The Agent Modeling Language (AMOLA)

Nikolaos Spanoudakis[1,2] and 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 MOdeling LAnguage (AMOLA). This language provides the syntax and semantics for creating models of multi-agent systems covering the analysis and design phases of the software development process. It 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. However, in this paper we focus in presenting only the individual agent development process. AMOLA is used by ASEME, a general agent systems development methodology.

Keywords: Multi-agent systems, Tools and methodologies for multi-agent software systems.

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. In our understanding it aims to provide the system developers with adequate engineering concepts that abstract away the complexity that is inherent to such systems. Moreover, it should allow for modular development so that successful practices can easily be incorporated in new systems. Finally, it should cater for model transformation between the different software development phases so that the process can be automated.

In the past, we introduced the Gaia2JADE process for Multi-agent Systems (MAS) development ([12]) and since then it has been used for the successful development of a number of MAS, see e.g. [8]. However, our more recent work (i.e. [11]) was about a more complex MAS that called for modularity, abstraction and support for iterative development. At the same time, we observed that the "services model" of Gaia [19] didn't apply to modern agents who provide services through agent interaction protocols. Moreover, we had no specific requirements analysis models that would be

D. Dochev, M. Pistore, and P. Traverso (Eds.): AIMSA 2008, LNAI 5253, pp. 32–44, 2008.

transformed to analysis models. Furthermore, the protocol model of Gaia did not provide the semantics to define complex protocols and the Gaia2JADE process additions remedied this situation only for simple protocols.

In order to address these issues, we used method fragments [4] from other methodologies and introduced a new language for the analysis and design phases of the software development process, namely the *Agent MOdeling LAnguage (AMOLA)*. All these changes led to the proposal of a new methodology, ASEME, an acronym of the full title *Agent SystEms Methodology*. Through this methodology, we were given an opportunity to express our point of view on modular agent architectures (see e.g. [9]) that we have been supporting several years now.

AMOLA not only modernizes the Gaia2JADE process but offers to the system developer new possibilities compared to other works proposed in the literature (a large set of which can be found in [7]). It allows models transformation from the requirements analysis phase to implementation. It defines three levels of abstraction, namely the society level, the agent level and the capability level, in each software development phase, thus defining a top-down analysis and design approach. Finally, using in an original way the statecharts and their orthogonality feature, it defines the inter- and intra-agent control models, the first for coordinating a group of agents and the second for coordinating an agent's modules.

In this paper, we present AMOLA focusing 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. Firstly, we discuss AMOLA's basic characteristics in section two, followed by the analysis and design phase models, in sections three and four respectively. For demonstrating the different elements of AMOLA we use a real-world system development case-study. Then, we discuss related work in section five and, finally, section six concludes.

2 The Basic Characteristics of AMOLA

The Agent Modeling Language (AMOLA) describes both an agent and a multi-agent system. Before presenting the language itself we identify some key concepts. Thus, we define the concept of *functionality* 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.

The originality in this work is the intra-agent control concept that allows for the assembly of an agent by coordinating a set of modules, which are themselves implementations of capabilities that are based on functionalities. Here, the concepts of capability and functionality are distinct and complementary, in contrast to other works where they refer to the same thing but at different stages of development (e.g. Prometheus [14]). The agent developer can use the same modules but different assembling

strategies, proposing a different ordering of the modules execution producing in that way different profiles of an agent (like in the case of the KGP agent [1]). Using this approach, we can design an agent with the *reasoning capability* that is based on the *argumentation based decision making functionality*. Another implementation of the same capability could be based on a different functionality, e.g. abductive reasoning.

Then, in order to represent our designs, AMOLA is based on statecharts, a well-known and general language and does not make any assumptions on the ontology, communication model, reasoning process or the mental attitudes (e.g. belief-desire-intentions) of the agents giving this freedom to the designer. Other methodologies impose (like Prometheus [14] or Ingenias [16]) or strongly imply (like Tropos [3]) the agent meta-models (see [7] for more details). Of course, there are some developers who want to have all these things ready for them, but there are others that want to use different agent paradigms according to their expertise. For example, one can use AMOLA for defining Belief-Desire-Intentions based agents, while another for defining procedural agents.

