Implementation Level Issues in MAS Modeling

Cossentino Massimo
Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
Consiglio Nazionale delle Ricerche (CNR)
cossentino@pa.icar.cnr.it

Poggi Agostino, Rimassa Giovanni and Turci Paola
AOT Lab - Dipartimento di Ingegneria dell'Informazione
Universita' degli Studi di Parma
{poggi, rimassa, turci}@CE.UniPR.IT

***Abstract*— The aim of this paper is to focus on the issues connected with the diagrammatic notations and tools, which should support developers when moving from the design phase towards the implementation phase of MASs. The paper deals with three main research lines: the representation of the interactions between agents and their relationship with the internal architecture of the single agent, the deployment of MASs, and the pattern reuse.**

I. Introduction

As evinced from an analysis of the majority of the proposed languages and methodologies, a great effort has been spending in defining new metaphors, symbols and diagrammatic notations. But despite this effort the modeling languages and methodologies proposed still remain incomplete.In particular, up to now, the study of the issues connected with the multiagent system implementation phase has not been stressed enough. The only methodology, which has dealt with these issues, is the PASSI methodology [1]; and we take it as a good starting point. The aim of this paper is to focus on the issues connected with the diagrammatic notations and tools, which should support developers when moving from the design phase towards the implementation phase of MASs. We have identified three main research lines: the representation of the interactions between agents and their relationship with the internal architecture of the single agent, the pattern reuse and the deployment of MASs. The reason for our choice is to be found in the nearest antecedent technology: object oriented software technology. We believe that this technology can give us the right means to understand what we could need in the agent technology field. In UML 2.0 the deployment diagram is the only form of implementation diagram and its role is to show how Artifacts are allocated to Nodes. None of the existing agent oriented modeling languages provide concepts and notations to fully capture the multiagent system deployment. Concerning the MAS implementation phase, we are convinced of the usefulness of the deployment diagram. Furthermore we believe that we need another diagram, showing the interactions between agents and the relationship with the activities flow inside the single agent, which would be a valid support for developers. We intend to refer to Agent UML (AUML) [2] [3], a modeling language based on UML, as a modeling language used to express the mentioned diagram. The AUML work is organized as an activity within the FIPA Modeling Technical Committee [4]. In this paper, we focus on a new subset of an agent-based UML extension for the specification of the interactions between agents, and their relationship with the internal structure of a single agent, and the deployment of a multiagent system. This is accomplished by extending the formally-based UML metamodel to support the semantics of agents and their supporting platforms. In particular, we aim at showing how UML can be exploited and extended to drive the implementation phase and to model the deployment of multiagent systems at the agent level. The presented notation takes advantage of stereotypes to associate an agent-oriented semantic with the model elements involved in the diagram. In order to be competitive today, the modern development process must allow customers to maximize business opportunities by enabling them to identify new ones; by ensuring a return on their investment as early and frequently as possible; and by allowing them to make changes when they feel the need. Multiagent systems, as any software development approach, benefit from sound methodology and notation but there is more to them. Concrete issues such as APIs, libraries and infrastructure are also very important to achieve a success when adopting MAS development in a software project. Moreover our conviction is that pattern reuse is a very challenging and interesting issue in multiagent systems as it has been in object-oriented ones. The introduction of an extensive pattern reuse practice in the implementation phase can be determinant in cutting down the time and cost developing the multiagent systems. Many works have been proposed about the patterns of agents; our concept of pattern addresses an entity composed of portions of design and the corresponding implementation code. The next section describes the behavioral diagrams, which will drive the implementation phase, giving useful information concerning the implementation of the interactions between agents and their relationship with the internal structure of a single agent. Section three deals with the relevance and the peculiarity of the configuration and deployment in multiagent systems, describing the rationale used in defining the AUML deployment diagram, specifying the standard representation for its elements in UML models. Section four deals with the importance of pattern reuse in the implementation phase. Finally, the fifth section concludes with a discussion about implementation issues and future work.

II. Implementation-Level MAS Interactions

The increased focus on interaction is one of the distinctive traits of the agent-oriented development approach, and it permeates the whole MAS development process, starting from

role-model based system analysis. However, dealing with the interaction dimension within multi-agent systems on a general basis is not the purpose of this paper. Instead, we wish to concentrate on the peculiar view over agent interaction that becomes dominant when the development process is very close to actual implementation of a multi-agent system. In the first part of this section we will try to define a two-dimensional plane and a point on that plane where a diagram showing implementation-level MAS interactions should be placed.

