Chapter 5
Porto - An Improvement to Gaia

Abstract This chapter[1] is the first out of the four main chapters of this dissertation containing original contributions. Here we propose an improvement to the *GAIA* methodology called *PORTO*, which is one of the main contributions of this thesis, and is the result of its application to the analysis, design and development of the MASDIMA - Multi-Agent System for Disruption Management. We present each phase of the methodology using as an example the MASDIMA system. This will allow the reader to better understand the concepts together with the proposed methodology and, at the same time, get acquainted with the system we have developed as a proof of concept for our approach to the disruption management problem.

5.1 Introduction

In Section 3.6, we have provided some background information about Agent Oriented Software Engineering (AOSE), including a short survey of the state of the art in this field. In this chapter we will present *PORTO*, a methodology that complements another, already proposed, methodology called *GAIA* (Zambonelli *et al.*, 2003).

It is important to have a methodology for the development of a software system, specially for systems that are complex (but not only). A software development methodology allows to structure, plan, and control the process of developing such a software system, resulting in systems that behave better and according to the requirements. A study conducted by NIST[2] in 2002 (Newman, 2002) reports that software defects cost the U.S. economy $59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed. Additionally, it is commonly accepted that the earlier a defect is found the cheaper it is to fix it. Table 5.1 shows the cost of fixing the defect depending on the stage it was found according to (McConnell, 2004). For example, if a problem in the requirements is found only post-release, then it would cost 10 to 100 times more to fix than if it had already been found in the requirements review.

Table 5.1 Cost to fix a defect (McConnell, 2004)

		Time Detected				
		Requirements	Architecture	Construction	Testing	Post-Release
Time Introduced	Requirements	1x	3x	5 to 10x	10x	10 to 100x
	Architecture	–	1x	10x	15x	25 to 100x
	Construction	–	–	1x	10x	10 to 25x

Due to the above reasons and personal work experience of the author, we decided to use an AOSE to develop the MASDIMA system (see Chapter 7 and Appendix A) that supports the concepts and approach we propose in this dissertation. As such, this chapter proposes an improvement

[1] A previous version of this work was presented in *Antonio J.M. Castro and Eugenio Oliveira The Rationale Behind the Development of an Airline Operations Control Center using Gaia based Methodology* (Castro & Oliveira, 2008).

[2] National Institute of Standards and Technology

to the *GAIA* methodology, called *PORTO*, that capitalizes on the experience we gained during the analysis and design of MASDIMA. Particularly, this chapter points out:

- The rationale behind the analysis, design and implementation of our software system.
- How we have used a goal-oriented early requirements analysis to complement *GAIA*. We also present the advantages that we believe emerge with the use of our approach to the modeling phase.
- How we used the early requirements analysis to sub-divide the system into sub-organizations and how we have described and represented the environment model.
- How we created the preliminary role model as well as the interaction model and how we found useful to have a graphical representation of the preliminary role and interaction models together with the environment model, i.e., a *UML*[3] *combined diagram*.
- How we used the previous models to define the organizational structure of our system and how we represented it in UML, including how we mapped the abstractions to UML metaclasses plus the stereotypes we created.
- The steps we took to complete the role and the interaction models and how we represented those two models in UML including the mappings we done.
- How we defined the agent model and the service model and how we represented those models in UML including how we mapped the abstractions to UML metaclasses.
- How we included an implementation phase, with steps that allow to identify the concepts and actions as well as to perform the mapping between the services and the required behaviours.
- How we included a test and validation phase that allows to test and validate the system according to the requirements.

It is important to point out that the main goal of our work was not about *AOSE*. However, the contributions in this area appear due to the need we had to model a complex and realistic MAS.

The rest of this chapter is organized as follows. In Section 5.2 we present a quick overview of the *PORTO* methodology. In Section 5.3 we explain the *Requirements Analysis* phase and in Section 5.4 the *Analysis* phase, the two phases that allow to understand the system-to-be.

In Section 5.5 and 5.6 we present the *Architectural Design* and *Design* phases, respectively, that are related to the design of the system according to the requirements and specifications. In Section 5.7 we present processes and artifacts of the *Implementation* phase and in Section 5.8 the *Test and Validation* phase. We end with a chapter summary in Section 5.9.

5.2 Methodology Overview

As stated in the previous section the goal of *PORTO* is to complement the *GAIA* methodology with the inclusion of a *Requirements Analysis*, *Implementation* and *Test and Validation* phases in the process of building software, as well as the inclusion of new or replacement documents and artifacts. In Figure 5.1 we present an overview of the proposed methodology. The left column shows the phase name, the center column the processes that should be performed in each phase, and the right column, the documents and artifacts produced during each phase. A brief explanation of each phase of the proposed methodology, follows:

[3] Unified Modeling Language (http://www.uml.org)

