d) and, …, and accept(p_n,m,d)
send fixed date
inform(m, p_1, d) and, …, and inform(m, p_n, d) /arrangedMeetingDate (meeting, d)
receive proposed date
propose(m, p, d)
decide response
send results
accept(p,m,d) or reject(p,m,d)
propose(m, p, d)
receive outcome
inform(m, p, d) /arrangedMeetingDate (meeting, d)

Figure 10. The inter-agent control model

The second work definition of the ASEME Design phase, i.e. "Define intra-agent control model", is detailed in Figure 11. The intra-agent control is created by transforming the liveness model of the role to a state diagram. We achieve that, by interpreting the Gaia operators in the way described in Table 2. Initially, the statechart has only one state named after the left-hand side of the first liveness formula of the role model. Then, this state acquires substates. The latter are constructed by reading the right hand side of the liveness formula from left to right, and substituting the operator found there with the relevant template in Table 2. If one of the states is further refined in a next formula, then new substates are defined for it in a recursive way. The transformation (SRM2PIM) process is shown graphically in Figure 6. The next activity imports the transition expressions from the inter-agent control for the part of the statechart containing a protocol (i.e. the part of the statechart produced from the formula whose left hand side is a protocol capability). The last activity of this work definition is about enriching the rest of the statechart with transition expressions.

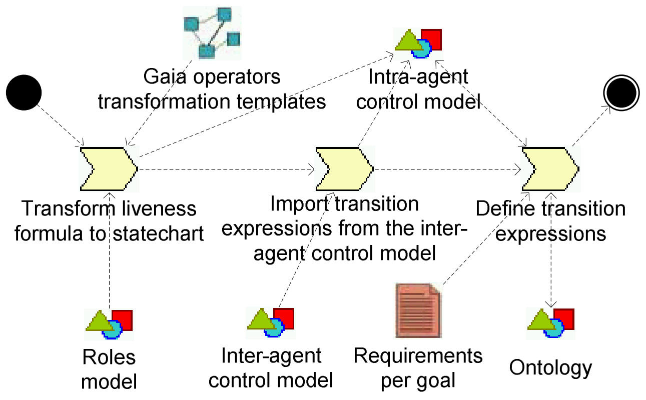

Gaia operators transformation templates
Intra-agent control model
Transform liveness formula to statechart
Import transition expressions from the inter-agent control model
Define transition expressions
Roles model
Inter-agent control model
Requirements per goal
Ontology

Figure 11: The "Define Intra-agent Control Model" work definition

The final work definition of the ASEME design phase (i.e. "Define components") is about designing the activities that are executed in each state. The input needed is the "Functionality table" to indicate the technology (e.g. which library to import and which programming language to use), the "Ontology" to show the data structures that will be used by this activity and the "Intra-agent control model" that lists all the activities as states. The output depends on the technology used for each activity and can be declarative or procedural knowledge (or both).

6. IMPLEMENTATION PHASE

The implementation phase's goal is to transform the platform independent model to a platform dependent model, i.e. an implementation model. This phase can have different instantiations according to the implementation platform. In Figure 12 we present the ASEME implementation phase for the JADE platform. The JADE implementation phase details a transformation process of the PIM to PSM. The transformation (PIM2PSM) is shown graphically for our working example in Figure 13. The PIM shown in the figure is part of the Personal assistant agent definition. It shows the "negotiate meeting date" part of the statechart shown in Figure 6 having added the states transition expressions. We show the part of the program whose production can be automated in the PSM model of Figure 13 (showing the automatically generated agent code, a complex behavior and a simple behaviour).

