The Agent Systems Methodology (ASEME): A Preliminary Report

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
{nikos, pavlos}@math-info.univ-paris5.fr

Abstract. This paper presents the Agent SystEms MEthodology (ASEME) for developing multi-agent systems. ASEME proposes a modular agent design approach and introduces the concept of intra-agent control. The latter defines the agent's lifecycle by coordinating the different modules that implement his capabilities. The modeling of the intra-agent control is based on statecharts. The analysis phase builds on the concepts of capability and functionality. ASEME covers all the phases of the software development process and deals with both the individual and societal aspect of the agents. However, in this paper we focus in presenting only the individual agent development process.

Keywords: Agent Oriented Software Engineering, Multi-agent Systems Development, Intra-agent Control, Agent Capability, Agent Functionality, Agent Architecture

1 Introduction

Agent oriented development emerges as the modern way to create software. Its main advantage – as referred to by the literature – is to enable intelligent, social and autonomous software development. These three qualities are argued to be the difference with the classic object-oriented design paradigm. However, we should not focus only on software needs as they are identified by analysts. If we want modern software with these capabilities we must also look into the software developer's needs. These change in two directions. One is the need to produce more software (in lines of code) because such systems are more complex than traditional systems. The second is the need to include and integrate diverse, and advanced technologies like computer vision, planning, argumentative reasoning, etc, that create the need for large software developer teams including many different skills. These needs can only be addressed by a methodology that provides many abstraction levels so that complex software can be modeled. In addition, this methodology must indicate the necessary technologies from the early design stages and allow for clear development tasks decomposition. Moreover, a modular design approach will lead to a clear and quick composition process. Finally, we need new metaphors that will encapsulate the new qualities of agent technology.

Based on our past experience in the Agent Oriented Software Engineering field and also in the development of several real-world multi-agent systems applications (e.g. [18], [19] and [20]), we recognized several shortcomings in the different existing methodologies and we decided to propose a new one. Thus, we were given an opportunity to express our point of view on modular agent architectures ([16], [17]) that we have been supporting several years now. In order to achieve that, we worked on new method fragments for the analysis and design phases of the software development process.

Thus, emerged the Agent SystEms Methodology (ASEME) with significant added value compared to the existing methodologies (e.g. [4], [5], [7], [13] and [10]). This methodology gives new definitions to the terms of capability and functionality, as compared to those of different authors in the past (e.g. [3], [4] and [5]), aiding the representation of modular agents architectures. ASEME synthesizes successful elements of previous agent development methodologies such as Gaia [6], MaSE [13] and Tropos [7]. Finally, it introduces the concepts of inter and intra-agent control, the former allowing for the representation of agent interaction protocols, the latter allowing for the representation of the dynamic interaction between the modules of an agent.

In this paper, we will present an overview of the Agent Systems Methodology (ASEME) and focus in the individual agent development issues, leaving aside, for the moment, the society issue. However, the reader will get an idea of how this is accomplished. An overview of the ASEME iterative process is presented in section two. Then, we define the new Agent MOdeling LAnguage's (AMOLA) concepts and models for the analysis and design phases in section three. The AMOLA models are used throughout this section for developing a sample multi-agent system. In section four we discuss related work and, finally, in section five, we conclude.

2 The Agent Systems Methodology

ASEME defines a software development process. Before viewing this process in detail we present some key concepts.

2.1 Definition of Key Concepts

In this paragraph we define the terms of capability, functionality, intra-agent control and inter-agent control in a way that allows for the modeling of simple or complex – human-like – artificial agents.

Thus, we define the concept of functionality as the mean to represent the thinking, thought and senses characteristics of an agent. Then, we define the concept of capability as the ability to achieve specific tasks that require the use of one or more functionalities.

The agent is an entity with certain capabilities, also capable for inter and intra-agent communication. Each of the capabilities requires certain functionalities and can be defined separately from the other capabilities. The capabilities are the modules that are integrated using the intra-agent control concept to define an agent. Each agent is considered a part of a community of agents. Thus, the community's modules are the agents and they are integrated into it using the inter-agent control concept.

2.2 The Phases of the ASEME Process

ASEME is outlined in Figure 1. We distinguish the six different phases of the software development process, through which the system developer can iterate. From every phase we can return to any previous one and continue from there again. The system can be described in platform independent format (output of the design phase) and in platform specific format (product of the implementation phase) being in line with the recently emerging Model Driven Architecture (MDA, [1]) paradigm.

