

PASSI: a Process for Specifying and Implementing Multi-Agent Systems Using UML

M. Cossentino
DIE, University of Palermo
V.le delle Scienze
90128 Palermo (Italy)
+39-091.6566273
cossentino@unipa.it

C. Potts
CoC, Georgia Institute of Technology
801, Atlantic Drive
Atlanta (GA) 30332-0280 (USA)
(+1)(404) 894-5551
potts@cc.gatech.edu

ABSTRACT

Multi-agent systems (MAS) differ from non-agent based systems because agents are intended to be autonomous units of intelligent functionality. As a consequence, agent-based software engineering methods must complement standard design activities and representations with models of the agent society. Some methods coming from artificial intelligence community address social knowledge and relationships but have high-level design abstractions as their end points. This paper describes PASSI a step-by-step requirement-to-code method for developing multi-agent software that integrates design models and philosophies from both object-oriented software engineering and MAS using UML notation. The method has evolved through several stages; it has been previously applied in the synthesis of embedded robotics software and we are currently exploring its applications to the design of various agent-based information systems.

Keywords

Agent-based software engineering, multi-agent systems, UML, design, software architectures.

1. UNIQUENESS OF AGENT-BASED SOFTWARE ENGINEERING

An important factor in modeling software is *fidelity*, which Robbins et al. [5] define as the distance between a model and its implementation. Low fidelity models are problem-oriented; whereas high fidelity models are more solution-oriented.

Several proposals exist for methods and representations for agent-based systems [1][2][3][4][7][28]. Because research in agent interaction is still an open topic, some proposed representations involve abstractions of social phenomena and knowledge [1][7][28] and therefore have low fidelity. Others, however, address implementation issues and have higher fidelity models (e.g. [2][3][4]).

One response to these developments is to treat agent-based software the same as other types. That is, there is no need for methods or representations specifically for agent-based software. We reject this view because it is more natural to describe agents in psychological and social language. For example, Yiu and Li [6] say that “an agent is an actor with concrete, physical manifestations, such as a human individual. An agent has

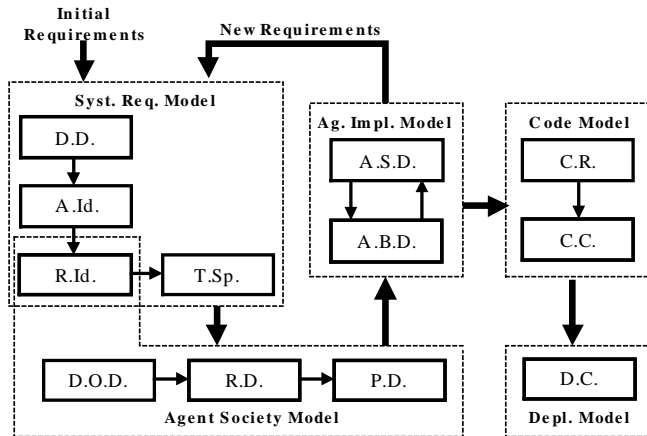
dependencies that apply regardless of what role he/she/it happens to be playing”. Jennings [1] defines an agent as “an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives”. Wooldridge and Ciancarini see the agent as a system that enjoys the properties autonomy, reactivity, pro-activeness, and social ability [16].

Thus, multi-agent systems (MAS) differ from non-agent based systems because agents are intended to be autonomous units of intelligent functionality. As a consequence, agent-based software engineering methods must complement standard design activities and representations with models of the agent society.

Two further responses are possible, that we also reject in their extreme forms. They argue that agents differ from conventional software but disagree about where the differences are. One view, held by proponents of low-fidelity representations is that what distinguishes agents are their social and epistemological properties, and it is only these that require different abstractions. The complementary argument made by proponents of high-fidelity representations is that it is the deployment and interaction mechanisms that make the difference. According to DeLoach [2], “an agent class is a template for a type of agent in the system and is analogous to an object class in object-orientation. An agent is an actual instance of an agent class”, and “... agent classes are defined in terms of the roles they will play and the conversations in which they must participate”.

A designer may want to work at different levels of detail when modeling a system. This requires appropriate representations at all levels of detail or fidelity and, crucially, systematic mappings between them.

This paper describes PASSI a step-by-step requirement-to-code method for developing multi-agent software that integrates design models and philosophies from both object-oriented software engineering and MAS using UML notation. It is closer to the argument made above for high-fidelity representations, but addresses the systematic mapping between levels of detail and fidelity. PASSI has evolved from a long period of theory construction and experiments in the development of embedded robotics applications (see [11],[13],[14],[15]). Its precursor, AODPU has been applied in the synthesis of embedded robotics software and is the basis of teaching materials in agent-based software engineering [12].



Key:

- D.D. – Domain Description
- A.ID. – Agents Identification
- R.Id.– Roles Identification
- T.Sp. – Task Specification
- A.S.D. – Agents Structure Definition
- A.B.D. – Agents Behavior Description
- D.O.D. – Domain Ontology Description
- R.D. – Roles Description
- P.D. – Protocols Description
- C.R. – Code Reuse
- C.C. – Code Completion
- D.C. – Deployment Configuration

Figure 1. The models and phases of the PASSI methodology

In PASSI, an *agent* is a significant unit of software at both the abstract (low-fidelity) and concrete (high-fidelity) levels. According to this view, an agent is an instance of an agent class that is the software implementation of an autonomous entity capable of pursuing an objective through its autonomous decisions, actions and social relationships. An agent may occupy several functional *roles* during interactions with other agents to achieve its goals, where a role is the collection of *tasks* performed by the agent in pursuing a sub-goal. A *task*, in turn, is defined as a purposeful unit of individual or interactive behavior.

The next section provides an overview and justification for PASSI. Section 3 gives a detailed description of the steps and the use of UML notations within each of them. Finally, in Section 4 we discuss our experiences with the method and future plans.

2. THE PASSI METHODOLOGY

2.1 A Quick Overview

In the following sections, we will introduce the elements of the PASSI methodology. It is composed of five models (Figure 1) that address different design concerns and twelve steps in the process of building a model¹.