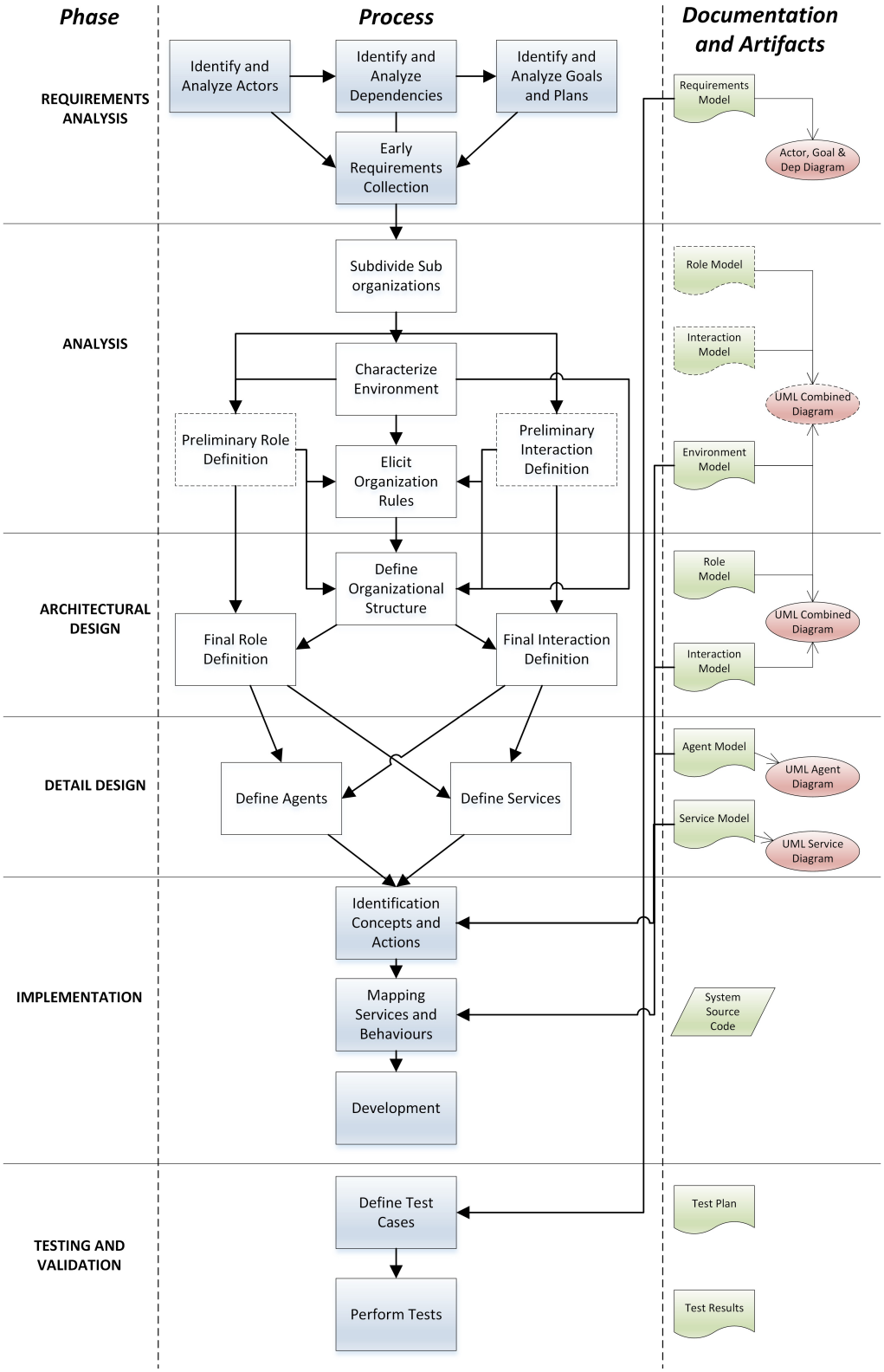

Fig. 5.1 Overview of PORTO Methodology

- *Requirements Analysis*: The goal of this phase is to understand the requirements from the point of view of the stakeholders[4] and users. *GAIA* does not propose a method to model the requirements and, as such, in *PORTO*, we propose to use a goal-oriented requirements analysis similar to the one presented in *TROPOS* (Bresciani *et al.*, 2004) and make the necessary changes to integrate it in the proposed methodology. The artifacts of this phase are the requirements model and the actor, goal and dependency diagrams.
- *Analysis*: The goal of this phase is to understand what the software system (here a MAS) will have to be, considering the requirements model. Like in *GAIA*, it will be produced an *environment*, *preliminary role* and *preliminary interaction* model as well as a set of *organizational rules*. This will be done considering the (possible) sub-organizations that might exist according to the requirements of the system-to-be. In *PORTO* we propose to use UML Diagrams as a notation to describe the several models produced in this phase (instead of the original ones) and introduce a new diagram called *UML Combined Diagram*.
- *Architectural Design*: The two previous phases are about understanding the MAS-to-be. Here, we start to make decisions about the actual characteristics of the MAS. This phase is not only about refining the roles and interaction models. It is also here that important decisions about the MAS organizational structure are made. As with the previous phase we follow the guidelines of *GAIA* regarding this phase, and we refine the *UML Combined Diagram* introduced in our approach.
- *Detail Design*: This phase is responsible for identifying the agents and services that will implement the roles, functions and interactions identified so far. It will take into consideration the spatial and physical distribution that is going to be adopted by the MAS. The outputs are the *agent* and *service* models. The specifications described in these models are neutral regarding the programming language or middleware used for implementation. We have replaced the agent and service model notation used by *GAIA*, by the *UML Agent* and *UML Services* diagram, respectively.
- *Implementation*: *GAIA* does not provide an implementation phase. In *PORTO* we propose to include some processes that will help the developers to identify the concepts and actions that need to be defined during implementation as well as a mapping of the services and agent behaviours. Although the steps presented here are valid for any programming language, we have used JAVA[5] and JADE[6] as an example to show how they can be applied. The output of this phase is the *system source code*.
- *Test and Validation*: During and after the implementation it is necessary to perform tests to see if the system works according to the specifications. This is a new phase that does not exist in *GAIA*. Here, the test cases are defined according to the specifications and the execution of these test cases will validate (or not) the system, showing if the system is working according to the requirements model. The *unit tests* are not considered at this stage but only in the development step of the implementation phase.

In the next sections we will detail each of the phases using as an example the analysis and design we have performed for the MASDIMA system (see Chapter 7 and Appendix A).

[4] A person, group or organization that has interest or concern in the system-to-be.

[5] http://www.java.com

[6] http://jade.tilab.com

5.3 Requirements Analysis

The *GAIA* methodology uses as an input a collection of requirements. The methodology does not propose a method to model these requirements. Although we could just list the requirements of the system-to-be, from interviews of the stakeholders and users, we believe it is important to have a better understanding of those requirements early in the process.

In *PORTO* we chose to adopt part of the goal-oriented requirements analysis of *TROPOS* (Bresciani *et al.*, 2004), i.e., the *Early Requirements Analysis*. In a goal-oriented requirements analysis the domain stakeholders are modeled as actors, depending on one another to achieve their goals. Plans to be performed and resources to be furnished are also modeled here. The key concepts or abstractions used in this phase are (Bresciani *et al.*, 2004):

1. *Actor*: Represents the stakeholders, the users of the system as well as the roles. It can be a physical, social or software entity that has strategic goals or concerns within the system or organizational setting.
2. *Goal*: Represents the actors' interests. There are *hardgoals* and *softgoals*. The former should be *satisfied* by the system, i.e., the system should have functionalities that allow the goal to be satisfied. The latter are non-functional requirements, i.e., the system does not necessarily implements functionalities to achieve the softgoals but might be operated in an environment (hardware and network infrastructure, for example) that will satisfy these softgoals. From now on and to simplify the reading, we will use the word goal as a synonym of hardgoal.
3. *Plan*: Represents a *way of doing something*. By executing a plan an hard or softgoal can be satisfied, i.e., a plan is a mean for satisfying a goal.
4. *Resource*: Represents an informational (e.g., a database) or physical (e.g., a sensor) entity.
5. *Dependency*: Dependencies exist between actors, meaning that one actor depends on another to achieve some goal, execute some plan or deliver/access some resource.

The *Requirements Analysis* phase has three main processes (see Figure 5.1), that will allow to collect all requirements in a *Requirements Model*. A description of these processes follows:

1. *Identify and Analyze Actors*: The objective is to identify and perform an analysis of all application domain stakeholders and users and their intentions as actors which want to achieve goals.
2. *Identify and Analyze Dependencies*: The objective is to focus on the dependencies between actors, modeling those dependencies as goals, plans or resources that depend on one or another to be achieved.
3. *Identify and Analyze Goals and Plans*: The idea is, from the point-of-view of the actor, perform an analysis of the actor's goals and plans, by using three basic reasoning techniques: *means-end analysis*, *contribution analysis* and *AND/OR decomposition*. The first one, aims at identifying plans, resources and softgoals that provide means to achieve a hardgoal. The second one, identifies goals that can contribute to the fulfillment of the analyzed goal. The last one, combines AND and OR decompositions of a root goal or plan into sub-goals or sub-plans, respectively, allowing to get a finer goal and plan structure.

By performing the above processes it is possible to summarize them in the *Early Requirements Collection* process.

Typically, to perform the above processes, the *requirements analyst* will interview all the potential stakeholders and users and read all the documentation that is available, to fully understand all the requirements. It is an iterative process, going back and forth as many times as needed.

The main graphical tool available for the analyst to help him/her is the *Actor, Goal and Dependencies Diagram*. Figure 5.2 shows a partial actor, goal and dependency diagram for the MAS-