The AMOLA models related to the analysis and design phases of the software development process are shown in Figure 1. These models are part of a more general methodology for developing multi-agent systems, ASEME (Agent Systems Methodology, a preliminary report of which can be found in [17]). ASEME is a model-driven engineering (MDE) methodology. 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[1], [10]) paradigm. MDA's strong point is that it strives for portability, interoperability and reusability, three non-functional requirements that are deemed as 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.

The system is described in platform independent format at the end of the design phase. We will provide guidelines for implementing the AMOLA models using JADE or STATEMATE (two different platform specific models).

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 *capability level*. In the first three phases the process is top-down, while in the last three phases it is bottom-up.

[1] The Model Driven Architecture (MDA) is an Object Management Group (OMG) specification for model driven engineering, http://www.omg.org/mda/

AMOLA is concerned with the first two levels assuming that the analysis and design in the capability level can be achieved using classical software engineering techniques.

	Development Phase	Levels of Abstraction		
		Society Level	Agent Level	Capability Level
	Analysis Phase *AMOLA Models*	Roles and Protocols Use case Diagram Agent Interaction Protocols	Capabilities Roles Model	Functionalities
Platform Independent Model	**Design Phase** *AMOLA Models*	Agent Types And Organization Inter-agent control model	Modules Intra-agent control model	Components

Fig. 1. The Agent Systems Methodology (ASEME) phases in three levels of abstraction and the AMOLA models related to each level

For the AMOLA models demonstration we present the analysis and design models of an agent participating in a real-world system that we developed. Our requirements were to develop a system that allows a user to access a variety of location-based services (LBS) that are supported by an existing brokering system, using a simple service request protocol based on the FIPA Agent Communication Language (ACL). The system should learn the habits of the user and support him while on the move. It should connect to an OSGi[2] service for getting the user's coordinates using a GPS device. It should also handle dangerous situations for the user by reading a heart rate sensor (again an OSGi service) and call for help. A non-functional requirement for the system is to execute on any mobile device with the OSGi service architecture. For more details about the real-world system the reader can refer to [11].

3 The Analysis Phase Models

The main models associated with this phase are the *use case model* and the *roles model.* The former is an extended UML *use case diagram* and the latter is mainly inspired by the Gaia methodology [19] (a Gaia roles model method fragment can be used with minimal transformation effort).

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 change. Firstly, the actor "enters" the system and assumes a role. *Agents* are modeled as roles, either within the system box (for the agents that are to be developed) or outside the system box (for existing agents in the environment). Human actors are also represented as roles outside the system box (like in traditional UML use case diagrams). We distinguish the human roles by their name that is written in italics. 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

[2] The Open Services Gateway initiative (OSGi) alliance is a worldwide consortium of technology innovators defining a component integration platform, http://www.osgi.org

one artificial agent role. 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 participation of the agent in an interaction protocol (e.g. negotiation). 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 functionality. A use case can *include* a second one showing that its successful completion requires that the second also takes place.

Referring now to our case study, 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 relevant actor diagram that was the result of the previous phase is presented in Figure 2.

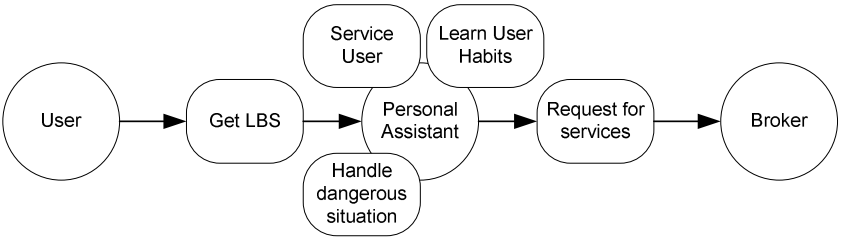


Fig. 2. Actor diagram. The circles represent the identified actors and the rounded rectangles their goals. It is the output of the requirements analysis phase.