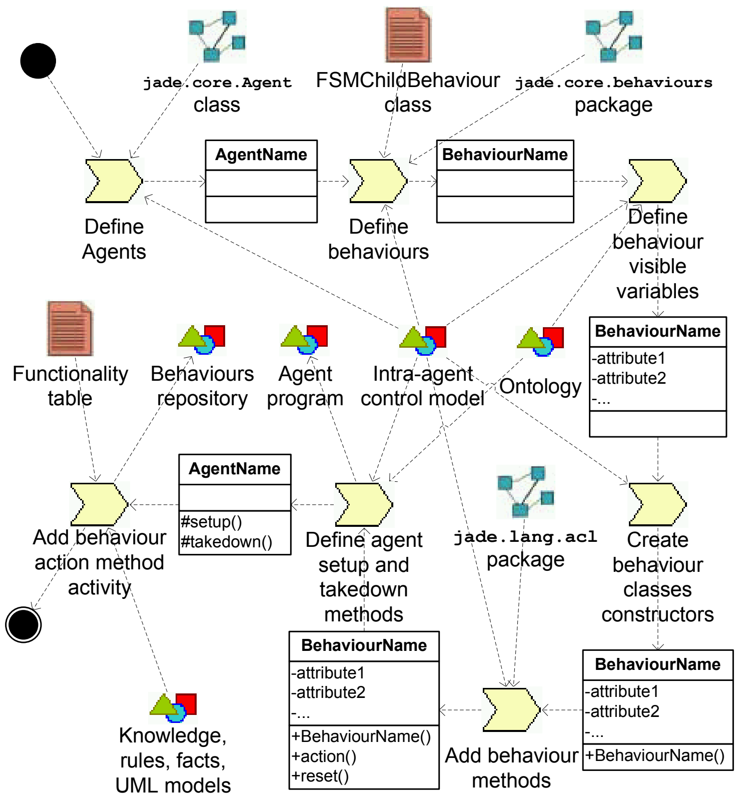
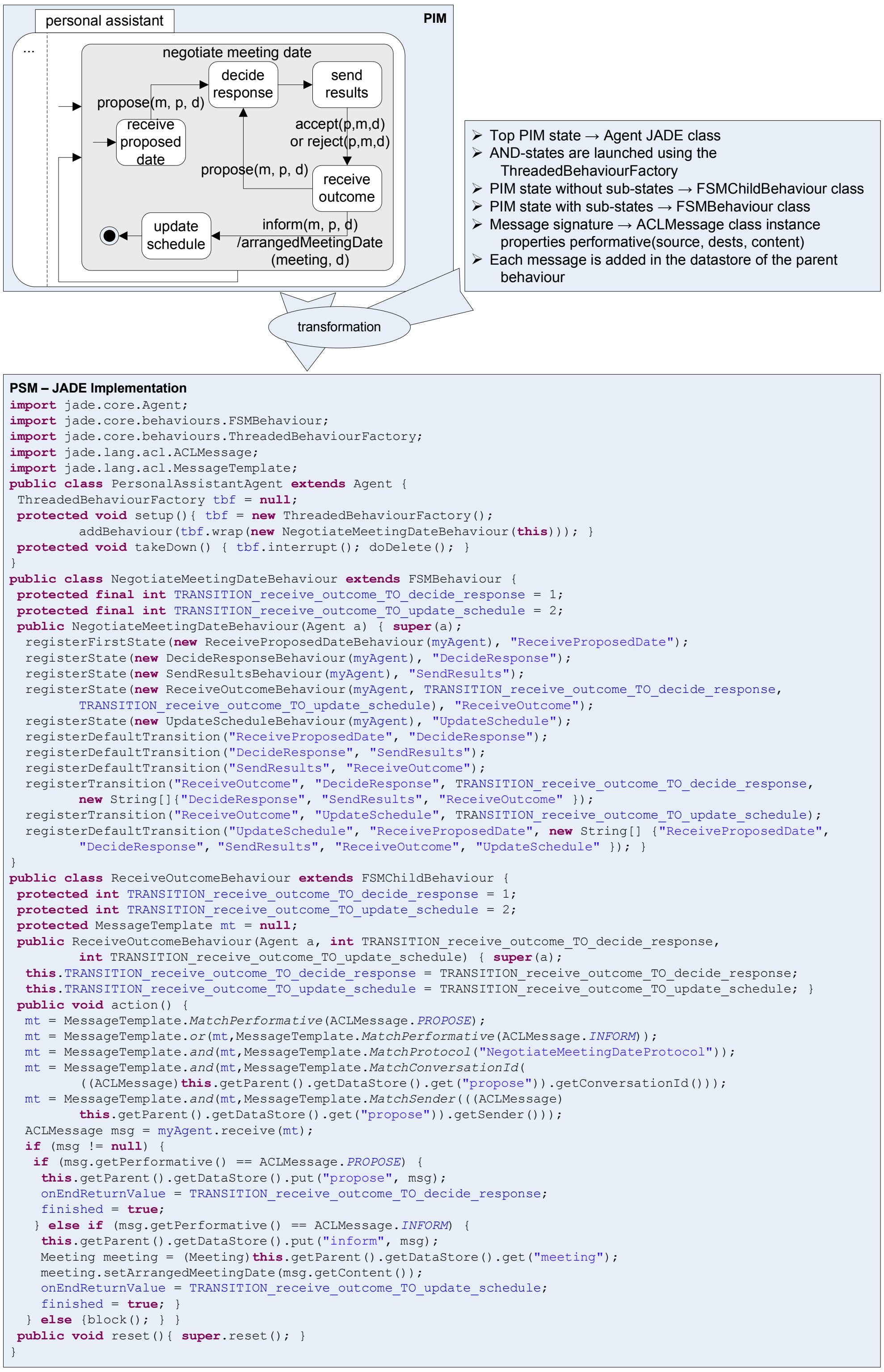