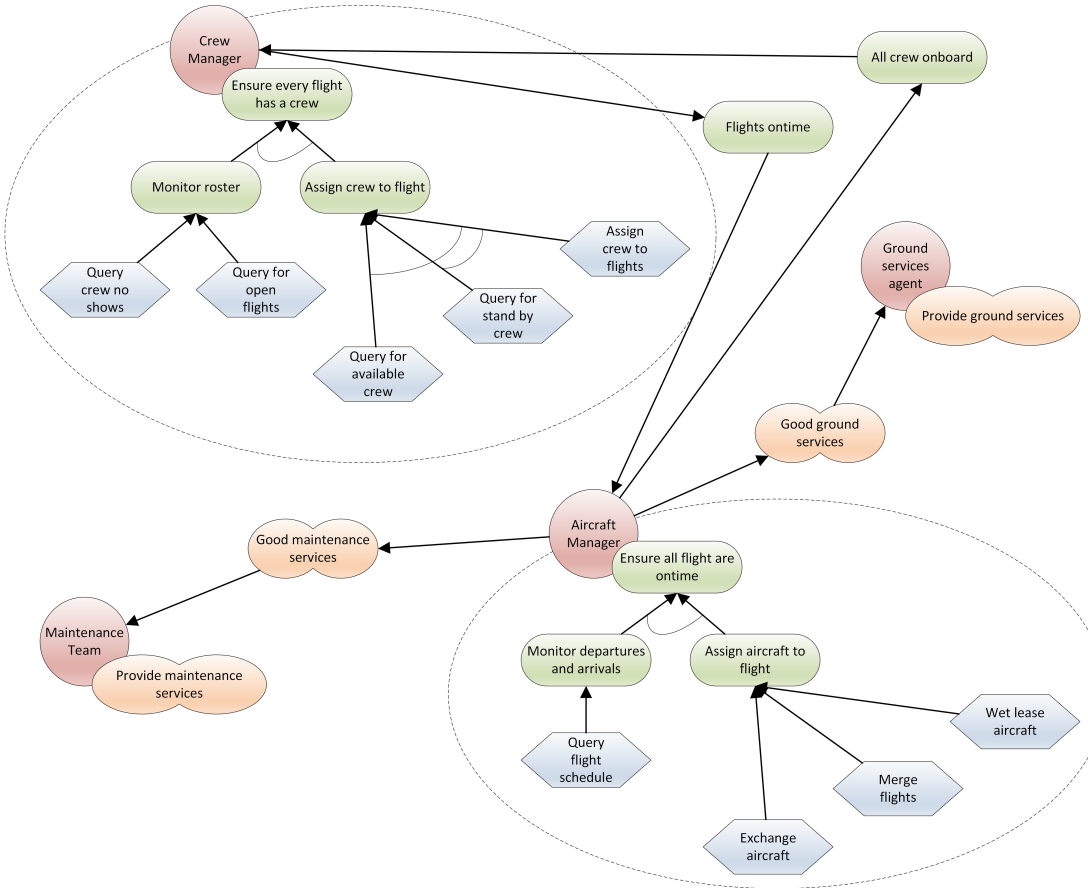

Fig. 5.2 Partial Actor, Goal and Dependency diagram

DIMA system, generated during the goal-oriented analysis. Here, the *red circles* represent actors, the *green ellipses* represent hardgoals, the *cloud ellipses* represent softgoals, the *blue hexagons* represent plans and the *large dash circle* represents the actor's perspective. The *and-decomposition* appears with a semi-circle connecting two arrows. Likewise, two arrows without a semi-circle mean a *or-decomposition*.

Looking at Figure 5.2, the main goal of the *Crew Manager* actor is to *Ensure every flight has crew*, meaning that before departure, it is necessary to guarantee that all flights have all crew members assigned according to the regulations. To be able to achieve this goal it is necessary to do an *and-decomposition*, meaning that it is necessary to achieve the sub-goals *Monitor roster* **AND** *Assign crew to flight*. To fulfill the sub-goal *Monitor roster* **any** of the following plans can be executed:

- *Query crew no shows*, meaning that it is necessary to query one of the resources available in the environment to obtain the name of the crew members that did not report for duty.
- *Query for open flights*, i.e., to query for flights with open crew positions.

To fulfill the sub-goal *Assign crew to flight* it is necessary to execute **all** of these three plans:

- *Query for available crew*, obtain a list of crew members that are available to be assigned to a specific flight. In this case, available means that the crew member does not have any kind of activity assigned, including a day off, and that, according to the regulations, can be assigned to the flight.
- *Query for stand by crew*, obtain a list of crew members that have assigned a stand by activity and that, according to the regulations, can be assigned to the flight.
- *Assign crew to flights*, finally, from the two lists obtained from the previous plans, to choose the best crew member according to criteria defined by the company, and assign him/her to the flight.

The execution of the referred plans is a mean to achieve the previously mentioned goals. To achieve its main goal the *Crew Manager* actor also depends on actor *Aircraft Manager*, through the dependency *Flights on time*.

Figure 5.2 also shows an example of a soft-goal, that is, a goal without a clear definition and/or criteria for deciding if it is satisfied or not (typically used to model non-functional requirements). Actor *Aircraft Manager* depends on actor *Maintenance Team* through the soft-dependency *Good maintenance services*.

5.3.1 Advantages

We found that the use of a goal-oriented analysis for eliciting the early requirements helped, not only in gathering and understanding the collection of requirements, but also in the analysis phase, namely:

- The modeling of the requirements, in terms of actors, their roles and their goals and dependencies among them, is more similar to the organization we have found in the airline AOCC, allowing us to better specify the software system-to-be.
- In subdividing the system: modeling the desires, intentions and dependencies of the stakeholders and specifying the system-to-be in terms of goals and softgoals, helped identifying the specific organizations and sub-organizations dedicated to the achievement of a given sub-goal. The teams in the AOCC exhibiting behaviours specifically oriented to the achievement of a goal and the corresponding mapping to sub-organizations are good examples of this advantage.
- In the preliminary role model: identifying the basic skills (functionalities and competences) required by the organization to achieve its goals. Again, the modeling of the system-to-be in terms of actors involved and their goals helped to identify the preliminary roles (basic skills). This is in accordance with the statement of the section 4.1.3 in (Zambonelli *et al.*, 2003).
- In the environment model: having a deeper understanding of the environment where the software must operate as well as of the interactions between software and human agents, during the modeling of the system-to-be, helped in identifying the roles of active components, allowing the distinction between active components that are only resources and others that should be agentified.
- In the preliminary interaction model: identifying the basic interactions that are required for the exploitation of the basic skills. This is not a direct advantage of applying a goal-oriented early requirements analysis. It appears in the identification of the basic skills (in the preliminary role model). However, having a better understanding of the actors and their dependencies, as

the result of the goal-oriented analysis, during the modeling of the system-to-be, facilitates the identification of the basic interactions.

5.4 Analysis

The objective of this phase is to understand what the system will have to be, considering the requirements model. From Figure 5.1 we see that it has five processes: (i) subdivision of the system into sub-organizations, (ii) characterization of the environment model, (iii) definition of the preliminary role model, (iv) definition of the preliminary interaction model and, (v) elicitation of the organizational rules. The outputs of this phase are the *preliminary role and interaction model* and the *environment model*. The main diagram used is the *UML Combined Diagram*.

Before proceeding to the explanation of each of the processes, it is important to define some of the abstractions used in this phase (and on the following ones) as well as the mappings we have done to the UML abstractions. This allows to better understand the processes and the UML diagrams used. The main abstractions and mappings are as follows:

- *Plans*: As stated in Section 5.3 it represents a *way of doing something*. We mapped this abstraction to a *UML class*. A class represents a group of things that have a common state and behavior. A class can represent a tangible and concrete concept, such as an invoice, or it may be abstract, such as a document. Using the class *attributes*, *methods* and *comments* abstraction we can represent the plan concept in a satisfactory way.
- *Resource*: Also as stated in Section 5.3 it represents an informational or physical entity. We found useful to map this abstraction to a *UML Table entity*.
- *Role*: We also mapped this abstraction to a *class*. A role represents functionalities and competences that need to be characterized. We found, at this point, that this mapping helped to visualize the roles and the relations among them. The role's safety properties are mapped as attributes and the activities as methods. However, as stated in (Bauer & Odell, 2005) "roles cannot be modeled in the necessary detail with any UML 2.0 diagrams". As we will show in Section 5.4.3 we use a role schema description table for each role to complete the information provided by the diagram. Nevertheless, we think that with our UML representation we are able to include almost all the necessary information.
- *Actions*: This concept represents the actions that roles or agents can perform on the environment resources (typically by executing a plan). For example, by reading or changing an informational resource. We mapped this concept to the *Dependency relationship* in UML adding the type of action to it.
- *Protocol*: The activities that involve interactions with other roles (protocols) are represented by an *Association relationship* in UML. We have created a protocol stereotype associated to the *association* metaclass in UML. Although some of these mappings might not be the appropriate one for the implementation phase, it did help us to visualize and model the organization with their roles, activities and protocols, using a widely used notation supported by several commercial tools.