In the Analysis phase, the actor diagram is transformed to the use case diagram presented in Figure 3. The activities that will be contained in each capability are the use cases that the capability includes. Then we define the role model for each agent role (see Figure 4). Firstly we add the interaction protocols that this agent will be able to participate in. In this case it is a simple service protocol with a requester (our agent) and a responder. We continue with the definition of the liveness model inside the roles model. The liveness model has a formula at the first line (*root formula*) where we can add activities or capabilities. A capability must be decomposed to activities in the following line.

In the capability abstraction level the activities are associated to generic functionalities. The latter must clearly imply the technology needed for realizing them (see Figure 5). In this case, returning to our running example, the requirement for functioning on any mobile device running OSGi services reveals that such a device must at least support the Java Mobile Information Device Profile (MIDP), which offers a specific record for storing data. Therefore, the activities that want to store or read data from a file must use the MIDP record technology. Finally, 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. JADE[3].

[3] The Java Agent Development Environment (JADE) is an open source framework that adheres to the FIPA standards for agents development, http://jade.tilab.com

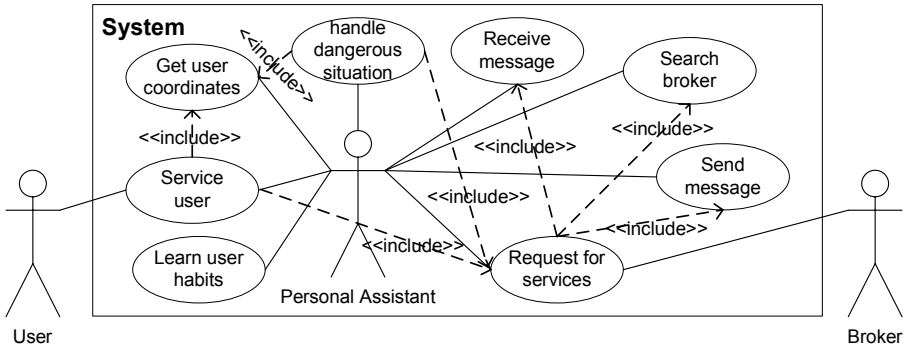


Fig. 3. Use case diagram

Role: Personal Assistant
Protocols: Service protocol: initiator
Liveness:
personal assistant = (service user)$^{\omega}$ || (handle dangerous situation)$^{\omega}$
service user = get user order. get user coordinates. get user preferences. request for services. present information to the user. learn user habits.
handle dangerous situation = invoke heart rate service. determine user condition. [get user coordinates. request for services]
request for services = search broker. [send message. receive message]
learn user habits = learn user preference. update user preferences.

Fig. 4. The role model, including five liveness formulas

4 The Design Phase Models

The models associated with the *Design phase* are the *inter-agent control* and *intra-agent control*. They define the functional and behavioral aspects of the multi-agent system. In the past, the MaSE methodology [5] defined agent behavior as a set of concurrent tasks, each specifying a single thread of control that integrates inter-agent as well as intra-agent interactions. Our model goes one step further by modeling the interaction among the capabilities of an agent, i.e. what they call the different threads of control, but also their execution cycle. The model associated to the first level of this phase is the inter-agent control, which defines interaction protocols by defining the necessary roles and the interaction among them. The implementation of the inter-agent control is done at the agent level via the capabilities and their appropriate interaction defined via the intra-agent control. 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 Model (PIM).