jade.core.Agent class
FSMChildBehaviour class
jade.core.behaviours package
AgentName
BehaviourName
Define Agents
Define behaviours
Define behaviour visible variables
BehaviourName -attribute1 -attribute2 -...
Functionality table
Behaviours repository
Agent program
Intra-agent control model
Ontology
AgentName #setup() #takedown()
Add behaviour action method activity
Define agent setup and takedown methods
jade.lang.acl package
Create behaviour classes constructors
BehaviourName -attribute1 -attribute2 -... +BehaviourName() +action() +reset()
Add behaviour methods
BehaviourName -attribute1 -attribute2 -... +BehaviourName()
Knowledge, rules, facts, UML models

Figure 12: The ASEME JADE implementation phase

Following the process in Figure 12, we firstly define the agents, naming them after the top level state of each statechart in the PIM model. In this case we define the *PersonalAssistantAgent* extending the JADE *Agent* class. Then, we define the behaviours. Those states that have sub-states extend the *FSMBehaviour* JADE class (one such is presented in Figure 13, the *NegotiateMeetingDateBehaviour*), while the basic states extend the *FSMChildBehaviour* class (one such is presented in Figure 13, the *ReceiveOutcomeBehaviour*), originally proposed in [12] and extended by this work (see Figure 14) so that it implements the *reset()* method allowing the behaviour to be started again.

In the next activity we define the behaviour's attributes. Those are the possible transitions named after the convention *TRANSITION_<state exited>_TO_<state entered>*. The *FSMBehaviour* descendant class must define all the transitions between its sub-states, while the *FSMChildBehaviour* descendants need only define the transitions that have them as their source. Then, the message receiving behaviours (those whose name starts with the word "receive") instantiate a *MessageTemplate* type variable for defining the attributes of the expected message.

The next activity defines the behaviours' constructors. The *FSMChildBehaviour* descendants receive the *Agent* class and the possible transition integers as inputs and initialize their variables. The *FSMBehaviour* descendants are more complex as they define the states and the transitions between them. When a transition will cause some states to be executed again they need to be reset. Then, we add the behavior methods, specifically the *reset* and *action* methods. In *reset* there is just a call to the super class method. In the *action* method we define two things. Firstly, if the possible transitions exiting the state are more than one we add an *if-else* statement catering for all different possibilities. The transition events and conditions become the *if-else* conditions. The transition actions become part of the enclosed statements.



- Top PIM state → Agent JADE class
- AND-states are launched using the ThreadedBehaviourFactory
- PIM state without sub-states → FSMChildBehaviour class
- PIM state with sub-states → FSMBehaviour class
- Message signature → ACLMessage class instance properties performative(source, dests, content)
- Each message is added in the datastore of the parent behaviour