The process for each phase is hierarchical, starting from the societal level, then focusing on the different actors-roles-agent types, and, finally, detailing the different parts-modules of the agent (the lines between the different levels in the figure aim to demonstrate the increase in detail – decomposition – as we move from the society level to the capability one). Thus, we define three levels of abstraction in each phase. The first is the societal level. There, the whole agent community 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 third level. In the first three phases the process is top-down, while in the last three phases it is bottom-up. It is a classic top-down development approach and our contribution is in defining the levels of abstraction and the information flow between them.

The ASEME process can be described using SPEM[1], as the FIPA Methodology Technical Committee[2] has proposed it for defining Agent Oriented Software Engineering methodologies, but the limited space in this paper does not allow for detailed descriptions of work definitions and work products. However, the reader can get a quick description of each development phase and the main work products that are defined during each step. We have also taken care so that it is as open as possible so that engineers can use method fragments from other methodologies and combine them with ASEME. 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 [15] for details).

Requirements Analysis Phase

During this phase the participating actors are identified along with the goals associated to each of them. Moreover, information about specific requirements that dictate the expected system functionality is collected. To model actors and their goals

---

[1] The Software Process Engineering Metamodel (SPEM) was specified by the Object Management Group (OMG) in 2002, http://www.omg.org

[2] The Foundation for Intelligent Physical Agents (FIPA) is an IEEE Computer Society standards organization. The FIPA Methodology Technical Committee focuses on the identification of a methodology for developing multi-agent systems, http://www.fipa.org

we use the actor diagram – quite similar to the TROPOS actor diagram [7] (which means that a Tropos requirements analysis method fragment could be combined with minimal effort with ASEME). In the first level, the analyst identifies the actors that will act in the system. Then, in the second level each actor is associated with goals, some of which may involve the participation of other actors. In the third level we define the specific requirements related to each goal of each actor. The actor diagram reflects the Computation Independent Model (CIM) of MDA.

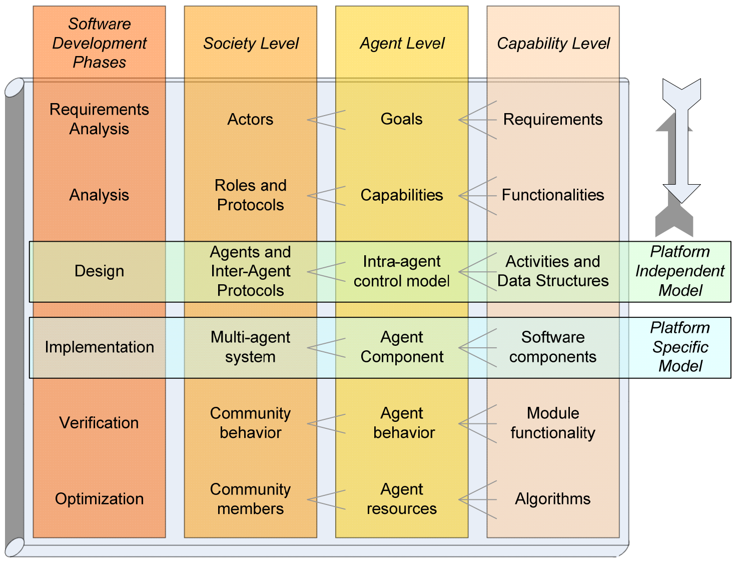


Fig. 1. The Agent Systems Methodology (ASEME) phases and products.

Analysis Phase

The requirements analysis has made a first approach to recognizing the actors, their goals and interactions. An extended Unified Modeling Language (UML, [8]) use case diagram refines the actors to roles and the goals to capabilities that are necessary for achieving them. The main models associated with this phase are the use case model and the roles model, the latter mainly inspired by the Gaia methodology [7] (which means that a Gaia roles model method fragment could be combined with minimal effort with ASEME).

In the first level of abstraction, actors of the previous phase are transformed to roles. Roles can be more abstract than those of the specific actors related to the application domain. E.g., the “housekeeper” and “greengrocer” actors can transform to the concrete “housekeeper” and “greengrocer” roles and the abstract “buyer” and “seller” roles. The use case diagram adds new information in relation to the actor diagram. Firstly, it shows the actors (transformed to roles) that will be developed in the system box. Secondly, it can add abstract roles for use in defining agent interaction protocols. Thirdly, it views the goals from a development perspective, adding implementation-related sub-goals in the form of use cases.

In the second level of abstraction the use cases define the agent capabilities. The roles model is constructed by these capabilities and their decomposition into simple activities and the definition of liveness formulas that depict the process that the agent realizes when activated. Compared to the Gaia roles model, our model is extended allowing for the formalization of multiple concurrent instances of an activity and for defining the needed technology (functionality) for each activity.

In the third level, the analyst points out the technologies that should be used for the realization of each activity. These are the functionalities.