For the Design Phase models we use the language of statecharts as it is defined in [6]. 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

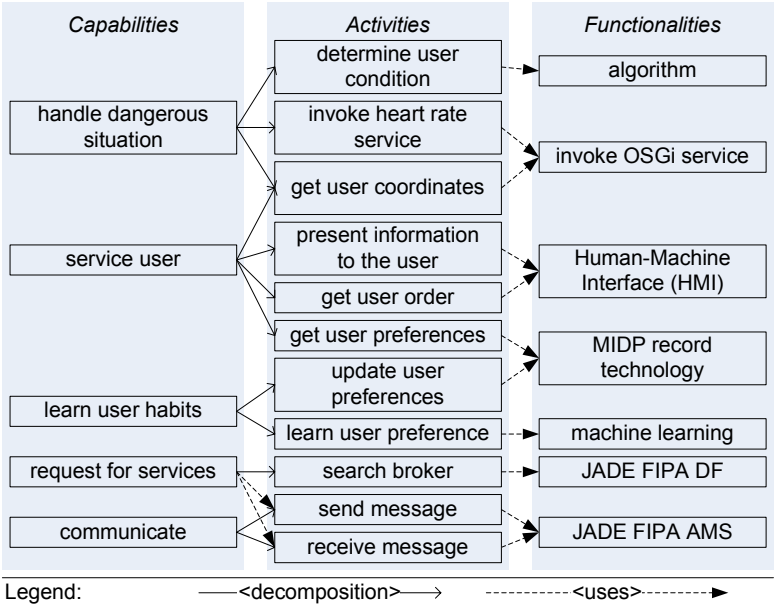


Fig. 5. Capabilities, activities and functionalities

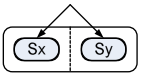
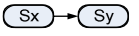
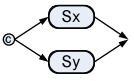
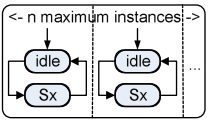
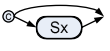
system are captured by activities and the data elements and signals that can flow between them. The behavioral aspects of these activities (what activity, when and under what conditions it will be active) are specified in statecharts. There are three types of states in a statechart, i.e. OR-states, AND-states, 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 modeling agent behaviors in MaSE [5]. 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 also between 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.

In the agent level, 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 1. 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 1. If one of the states is further refined in a next formula, then new substates are defined for it in a recursive way.

Table 1. Templates of extended Gaia operators (Op.) for Statechart generation

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

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.

In Figure 6 we present the statechart that is derived from the liveness model of our example presented in Figure 4. 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.

Finally, the designer defines the modules that will be used for the agent. The modules are typically as many as the agent capabilities. 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. The agent implements the root formula of the statechart. The substates are implemented in the relevant modules. The modules are ready for development by transforming the statecharts to code, not restricted to JADE development like in [12], but using any tool that transforms statecharts to code, e.g. STATEMATE [6] for object oriented languages. In order to transform the models