The implementation phase typically needs a very precisely defined model, which fully refines previous design models to achieve a specification that is unambiguous enough to be fed to automated tools such as compilers and code generators. Producing such a detailed model generally entails matching the conceptual entities, gathered from analysis and architectural design deliverables, to the actual technological artifacts that are going to be used to build the system. Applying the above consideration to the case of multi-agent systems development, we have to recognize that with the current state of the technology the final implementation of a multi-agent system will most likely use object-oriented technology. While some agent-level concepts have no direct counterpart in object-oriented languages, the "every software is simulation software" vision that lies at the heart of the object-oriented approach allows agent developers to define those concepts at the infrastructure layer. In the past years several libraries, frameworks and middleware platforms have been made available, that provide MAS application developers with agent-level abstractions implemented with object-oriented languages and tools. When adopting such support systems, developers express interaction, coordination and deliberation concerns through an agent-level API, resorting to plain object orientation when specifying ordinary computation tasks. Therefore, the first dimension of our conceptual plane is the *component* dimension, and we claim that an effective implementation-level diagram for MAS interactions must be placed along this dimension where agents melt into objects. This implies that looking at the diagram one has to be able to recognize both the agent-level entities represented and the object-oriented artifacts with which they are expressed. The second dimension that we consider is the *interconnection* dimension (in a way similar to architecture definition languages, which specify software architectures as made by *components* and *connectors*). Along this dimension, willing to represent MAS interactions at the implementation level, we notice that the focus is to be put on the *boundary* between individual agents and the society they participate in. This is because most implementation effort goes into writing the components, and the connectors are generally implemented by how the code within a component interacts with code within another one. So, just as our diagram is placed at the agent-object boundary in the component dimension, we place it at the agent-society boundary in the interconnection dimension. This means that, when looking at the diagram, one has to be able to see the (part of the) social structure the agents participate in as a whole, but also which internal parts of each agent get actually involved in the ongoing interaction. Considering the component vs. interconnection plane, our interaction diagram is placed at the contact point between the agent internal structure and the external environment. This entails that the diagram cannot be independent from the actual agent architecture; the implementation-level interaction has to be specified differently for different agent architectures.

Several agent architectures have been proposed in the literature, each with its own strengths and weaknesses, its own preferred application domains and its own supporters. We strive here for a *parametric diagram*, taking the concrete architectures of the agents participating to the interaction as parameters. The aim is to support both well-known agent architectures and custom agent architectures expressed through an object-oriented model of the agent internals.

Two additional, but important nevertheless, requirements for an implementation-level representation of agent interactions are *traceability* and *tool integration*. Since we operate very closely to the source code of the system, a bidirectional mapping between code and diagrams is highly desirable and much easier to attain than, say, in the case of requirements traceability. Tool integration is paramount at this level; interesting and desirable features for an interaction diagram are the ability to generate the possible interaction traces or the visual animation of the interaction.

A. Choosing the Perspective: UML Activity Diagrams

Representing the dynamics of a MAS is quite different from describing the flow of control of an object-oriented system; we will come back on this issue in the next sub-section but by now we could simply argue that the first determinant difference is in the greater encapsulation of the agents; in fact despite having a very complex inner structure (they are often composed of several classes), they interact with the remaining part of the system as a whole (the agent) that can be resembled to an unicellular organism that aggregates with the others creating complex structures where each individual plays a precise role and has a specific objective. Another important issue is that agents, usually, cannot directly relate to each others but they need a message transport service that in many architectures is provided by the middleware. These interactions are obviously totally different in nature by the direct method invocations that could take place within the agent among the classes that constitutes it. As already stated we decided to support the AUML initiative and therefore we mainly consider UML diagrams as the source for our notations. In order to describe the agents interactions, we should choose a diagram that allows the description of the behavior of the component dimension during time also in terms of the interconnection dimension. This overall requisite can be decomposed in some specific needs:

- Being focused on the implementation issues, we aim at obtaining and high level of detail. This sometimes produces very crowded diagrams that going deeply into the particulars fail in providing the designer with a global description of the situation; in the solution we will provide a zoom capability that allows the representation