Design Phase

The models associated with this phase are the *agent interaction protocol* and *intra-agent control* models that define the functional and behavioral aspects of the multi-agent system. In the past, the MaSE methodology defined agent behavior as a set of concurrent tasks, each specifying a single thread of control that integrated inter-agent as well as intra-agent interactions. Compared to MaSE, our model offers to the designer the possibility to model the interaction among the capabilities of an agent. Moreover, all modeled capabilities of the agent are reusable encapsulated software modules.

In the first level we define the inter-agent control which implements a specific interaction protocol by defining the necessary roles and the interaction among them. However, the implementation of the inter-agent control is done at the agent level via the capabilities defined at the agent level and their appropriate interaction defined via the intra-agent control. Thus, a "housekeeper" agent can implement the "buyer" abstract role differently than a "tourist" agent, the former knowing more about the regional "seller" agents. Finally, in the third level each capability is defined with regard to its functionality, what technology is used, how it is parameterized, what data structures and algorithms should be implemented. The models defined in this phase are the Platform Independent Models (PIM) with regard to MDA.

Implementation Phase

The platform or the programming languages (e.g. procedural, declarative or a combination of both) for the different agents' modules development are selected. The development phase is bottom-up, which means that firstly the third level software components are developed, secondly an agent is assembled by them in the second level, and, finally, the multi-agent system is instantiated in the selected platform. In this phase, depending on the technology and the methodology fragment that someone will use for implementing the agents, the MDA Platform Specific Models (PSM) are created.

Verification Phase

During this phase the system's functionality is verified in comparison to its requirements. The verification phase could 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.

Optimization Phase

This phase is concerned with the optimization of the system. The algorithms in the capability level can be optimized in execution time or resource consumption. The concurrently executed capabilities number can be optimized in the agent level (based on the available resources that the agent will have) and, 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 can be optimized.

Table 1. The ASEME models

Levels of abstraction **Phases**	**Society level**	**Agent level**
Requirements	Actor diagram (actors)	Actor diagram (goals)
Analysis	Use case diagram Agent interaction protocols	Roles model
Design	Inter-agent control model	Intra-agent control model Component diagram

In Table 1, the reader can see the main models related to each phase of the software development process. The capability level models are not presented since these are related to the development of a simple software component and are dependent on the technology used. For example, for the use of object-oriented components we would use UML models.

3 The Agent-Based Systems Modeling Language

The Agent-Based Systems Modeling Language (AMOLA) describes both an agent and a multi-agent system. It is used for creating the models of the analysis and design phases of ASEME. For demonstrating the different elements of AMOLA we will use an example running throughout this section. The reader should note that in this example we focus in presenting only an individual agent development process.

We assume some simple requirements for a system. The system is responsible for emptying rooms taking orders from humans. In the requirements phase and at the first level (societal) we recognize the actors. There is an external actor (human) that orders a room to be emptied (master) and a system actor that empties the room (worker). Then, in the second level each actor is associated with goals, some of which may involve the participation of other actors. The goals of the worker are to move the objects. In the third level we define the specific requirements related to each goal of each actor. The requirement for workers is that they are small enough to fit in a closet. This requirement implies that a single worker will not be able to move heavy objects, thus we refine our model to allow for a worker that has a heavy object to move to share the task with another. The relevant actor diagram is presented in Figure 2. The reader should notice that when defining a goal dependency form one actor (e.g. ‘worker’) to another that is of the same type then he is named using the word “other” in front of the original name (e.g. ‘other worker’).

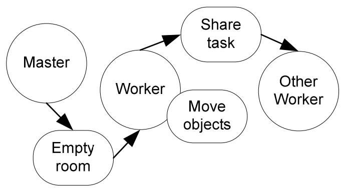


Fig. 2. Actor diagram

3.1 Analysis Phase

In the analysis phase the following concepts are the main ones:

Roles: Human and artificial (agent) roles. They can correspond to the actors of the previous phase (i.e. requirements analysis) as concrete roles, but they may also appear at this stage as abstract agent interaction protocols participants. Roles are initially depicted in the use cases diagram and then refined in the roles model. The latter includes the role name, the inter-agent protocols that the role participates in and its Liveness model.

Use Cases: The use cases define the interaction of the system with its environment and the functionality of the system. Use cases are extracted from the goals of the requirements analysis actor diagram.

Capabilities: The capabilities of a role correspond to the tasks that the role can achieve either by itself or through interaction with other roles and other-external systems. In general, an agent's capabilities describe what the agent "can do". Capabilities are extracted from the use case diagram.