5.4.1 Subdivide the system into sub-organizations

This is the first process to be performed in the Analysis phase. The objective is to see if the overall system can be subdivided into sub-organizations. Sub-organizations can be found when there are portions of the overall system that have **any of these conditions**:

- Exhibit a behavior specifically oriented towards the achievement of a given sub-goal.
- Interact loosely with other portions of the system.
- Require competences that are not needed in other parts of the system.

Continuing with MASDIMA as an example, we see that from the requirements analysis (see Figure 5.2) it is possible to determine three candidate sub-organizations that fulfill at least one of these conditions. The sub-organizations identified are described in table 5.2.

Table 5.2 Identification of sub-organizations

Sub-organization	Description
Crew Manager	The subgoal to achieve is *Ensure every flight has a crew*. It will loosely interact with other portions of the system because of the dependency *Flights on time* with the Aircraft Manager actor.
Aircraft Manager	The subgoal to achieve is *Ensure all flights are on time*. It will loosely interact with other portions of the system because of the dependencies *All crew onboard* with the Crew Manager actor and *Good maintenance services* with the Maintenance Team actor and *Good ground services* with the Ground Services handling agent actor.
Passenger Manager	The subgoal to achieve is *Ensure all passengers arrive at the destination*. Its interactions depend on *Flights on time* with the Aircraft Manager actor and *Good ground services* with the Ground Services handling agent actor. Please note that the Passenger Manager actor and dependencies are not included in the partial actor, goal and dependencies diagram on Figure 5.2.

5.4.2 Characterize the Environment

The second process of the analysis phase is to define the environment and, in this process, the first decision to be made is to distinguish between resources and active components. Resources might be, according to *GAIA*, ”variables or tuples, made available to the agents for sensing (e.g. reading their values), for effecting (e.g. changing their values) or for consuming (e.g. extracting them from the environment)”. On the other hand, active components are those components and services capable of performing complex operations with which agents in the MAS will have to interact. Computer-based systems or humans in a process are examples of active components that should not be treated as part of the environment but, instead, they should be agentified. In the MASDIMA case, we have one ”human” in the Operations Control Center that has an important role in the process (see Table 5.3) and to which the agents in the MAS will have to interact. This Active Component will be agentified through one agent. Table 5.4 shows some of the resources that are available in the environment.

Table 5.3 Active Components (partial)

Name	Description
Operational Control Supervisor	Final human authority regarding: initiating, canceling, consolidating or advancing flights, wet lease of airplanes, exchange of airplane, delay of flights, etc. In summary, this human has to authorize the application of any solution found in the operational plan.

Table 5.4 Resources (partial)

Name	Description
Crew Sign On	Contains information regarding the crew sign on for flights. It will be possible to know if a crew member did not report for duty. It will allow to implement the plan *query crew no shows* indicated in the crew manager goal diagram of Figure 5.2.
Pairings	Contains information regarding the pairings (and flights) that need to have crew members assigned. It will allow implementing the plan *query for open flights* and *assign crew to flights* indicated in the same diagram as the previous one.
Roster	Contains information regarding the roster of all crew members. It will allow implementing the plans *query for available crew*, *query for stand by crew* and *assign crew to flights* indicated in the same diagram.

For the environment model to be complete, we need to list or represent the resources and actions that will be performed to access them, from the environmental perspective. We proposed a UML representation of the environment model that includes the resources as well as the plans that will be executed using those resources. In our opinion, the inclusion of the plans allows to better model the environment.

Figure 5.3 shows a partial environment model taken from the MASDIMA. Regarding the *CrewSignON* resource we can see that the action will be to *read* information from attributes *dutyID* and *crewNumber* and the plan to be used is *QueryCrewNoShow*. That plan executes the *read* action performing the condition described in the comments, that is, returns the records where the *current date* is equal or greater than the *dutyDateTime* plus an additional time (in minutes) and where the *signOnDateTime* attribute is null (meaning that the crew did not report for duty). An example of an action that *changes* the resources is presented in the *Pairing* resource. It is possible to see that the plan *AssignCrewFlights* will *change* the attributes *cmdOpen*, *foOpen*, *csOpen*, *cfaOpen*, *faOpen*, subtracting to the existing value the number of crew members assigned. That same plan has *read* access to the *Roster* resource.

In the Preliminary Role Definition process we will start to use a UML Combined diagram that includes the environment model and the roles, showing the actions that those roles perform on the resources (through the execution of plans).

5.4.3 Preliminary Role Definition

In this process the objective is to identify the basic skills, that is, functionalities and competences, required by the organization to achieve its goals. Those basic skills are the preliminary roles and we need to identify the ones that will be played whatever the organization structure that will be

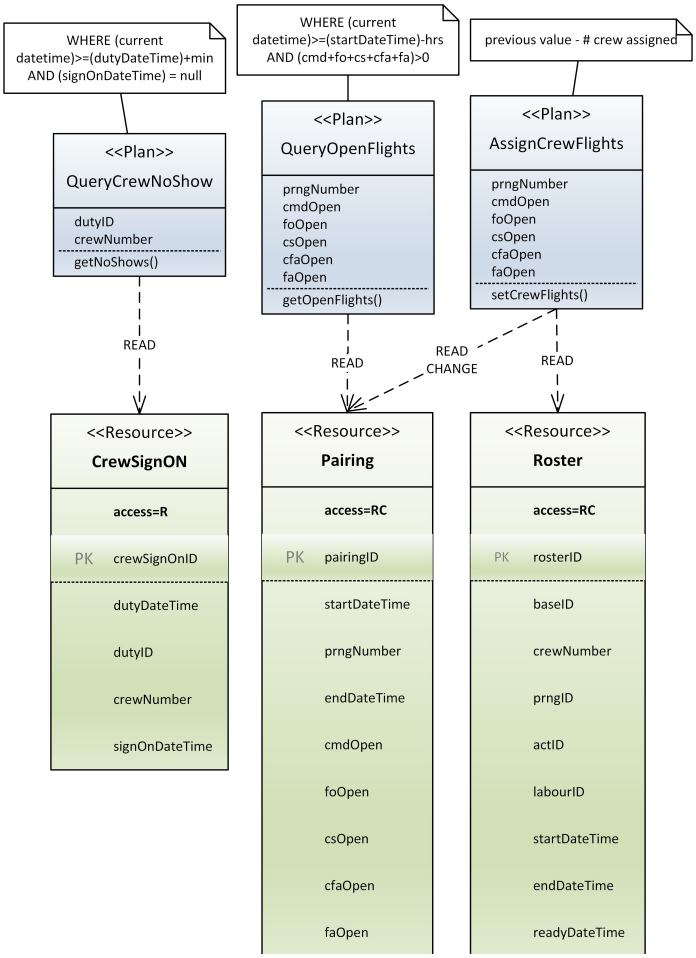

Fig. 5.3 Partial UML Environment Model Diagram

adopted later on during the Architectural Design phase. *GAIA* adopts an abstract, semiformal description to express the capabilities and expected behaviours of the preliminary roles. These are represented by two main classes: *Permissions* (actions allowed on the environment to accomplish the role) and *Responsibilities* (attributes that determine the expected behavior of a role, divided in Liveness Properties and Safety Properties). According to (Zambonelli *et al.*, 2003), *Liveness properties* "describe those states of affairs than an agent must bring about, given certain conditions". In contrast, *Safety properties* are *invariants*, i.e., an "acceptable state of affairs is maintained". The authors of *GAIA* propose the use of regular expressions to define these properties. For example, a *liveness expression* has the general form:

$$RoleName = expression$$

where *expression* are *activities* or *protocols*. Table 5.5 shows the operators used for *liveness* expressions. We recommend the reading of Section 4.1.3 of the *GAIA* methodology, for more information regarding these properties.