PSM – JADE Implementation

```
import jade.core.Agent;
import jade.core.behaviours.FSMBehaviour;
import jade.core.behaviours.ThreadedBehaviourFactory;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
public class PersonalAssistantAgent extends Agent {
 ThreadedBehaviourFactory tbf = null;
 protected void setup(){ tbf = new ThreadedBehaviourFactory();
        addBehaviour(tbf.wrap(new NegotiateMeetingDateBehaviour(this))); }
 protected void takeDown() { tbf.interrupt(); doDelete(); }
}
public class NegotiateMeetingDateBehaviour extends FSMBehaviour {
 protected final int TRANSITION_receive_outcome_TO_decide_response = 1;
 protected final int TRANSITION_receive_outcome_TO_update_schedule = 2;
 public NegotiateMeetingDateBehaviour(Agent a) { super(a);
  registerFirstState(new ReceiveProposedDateBehaviour(myAgent), "ReceiveProposedDate");
  registerState(new DecideResponseBehaviour(myAgent), "DecideResponse");
  registerState(new SendResultsBehaviour(myAgent), "SendResults");
  registerState(new ReceiveOutcomeBehaviour(myAgent, TRANSITION_receive_outcome_TO_decide_response,
        TRANSITION_receive_outcome_TO_update_schedule), "ReceiveOutcome");
  registerState(new UpdateScheduleBehaviour(myAgent), "UpdateSchedule");
  registerDefaultTransition("ReceiveProposedDate", "DecideResponse");
  registerDefaultTransition("DecideResponse", "SendResults");
  registerDefaultTransition("SendResults", "ReceiveOutcome");
  registerTransition("ReceiveOutcome", "DecideResponse", TRANSITION_receive_outcome_TO_decide_response,
        new String[]{"DecideResponse", "SendResults", "ReceiveOutcome" });
  registerTransition("ReceiveOutcome", "UpdateSchedule", TRANSITION_receive_outcome_TO_update_schedule);
  registerDefaultTransition("UpdateSchedule", "ReceiveProposedDate", new String[] {"ReceiveProposedDate",
        "DecideResponse", "SendResults", "ReceiveOutcome", "UpdateSchedule" }); }
}
public class ReceiveOutcomeBehaviour extends FSMChildBehaviour {
 protected int TRANSITION_receive_outcome_TO_decide_response = 1;
 protected int TRANSITION_receive_outcome_TO_update_schedule = 2;
 protected MessageTemplate mt = null;
 public ReceiveOutcomeBehaviour(Agent a, int TRANSITION_receive_outcome_TO_decide_response,
        int TRANSITION_receive_outcome_TO_update_schedule) { super(a);
  this.TRANSITION_receive_outcome_TO_decide_response = TRANSITION_receive_outcome_TO_decide_response;
  this.TRANSITION_receive_outcome_TO_update_schedule = TRANSITION_receive_outcome_TO_update_schedule; }
 public void action() {
  mt = MessageTemplate.MatchPerformative(ACLMessage.PROPOSE);
  mt = MessageTemplate.or(mt,MessageTemplate.MatchPerformative(ACLMessage.INFORM));
  mt = MessageTemplate.and(mt,MessageTemplate.MatchProtocol("NegotiateMeetingDateProtocol"));
  mt = MessageTemplate.and(mt,MessageTemplate.MatchConversationId(
        ((ACLMessage)this.getParent().getDataStore().get("propose")).getConversationId()));
  mt = MessageTemplate.and(mt,MessageTemplate.MatchSender(((ACLMessage)
        this.getParent().getDataStore().get("propose")).getSender()));
  ACLMessage msg = myAgent.receive(mt);
  if (msg != null) {
   if (msg.getPerformative() == ACLMessage.PROPOSE) {
    this.getParent().getDataStore().put("propose", msg);
    onEndReturnValue = TRANSITION_receive_outcome_TO_decide_response;
    finished = true;
   } else if (msg.getPerformative() == ACLMessage.INFORM) {
    this.getParent().getDataStore().put("inform", msg);
    Meeting meeting = (Meeting)this.getParent().getDataStore().get("meeting");
    meeting.setArrangedMeetingDate(msg.getContent());
    onEndReturnValue = TRANSITION_receive_outcome_TO_update_schedule;
    finished = true; }
  } else {block(); } }
 public void reset(){ super.reset(); }
}
```

Figure 13: The PIM2PSM transformation

All inter-agent messages and variables appearing on state transition expressions are stored in the data store of the state that is the *scope* of the transition. Thus, the agent protocol related behaviours can access the message history based on the performative of the message (i.e. as soon as a new *propose* message arrives, the previous one is deleted from memory). The atoms appearing on transition expressions are handled as properties of objects that must have been defined in the ontology. The *arrangedMeetingDate(meeting, d)*, for instance, implies that the object *meeting* has a property *arrangedMeetingDate* whose value is *d*.

```
import jade.core.Agent;
import jade.core.behaviours.SimpleBehaviour;
public class FSMChildBehaviour extends SimpleBehaviour {
  protected boolean finished = false;
  protected int onEndReturnValue;
  public FSMChildBehaviour(Agent a) { super(a); }
  public void action() { }
  public boolean done() { return finished; }
  public int onEnd(){ return onEndReturnValue; }
  public void reset(){
    finished = false;
    onEndReturnValue = 0;
  }
}
```