In PASSI we use UML as the modeling language because it is widely accepted both in the academic and industrial worlds. Its

extension mechanisms (constraints, tagged values and stereotypes) facilitate the customized representation of agent-oriented designs without requiring a completely new language. Within the agent design community, a variant or dialect of UML, AUML, has been proposed as a standard [26], which goes beyond the use of standard UML extension mechanisms but still stays within the representational philosophy of object orientation, use cases, temporal interactions and modular architectural structure. We adopt these extensions below without detailed comment.

The models and phases of PASSI are:

1. **System Requirements Model.** An anthropomorphic model of the system requirements in terms of agency and purpose. Developing this model involves four steps:
 - Domain Description (D.D.): A functional description of the system using conventional use-case diagrams.
 - Agent Identification (A.Id.): Separation of responsibility concerns into agents, represented as stereotyped UML packages.
 - Role Identification (R.Id.): Use of sequence diagrams to explore each agent’s responsibilities through role-specific scenarios.
 - Task Specification (T.Sp.): Specification through a use case diagram and auxiliary descriptions of the capabilities of each agent.
2. **Agent Society Model.** A model of the social interactions and dependencies among the agents involved in the solution. Developing this model involves three steps in addition to part of the previous model:
 - Role Identification (R.Id.). See the System Requirements Model.
 - Domain Ontology Description (D.O.D.). Use of class diagrams and OCL constraints to describe the knowledge ascribed to individual agents and the pragmatics of their interactions.
 - Role Description (R.D.). Use of class diagrams to show distinct roles played by agents, the tasks involved that the roles involve, communication capabilities and inter-agent dependencies.
 - Protocol Description (P.D.). Use of sequence diagrams to specify the grammar of each pragmatic communication protocol in terms of speech-act performatives.
3. **Agent Implementation Model.** A model of the solution architecture in terms of classes and methods, the development of which involves the following steps:
 - Agent Structure Definition (A.S.D.). Use of conventional class diagrams to describe the structure of solution agent classes.
 - Agent Behavior Description (A.B.D.). Use of activity diagrams or statecharts to describe the behavior of individual agents.

¹ Authorities differ about whether PASSI stands for “Process for Agent Societies Specification and Implementation” or is from the Italian word for “steps.”

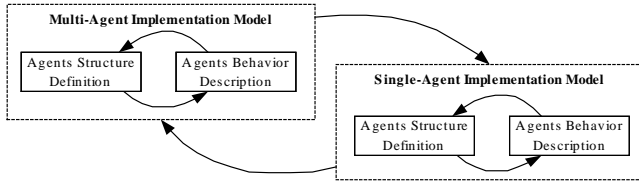


Figure 2. The agents' implementation iterations

4. **Code Model.** A model of the solution at the code level requiring the following steps to produce:
 - Code Reuse Library (C.R.). A library of class and activity diagrams with associated reusable code.
 - Code Completion Baseline (C.C.). Source code of the target system.
5. **Deployment Model.** A model of the distribution of the parts of the system across hardware processing units, and their migration between processing units. It involves one step:
 - Deployment Configuration (D.C.). Use of deployment diagrams to describe the allocation of agents to the available processing units and any constraints on migration and mobility.

2.2 Iterations

In common with widely accepted object-oriented methods (e.g. the U.D.P. [9]), PASSI is iterative. In contrast, most agent-based software methods (e.g., [7]) are sequential and top-down. There are two kinds of iteration in PASSI relating to requirement elicitation and MAS implementation, respectively. The first type of iteration is driven by new requirements, and involves dependencies between three models: the requirement analysis and agent society models on one hand, and the agent-implementation model on the other. We exclude the code model from requirements-driven iteration. Even though the code changes, we can abstract from the code level and generate the code using a CASE tool.

The second iteration takes place inside the agent implementation model. Two different levels of iteration compose it (Figure 2). The outer iteration arises from dependencies between the multi-agent and single-agent perspectives. In the multi-agent perspective, we describe the structure of the agents in a single class diagram in terms of tasks (classes internal to the agent class); the behavior is depicted with an activity diagram showing a flow of events, methods (often invoked by events) and messages between agents. At the single-agent level, however, a class diagram is drawn for each agent; all the details of the structure are introduced both for the agent class and the task subclasses. The behavior of each agent at this level can be specified in sequence and collaboration diagrams.

Within the outer iteration, there is another that arises from dependencies between the structural and behavioral perspectives at both the multi-agent and single-agent levels.

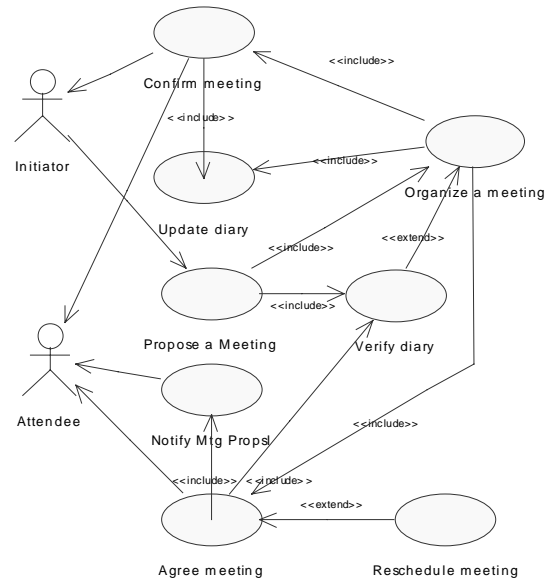


Figure 3. The details of the meeting scheduler functionality of the Personal Assistant (PA) example

3. THE PHASES OF THE PASSI METHODOLOGY

In this section we discuss the steps of PASSI, using the widely implemented standard FIPA architecture [19],[20] as an implementation environment. The example MAS referred to throughout is a Personal Assistant (PA) [21] that is often used by the FIPA community for comparisons and benchmarking.

3.1 Domain Description Phase

Requirements elicitation may involve the elaboration of system goals [6][29][30][31], and this is even more appropriate in MAS design, which are teleological in nature. DeLoach et al. [2] describe requirements using goals, which they refine into a system description using use-case and sequence diagrams. We think that in so doing, they perform an abstraction passing from requirements to goals and then they come back to the requirements level with use cases. These changes, which could be useful in understanding some systems, more likely introduce errors of redundancy and inconsistency in the design.