From the Actors, Goals and Dependencies diagram in figure 5.2, we can identify several roles that will exist independently of the final organization of our MAS. A partial list of those roles is:

Table 5.5 Operators for Liveness Expressions

Operator	Interpretation
$x.y$	x followed by y
$x\|y$	x or y occurs
$x*$	x occurs 0 or more times
$x+$	x occurs 1 or more times
x^w	x occurs indefinitely often
$[x]$	x is optional
x_y	x and y interleaved

- *RosterCrewMonitor*, role associated with monitoring the crew roster for events related to crew members that do not report for duty and/or flights with open positions.
- *CrewFind*, role associated with finding the best crew member to be assigned to a flight after an event triggered by the *RosterCrewMonitor* role.
- *CrewAssign*, role associated with assigning the crew member found by the *CrewFind* role.

For each role it is necessary to define the permissions and the responsibilities and, an important step, it is necessary to identify any inconsistencies between what operations the environment allows and what the roles (agents) need or must be allowed to do. *GAIA* refers the need to create a *Role Schema* for each role, where these properties will be indicated. However, we found that a diagram that includes the Environment and Preliminary Roles will help to better identify these inconsistencies.

In the *UML Combined Diagram with Preliminary Roles and Environment* in figure 5.4 the *actions* allowed by the environment are labeled with an *R* for *reading* and a *C* for *changes*, for each resource through the *access* attribute. The *actions* the roles need or must be allowed to do are indicated by the *dashed arrows*. We can see that the *CrewAssign* role needs to read and change information from the resources *Pairings* and *Roster*. Looking to those resources representation, it is possible to see that these operations are allowed (both have the *access* attribute equal to *RC*). After analyzing this diagram we can see that there are no inconsistencies between the operations allowed by the environment and what the agents need to do.

To complete the preliminary role model, we have filled a role schema for each of the roles identified, with the information collected so far. It is possible to see an example in table 5.6.

It is important to point out that the *plans* that came from the requirements model and later represented in the environment diagram (see Figure 5.3), are replaced by *activities* performed by the roles. For example, the plans *Query crew no shows* and *Query for open flights* that allow the *Crew Manager* actor to reach goal *Monitor Roster*, are included in the activity CheckNewCrewEvents that appear in the *Role Schema*. In the UML combined diagram in Figure 5.4 these activities appear inside the role as methods with stereotype $<<act>>$.

The *permissions* in the role schema are represented as actions in the diagram and the *safety* rules as attributes inside the role. For example, attribute *conSignON* allows to validate rule *successful_connection_with_CrewSignON*. The only thing that the UML Combined diagram cannot capture very well is the *liveness* expressions. As it is possible to see in Figure 5.4 we have used a comment to represent the liveness expression for the *RosterCrewMonitor* role.

The preliminary role model will be finished in the Architectural Design, giving its place to the full Role Model.

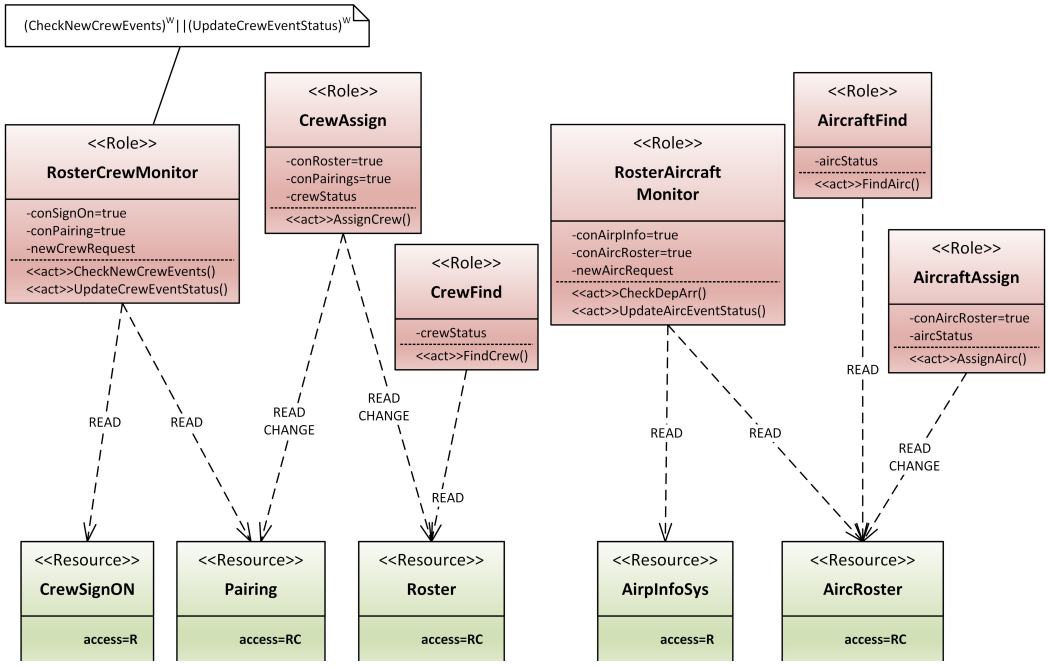

Fig. 5.4 UML Combined Diagram with Preliminary Roles and Environment (partial)

Table 5.6 RosterCrewMonitor preliminary role

Role Schema: RosterCrewMonitor

Description: This preliminary role involves monitoring the crew roster for events related to the crew members not reporting for duty and/or flights with open positions. After detecting one of these events it will elicit a solution from the organizer. It should be able to trace previous requests, avoiding duplicates, until it receives a message regarding the status of the request.

Protocols and Activities: CheckNewCrewEvents.UpdateCrewEventStatus

Permissions:
reads CrewSignON (to obtain all who did not report for duty)
reads Pairings (to obtain all flights with open positions)
Responsibilities:
Liveness:
$RosterCrewMonitor = (CheckNewCrewEvents)^{W}\ \ (UpdateCrewEventStatus)^{W}$
Safety:
$successful\ connection\ with\ CrewSignON = true$
$successful\ connection_{w}with\ Pairings = true$
$new\ crew\ request <> existing\ unclosed\ crew\ request$

5.4.4 Preliminary Interaction Definition

The objective of this process is to capture the dependencies and relationships between the various roles in the multi-agent system organization. This is done with one protocol definition for each type of inter role interaction. *GAIA* proposes a very simple notation to specify the protocols, i.e., a table with the protocol name, description, initiator and partner role(s) and input and outputs of the protocol. We propose to use a UML Interaction diagram, since it is more expressive than the tabular notation.

Figure 5.5 shows the preliminary definition of the *requestCrew* protocol identified in the MAS-DIMA system. The protocol has the *RosterCrewMonitor* role as *initiator* and the *CrewFind* role as *partner*. After detecting an event, i.e., a crew member that does not report for duty or a flight with open positions (represented in the diagram by the parameter *crewEvents* in the *request* performative), the *RosterCrewMonitor* requests a solution to the *CrewFind* role (represented by the parameter *crewMembers* in the $inform - result$ performative). It starts by issuing a request and, then, the *CrewFind* has two alternatives: refuse to present a solution or accept the request. Accepting the request, if it does not have success in looking for a solution, it answers back with a *failure* performative that includes the reason. Having success, it informs when the job is done through the $inform - done$ performative and, then, sends the solution for the request through the $inform - result$ performative. This protocol is according to the *FIPA Request* protocol definition (FIPA, 2002c).