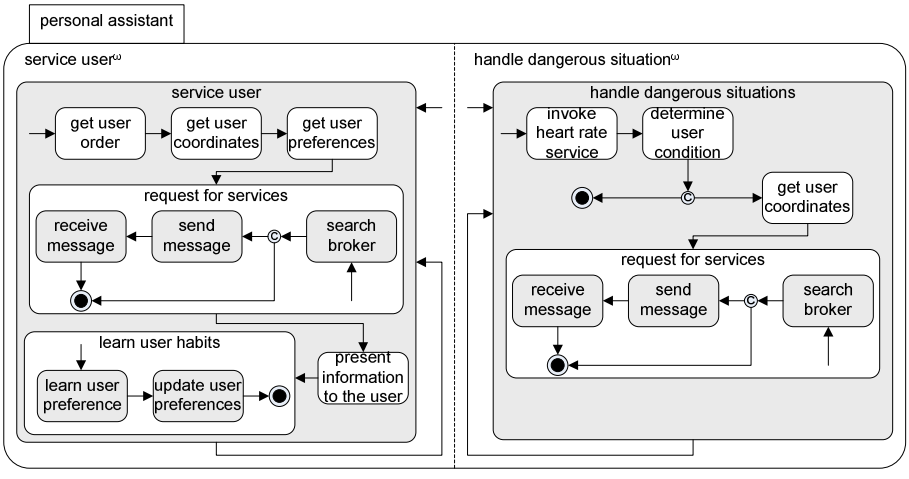


Fig. 6. The intra-agent control model

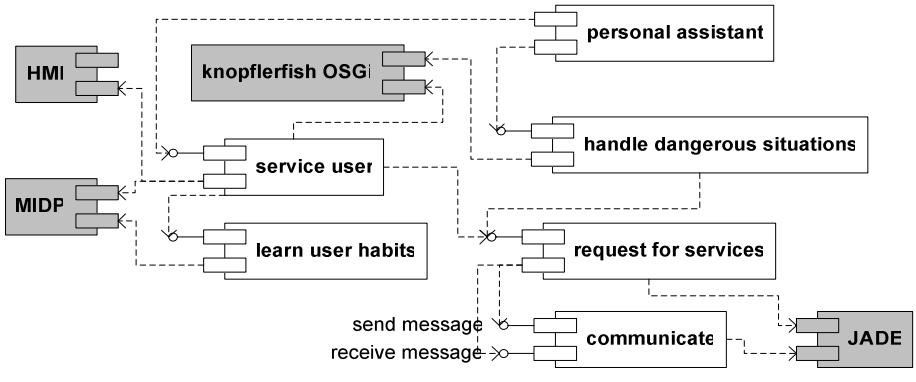


Fig. 7. The agent modules

to JADE code the developer should transform complex states to instances of the *FSMBehaviour* class and the simple states to *SimpleBehaviour* instances in a fashion relevant to the one in [12].

In the next ASEME phase the modeler firstly creates the platform specific models and then implements the system. 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 reader will notice that the

abstract functionalities like *algorithm* are not shown in this diagram as they do not refer to an existing software component. They are analyzed, designed and programmed like any other software component using an existing method, e.g. UML.

5 Related Work

Comparing AMOLA with the Gaia methodology [19] we first notice that the latter does not support the requirements analysis phase and its agent design models do not lead in a 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 [12], 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 [12] we offered some extensions but they were in rather practical than conceptual level. These extensions allowed for easily conceiving and implementing relatively simple agents. Finally, its models cannot be used for simulation-optimization. AMOLA can be connected with successful method fragments in the requirements analysis phase (such as the actor diagram of Tropos) and its models are statecharts. The latter is a well known language for which there are numerous tools for code generation and simulation/optimization.

TROPOS [3] 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 AMOLA. 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 [13] (as well as in the work presented in [14]). However, AUML has specific shortcomings when it comes to defining complex protocols (the reader can refer to [15] for an extensive list). Finally, Tropos has been applied for modeling relatively simple agents, not complex ones [7].

AUML has been proposed as a language for modeling multi-agent systems. However, it does not come along with a methodology or a complete process for software development. Thus, many methodologies just use some of its models, mainly the agent interaction protocol (AIP) model that has been defined as an extension to the UML sequence diagram. The Layered approach to protocols provides a mechanism for specifying the program that implements a protocol but does not specify how it is integrated with other such programs (other protocols), or how to integrate it with the other agent capabilities. AMOLA caters for this issue by using the same formalism (statecharts) for modeling inter and intra-agent control.

MaSE [5] 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 AMOLA more implementation possibilities are allowed.

In Prometheus [14] 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. argumentation, 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. The support for implementation, testing and debugging of Prometheus models is limited and it has less focus on early requirements and analysis of business processes [7]. AMOLA's capability to be integrated with method fragments and the fact that its design models are statecharts overcomes these issues.

One interesting approach that is based on UML is Ingenias [16], which proposes several meta-models that define specific structures for different concepts like *agent*, *role*, *resource*, etc. For some developers it could be useful, but for others it could be considered as a constraint, as it imposes a particular structure for an agent and agent organizations. Moreover, Ingenias does not offer the convenience of gradually modeling a multi-agent system by considering it at three levels of abstraction.