Because of these potential problems, and despite a commitment to the use of goals in requirements engineering [29][30][32], we have chosen a different approach. In PASSI, we describe the requirements directly in terms of use-case diagrams that are obtained either through standard requirements elicitation precursors to object-oriented methods [9][10] or through the informal application of a scenario-based teleological method such as GBRAM [30] or ScenIC [32].

As a result, a functional description of the system is provided through a hierarchical series of use-case diagrams, the uppermost of serves as a context diagram. Scenarios of the detailed UC diagrams are explained using sequence diagrams.

3.2 Agent Identification Phase

One of the central issues in PASSI is the identification of agents early in the development process. If agents were merely MAS implementation components, their identification during the analysis of requirements would be premature. However, if a MAS is a society of anthropomorphic entities, it is more reasonable to divide the description of required behaviors into loci of responsibility from the start. This is particularly so in cases where the “system” is a heterogeneous society of intended and existent agents that in Jackson’s terminology can be “bidden” or influenced but not deterministically controlled [33].

Agent identification starts from the use-case diagrams of the previous step. (See Figure 3 for an example: the scheduling capability of the FIPA Personal Assistant.) According to our definition of agents given in Section 1, it is possible to see an agent as a use case or package of use cases. As is standard, we represent external entities interacting with our system (people, devices, conventional software systems) as actors.

Less standard are our assumptions about agent interaction and knowledge:

- Interactions between agents and external actors consist of *communication* acts;
- An agent acts to achieve its objectives on the basis of its local knowledge and capabilities;
- Each agent can request help from other agents;
- An agent’s knowledge can increase through communication with other agents or exploration of the real world.

It is important to underline that the social relationships among different agents in the use case diagram are characterized by a “communication” stereotype. Standard stereotypes for non-MAS (for example “extend,” “include” or “generalize”) are not typical.

Communications among agents may differ significantly from those between agents and other entities. Among agents, communication patterns follow a specific protocol and communication content utilizes a specific ontology. Communications between external actors and agents are not necessarily implemented like that. For example, agents could receive information directly from actors through sensing devices, whose communication protocols are constrained by the hardware devices and their drivers.

Starting from a sufficiently detailed diagram of the system functionalities description (Figure 3), we group one or more use cases into different packages in a new diagram (Figure 4). The name of the package is the name of the agent. Each package defines the functionalities of a specific agent.

Relationships between use cases of the same agent follow the usual UML syntax and stereotypes, whereas relationships between use cases of different agents are stereotyped as “communication.”

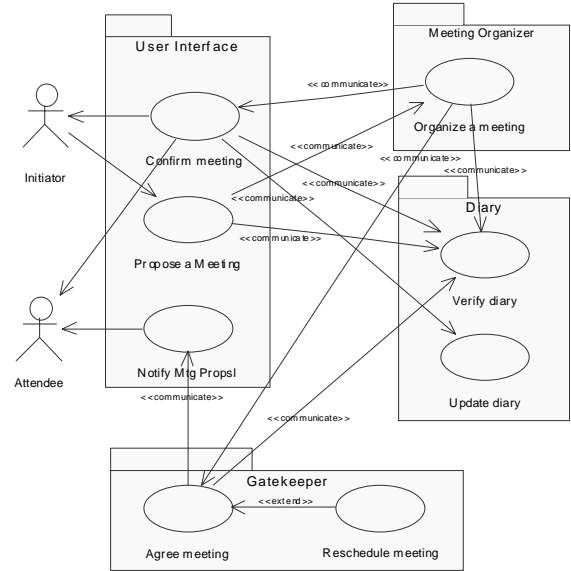


Figure 4. The Agents Identification diagram obtained from the requirements described in the previous phase.

3.3 Role Identification Phase

This phase occurs during requirement analysis because it describes the MAS functionally rather than structurally. We are concerned with an agent’s externally visible behavior, so any representation of its internal behavior is approximate. A role is also a social concept. Consequently, we consider role identification also to be part of the agent society model (see Figure 1).

Role identification produces a set of sequence diagrams that specify the most important scenarios from the agents’ identification use case diagram. In general, one sequence diagram is drawn for each different path (Figure 5). It is important to investigate all the paths involving inter-agent communications, but fortunately representational constraints are useful as analysis guidelines: (1) Such communication paths are shown in the A.Id. diagram by the presence of a relationship between two agents; (2) Each such relationship may belong in several scenarios (i.e. paths among use cases); (3) For each relationship in a specific scenario of the A.Id. diagram, there is at least one message in the sequence diagram of the R.Id. phase.

The several roles that an agent can play are introduced as objects in the appropriate sequence diagram. An agent may participate in several scenarios playing distinct roles in each, and may even appear as more than one object in a single sequence diagram (e.g. the Diary agent in Figure 5).

Inter-object messages in the sequence diagram either represent events generated by the external environment or system, or parts of communications between the roles of one or more agents. A message specifies what the role is to do and possibly data to be provided or received. Data and tasks that are mentioned in the sequence diagram are specified in more detail later in D.O.D. and R.D. diagrams, respectively. For each message in a sequence diagram, a corresponding relationship should appear in the

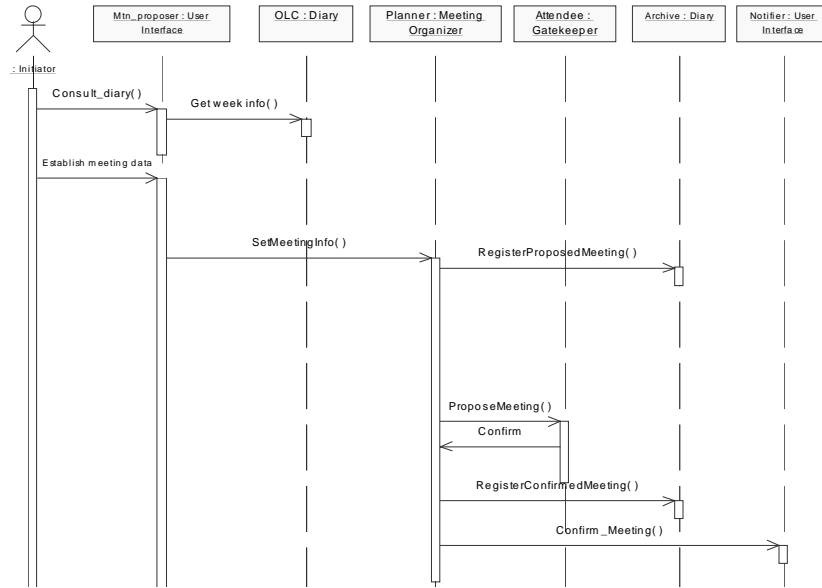


Figure 5. The Roles Identification Diagram for a scenario in which the initiator proposes a meeting and an attendee agrees.