Functionalities: A functionality is related to the different kind of technologies that one can use for implementing reasoning mechanisms (e.g. preference reasoning, abduction, induction) or mechanisms for interaction with the environment (i.e. sensors and effectors). A capability may involve the employment of one or more functionalities. Therefore, functionalities are generic and application independent while capabilities are goals specific.

Activities: Each capability is decomposed to simple activities. For example a "share task" capability can be decomposed to the "search partner" activity, which does the matching between the characteristics of an agent and those of the task, the "choose partner" activity, which compares the selected agents according to different criteria and choose the best one, and, the "send message" activity, which informs the chosen agent on the task that he will have to achieve by sending simultaneously all the necessary input. Each of the activities is related to a different functionality. The decomposition of capabilities to activities starts from the use case diagram when a use case (capability) includes another use case (implying one or more activities) and is completed in the roles model.

Agent Interaction Protocols: A protocol is implemented as a capability. That means that different activities will implement the behavior of an agent in a particular protocol (for example, the decision making activity for determining whether to accept, refuse, or propose an offer within a negotiation protocol). Protocols allow either the achievement of collective tasks or the possibility for agents to achieve individual tasks avoiding harmful interactions with other agents.

Liveness model: A process model that describes the dynamic behavior of the role. It is highly related to the Gaia Liveness property of roles. Each liveness model is defined by formulas that connect all the role's capabilities and associated activities using the Gaia operators [6]. The Liveness model defines the dynamic aspect of the role, that is which activities execute sequentially, which concurrently and which are repeating.

Having presented the above concepts we can now discuss their use in the analysis phase. The use case diagram helps to visualize the system including its interaction with external entities, be they humans or other systems. No new elements are needed other than those proposed by UML. However, the semantics need to change. Firstly, the actor "enters" the system and assumes a role. Thus, we transform the actors identified in the requirements analysis phase to roles. Agents are modeled as roles within the system box. Human actors are also represented as roles, but outside the system box (like in traditional UML use case diagrams). This approach aims to show the concept that we are modeling artificial agents interacting with other artificial agents or human agents. Secondly, the different use cases must be directly related to at least one artificial agent role. Every goal identified during the requirements analysis phase must be satisfied by one use case, which implies an agent capability. These general use cases can be decomposed to simpler ones using the <<*include*>> use case relationship. Based on the use case diagram the system modeler can define the roles model.

A use case that connects two or more (agent) roles implies the definition of a special capability type: the agent interaction protocol. When two or more such use cases can be substituted by a more general one then we introduce new abstract roles that are connected by that general use case. In this case, the protocol description within each role's role model must refer to the assumed role. For example, a use case may connect the role "mother" to the role "greengrocer" and the use case name is "buy tomatoes". Another may connect the role "farmer" to the role "greengrocer" and the use case name is "buy tomatoes". The modeler decides that both use cases are instances of a "buy-sell" protocol and involve the roles "buyer" and "seller". He draws two new roles in the system box with the names "buyer" and "seller" and connects them with the use case "buy-sell". The role model for "mother" will include participation in the "buyer-seller" protocol as "buyer". The role model for "greengrocer" will include participation in the "buyer-seller" protocol both as "buyer" (with the "farmer") and as "seller" (with "mother").

A use case that connects a human and an artificial agent implies the need for defining a human-machine interface (HMI). The latter is modeled as another agent capability.

Referring now to our running example, in the Analysis phase the actor diagram is transformed to a use case diagram. Actors are transformed to roles (societal level) and goals to use cases. The latter are further decomposed to simpler ones using the <<*include*>> relationship. Thus, the use case diagram presented in Figure 3 is created.

In the agent level we define the agent's capabilities as the use cases that correspond to the goals of the requirements analysis phase. The activities that will be contained in each capability are the use cases that are *included* by that capability. Thus, in order to achieve the “move objects” use case, the worker role needs to generate and execute a plan defining the order in which the objects will be removed. He also needs to collaborate with other agents to move heavy objects, because he is only able to move light objects. In order to share a task the agent searches for an available partner, then chooses one among those that match his needs and, finally he sends a message to him to allocate the task.

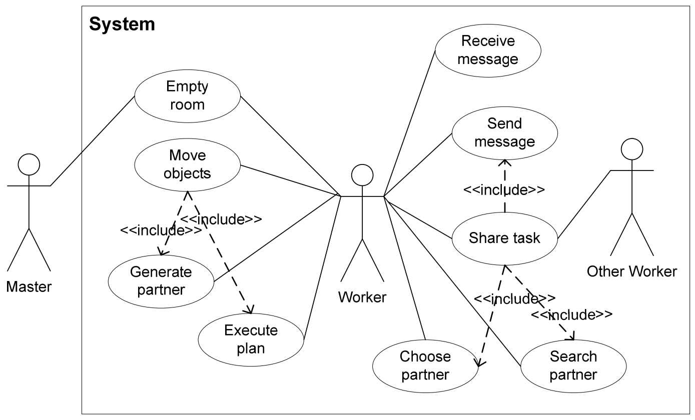


Fig. 3. Use case diagram

In the capability abstraction level the capabilities decomposition to simple activities is concluded and all activities are associated to generic functionalities. The latter must clearly imply the technology needed for realizing them (see Figure 4). 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. This capability is shared by all agents and is defined separately because its implementation is relative to the functionality provided by the agent development platform (*messages handling*) that will be used for implementation.