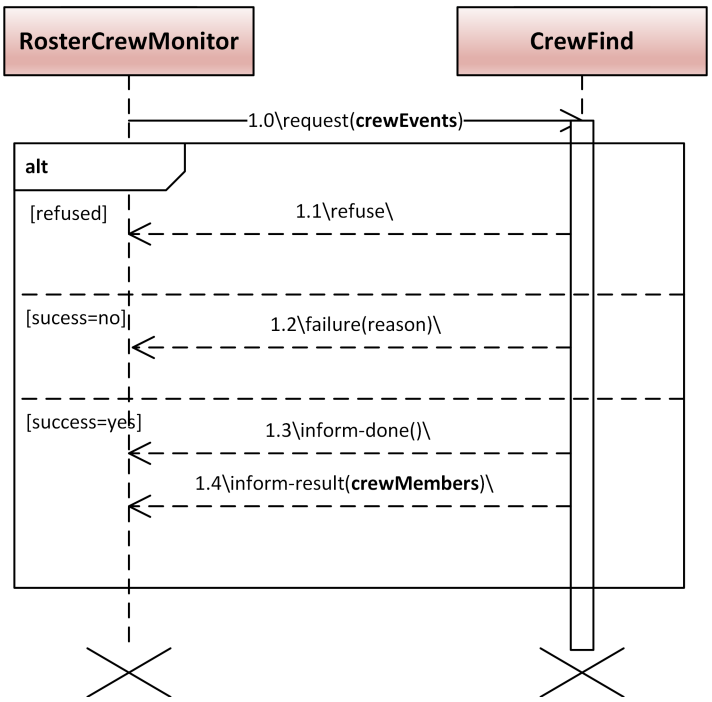

Fig. 5.5 UML Interaction Diagram for *requestCrew*

To have a better overview of the whole system, we found useful to complement the UML Combined diagram with the preliminary interactions (protocols) as presented in figure 5.6.

5.4.5 Elicit Organizational rules

According to the authors of *GAIA* "(. . .) there may be general relationships between roles, between protocols, and between roles and protocols that are best captured by organizational rules". Organizational rules are seen as responsibilities of the organization as a whole. In the preliminary role model we have already defined or approached the roles' responsibilities. As in that model, organizational rules also have safety and liveness rules, or, as in (Zambonelli *et al.*, 2003), constraints and relations, respectively:

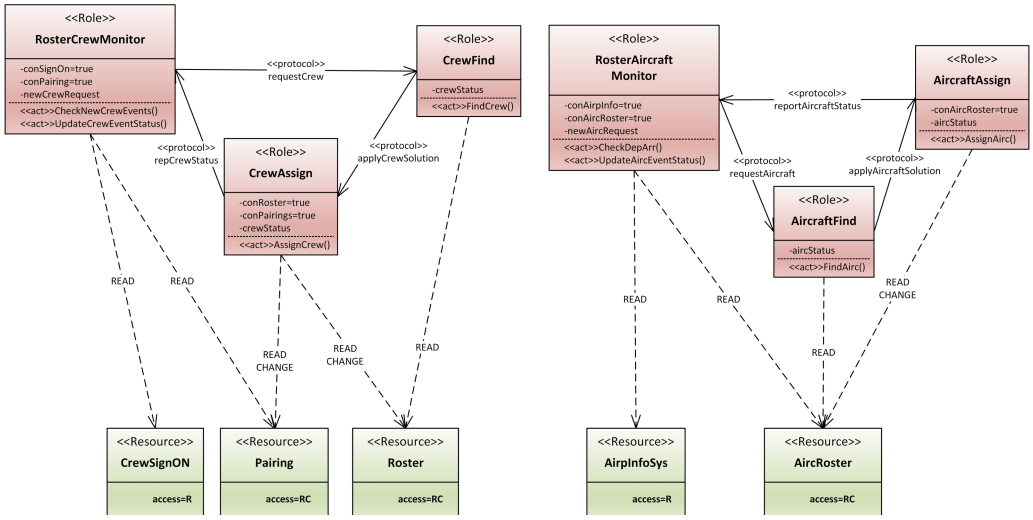

Fig. 5.6 UML Combined Diagram with Preliminary Roles, Protocols and Environment (partial)

- *Liveness organizational rules (relations)*, define ”how the dynamics of the organization should evolve over time”. For example, a specific role can be played by an entity only after it has played a given previous role.
- *Safety organizational rules (constraints)*, define ”time-independent global invariants for the organization that must be respected”. For example, two roles cannot be played by the same entity.

The formalism to express these rules can be the same used for the liveness and safety rules for the roles. They will be expressed by liveness and safety expressions respectively.

In summary, liveness expressions detail properties related to the dynamics of the organization, that is, how the execution must evolve and safety expressions detail properties that must always be true during the whole life of the MAS. A partial list of the liveness organizational rules (relations) we have defined for MASDIMA can be found in table 5.7 and in table 5.8 a partial list of the safety organizational rules (constraints).

Table 5.7 Liveness rules (relations)

Liveness rule or relations
$applyCrewSolution(CrewAssign(crew(x))) \rightarrow repCrewStatus(CrewAssign(crew(x)))$
Protocol *applyCrewSolution* must necessarily be executed by role *CrewAssign* for a specific crew solution *crew(x)* before role *CrewAssign* can execute protocol *repCrewStatus* for that crew solution.
~~$requestCrew(CrewFind(request(x))) \rightarrow applyCrewSolution(CrewFind(crew(x)))$~~
Protocol *requestCrew* must necessarily be executed by the role *CrewFind* for a specific request *request(x)* before role *CrewFind* can execute protocol *applyCrewSolution* for the solution found.

Table 5.8 Safety rules (constraints)

Safety rules or constraints	Description
¬(*RosterCrewMonitor*)|*CrewFind*)	Role *RosterCrewMonitor* and role *CrewFind* can never be played concurrently by the same entity.
¬(*RosterCrewMonitor*)|*CrewAssign*)	Role *RosterCrewMonitor* and role *CrewAssign* can never be played concurrently by the same entity.

5.5 Architectural Design

The analysis phase, presented on the previous section, has the objective of understanding what the MAS *will have to be*, i.e., what it is expected to do. The deliverables of the analysis (i.e., environment model and preliminary roles and interaction model) express the functionality and operational environment of the MAS. In the architectural design phase it is necessary to make decisions regarding the actual characteristics of the MAS. So, besides completing and refining the preliminary models, the design will rely in actual decisions about the organizational structure and in modeling the MAS based on the specifications produced.

The architectural design phase has three processes: (i) define the organizational structure; (ii) define the final role model and (iii) define the final interaction model. The outputs of this phase are the role and interaction model and the main diagram used is the UML Combined diagram.

The choice of the organizational structure is very important and will affect the development of the succeeding phases. For that we need to choose the desired topology and control regime to be applied. The *GAIA* paper (Zambonelli *et al.*, 2003) has a very good explanation of this important step. Another very useful reading is Mark S. Fox's paper on organizational theory (Fox, 1981).

5.5.1 Defining the organizational structure

To define the organizational structure we will continue to use as an example the MASDIMA system. From the specification documents of the analysis phase, we found the following main requirements to consider for defining the organizational structure:

- *From the Actors, Goals and Dependencies diagram*: The main organization (AOCC) has three sub-organizations, that is,
  1. Aircraft Manager.
  2. Crew Manager.
  3. Passenger Manager.
- *From the Environment model*: We have identified the following active component (resources that will be agentified), that is, Operational Control Supervisor (human authority).
- *From the Preliminary Role model*: We have identified a requirement that the *CrewFind* role and *AircraftFind* role use different techniques to find the solutions.
- *From the Organizational rules*: We have identified the roles that cannot be played concurrently by the same entity, such as (this is a partial list), *RosterCrewMonitor* and *CrewFind*, *RosterCrewMonitor* and *CrewAssign*, *AircraftFind* and *PaxFind.*

Having this information we defined the organization structure of our MAS. Table 5.9 gives a summary of the topologies and control regimes applied.

Table 5.9 Topologies and control regime

Organization	Topology	Control Regime
AOCC	Multilevel hierarchy	Mixed: cooperative and authoritative.
Crew Manager	Multilevel hierarchy	Work specialization.
Aircraft Manager	Multilevel hierarchy	Work specialization.
Passenger Manager	Hierarchy	Work specialization.