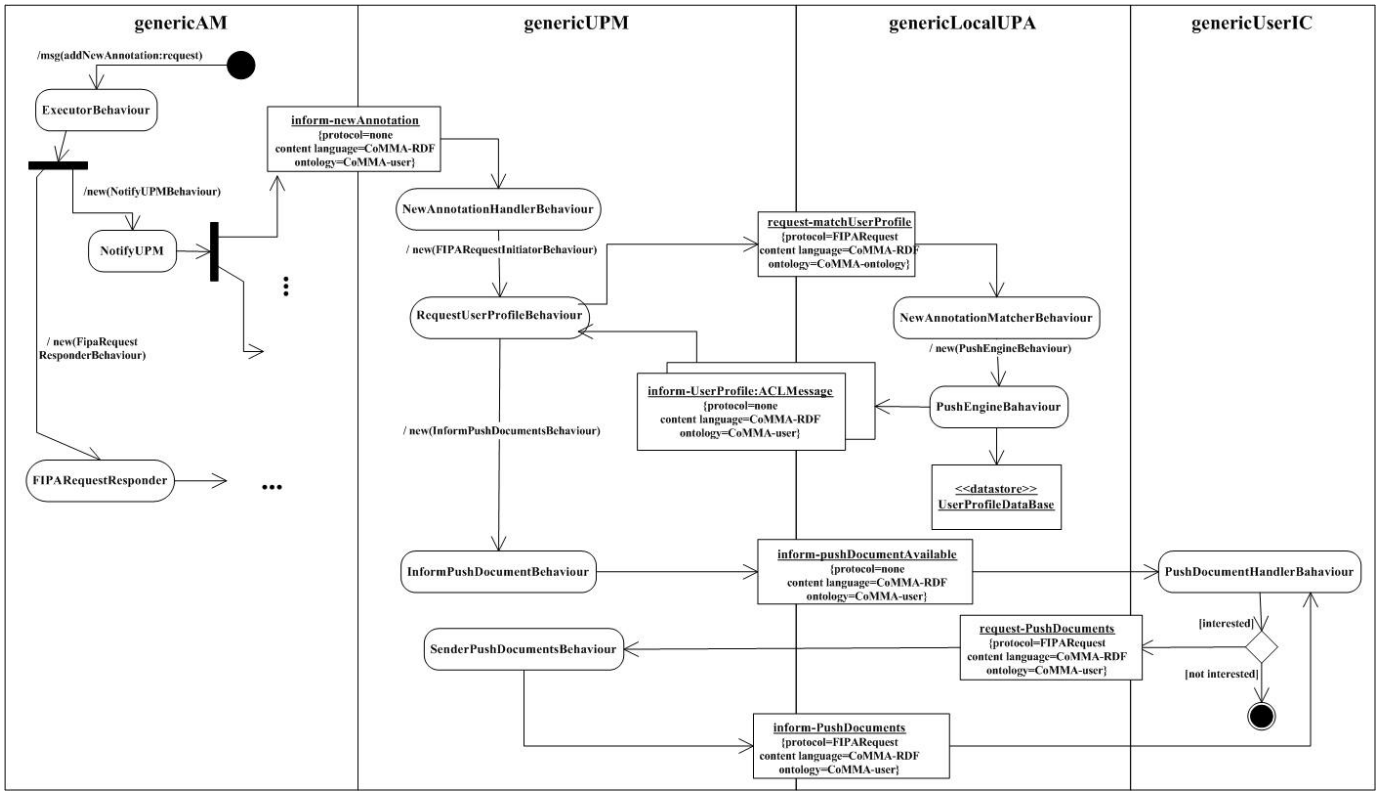

Fig. 1. Diagram for FSM-based agents

of different levels of details in the same diagram for different agents.

- Both agents and objects should be described (if necessary) in the same diagram in order to represent not only the agent but also its internal classes (this structure is essentially OO) and the interactions of the MAS with external not agent-based systems;
- Possible interactions that should be described are the communications among different agents (inter-agents relationships) and the classical object-oriented interactions within a single agent (intra-agent relationships).

A good solution to these problems could come from the use of activity diagrams with some minor extensions of their syntax. These diagrams allow the description of both the structural and behavioral aspects of the MAS; in fact, concepts like swim lanes show the relationships among the software entity related to that swim lane and its dynamical representation in terms of activities/actions. In activity diagrams, time constraints can be explicitly represented (this allows for example the representation of real-time situations) just like concurrency (useful in order to explicitly model agent pro-activeness) and other control-flow structures like branches, forks and joints. An extension of the 'classical' UML activity diagram is necessary in order to obtain the proper level of detail for each different situation. We think that these diagrams should not address only one specific level of abstraction but a range of possible levels. The coarse-grained representation could include a diagram with one swim lane for each agent. Activities within each swim lane group the dynamics of an agent. At this level of abstraction we could think that activities granularity is comparable to an agent behavior. The opposite situation is described in a diagram composed of one different swim lane for each agent class or agent behavior; in this case each activity can address a method level entity. There should be no duality nor separation between these two extremes; we consider that a specific CASE tool could support a zooming operation so that the designer could start looking at the whole system and then in order to refine his/hers work he could exploit the representation of one agent down to the lowest level of detail.

B. Example: FSM-Based Agent Architecture

As a working example, we consider here the very simple yet popular agent architecture that builds software agents as active objects with basic task scheduling capabilities but no reasoning and planning modules. In most cases, tasks are represented as first class objects and can be composed according to some rules to aggregate complex tasks out of simpler ones. The most common execution model for this kind of agent relies on Finite State Machines. An object-oriented implementation of the above, cast in a class-based language ends up with an *agent class* (that is called `jade.core.Agent` in the JADE framework) and a *task* class (`jade.core.behaviours.Behaviour` in JADE). When applying UML Activity Diagrams to FSM-based agents,