Figure 14: The FSMChildBehaviour class

The next step finalises the *Agent* descendant classes by adding code in their *setup* and *takedown* methods. If there are orthogonal components in the statechart the *ThreadedBehaviourFactory* class is used for adding the relevant behaviours as independent threads. If there are no orthogonal components then the behaviours are added in the agent's execution queue normally as if the agent class was a super-state behavior.

Finally, the last activity of the implementation phase is about defining the details of the action method of each behavior. This refers to the capability level of abstraction and the implementation must reflect the designs for each component.

7. VERIFICATION AND OPTIMIZATION PHASES

During the Verification Phase the system's functionality is verified in comparison to its requirements. The verification phase can be carried out in parallel for the three different abstraction levels; however, the best approach is sequential: the software components are tested for the successful implementation of algorithms, the agents for the successful implementation of capabilities and the MAS for its overall correct operation.

The Optimization Phase is concerned with the optimization of the system. The algorithms in the capability level can be optimized in execution time or resource consumption. In the agent level, the number of concurrently executing capabilities can be optimized, i.e. how many instances of the "negotiate meeting date" capability should the personal assistant be executing concurrently to meet his requirements. Finally, in the societal level, the number of agents that will be instantiated and the strategy for instantiating or destroying agents while the system is in operation is optimized.

The interesting part of these phases is that they can be present in all development iterations since the AMOLA models of the design phase are statecharts. Moreover, the latter can be transformed to process models, since all states represent an activity that is executed by a specific agent role resource. Both statecharts and process models are supported by commercial and open source tools (e.g. STATEMATE [7] for statecharts and SIMPROCESS [1] for process models) that allow for simulation and thus, can greatly aid the verification and optimization phases. SIMPROCESS can also be used for optimizing process models. Using, for instance, the open source Intalio tool (see http://bpms.intalio.com/ for more details) we transformed the statechart in Figure 10 to the business process diagram presented in Figure 15. In Intalio, the message reception activities are represented as circles with an envelope inside. The circle with the square that starts the *Meetings Manager* process resembles the need for some conditions to be satisfied so that the process starts. Finally, a diamond with an X resembles an exclusive choice (either one or the other). The states of the inter-agent control model (see Figure 10) are transformed to processes in the business process diagram. The transitions between states are transferred as they are except in the case of transitions that are enabled by an inter-agent message event. The latter are becoming transitions from the sending process to the receiving process. The conditions on the transitions can be represented in Intalio and the process can be directly simulated, or even deployed to executable code (only if the activities can be implemented with the available Intalio tools). The *meetings manager* role interacts with all the *personal assistant* roles in the way presented in Figure 15.

Here we should also note that the inter-agent control model does not impose a specific way for interpreting the exchanged messages or a technology for exchanging them. These issues are defined by the developers according to the platform that they will use for deploying their system and their expertise. For example, in FLBC (an agent communication language, see [11]) the effects of a *request* message are linked to the beliefs of the sender which may not be the case in another communication language with different semantics. Thus, a procedural agent might not have a model of beliefs in contrast with a BDI (i.e. belief-desire-intention, see [5]) agent.