The authors of [2] 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 the capability concept for us is more general (not limited to BDI agents) and it leads to the definition of the module that can be a reusable software component.

Capability in AML [18] 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 AMOLA 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.

6 Conclusion

Concluding, in this paper we defined AMOLA, a language for modeling agent systems that has many qualities compared to other relevant methodologies (e.g. the Prometheus, Gaia, Tropos, PASSI and MaSE discussed in [7]):

- 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. 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 the intra-agent control, thus simplifying the designer's 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 (e.g. like in Ingenias [16] and Prometheus [14])
- 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.

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.

References

1. Bracciali, A., Demetriou, N., Endriss, U., Kakas, A., Lu, W., Stathis, K.: Crafting the Mind of a PROSOCS Agent. Applied Artificial Intelligence 20(4-5) (April 2006)
2. Braubach, L., Pokahr, A., Lamersdorf, W.: Extending the Capability Concept for Flexible BDI Agent Modularization. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2005. LNCS (LNAI), vol. 3862. Springer, Heidelberg (2006)
3. 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 (2004)
4. 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 1(1), 91–121 (2007)
5. Deloach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent Systems Engineering. Int. Journal of Software Engineering and Knowledge Eng. 11(3), 231–258 (2001)
6. Harel, D., Naamad, A.: The STATEMATE Semantics of Statecharts. ACM Transactions on Software Engineering and Methodology 5(4), 293–333 (1996)

7. Henderson-Sellers, B., Giorgini, P. (eds.): Agent-Oriented Methodologies. Idea Group Publishing (2005)
8. Karacapilidis, N., Lazanas, A., Megalokonomos, G., Moraitis, P.: On the Development of a Web-based System for Transportations Services. Information Sciences 176(13), 1801–1828 (2006)
9. Karacapilidis, N., Moraitis, P.: Intelligent Agents for an Artificial Market System. In: Proc. fifth Int. Conf. on Autonomous Agents (AGENTS 2001), Montreal, Canada, pp. 592–599 (2001)
10. Kleppe, A., Warmer, S., Bast, W.: MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley, Reading (2003)
11. Moraitis, P., Spanoudakis, N.: Argumentation-based Agent Interaction in an Ambient Intelligence Context. IEEE Intelligent Systems 22(6), 84–93 (2007)
12. Moraitis, P., Spanoudakis, N.: The Gaia2JADE Process for Multi-Agent Systems Development. Applied Artificial Intelligence Journal 20(4-5) (2006)
13. Odell, J., Parunak, H.V.D., Bauer, B.: Representing Agent Interaction Protocols in UML. In: Ciancarini, P., Wooldridge, M.J. (eds.) AOSE 2000. LNCS, vol. 1957. Springer, Heidelberg (2001)
14. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. Wiley, Chichester (2004)
15. Paurobally, S., Cunningham, R., Jennings, N.R.: Developing agent interaction protocols using graphical and logical methodologies. In: Dastani, M., Dix, J., El Fallah-Seghrouchni, A. (eds.) PROMAS 2003. LNCS (LNAI), vol. 3067. Springer, Heidelberg (2004)
16. Pavón, J., Gómez-Sanz, J.J., Fuentes, R.: The INGENIAS Methodology and Tools. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 236–276. Idea Group Publishing (2005)
17. Spanoudakis, N., Moraitis, P.: The Agent Systems Methodology (ASEME): A Preliminary Report. In: Proceedings of the 5th European Workshop on Multi-Agent Systems (EUMAS 2007), Hammamet, Tunisia, December 13 - 14 (2007)
18. Trencansky, I., Cervenka, R.: Agent Modelling Language (AML): A comprehensive approach to modelling MAS. Informatica 29(4), 391–400 (2005)
19. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: the Gaia Methodology. ACM Trans. on Software Eng. and Methodology 12(3), 317–370 (2003)