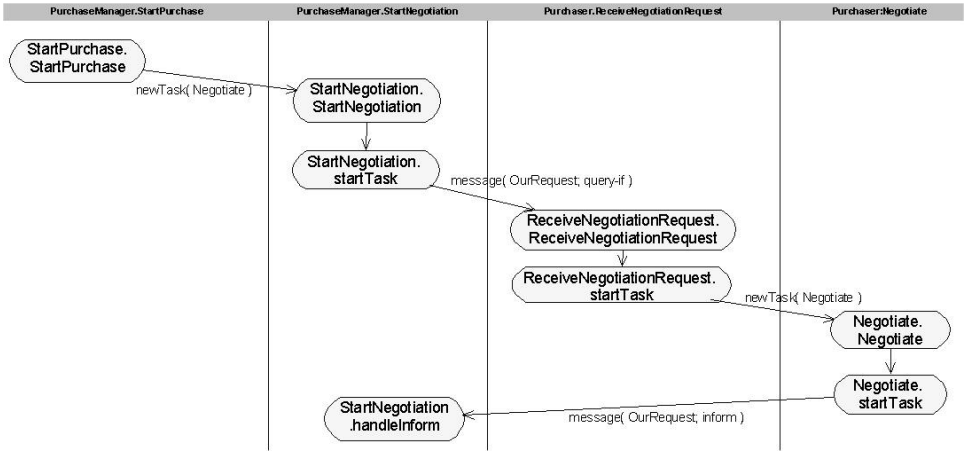

Fig. 2. A more detailed Activity Diagram

agent tasks are naturally mapped to activities and swim lanes can be exploited to gather together the tasks belonging to the same agent. The links between activities can be used to represent message exchanges when they cross swim lanes, and task creation or activation when they connect activities within the same swim lane. The following figure illustrates this diagram with respect to a sample agent interaction. The PASSI methodology [1] has a diagram, named *Multi-Agent Behavior Description*, which uses UML Activity Diagrams in a way similar to the one we are proposing here. However, the PASSI diagram is more detailed than the one above. It uses one swim lane per agent task instead of one swim lane per agent, and shows actual methods of the agent and task classes as activities. Therefore, exploiting the recursive aggregation of activities allowed by UML Activity Diagrams, one can have both the diagram represented in Figure 1, and the more detailed diagram used by PASSI and shown in Figure 2. Observing the PASSI diagram, some issues can be raised about whether modeling methods as activities can be a proper approach. Methods are rather requests for activities and as such they have no state; on the other hand, the diagram in Figure 2 closely matches the actual code that application programmers write when implementing agents according to the "agent class and task class" style. This apparent contradiction is a good example of the peculiar status of this kind of diagram; being at the border between agents and their object-oriented implementation, this diagram shows exactly the slight twist of object-oriented idioms that is to be used to effectively represent and implement agent oriented concepts. The relation between the agent class and the task classes is very specific. Firstly, it is an exclusive aggregation: the agent has control over the life cycle of its tasks, which cannot be accessed from the outside. Tasks, in turn, can use agent internal data as a common repository to share information among them. This means that the methods of the agent and task classes that appear as activities in the diagram of Figure 1 are not to be invoked from the outside, and do not belong to the agent external interface. In Java based implementations, it is customary to use private inner classes to realize agent tasks, so that they are only accessible from their agent class, and can even call its private method. When such a code pattern is followed, the agent-task aggregation is protected at the Java language level. This means that the mutual method calls that occur between an agent class and its task classes are not to be considered ordinary invocation, according to Design by Contract. Rather, they are but a language level representation of the distribution of an agent state between its knowledge base and the conversational state of its ongoing activities. Since the knowledge base and the conversational states have to be coherent, and since their concrete implementation is expressed with an object-oriented language whose only computation device is method invocation, we have to use method calls to synchronize the activities (whose data resides within the task classes) and the knowledge base (contained within the agent class). Therefore, the Activity Diagram depicted in Figure 2 meaningfully details the one in Figure 1, and its representing methods as activities is correct modeling because those methods are special methods that indeed serve as links between the different agent activities.

III. Modeling Deployment Issues in MASs using AUML

The role of the deployment diagram in UML is to show the configuration of run-time processing elements and the software components, processes, and objects that execute on them [5]. In UML 2.0 it is the only form of implementation diagram. In several systems, these aspects are quite evident and a deployment diagram does not add real value to the modeling phase. In these cases it can be useful to produce one a posteriori for documentation completeness. Complex systems with several nodes with significantly different computational responsibilities may benefit from the deployment diagram right from the beginning. Considering the software engineering process in more detail, promoters of the MAS approach generally stress its suitability for heterogeneous distributed systems. Those systems are exactly the ones where

deployment issues can become nontrivial and deserve to be analyzed and addressed with properly designed techniques. Regarding the MAS, the deployment diagram has to represent hosts (servers, front-ends, etc.), resources, physical agents and their acquaintance graphs, and, depending on the framework used in the implementation, MAS platforms. The deployment diagram is very useful to model highly distributed MAS, that is systems in which it is important to visualize the system's current topology and distribution of components and agents, and to reason about the impact of changes on the topology. There is a need of defining new model elements to represent the new entities. Our starting point is AUML and we call the diagram, AUML Deployment Diagram.