D.O.D. diagram together with a description of knowledge exchanged or shared.

3.4 Task Specification Phase

While the use case diagram in the Agent Identification phase represents multi-agent collaboration (the whole agent society), in the Task Specification phase we look at one agent at a time, drawing one use case diagram per agent. The two forms of use-case diagram are complementary and non-redundant. In task specification, the scope of a diagram is limited to capabilities of the agent viewed as an asocial problem solver [27].

In this diagram, each use case represents a task that the agent can do. The UC diagram collects all the capabilities of the agent without temporal relations, and showing only functional dependencies between different facets of the agent. That is, we summarize what the agent is capable of doing, ignoring information about roles that an agent plays when doing particular tasks.

The relationships between the use cases represent within-agent communications between tasks. These communications are not speech acts; they are often signals addressing the necessity of beginning an elaboration that is a work of a specific task or signals to delegate another task to do something. But it is also possible that some relationships represent a message from an instance of an agent performing a certain role to another one performing a different role. Interacting agents are represented as actors.

The behavior represented by each task can be specified in more detail by sequence diagrams, activity diagrams or semi-formal text with fields for preconditions, triggers, etc.

Information needed to produce this diagram comes from the R.Id. sequence diagrams. In Figure 6 we see the tasks of a MeetingOrganizer agent in the PA example. One listener task is

introduced in order to pass incoming communications to the appropriate task. A task is introduced for each incoming message in the R.Id. sequence diagrams, and is connected to the listener. Similarly, a task is introduced for each outgoing message of the R.Id. sequence diagrams, with a “communicate” relationship emanating from this use case to the interacting agent.

In addition to listeners, the designer may have to introduce other tasks that are not visible in an exterior view of the agent. These could arise spontaneously from the description of the tasks identified in the external behavior of the agent or from the opportunity of better decompose the parts of the agent (for example to facilitate reuse, perform calculations, access external devices).

3.5 Domain Ontology Description Phase

An agent-based system may achieve its semantic richness through explicit ontologies, or domain-specific terminologies and theories. In order to detail the resulting ontology of the solution we introduce the D.O.D. (Domain Ontology Description) phase in which we use a class diagram describing the involved entities (agents as well as external actors), their knowledge and their communication relationships.

Following other authors ([23][24]), we express this information using class diagrams. We describe an agent’s knowledge types through the attributes of each class representing an agent, and we represent the content of the messages through associations.

As an example we can consider (Figure 7) that the *Diary* agent has a *calendar* attribute containing the scheduled (and confirmed) appointments of the user, and it has a *proposed_events* attribute that lists the appointments that are not confirmed. The *Diary* agent provides this information to the *User Interface* agent that shows it to the user when he or she wants to schedule a new meeting or to consult a weekly agenda.

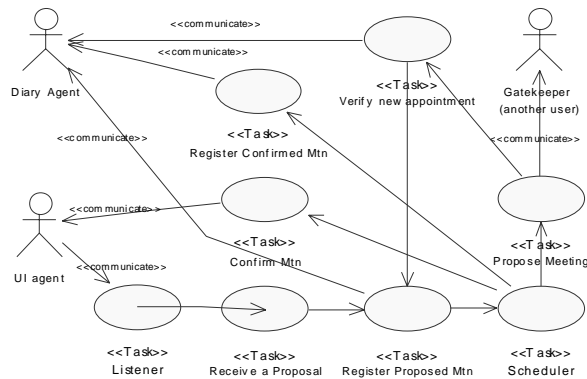


Figure 6. The tasks of an agent (Meeting Organizer) of the PA example.

This diagram can also be helpful in defining the content of the messages and the attributes of the classes for the A.S.D. diagrams. In designing this diagram we start from the results of the A.Id. phase. A class is introduced in this diagram for each identified agent, and an association is introduced for each message of a certain agent (ignoring for the moment distinctions between an agent’s roles). Obviously it is necessary to introduce a proper data structure in each agent in order to store the received/produced data.

3.6 Role Description Phase

The scope of this phase is modeling the lifecycle of each agent, looking at the roles it can play, at the collaboration that it needs, and the communications in which it participates. The R.D. diagram is also where we introduce all the rules of the society (organizational rules, [7]), laws of the society and the domain in which the agent operates (e.g. trade laws) and the behavioral laws considered by Newell in his “social level” [27]. Although these rules could be expressed in plain text we prefer OCL in order to have a more precise, formal description.

The information presented in this class diagram derives from the output of the R.Id. and T.Sp. phases. From the former, we deduce information about the existing roles of an agent and changes of role; from the latter, we obtain the names of tasks belonging to each role. (Recall that in this last diagram we have all the tasks required by an agent, some of which are absent from the sequence diagrams of the R.Id. phase.) This diagram therefore presents information that is already in the design but which is now assembled from a different viewpoint. The role is the focus and related data are as follows:

- The agent class playing the role;
- The tasks involved in it (it is possible that the same task is used in different roles of the same agent);
- The communications between the different roles, some of them involving messages between different agents, and others remaining within the same agent;
- The dependencies between different agent roles.

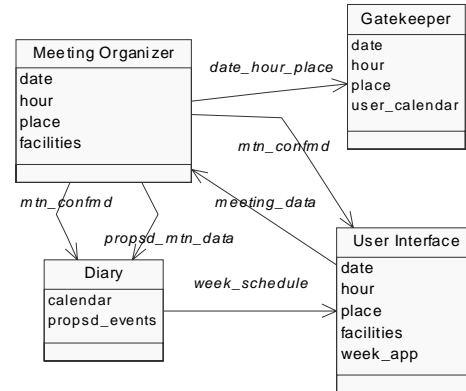


Figure 7. The ontology of the PA system.