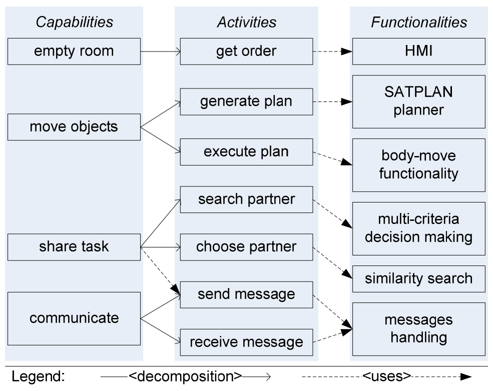


Fig. 4. Capabilities, activities and functionalities

We continue with the definition of the liveness model that is presented in Figure 5 inside the roles model. The liveness model of the agent has a formula at the first line (**root formula**) where we can add activities or whole capabilities. In the latter case the capability must be decomposed in the following line. The || Gaia operator should only be used in the root formula because the statechart orthogonal components (that will be derived from the liveness model) always appear at the outermost state (at the root level). Thus, all capabilities that want to be executed concurrently must be put at the root formula. Finally, the requirements defined in the previous phase may imply or have impact on some system functionality.

Role: Worker
Protocols: Share task: initiator, Share task: responder
Liveness:
agent = get order$^{\omega}$ || receive message$^{\omega}$ || move objects$^{\omega}$
move objects = generate plan. share task*. execute plan+
share task = search partner. choose partner. send message.

Fig. 5. The role model, including three liveness formulas

3.2 Design phase

We provide a brief introduction to the language of statecharts, as the design models are based on it. Then, we present the design phase concepts and work products.

3.2.1 The Statecharts

We use the language of statecharts as it is defined in [9]. Statecharts 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. The behavioral aspects of these activities (what activity, when and under what conditions it will be active) are specified in statecharts.

There are three types of states in a statechart, i.e. OR-states, AND-states, and basic states. OR-states have substates that are related to each other by “exclusive-or”, and AND-states have orthogonal components that are related by “and” (execute in parallel). Basic states are those at the bottom of the state hierarchy, i.e., those that have no substates. The state at the highest level, i.e., the one 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. Action a can be the special action *start(P)* that causes the activity P to start. The scope of a transition is the lowest OR-state in the hierarchy of states.

Multiple concurrently active statecharts are considered to be orthogonal components at the highest level of a single statechart. If one of the statecharts becomes non-active (e.g. when the activity it controls is stopped) the other charts continue to be active and that statechart enters an idle state until it is restarted.

In the past, statecharts have been used for defining reactive systems [9]. They have been used in the past for modeling agent behaviors in MaSE [13]. In our work we use statecharts to model intra-agent control. As we said before, it corresponds to modeling the interaction between different capabilities, defining the behavior of the agent. This interaction defines the interrelation in a recursive way between capabilities and activities of the same capability that can imply concurrent or sequential execution. This is the basic and main difference with the way that statecharts have been used in the past. Moreover, we use statecharts in order to model agent interaction, thus using the same formalism for modeling inter and intra-agent control which is also a novelty. However, the use of statecharts for the inter-agent control is out of the scope of this paper.

3.2.2 Design Models

The design models focus in defining the agent's capabilities along with the inter-agent and intra-agent control. The following concepts are defined at this phase:

Agent Types, which are the roles that become agents.

Agent Actions, which are the effects that an agent can generate. They can be of two types:

- Inter-agent communicative actions (e.g. send messages to other agents). It is a common functionality for all agents.
- Actions on an external system or on the environment (erase a computer file, invoke a web service)

Agent Percepts, which are the changes in the environment that the agent can sense. Such can be:

- The receipt of an inter-agent message
- A change in the environment as it is perceived by a sensor's functionality

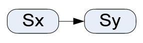
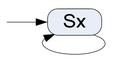
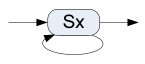
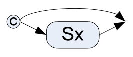
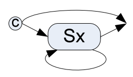
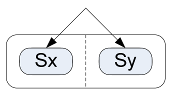
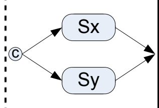
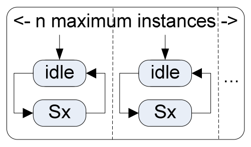
The inter-agent control is defined through statecharts but the way that this is done is out of the scope of this paper..