A. Multiagent System Architecture and Configuration

The architecture of a MAS is a structure that portrays the different kinds of agents and the relationships among them. The architectural description is studied and fixed when designing the MAS. A configuration is an instantiation of an architecture with a chosen arrangement and an appropriate number of agents. One frozen architecture can lead to several configurations. The configuration is tightly linked to the topology and the context of the place where the MAS is rolled out. The architecture is designed so that the possible configurations cover the different system organizational layouts foreseeable in the context of a project. Agents can be arranged among various machine configurations in order to more effectively use available processing power and network bandwidth. The deployment of a multiagent system therefore is driven by: the system organizational layout, the network topology and the interests area.

B. AUML Deployment Diagram Notation

A deployment diagram is a graph of nodes connected by communication associations (see Figure 3). The most common kind of relationships among nodes are associations that represent physical connections. Generic components are depicted, as they are in UML standard. The concrete agents may be contained within the component instance symbols to indicate that the items reside on the component instances. A concrete agent is rendered as rectangle with a name. Two primary forms of information may be supplied for an agent name: instance, and class. The general form of describing the agent name in AUML is:

```
instance-name : class
```

Agents belonging to the same agent platform are grouped together. An agent platform is a kind of Component, indicating which agents are housed on the platform itself. Every agent platform must have a name that distinguishes it from other platforms; a name is a textual string. Agents are connected to other agents by acquaintance relationships; this indicates that one agent could communicate with the "known" agents by means of interactions protocol. A directed graph is used to show the agent acquaintance graph. The directed graph identifies communication pathways between concrete agents

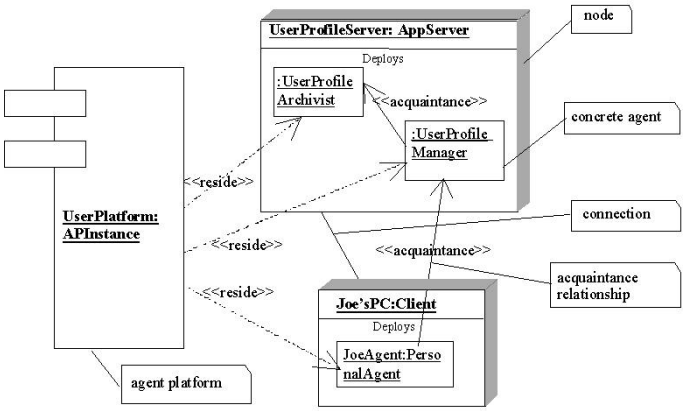

Fig. 3. AUML deployment diagram notation

playing the roles involved in an interaction scenario. A non-directed edge denotes that both concrete agents, playing the roles, know each others.

C. AUML Deployment Diagram Semantics

The aim of this section is to provide complete semantics for all modeling notations used in the AUML deployment diagram. In the following we try to give a precise definition of the terms involved in the deployment diagram, mapping them in UML model elements. For the following, refer to Figure 4. We consider a starting point of our dissertation the definition of the model element "agent" as a stereotype of the metaclass Class [6]. The stereotype extension mechanism provides a way of defining virtual subclasses of UML metaclasses with new metaattributes and additional semantics. What is applicable to a metaclass Class is therefore, by definition, applicable to an Agent Class. Agent Class defines a set of elements, that we call "Agents"[1] (instances of the Agent Class), which have the same structural and behavioral characteristics. Moreover Agent Class is a conceptual element declared in an intensional way as a collection of features and inherits participation in Associations. The stereotype "acquaintance" is applied to an Association between Agent Classes to denote that messages may be sent between their instances. An Artifact is a concrete element[2] that we can define with a good approximation as a structured set of bytes. The implementation of an Agent Class can be memorized in one or more Artifacts. Executable Artifacts can be loaded in memory and be associated to one or more executable threads. If an Artifact contains an implementation of an Agent Class, we can say that the copy, in memory, is able to create concrete elements that implement Agents, instances of the Agent Class itself. An Artifact may constitute the implementation of a deployable Component. We consider Component as a conceptual element.

[1] An Agent has at least one thread of control and runs concurrently with other Agents; Agent Class is therefore a subclass of Class with the attribute "isActive" always true.

[2] In general we use the term "concrete element" to denote an active process, a dynamic library, an instance of an Implementation Agent Class, etc. We use the term "conceptual element" to denote for example an Agent Class.