Dependencies in PASSI are similar to those in i^* [6]. In i^* , the Strategic Dependency Model is a dependency graph, in which each node represents an actor, and each arc represents a directed dependency from one actor to the another.

Dependencies are a direct consequence MAS cooperation (or competition), but do not always hold when a MAS runs. Agents are autonomous and could therefore refuse to provide a request for service. For this reason, the designer needs a schema in which it is possible to analyze these dependencies and, if necessary, provide alternative ways to achieve the goal.

We consider the following kinds of dependency:

- Service dependency – An actor depends on another to accomplish a goal or perform an activity. The other actor may provide or deny the required service. This dependency unifies the goal and task i^* .
- Resource dependency - An actor depends on another for the availability of an entity. This is the same as i^* .
- Soft-Service and Soft-Resource dependency – Soft versions of the Service and Resource dependencies for which the requested service or resource is useful or desirable for accomplishing the requesting actor’s final goal, but not essential for it. This further classifies i^* ’s “softgoal” dependencies.

Unlike the special notation of i^* , in PASSI we introduce dependencies into the role diagram. However, we need to introduce some further syntax (see Figure 8) and stylistic conventions as follows:

- Classes are roles. Each role class is designated with the name of the agent class to which it belongs and the name of the role separated by a colon.
- Methods of these classes correspond to tasks of the agent.
- Communications between roles of different agents are shown with a solid line (i.e. a UML association) and named by the name of the used protocol. Communications between roles of the same agent are shown with solid line without any protocol name. The direction of a communication is shown by an arrow that points at the receiver of the first message of the conversation.

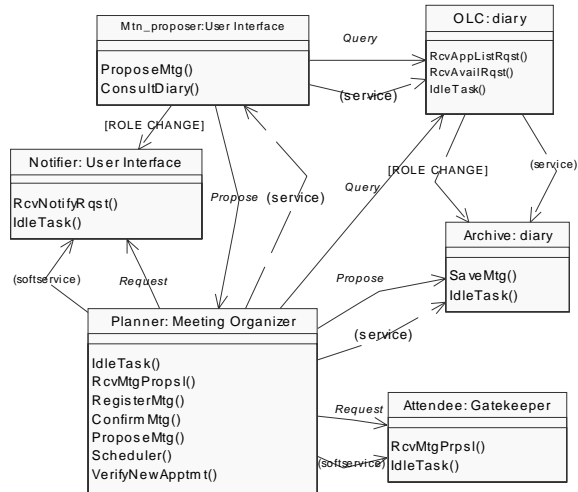


Figure 8. The Roles Description diagram for the Personal Assistant example

- A change of role by an agent is shown by a dashed line (i.e. a UML flow) from the old role to the new one with the attribute ‘(role changed)’ on the line.
- A service dependency is shown by a dashed line (i.e. a UML dependency), with the attribute ‘(service)’ on the line. The direction is towards the ‘server’ role.
- A resource dependency is shown by a dashed line (i.e. a UML dependency) with the attribute ‘(resource)’ on the line. The direction is towards the ‘server’ role.
- Soft-Service and Soft-Resource dependencies are shown by dashed lines (i.e. UML dependencies), with the attribute ‘(softservice)’ or ‘(softresource)’ on the line. The direction is towards the ‘server’ role.

Some sample rules can be enumerated for this diagram:

- One class is drawn for each role of the same agent class of the R.Id. diagrams.
- For each communication of the R.Id. sequence diagrams a task is introduced in the R.D. for the role that has received the first message of that communication. According to the FIPA architecture, each agent that receives a communication needs a listener task (often called *IdleTask*) to properly handle arriving messages.
- The role from which the first message is sent needs a communication task in order to ask for the service from the other agent.
- All the tasks of the previous two steps are also present in the T.Sp. diagrams. Moreover in these we can find all the others not related to communication. These must be introduced in the right role. At the end, each task of the agent in the T.Sp. phase should be present in at least one role.

For example in Figure 8 we can see that the *Planner* role of the *MeetingOrganizer* agent receives a communication from the

Mtn_proposer role using the *Propose* standard FIPA protocol. *IdleTask*, the listener, passes the message to an appropriate task (*RcvMtgPrpsl*). This role can also communicate with other entities, for example with the *Attendee* using the *Request* standard FIPA protocol and the *ProposeMtg* task. It is also possible to find the same task in different roles (see *IdleTask* of the *Diary* agent).

3.7 Protocols Description Phase

In the previous phase a protocol has been chosen for each communication. In the example of Figure 8 the protocols used are all standard FIPA protocols [25]. In this specific case the documentation of the communication acts is part of the FIPA specification, usually in form of AUML sequence diagrams [8], and therefore the designer does not need to specify them. In some cases, however, existing protocols are insufficient, and a dedicated protocol should be designed; this can be done using the same approach of the FIPA documents (AUML sequence diagrams).

3.8 Agents Structure Definition Phase

As discussed in section 3.2, this phase is strongly related to the Agent Behavior Description phase through a double level of iteration. The outer level of iteration deals with the switching from multi-agent models to single-agent models. The second level of iteration represents the process of adding parts of the structure to an agent and using them in its behavior.

The Agent Structure Definition phase is composed of several class diagrams. We begin the design of the system that will be implemented with them. The diagrams logically belong to two different levels of detail in the design: the multi-agent and the single-agent. In the first we focus on the general architecture of the system and therefore in it we can find agents and their tasks. In the second level we look at each agent’s internal structure, showing all the attributes and methods of the agent class and of its internal task classes.

3.8.1 Multi-Agent Structure Definition Diagram

One diagram represents the MAS as a whole (Figure 9). (Of course, several diagrams could be used if the number of classes requires this.) One class is introduced in this diagram(s) for each agent of the A.Id. phase. The tasks of each agent are shown in the operations compartment and detailed only by name.

This diagram represents the structure of the agents in a simple compact form. Introducing attributes or other details in it could result in an unreadable schema.

3.8.2 Single-agent structure definition diagrams

In this diagram (Figure 10) we address the internal structure of the classes composing the agents. We produce one diagram for each agent, in which we introduce the agent class and its tasks as inner classes.