The last, but equally important concept is the *Intra-Agent Control* that is the main focus of this paper. The intra-agent control allows for modeling the interaction among the different capabilities of the agent. In the design phase the capabilities are transformed to modules. Thus, we define the intra-agent control 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. 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).

Initially the statechart has only one state named after the left-hand side of the first liveness formula of the role model (probably named after the agent type). Then, this state acquires substates. The latter are constructed 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.

At this stage, the activities that have been defined in the roles model are assigned to the states with the same name in the statechart. An agent percept, a monitored for environmental effect, an event generated by any other executing agent activity, or the ending of the executing state activity can cause a transition from one state to another.

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

Op.	*Template*	*Op.*	*Template*
x . y		x^{ω}	
x+		[x]	
x*		x || y	
X | y		$\|x^{\omega}\|^n$	

A capability can be directly related to another concept, the *module*. This allows for a modular representation of the agent's architecture and defines the right level of decomposition of an agent. Thus, it allows for the reusability of the modules as independent software components in different types of agents, having common capabilities. This is also a main difference with other methodologies. Modules for software agents include the following:

- Data structures or data sources
- Methods (relating to activities of the analysis phase) whose execution starts whenever the relevant state is reached in a statechart
- Software libraries implied by the implemented functionalities

In Figure 6 we present the statechart that is derived from the liveness model of our example presented in Figure 5. At this point, we need to define the events that cause transitions, their conditions and also the data elements that will be used for the statechart. These events can be inter-agent messages, or other kinds of events generated by the execution of the agent activities. Moreover, the designer can define algorithms or edit pseudo-code for each activity.

Finally, the designer defines the modules that will be used for the agent. The modules are presented in a UML component diagram; Figure 7 shows the modules of our example. The modules are typically as many as the agent capabilities. The aggregation of these modules leads to a new module, namely the agent. The agent module only implements the root formula of the statechart. The substates are implemented in the relevant modules. All these modules are now concrete components and could be reused in the future by another agent. The grey components in Figure 7 are the used functionalities. The modules are ready for development by transforming the statecharts to code, e.g., like in [10] where finite state machines are implemented in JADE[3], or in [9] for object oriented languages. The designer can simulate and validate his design models using tools such as STATEMATE [9] with different levels of detail thus allowing for iterative-incremental development.

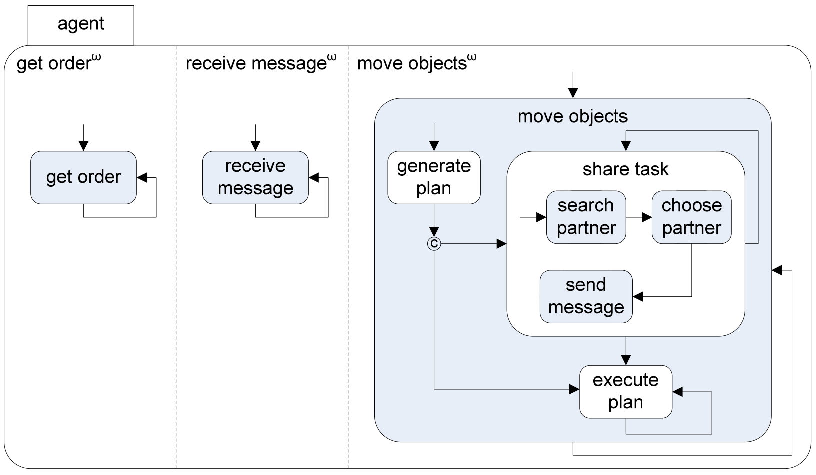


Fig. 6. The intra-agent control model

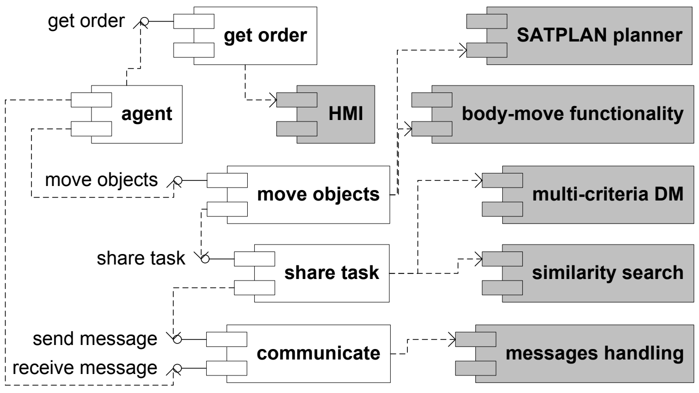


Fig. 7. The agent modules

4 Related Work

Comparing ASEME with the Gaia methodology we first notice that the latter does not support the requirements analysis phase and its agent design models do not lead in a