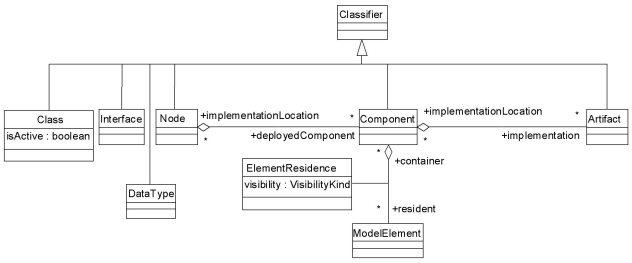

Fig. 4. UML 1.4 Core Package - Classifiers

The Component is defined as a container of one or more Artifacts. The property of the Component, as a conceptual element, is the property of being able to host other conceptual elements, like Agent Classes. At the implementation level, the meaning of the role "resident" (characterizing the Artifact) is that the Executable Artifact, binding to the Component, contains code, which is able to create, at the execution time, instances of the [implementation of the] Agent Classes, resident in the Component. The meaning that we usually give to ComponentInstance is an instance of the Executable Artifact associated to the Component itself. So when we speak of ComponentInstance we refer to its executable part. If a Component hosts conceptual elements, like Agent Classes, a ComponentInstance will host concrete elements, instances of the implementations of these conceptual elements; in this specific case, concrete agents[3]. A NodeInstance is an instance of a Node. A collection of ComponentInstances may reside on a NodeInstance. In the metamodel each ComponentInstance that resides on a NodeInstance must be an instance of a Component that resides on the corresponding Node.

1) Agent Platform and MAS Middleware: Among all the various UML diagrams, the Component and Deployment Diagrams are the ones most tightly related to concrete software infrastructures and middleware; therefore, when defining the AUML Deployment Diagram, it is natural to consider the agent oriented middleware standards and products available, to try and find useful abstractions to model them within AUML. Two UML metamodel elements were considered: Subsystem, which seems more appropriate for modeling the internal view of an agent platform and Component, which seems to be the natural choice on the basis of its characteristic of exhibiting services. Moreover, in the UML metamodel a Component is a subclass of Classifier, and as a Classifier it may also have its own Features and realize Interfaces. This assertion of "individuality" of the Component is important since it means the component not only exposes the interfaces of artifacts deployed in it but it can implement them directly. The choice falls on the Component, seen more as an infrastructure for agents deployed in it rather than as a mere agent container. We therefore define the model element "agentPlatform" as a stereotype of the metaclass Component. It should be noted that the concept of an agent platform does not mean that all agents resident on an agent platform have to be co-located on the same host computer. FIPA envisages a variety of different agent platforms from single processes containing lightweight agent threads, to fully distributed agent platforms built around proprietary or open middleware standards. One AgentPlatform Component, therefore, can span more than one Node, if the agents belonging to this platform are deployed on more than one Node. The model element AgentPlatformInstance represents an instance of an AgentPlatform Component and it can host concrete agents.

[3]The conceptual element Agent denotes an instance that originates from an Agent Class. The term "concrete agent" denotes an instance that originates from an implementation of an Agent Class.

D. Concrete and Situated Agents

A major feature of the multiagent system approach to software development is the reliance on the social level of abstraction: this allowed researchers in the MAS field to take inspiration and leverage results from social sciences, where they deal with complex and dynamically changing systems. However, the shift towards a social perspective in multiagent systems should not suggest forgetting the main attributes of the single agent, namely autonomy and situatedness. When inserted into a society, each member agent becomes situated in an hybrid environment, arising partly from social and institutional entities and partly from entities external to the agent society. The diagrammatic representation of a concrete MAS should be able to fully depict this hybrid situatedness, showing agents and their social and natural environment as a whole. The social aspect of agent situatedness can be captured by an oriented graph connecting agents with arcs. This graph is called "acquaintance" graph. In simple client/server systems, usually it is assumed that clients know the server beforehand but the server does not know a client until it is contacted by it. These strict assumptions, common in multi-tier client/server systems, make the acquaintance graph trivial, which explains why it is generally not included in the system diagrams, but a MAS architecture can result in arbitrary acquaintance graphs, so they have to be explicitly represented. From the above considerations it follows that even if the deployment diagram is part of the architectural models and its scope is to model the static deployment view of a system, some aspects of MAS deployment diagrams are related to the behavioral models. The AUML deployment diagram therefore must be more expressive than the corresponding reference UML deployment diagram, and must include the acquaintance relationships between agents.