We introduce all the methods that are needed in the different classes that have not been identified previously. These include constructors and the *shutdown* method required by the FIPAOS environment. The tasks devoted to exchanging information need

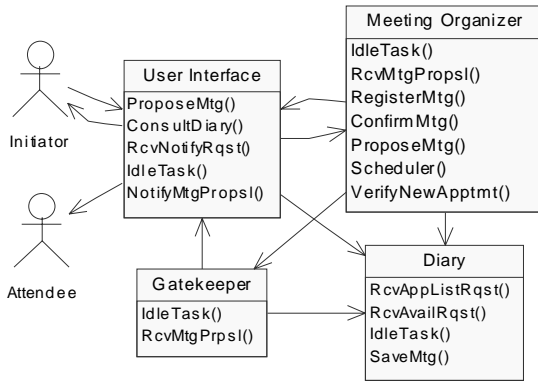


Figure 9. The Multi-Agent Structure Definition diagram for the Personal Assistant example

specific methods to deal with the communication events. (E.g. the *handleAcceptProposal* method in the *RegisterMtg* task, is invoked when the other agent responds with an accept-proposal communicative act to the request of registering a meeting.)

At this level of detail we have now described the structure of the software (classes, methods, attributes) in sufficient detail to implement it almost mechanically. The classes produced by following the steps described above are precisely the classes that need to be implemented in the agent coding language. What is still lacking, of course, is the description of the methods presented in these diagrams.

3.9 Agents Behavior Description Phase

As in the case of the A.S.D. phase we have two different levels of abstraction in this phase. In the multi-agent diagrams we represent the flow of events, the invoked methods and the messages exchanged between agents. In the single-agent diagrams we detail the implementation of the formerly used methods.

3.9.1 Multi-Agent Behavior Description diagrams

We draw one or more activity diagrams to show the agent's main class and the inner classes representing its tasks. In the swim-lanes we introduce the methods of the corresponding agent/task class. The syntax of UML activity diagrams supports the representation of concurrency and synchronization. Unlike DeLoach [2], we do not therefore need to introduce a specific diagram for concurrency.

Transitions in the activity diagrams either represent events (e.g. an incoming message or the closing of a task) or method invocations. A transition is introduced for each message identified in the previous phases. (We could take them from the R.Id. diagrams). In this transition we specify the message's performative taken from the the protocol chosen in the R.D. phase and the message's content as described in the D.O.D. phase. We have, therefore, a complete description of the communication including the precise methods involved.

With regards to our example in Figure 11 we can see that the *Forms* task of the *UserInterface* agent calls a new task (*ProposeMtg*) in order to communicate the request of arranging a

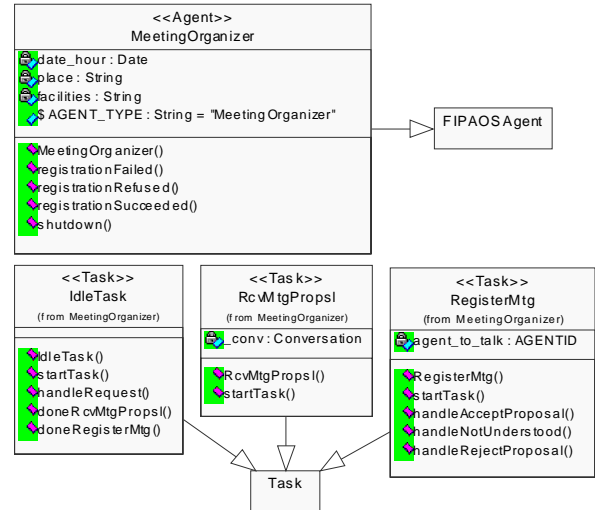


Figure 10. Part of the Single-Agent Structure Definition diagram for the MeetingOrganizer agent

new meeting coming from the user. The new task begins with its constructor and then according to the FIPA architecture, its *startTask* method is automatically invoked. From this method, a message starts towards the *MeetingOrganizer* agent. The performative of this message is "propose" as required by the *Propose* protocol. The *handleRequest* method of the *IdleTask* task in the *Meeting Organizer* agent receives the incoming communication and sends it to the *RcvMtgPropsl* instantiating the task.

3.9.2 Single-Agent Behavior Description diagrams

This is a conventional phase for describing the implementation of the methods introduced in the (S)ASD diagrams. We can choose the any effective way to describe the specific method.

3.10 Code Reuse Phase

Several studies have been carried on in the field of patterns for MAS and different approaches have been applied. In [3] we can find a classification of patterns for agent systems including three main categories:

- Traveling patterns (dealing with the movement capabilities of agents),
- Task patterns (dealing with the tasks that agents can perform),
- Interaction patterns (dealing with communications among the agents).

Another classification can be found in [4]. It is composed of seven levels (mobility, translation, collaboration, actions, reasoning, beliefs, sensory).

In both classification schemes we find common elements: the importance of the agent mobility, actions performed by agents, and agent collaboration or interaction. Nevertheless, the second classification is more detailed, and there we find patterns related specifically to sensors. This is a very important issue in MAS

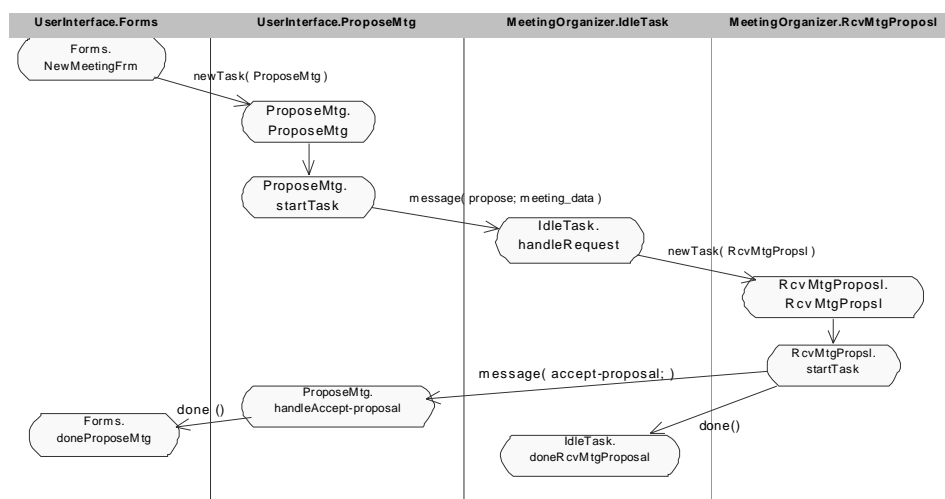


Figure 11. An example of Multi-Agent Behavior Description diagram

operating in the real world, such as we have in robotics applications.