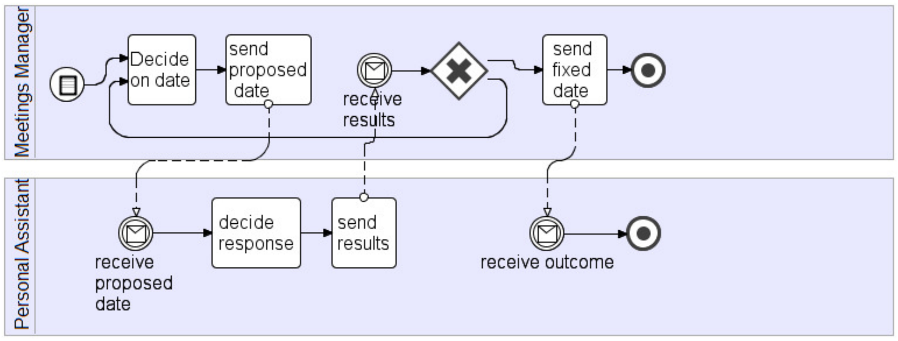


Figure 15. The Negotiate Meeting Protocol Process

8. RELATED WORK AND CONCLUSION

All, in all, in this paper, we presented the ASEME process a Model-driven Engineering methodology for Agent Oriented Software Engineering. We showed how the system modeler can proceed from capturing requirements to system implementation using models transformation throughout the development phases. This is the main contribution of this paper to the state of the art. We are currently reviewing the two most popular model transformation languages (ATL and QVT, see [9]) in order to select one for developing a graphical tool for aiding the developer throughout the different phases of the process. The main advantages of ASEME over existing methodologies are:

- The intra-agent control, whose novelty is to allow the modeling of interactions between the different capabilities of an agent. For this purpose we use statecharts and their orthogonality concept in an original way
- The inter-agent control that corresponds to the agent interaction protocol. It is realized through the use of statecharts like in the intra-agent control, thus simplifying the designer's integration task by using the same formalism
- There is a straightforward transformation process between the models of the analysis phase to those of the design phase and then to an implementation platform
- It defines three abstraction levels (the society, agent and capability levels), thus supporting the development of large-scale systems
- The models of AMOLA can lead to agent development without imposing constraints on how the mental model of the agent will be defined
- We define the terms of capability and functionality that have been used with different meanings in the past in order to provide new concepts for modeling agent-based systems with relation to previous methodologies like, e.g. for object-oriented development.

9. REFERENCES

[1] April, J., Better, M., Glover, F., Kelly, J. and Laguna M. Enhancing Business Process Management With Simulation Optimization. In Proceedings of the 38th conference on Winter simulation (Monterey, California, USA, December 3-6, 2006). WSC '06, 642–649.

[2] Beydeda, S., Book, M., Gruhn, V. 2005. Model-Driven Software Development. Springer.

[3] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. and Perini, A. 2004. TROPOS: An Agent-Oriented Software Development Methodology. J. Auton. Agent. Multi-Ag. 8, 3 (October 2004), 203-236.

[4] Cossentino, M., Gaglio, S., Garro, A. and Seidita, V. 2007. Method fragments for agent design methodologies: from standardisation to research. Int. J. of Agent-Oriented Software Engineering, 1, 1 (April 2007), 91-121.

[5] Dastani, M., van Riemsdijk, M.B. and Meyer, J.-J.Ch. Programming multi-agent systems in 3APL. In 2005 Multiagent Systems, Artificial Societies, and Simulated Organizations, 15. Springer-Verlag, 39-67.

[6] Deloach, S.A., Wood, M.F. and Sparkman, C.H. 2001. Multiagent Systems Engineering. Int. J. Softw. Eng. Know. 11, 3 (June 2001), 231-258

[7] Harel, D. and Naamad, A. 1996. The STATEMATE Semantics of Statecharts. ACM T. Softw. Eng. Meth. 5, 4 (October 1996), 293-333.

[8] Henderson-Sellers B. and Giorgini P. 2005 Agent-Oriented Methodologies. Idea Group Publishing.

[9] Jouault, F. and Kurtev, I. On the Architectural Alignment of ATL and QVT. In Proceedings of the 2006 ACM Symposium on Applied Computing (Dijon, France, April 23-27, 2006). SAC 06. ACM Press, 1188-1195.

[10] Kleppe, A., Warmer, S. and Bast, W. 2003 MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley.

[11] Moore, S.A. On conversation policies and the need for exceptions. In 2000 Issues in Agent Communication, Lecture Notes in Artificial Intelligence 1916/2000. Springer, 144–159.

[12] Moraitis, P. and Spanoudakis N. 2006. The Gaia2JADE Process for Multi-Agent Systems Development. J. Appl. Artif. Intell. 20, 2-4 (February-April 2006), 251-273.

[13] Unified Modeling Language Specification. ISO 19501:2005

[14] Zambonelli, F., Jennings, N.R. and Wooldridge, M. 2003. Developing multiagent systems: the Gaia Methodology. ACM T. Softw. Eng. Meth. 12, 3 (July 2003), 317-370.