IV. Impact of Patterns in the Implementation

Patterns for MAS have been studied since many years [7] [8] and the interest about them is still very high and new works often appear [9]. Despite the number and differences of these works two issues have not received a definitive solution: the definition of pattern and the implementation aspects. About the definition, the different works, sometimes, propose new arguments that try to introduce the concept of agent in it. We see a pattern as a recurrent problem and an

associated solution expressed in terms of portions of design and the related implementation code. Four different possible granularities can be identified for our patterns: the service pattern is related to the composition of two or more agents, the component pattern looks at a single agent as the solution of the problem, the task/behavior pattern regards a significant portion of the agent behavior and the action pattern is a simple action done by an agent. Three different steps should be considered in reusing patterns: the identification, the solution design at the conceptual, social level and, finally, the implementation. About pattern identification we think that patterns rely on the problem domain rather then in the solution one and in fact our definition starts from the 'classical' one proposed by Christopher Alexander [10]: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core solution to that problem". As a consequence, we can identify a pattern wherever a problem is present. Typical scenarios for pattern identification are:

- collaborations among agents. The scope of these patterns can be related to two or more different agents and therefore it spreads across a large part of the system;
- specific vocation of a single agent (for example the ability to offer some kind of proxy service). This is usually limited to the single specific agent and therefore can be considered as a kind of component scoped pattern;
- agents' behaviors or roles. Patterns of this kind are related to minor portions of the agent capability (for example the possibility of communicating with a specific interaction protocol) and therefore are smaller (behavior-sized) than the previous one.

In the next two subsections we will discuss the other two aspects of patterns: the design with the related modeling problems and the implementation issues.

A. Using patterns during the design

There is an evident duality in software between its structure and dynamic behavior; in dealing with patterns this has been well captured in the approach of Gamma et al. [11]. They classified patterns according to two criteria: purpose and scope. With purpose the authors refer to what the pattern does and with scope they separate patterns that apply to classes or object. The same duality should be the guidance in representing patterns for agents. We found particularly useful to represent the structure of the pattern with a class diagram reporting the agent base class and all the agent's behaviors as different classes. This diagram results to be very near to the implementation since agent and behaviors (classes in the diagram) are represented with their attributes and operations. The pattern behavior representation is done with an activity diagram. In Figure 5 we can see an example of the proposed notation applied to the *Request Participant* pattern that is a behavior-sized pattern. The diagram is divided in two different swim lanes: the right one describes the flow of control within the pattern, the left one contains activities belonging to other agents that interact with the previous one. Exchanged information is represented as an object.

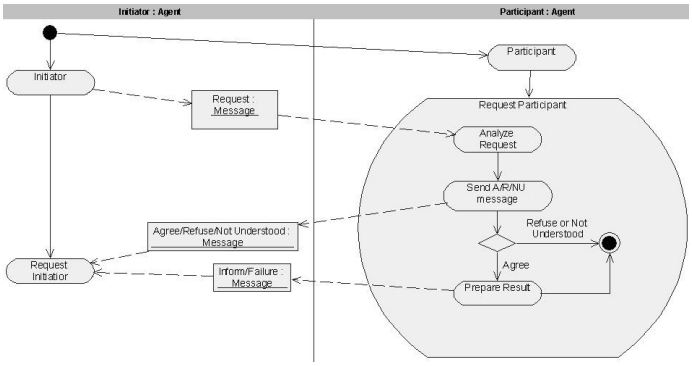

Fig. 5. The Request pattern behavior description

The *Request Participant* pattern, when applied to an agent adds to it the capability of playing the participant role in a communication based on the FIPA Request communication. As it can be seen, an external agent begins the communication sending the *Request* message (represented by the object "Request:Message"). The *Participant Agent* (right-hand swim lane) receives the message, analyzes its content (this part is domain specific and therefore it is not part of the pattern) and decides to send one of the three possible answers (Agree, Refuse, and Not Understood). This diagram is particularly expressive and provides the designer with a simple but complete description of the pattern functionalities. In the structural description, we represent the pattern in terms of the two constitutive elements of an agent: the agent itself and its behaviors. The first is the agent base class obtained specializing a generic *AgentShell* class (that, for example, represents the JADE *agent* class or FIPA-OS *Fipaosagent* class) while the behaviors are obtained specializing the *TaskShell* class (that for example in JADE stands for the *behavior* class and in FIPA-OS for the *Task* class) and are identified by the "Task" stereotype. We use to tightly relate (in terms of names used) the operations of these elements to the activities shown in the dynamic pattern representation in order to allow an easier understanding of the pattern functionalities.

B. Patterns Implementation

In the cited Gamma's approach, the implementation of the suggested patterns was provided only in a few cases with partial solutions in one specific language. This is obviously justified by the broad audience and the generality of that work but we think it remains a limit in a real application context. Working with agents, it exists a natural solution to this observation that consists in restricting the possible implementation platforms to the FIPA ones. This hypothesis has a great simplifying effect since it reduces the implementation language to JAVA, and as another consequence, all the platforms share a very similar structure and comparable services. At this point an obvious goal will be the definition of an architecture that allows the pattern platform-independent definition though maintaining the possibility of easily producing the pattern code for each specific platform. This