[3] The Java Agent Development Environment (JADE) is an open source software framework implemented in Java language for developing multi-agent systems (jade.tilab.com)

straightforward way to implementation. For example, the services model isn't concrete – does not relate to code. In the past, Gaia has been modified in order to cover the implementation phase [10], but certain aspects proved difficult to deal with, such as the definition of complex agent interaction protocols or the way to merge two roles in one agent. In [10] we offered some extensions but they were in rather practical than conceptual level. These extensions allowed easily conceiving and implementing relatively simple agents. Finally, its models cannot be used for simulation-optimization.

TROPOS provides a formal language and semantics that greatly aid the requirements analysis phase. It can also lead to successful requirements verification for a system. However, the user must come down to attributes definitions (extremely detailed design including data types) in order to use simulation. It is a process centric design approach, not a module based one, like ours. We believe that the module based approach proposes the right level of decomposition of an agent because it allows for the reusability of the modules as independent software components in different types of agents, having some common capabilities. Moreover, the detailed design phase of TROPOS proposes the use of AUML (as well as in the work presented in [5]). However, AUML has specific shortcomings when it comes to defining complex protocols (the reader can refer to [11] for an extensive list).

MaSE ([13], [14]) defines a system goal oriented MAS development methodology. They define for the first time inter and intra-agent interactions that must be integrated. However, in their models they fail to provide a modeling technique for analyzing the systems and allowing for model transformation between the analysis and design phases. Their concurrent tasks model derives from the goal hierarchy tree and from sequence diagrams in a way that cannot be automated. In our work the model transformation process is straightforward. For example, we provide simple rules for obtaining the design phase intra-agent control from the analysis phase liveness model. Moreover, we distinguish independent modules that are integrated for developing an agent, which can be reusable components. We define agent types that originate from actors of the requirements phase, while the agents in MaSE are related to system goals. This restricts the definition of autonomous agents. Finally, in MaSE agents are implemented using AgentTool while in ASEME a larger number of implementation possibilities are allowed.

In addition to the above, we would like to discuss the terms of capability and functionality that have been used with different meanings in the past. ASEME clearly defines their use and with the help of these terms it can greatly aid the analysis phase defining both what the agent will be able to do but also what technologies should be used for his development.

The authors of [3] proposed a capability concept for BDI agents. In their view, capability is "a cluster of plans, beliefs, events and scoping rules over them". Capabilities can contain sub-capabilities and have at most one parent capability. Finally, the agent concept is defined as an extension of the capability concept aggregating capabilities. The differences of our work in comparison to this one is the fact that capability for us is more general (not limited to BDI agents) which leads to the definition of the module that can be a reusable software component.

Capability in AML [4] is used to model an abstraction of a behavior in terms of its inputs, outputs, pre-conditions, and post-conditions. A behavior is the software component and its capabilities are the signatures of the methods that the behavior realizes accompanied by pre-conditions for the execution of a method and post-conditions (what must hold after the method's execution). However, in our case the concept of capability is more abstract and is used for modeling an agent's abilities that are more general than method signatures. The latter are defined as functionalities and the activity within the capability defines when and how the functionalities are used by the agent.

In [5] the authors use the terms of functionality and capability. However, they correspond to different concepts compared to our work. In fact, functionalities and capabilities refer to same concept as it evolves through the development phases (i.e. the abilities that the system needs to have in order to meet its design objectives). In our work capabilities refer to a specific goal and functionality is related to the used technologies that are application independent (e.g. preference based reasoning, abduction, induction for reasoning mechanism implementation). Moreover, in our approach, with the proposal of the intra-agent control we are able to model in a recursive way the dynamic interaction between capabilities and between activities of the same capability.