The control regime was defined following the guidelines of (Zambonelli *et al.*, 2003; Fox, 1981). In the AOCC organization we have cooperative control regime between the *Managers'* roles due to the *peer relation* among them, and authoritative from *Operational Control Supervisor* to the *Managers'* roles due to the *control relationship* (for example, *Operational Control Supervisor* controls *Aircraft Assign* role). The work specialization control regime on the other organisations is derived from the fact that the role (for example, *CrewFind* or *AircraftFind*) provides specific services.

To represent the organizational structure, *GAIA* suggests a coupled adoption of a formal notation and of a more intuitive graphical representation. Table 5.10 is a formal notation of the organization structure that we defined for the *Passenger Manager* sub-organization. Please note that the relationship types identified here are neither mutually exclusive (for example, a control relation type may also imply a dependency relation type), nor complete (other types of relations may be identified). A (partial) UML representation of the organization structure is presented in figure 5.7 following the suggestions of (Bauer & Odell, 2005).

Table 5.10 Organization structure for passenger recovery

Statement/Comment
$\forall i,\quad OperationalControlSupervisor \overset{control}{\rightarrow} PaxApply[i]$
This means that the role *OperationalControlSupervisor* has an authoritative relationship with role *PaxApply*, controlling, in this case, all the actions of role *PaxApply*. Specifically, role *PaxApply* needs approval from *OperationalControlSupervisor* before applying the solution. Please note that role *OperationalControlSupervisor* is shared between this sub-organization and *Aircraft Manager* sub-organization.
$\forall i,\quad OperationalControlSupervisor \overset{depends\ on}{\rightarrow} PaxFind[i]$
This means that the role *OperationalControlSupervisor* relies on resources or knowledge (a solution found to solve a passenger problem) from the role *PaxFind* to accomplish its task (i.e., to authorize or not authorize the assignment of a specific solution).
$\forall i, j, PaxFind[i] \overset{depends\ on}{\rightarrow} PaxMonitor[j]$
This means that the role *PaxFind* relies on resources or knowledge (an event related to a passenger problem) from the role *PaxMonitor* to accomplish its task (i.e., to find a solution to the passenger problem).

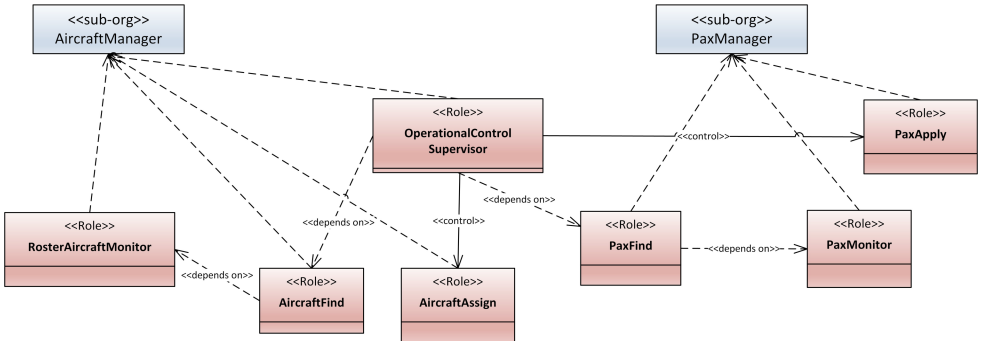

Fig. 5.7 Organization Structure (partial) in a UML Diagram

To be able to represent the organization structure in UML we made some mappings between the abstractions used here and the UML artifacts as well as created some stereotypes, as follows:

- *Organization Abstraction*: We have mapped this abstraction to a package. In UML packages provide a way to group related elements. Using package diagrams it is possible to visualize dependencies between parts of the system. If we see an organization as a package we can take advantage of these characteristics and model them using package diagrams. For the goal of the organization we can use a note or constraint to represent it. We have created a sub-organization stereotype associated to the package metaclass of UML.
- *Depends on Abstraction*: We have mapped this abstraction to a dependency relationship. The dependency relationship in UML is the weakest it is possible to define. A dependency between classes means that one class uses, or has knowledge of, another class. They are typically read as "... uses a ...". A *depends on* relation between two roles usually means that one role relies on resources or knowledge from the other role. We have created a *depends on* stereotype associated to the dependency metaclass of UML.
- *Controls Abstraction*: We have mapped this abstraction to an association relationship. Association relationships in UML are stronger than dependencies and typically indicate that one class retains a relationship to another class over an extended period of time. They are typically read as "... has a ...". A control relation between two roles usually means that one role has an authoritative relationship with the other role, controlling its actions. We have created a control stereotype associated to the association metaclass of UML.
- *Peer Abstraction*: We have also mapped this abstraction to a dependency relationship. A peer relation between two roles usually means that they are at the same level and collaborate to solve problems. We have created a peer stereotype associated to the dependency metaclass of UML.

5.5.2 Completing the role and interaction model

After having the organization structure it is possible to complete the role and the interaction model. Some roles' interactions result from the organization topology and the protocols that need to be executed from the control regime defined. The tasks that are necessary to be performed to complete both models are:

- Complete the *activities* in which a role is involved, including its liveness and safety responsibilities.
- Define *organizational roles*, that is, those whose presence was not identified during analysis and that result directly from the adopted organization structure.
- Complete the definition of protocols specifying which roles the protocol will involve.
- Define *organizational protocols*, that is, those whose identification derives from the adopted organization structure.

It is important to notice the distinction between characteristics that are *intrinsic* from the *extrinsic*. *Intrinsic* are independent of the use of the role and/or protocol in a specific organization structure. *Extrinsic* are the ones that derive from the adoption of a specific organizational structure. This distinction is important in terms of reuse and design for change.

Table 5.11 RosterCrewMonitor role (Final)

Role Schema: RosterCrewMonitor

Description: Monitors the crew roster for events related to crew members not reporting for duty and/or flights with open positions. After detecting one of these events, it will request a solution from the organization. Traces previous requests and avoids duplicates, until it receives a message regarding the status of the request.

Protocols and Activities: CheckNewCrewEvents, UpdateCrewEventStatus, requestCrew, repCrewStatus

Permissions:
reads CrewSignON (to obtain all who did not report for duty)
reads Pairings (to obtain all flights with open positions)
changes CrewEvents (keep log of events)
Responsibilities:
Liveness:
$RosterCrewMonitor = (\underline{CheckNewCrewEvents}^{W}.requestCrew)^{W}$
$(repCrewStatus^{W}.\underline{UpdateCrewEventStatus})^{W}$
Safety:
successful connection with CrewSignON = true
$successful\ connection_{w}\ with\ Pairings = true$
$successful\ connection_{w}\ with_{C}rewEvents = true$
new crew request <> existing unclosed crew request

Table 5.11 is an example of a final role specification taken from the MASDIMA complete role model. It is important to point out the differences from the final role schema when comparing with the preliminary one (from table 5.6). First, the liveness responsibilities were completely defined and, second, due to the work done during the architectural design, a new resource was identified: *CrewEvents*. This new resource will keep a record of events status. The permissions property was updated to reflect the access to this new resource. The liveness property specifies what activities and protocols the role will have. They express part of the role's expected behavior. In our example, *activities* appear underlined and express actions performed by the role that do not involve interaction with any other role (similar to a method in object oriented terms). *Protocols* are activities that do require interaction with other roles. From the liveness expression of our example

$$RosterCrewMonitor = (\underline{CheckNewCrewEvents}^{W} \,.\, requestCrew)^{W}$$
$$(repCrewStatus^{W} \,.\, \underline{UpdateCrewEventStatus})^{W}$$

We can see that role *RosterCrewMonitor* consists of executing the activity CheckNewCrewEvents indefinitely (marked by the *W* operator), followed by the execution of the protocol *requestCrew*. Both of these are performed indefinitely. In parallel (marked by the operator) it executes the protocol *repCrewStatus* indefinitely followed by the activity UpdateCrewEventStatus. Both also performed indefinitely.