Indeed all patterns are really useful only if they are well documented and obviously versatile. For this reason we have focused our work first in the production of highly reusable patterns and then in their documentation in order to obtain a quick identification of the best pattern for a specific issue.

In the Code Reuse Phase, we try to reuse existing patterns of agents and tasks. It is not correct to talk of patterns of code only, because the process of reuse take place in the design CASE tool where the designer looks at diagrams detailing the library of patterns, not at their code directly. Our patterns therefore are not only pieces of code. They are also pieces of design (of agents and tasks) that can be reused to implement new systems. At the programming level, the designer is too deeply involved in solving the details of the implementation of the various agents and does not have a sufficiently complete view of the system.

The best environment to try to reuse patterns is the design environment. We have used a commercial UML CASE tool that has proven very versatile, thanks to the binding of the design elements (classes, methods, attributes...) to the code language. We have therefore produced a series of pieces of reusable code that are documented with their (M)ABD and (S)ASD diagrams. In the first diagram we describe the behavior of the pattern through the sequence of events and methods implemented while in the (S)ASD we have a structural description of it in form of a class or a group of classes (for example an agent with its *IdleTask* and communication tasks for some protocols).

We have found that in our applications and with our specific implementation environment (FIPAOS), the most useful patterns are those that could be classified as 'interaction' patterns. This is due to the specific structure of FIPA language that delegates a specific task for each specific communication. Each time an agent needs to use that protocol, the pattern task could be easily reused and only the part of the code devoted to the information treatment could need of modification.

3.11 Code Completion phase

This is rather a conventional phase. The programmer completes the code of the application starting from the design, the skeleton produced and the patterns reused.

3.12 Deployment Configuration Phase

This is one of the key elements of the evolution between PASSI and the previous methodology developed by one of the authors (AODPU, [12]) more specifically for robotics applications. It is the response to the necessity of detailing the position of the agents in distributed systems or in mobile-agents contexts.

The deployment configuration diagram describes where the agents are located and which different elaborating units need to communicate in order to permit the communications among the agents. As usual, elaborating units are shown as 3-D boxes (see Figure 12). Agents are shown as components; their name is in the form *agent-name: agent-class*. Communications among agents are represented by dashed lines with the *communicate* stereotype, directed as in the R.D. diagram. For each communication described in the R.D. diagram occurring between agents in different elaborating units, a dashed line is drawn. The receiving agent has an interface to show that it is capable of dealing with that communication. (I.e. It understands the protocol used.) An extension of the UML syntax is used in order to deal with mobile agents moving from one computer to another. A dashed line with the *move_to* stereotype represents it (Figure 12).

In this diagram it is also possible to specify the hardware devices used by the agents (sensors and effectors) and the modes of communication among agents in different elaborating units (traditional/wireless networks, for example). If two agents in different elaboration nodes need to communicate (as stated in the previous phases of the design), a path of connection should be provided between the two nodes.

These constraints about the connections could also be dynamic. In fact, if agent A needs to communicate with agent C, but moves

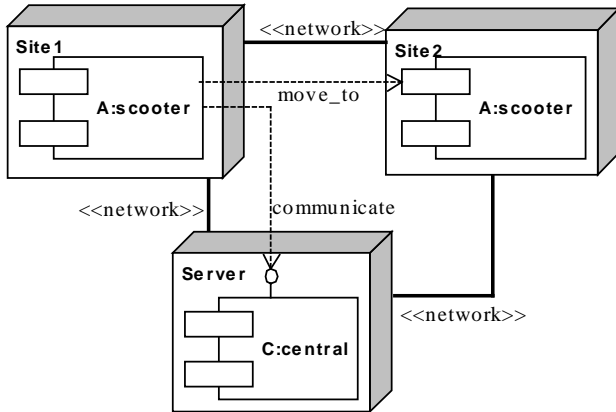


Figure 12. An example of D.C. diagram. The scooter agent moves from one node to another.

across the network (Figure 12), we need to introduce the connection constraints as dependent on agent A's position. We introduce an OCL constraint in all the needed connections for this specific purpose.

4. Experience and plans

PASSI is the result of more than two years of studies and experiments. It proved successful with multi-agent and distributed systems both for robotics and other kinds of applications. It has been used in several research projects (among the others in [14] and in [15]) and in the course of robotics at the university of Palermo and in the course of software development process at the Georgia Institute of Technology for final assignments. Students appreciated the step-by-step guidance provided by the methodology and have found it rather easy to learn and to use.

The implementation environments that we have used vary from a real-time C++ coded agent-based operating system (Ethnos, [17], [18]) to an implementation of the FIPA (Foundation for Intelligent Physical Agents) architecture [19], [20] which is Java-based. We have applied it also in distributed systems [14]. Although the code language should not condition the design, it necessarily imposes some constraints if we want to introduce an extended patterns reuse. For this reason our discussion has focused on a specific architecture, FIPA, for the implementation of the agents.

From these design and implementation experiences we identified limitations of AODPU and requirements for PASSI. We are currently investigating how multiple perspectives of agency, society, design architecture and physical deployment coexist and inform each other in MAS design. In particular, we are interested in how flows of information among the different domains involved in the design of a MAS and anthropomorphic metaphors of agency both afford and potentially impede design cognition. The perspectives are the following: (1) *architectural* (the MAS as software subsystems and interfaces); (2) *social* (the MAS as an artificial society or system of norms); (3) *epistemic* (the single, asocial agent as a problem solver and the MAS as a distributed cognition system), (4) *resource* (the MAS as an instance of recurring patterns), (5) *computing* (the MAS as a

runtime system configuration). Perspectives spread between different models and phases giving a richer (more conceptual, less operative) classification of the elements of the methodology than that described in this paper.