5 Conclusion

Having the background of engineering real-world applications using an existing methodology (i.e. [6] and [10]) we move one step further in defining ASEME, a methodology for developing multi-agent systems. ASEME has many qualities compared to other relevant methodologies (e.g. [4], [5], [7] and [10]):

- It defines the intra-agent control for integrating the different modules that constitute an agent allowing for a modular design approach. For this purpose we use statecharts in an original way for this purpose
- It defines the inter-agent control that corresponds to the agent interaction protocol. This part of the methodology is out of the scope of this paper but that which is important is the use of statecharts like in defining the intra-agent control, thus simplifying the designer's task by using the same formalism
- It can be used for developing functional or procedural agents and its design models are platform independent
- It covers the whole software development process and is iterative. There is a straightforward transformation process between the models of the different development phases. These qualities have been considered as very important for the development of software systems (e.g. for object-oriented development in [2])
- Multi-agent systems development is hierarchical at each process step, as it allows for top-down analysis and design decomposition and bottom-up development and integration, thus supporting the development of large-scale systems. The originality is in defining the society, agent and capability level of abstraction
- It can be combined with successful method fragments of existing methodologies with minimum effort. Engineers familiar with those will gain quick understanding of ASEME

Currently we are working on the society level using statecharts in order to model agent interaction protocols. Moreover, we are working on the way that these models will be integrated and implemented through the agent capabilities. Finally, we will provide a SPEM notation for all the phases of the methodology and will attempt to organize the development of a tool for supporting the development lifecycle.

6 References

1. Beydeda, S., Book, M., Gruhn, V. (eds): Model-Driven Software Development. Springer (2005)
2. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley (2003)
3. Braubach, L., Pokahr, A., Lamersdorf, W.: Extending the Capability Concept for Flexible BDI Agent Modularization. In: Proc. of the Third Int. Workshop on Programming Multi-Agent Systems (ProMAS'05), Springer Verlag (2005)
4. Trencansky, I. and Cervenka, R.: Agent Modelling Language (AML): A comprehensive approach to modelling MAS. Informatica 29(4), pp. 391–400 (2005)
5. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. Wiley, (2004)
6. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: the Gaia Methodology. ACM Transactions on Software Engineering and Methodology 12 (3), pp. 317-370 (2003)
7. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A.: TROPOS: An Agent-Oriented Software Development Methodology. Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers (2004)
8. The Unified Modeling Language Specification. Version 1.4.2, ISO/IEC 19501 (2005)
9. Harel D., Naamad, A.: The STATEMATE Semantics of Statecharts. ACM Transactions on Software Engineering and Methodology, 5(4), pp. 293-333 (1996)
10. Moraitis, P., Spanoudakis, N.: The Gaia2JADE Process for Multi-Agent Systems Development. Applied Artificial Intelligence Journal, 20(4-5), Taylor & Francis (2006)
11. Paurobally, S., Cunningham, R., Jennings, N.R.: Developing agent interaction protocols using graphical and logical methodologies. The First Int. Workshop on Programming Multiagent Systems languages, frameworks, techniques and tools (PROMAS 2003), Melbourne, Australia (2003)
12. Lonchamp, J.: A Structured Conceptual and Terminological Framework for Software Process Engineering. Proceedings of the Second International Conference on the Software Process: Continuous Software Process Improvement, Berlin, Germany. IEEE Computer Society, ISBN 0-8186-3600-9, pp. 41-53 (1993)
13. Deloach, S.A., Wood, M.F. and Sparkman, C.H.: Multiagent Systems Engineering. Int. Journal of Software Engineering and Knowledge Engineering, Vol. 11, No. 3, pp. 231-258 (2001)
14. Deloach, S.A.: Specifying Agent Behavior as Concurrent Tasks. Proceedings of the fifth international conference on Autonomous agents, Montreal, Canada, ACM, pp. 102-103 (2001)
15. Cossentino, M., Gaglio, S., Garro, A., Seidita, V.: Method fragments for agent design methodologies: from standardisation to research. Int. Journal of Agent-Oriented Software Engineering, Vol. 1, No.1 pp. 91-121 (2007)
16. Moraitis, P.: Decision Theoretic and Logic Based Agents for Multi-Agent Systems. Habilitation for Research Supervising, University Paris-Dauphine, France (2002)

17. Karacapilidis, N., Moraitis P.,: Intelligent Agents for an Artificial Market System. Proc. fifth International Conference on Autonomous Agents (AGENTS'01), pp. 592-599, Montreal, Canada (2001)
18. Spanoudakis N., Moraitis, P.: Agent Based Architecture in an Ambient Intelligence Context. Proceedings of the 4th European Workshop on Multi-Agent Systems (EUMAS'06), Lisbon, Portugal (2006)
19. Moraitis, P., Petraki, E. and Spanoudakis, N.: An Agent-based System for Infomobility Services. Proceedings of the third European Workshop on Multi-Agent Systems (EUMAS2005), Brussels, Belgium, (2005)
20. Moraitis, P., Petraki, E. and Spanoudakis, N.: Providing Advanced, Personalised Infomobility Services Using Agent Technology. Proc. of the 23rd SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence (AI2003), Peterhouse College, Cambridge, UK (2003)