architecture already exists and it is part of an OMG standard, it is the MDA (Model Driven Architecture) [12]. Its objective is separating the specification of the system operations from the implementation details (the way the system interacts with its platform). According to the MDA specifications, the system is to be described with three different and successive models. The first model is the Computation Independent Model (CIM) that is a sort of domain model that totally neglects all the details about the system structure but describes the situation in which the system will be used. The second model is the Platform Independent Model (PIM) that represents the system implementation without dealing with its use of the deployment platform (often modeled as a technology-neutral virtual machine). The last model is the Platform Specific Model where details on the way of implementing the system with a specific platform are provided. A similar approach has been already applied to agents with interesting results [13]; it starts considering a design representation stage of the pattern functionalities and initial structure that is usually performed with the diagrams described in the last subsection; this is similar to the MDA Context Independent Model. These representations, are related to a meta-pattern level, that is platform independent and contains all the elements that are common to patterns of the different environments. For example meta-patterns refers to class constructors, mother-classes from which agents and their standard elements inherit their behavior, setup and shutdown methods and so on. Meta-patterns are described using XML and can be regarded as an MDA Platform Independent Model. From this model, applying an XSL transformation we substitute the meta-level placeholders with the concepts used in the selected platform and if the case the values introduced by the designer (for example the specific name of the agent or some parameters). In the resulting XML file, an agent is described within an *Agent* tag and its properties (attributes and operations or tasks) are represented as inner elements of that structure. This is an MDA Platform Specific Model. The last phase consists in the code generation and this can be done with another XSLT application. This is possible because the pattern at this stage intrinsically represents an implementation viewpoint. It is, in fact, possible to look at the source code as one of the possible abstractions representing an agent: it is at an intermediate layer between agent design and agent deployment. The use of XSLT enables code generation for different FIPA-compliant frameworks by only changing the transformation sheet. At this stage the JAVA skeleton of the agent (and it's tasks) is complete. While this result is considered the final target by several CASE tools we think that (with agents) it is possible to go beyond this goal introducing significant parts of the inner code of methods. Specific portions of code (related to a precise action) are therefore stored in a database and the correct one, for each method, is selected referring to the value of a specific tag in the XML method description. The final result is therefore that starting from an high level design representation of the solution that the pattern offers for some problem, we obtained a platform independent description. From that we "instantiated" the platform specific model that can be used to generate code skeletons. These, completed with portions of code taken from a repository constitute a real executable implementation of the pattern.

V. Conclusions

The aim of the paper is to try to cope with the important issues connected to the transition from the design phase to the implementation phase. At first glance the work presented may seem composed of three loosely connected parts; in reality this is due to the nature of the problem. As a matter of fact moving towards the implementation phase means solving several issues connected with the refinement of the design models in order to move towards the code, with the deployment of the system and with the necessity of reducing the prototype implementation time. But, there is more. Indeed more work needs to be done; other issues should be tackled. An issue, that needs to be addressed, deals with non-social situatedness. An agent's surroundings comprise not only its acquaintances, but also several non-agent entities such as the resources it uses and manages, the events it can perceive, the concrete actions it can perform. Another issue deals with the testing of MAS. These further studies are left as subject for future work.

References

[1] M.Cossentino and C. Potts, "A case tool supported methodology for the design of multi-agent systems," The 2002 International Conference on Software Engineering Research and Practice. Las Vegas (NV), USA: SERP'02, June 24-27 2002.

[2] Agent UML - AUML Home Page. Available at http://www.auml.org.

[3] J. Odell, H. van Dyke Parunak, and B. Bauer, "Extending uml for agents," in *Proc. of the 2nd Int. Workshop on Agent-Oriented Information Systems*, G. Wagner, Y. Lesperance, , and E. Yu, Eds., Berlin. iCue Publishing, 2000.

[4] FIPA Modeling Technical Committee. Available at http://www.fipa.org/activities/modeling.html.

[5] OMG, "Uml2.0 superstructure, 2nd revised submission," March 2003, object Management Group, document ad/03-03-03.

[6] B. Bauer, "Uml class diagrams: Revisited in the context of agent-based systems," in *Agent-Oriented Software Engineering (AOSE)*, vol. Proc. of Agents 2001, Montreal, 2001.

[7] Y. Aridor and D. B. Lange, "Agent design patterns: Elements of agent application design," in *Second International Conference on Autonomous Agents*, Minneapolis, 1998, pp. 108–115.

[8] E. A. Kendal, P. V. M. Krishna, C. V. Pathak, and C. B. Suresh, "Patterns of intelligent and mobile agents," in *Second International Conference on Autonomous Agents*, Minneapolis, 1998, p. 9299.

[9] J. Lind, "Patterns in agent-oriented software engineering," in *AOSE Workshop at AAMAS 2002*, vol. Bologna (Italy), 2002.

[10] C. Alexander, *The Timeless Way of Building*. Oxford University Press, 1979.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994.

[12] OMG, "Guide version 1.0.1," June 2003, object Management Group, document omg/2003-06-01.

[13] M. Cossentino, L. Sabatucci, and A. Chella, "A possible approach to the development of robotic multi-agent systems," in *IEEE/WIC Conf. on Intelligent Agent Technology (IAT'03)*, vol. Halifax (Canada), 2003.