5. REFERENCES

- [1] Jennings, N.R. On agent-based software engineering. In *Artificial Intelligence*, 117 (2000), 277-296.
- [2] DeLoach, S.A., Wood, M.F., and Sparkman, C.H. Multiagent Systems Engineering. *International Journal on Software Engineering and Knowledge Engineering* 11, 3, 231-258.
- [3] Aridor, Y., and Lange, D. B. Agent Design Patterns: Elements of Agent Application Design. In *Proc. of the Second International Conference on Autonomous Agents* (Minneapolis, May 1998), 108-115.
- [4] Kendall, E. A., Krishna, P. V. M., Pathak C. V. and Suresh C. B. Patterns of intelligent and mobile agents. In *Proc. of the Second International Conference on Autonomous Agents*, (Minneapolis, May 1998), 92-99.
- [5] Robbins, J.E., Medvidovic, N., Redmiles, D.F., and Rosenblum, D.S. Integrating Architecture Description Languages with a Standard Design Method. II EDCS Cross Cluster Meeting in Austin, Texas. www.ics.uci.edu/pub/arch/uml/research/.
- [6] Yu, E., Liu, L. Modelling Trust in the i* Strategic Actors Framework. *Proc. of the 3rd Workshop on Deception, Fraud and Trust in Agent Societies at Agents2000* (Barcelona, Catalonia, Spain, June 2000).
- [7] F. Zambonelli, N. Jennings, M. Wooldridge. Organizational Rules as an Abstraction for the Analysis and Design of Multi-agent Systems. *Journal of Knowledge and Software Engineering*, 2001, 11, 3, 303-328.
- [8] J.Odell, H. Van Dyke Parunak, B. Bauer. Representing Agent Interaction Protocols in UML, *Agent-Oriented Software Engineering*, P. Ciancarini and M. Wooldridge eds., Springer-Verlag, Berlin (2001), 121-140.
- [9] Jacobson, I., Booch, G., Rumbaugh, J. *The Unified Process*. IEEE Software (May/June 1999).
- [10] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley (1992).
- [11] Chella, A., Cossentino, M., and Lo Faso, U. Applying UML use case diagrams to agents representation. *Proc. of AI*IA 2000 Conference*. (Milan, Italy, Sept. 2000).
- [12] Chella, A., Cossentino, M., and Lo Faso, U. Designing agent-based systems with UML in *Proc. of ISRA'2000* (Monterrey, Mexico, Nov. 2000).
- [13] Chella, A., Cossentino, M., Infantino, I., and Pirrone, R. An agent based design process for cognitive architectures in robotics in *proc. of WOA'01* (Modena, Italy, Sept. 2001).

- [14] Chella, A., Cossentino, M., Infantino, I., and Pirrone, R. A vision agent in a distributed architecture for mobile robotics in Proc. Of Workshop "Intelligenza Artificiale, Visione e Pattern Recognition" in the VII Conf. Of AI*IA (Bari, Italy, Sept. 2001).
- [15] Chella, A., Cossentino, M., Tomasino, G. An environment description language for multirobot simulations in proc. of ISR 2001 (Seoul, Korea, Apr. 2001)
- [16] M. Wooldridge, P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. Agent-Oriented Software Engineering, P. Ciancarini and M. Wooldridge eds., Springer-Verlag, Berlin (2001), 1-28.
- [17] Piaggio, M., and Zaccaria, R. An Efficient Cognitive Architecture for Service Robots. Journal of Intelligent Systems. Freund & Pettman, Endholmes Hall, England, 9:2 (March-April 1999).
- [18] Piaggio, M., and Zaccaria, R. Distributing a Robotic System on a Network - the ETHNOS Approach. Advanced Robotics, International Journal of the Robotics Society of Japan, 12, 8, (April 1998).
- [19] O'Brien P., and Nicol R. FIPA - Towards a Standard for Software Agents. BT Technology Journal, 16,3(1998),51-59.
- [20] Poslad S., Buckle P. and Hadingham R. The FIPA-OS Agent Platform: Open Source for Open Standards. Proc. of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents (Manchester,UK, April 2000), 355-368.
- [21] FIPA Personal Assistant Specification. Foundation for Intelligent Physical Agents, Document FIPA00083 (2000). www.fipa.org/specs/fipa00083/.
- [22] OMG Unified Modeling Language, version 1.3, June 99, Object Management Group document ad/99-06-08, <http://cgi.omg.org/docs/ad/99-06-08.pdf>
- [23] Cranefield, S., and Purvis, M. UML as an ontology modeling language. Proc. of the Workshop on Intelligent Information Integration, IJCAI-99 (Stockholm, Sweden, July 1999).
- [24] F. Bergenti, A. Poggi. Exploiting UML in the design of multi-agent systems. ESAW Workshop at ECAI 2000 (Berlin, Germany, August 2000).
- [25] Communicative Act Library Specification. Foundation for Intelligent Physical Agents, Document FIPA00037 (2000). <http://www.fipa.org/specs/fipa00037/>.
- [26] Odell, J., Van Dyke Parunak, H., and Bauer, B. Extending UML for Agents. AOIS Workshop at AAAI 2000 (Austin, Texas, July 2000).
- [27] Newell, A. The knowledge level, Artificial Intelligence, 18 (1982) 87-127.
- [28] Wooldridge, M., Jennings, N.R., and Kinny, D. The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and Multi-Agent Systems. 3,3 (2000), 285-312.
- [29] Antón, A.I., McCracken, W.M., and Potts, C. Goal Decomposition and Scenario Analysis in Business Process Reengineering in proc. of Advanced Information System Engineering: 6th International Conference, CAiSE '94 (Utrecht, The Netherlands, June 1994) 94-104.
- [30] Antón, A.I., and Potts, C. The Use of Goals to Surface Requirements for Evolving Systems, in proc. of International Conference on Software Engineering (ICSE '98), (Kyoto, Japan, April 1998), 157-166
- [31] van Lamsweerde, A., Darimont, R. and Massonet, P. Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt in Proc. 2nd International Symposium on Requirements Engineering (RE'95) (York, UK, March 1995), 194-203
- [32] Potts, C. ScenIC: A Strategy for Inquiry-Driven Requirements Determination in proc. of IEEE Fourth International Symposium on Requirements Engineering (RE'99), (Limerick, Ireland, June 1999), 58-65.
- [33] Jackson, M. Problem Frames: Analyzing and structuring software development problems. Addison Wesley, 2001