Regarding the final interaction model, Figure 5.8 shows an example taken from the MASDIMA complete interaction model. The important thing to point out in here is the distinction that is made

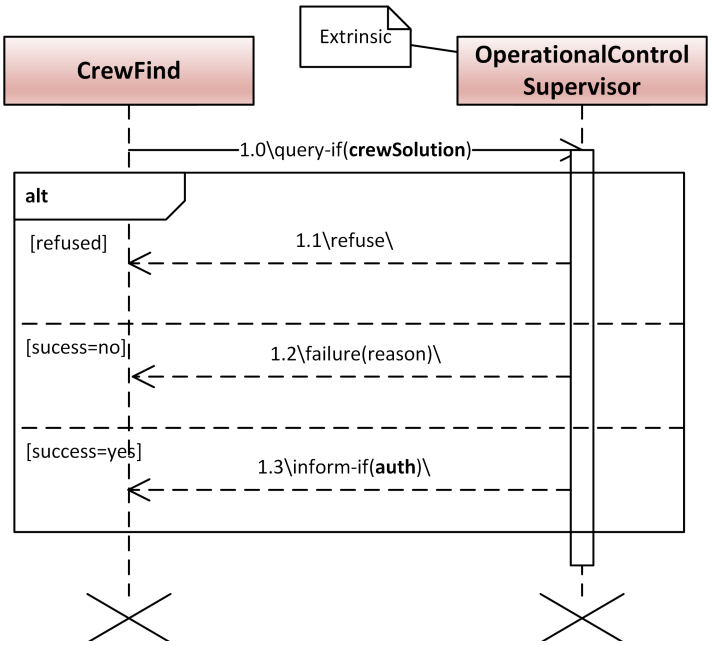

Fig. 5.8 UML Interaction Diagram for *sendCrewSolution*

regarding the extrinsic characteristic. In this specific example, it is the *OperationalControlSupervisor* partner that is specific to the organization structure defined. This information might be important if we, later, decide to change the organization structure.

At this stage it is desirable to draw a UML Interaction Diagram similar to the one in figure 5.8, for all protocol definitions of our interaction model. Finally, the preliminary UML Combined Diagram from the Analysis phase, should also be updated with the final role and interaction model. A partial example of such a diagram taken from MASDIMA is presented in Figure 5.9.

5.6 Detailed Design

This phase is responsible for identifying the agents and services that will implement the roles, functions and interactions identified so far. It will take into consideration the spatial and physical distribution that is going to be adopted by the MAS.

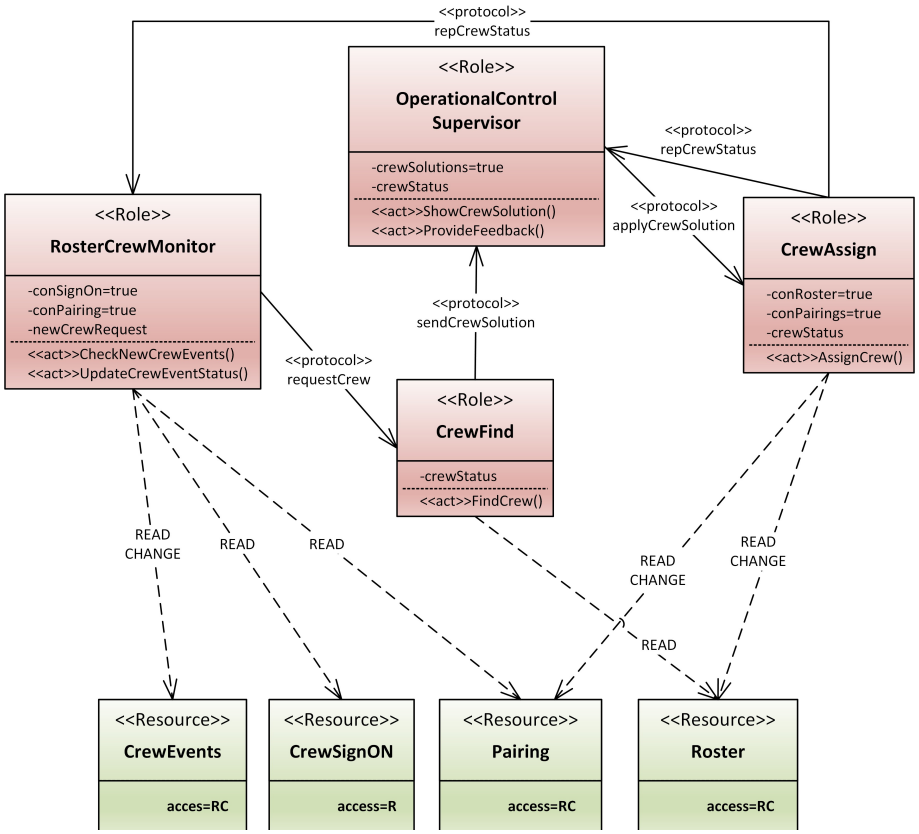

Fig. 5.9 UML Combined Diagram (partial) with final roles, interaction and environment

The Design phase has two processes, i.e., define agents and define services and the outputs are the Agent model and the Service model. They will show the agents that will be implemented as well as the services that will be necessary to implement by each one of them. It will be a programming language/middleware neutral specification. This phase uses two UML diagrams: the UML Agent Diagram and the UML Services Diagram.

5.6.1 Define Agents

To build the agent model it is possible to make a one-to-one correspondence between roles and agent classes. However, there are some advantages in trying to find a better mapping. The better one is to try to compact the design by reducing the number of classes and instances leading to a reduction in conceptual complexity. This has to be done without: affecting the organizational efficiency, violating the organizational rules and creating ”bounded rationality” problems (that is, without exceeding the amount of information it is possible to process in a given time). *GAIA* does not specify any special notation for showing the agent model although it implicitly suggests the adoption of a class model diagram. A simple way of representing the Agent Model is to use a table like the one we present in 5.12.

Table 5.12 Agent Model (partial)

Agent classes/roles
$OpMonitor^{1..n} \xrightarrow{play} RosterCrewMonitor, RosterAircraftMonitor, PaxMonitor$ This means that agent class *OpMonitor* will be defined to play the roles *RosterCrewMonitor*, *RosterAircraftMonitor* and *PaxMonitor*, and that we will have between one and *n* instances of this class in our MAS (*n* depends on the functional and physical distribution adopted).
$OpAssign^{1..n} \xrightarrow{play} CrewAssign, AircraftAsssign, PaxApply$ This means that agent class *OpAssign* will be defined to play the roles *CrewAssign*, *AircraftAssign* and *PaxApply*, and that we will have between one and *n* instances of this class in our MAS (*n* depends on the functional and physical distribution adopted).
$OpSupervisor^{1} \xrightarrow{play} OperationalControlSupervisor$ This means that agent class *OpSupervisor* will be defined to play the role *OperationalControlSupervisor*, and that we will one instance of this class in our MAS.

5.6.2 Define Services

The services derive from the protocols, activities and liveness expressions of the roles that each agent implements. Usually, there will be one service for each parallel activity of execution that the agent has to execute. According to *GAIA*, the service model requires that, for each service that may be performed by an agent, four properties are identified: inputs, outputs, pre-conditions and post-conditions. The *inputs* and *outputs* are derived from the interaction model and from the environment model. If the service involves elaboration of data and the exchange of knowledge between the agents, they will come from the protocols. If the service involves evaluation and modification of the environment resources, they will come from the environment. The *pre* and *post conditions* represent restrictions on the execution and completion, respectively, of the services. They derive from the role safety properties as well as from organizational rules. Applying the above guidelines we obtain the service model. In table 5.13 we represent some services for agent class *OpMonitor* and in table 5.14 some services for agent class *OpSupervisor*, both from the MASDIMA example.

5.6.3 UML Representation

As we stated before, *GAIA* does not propose any notation to represent the Agent and Service model besides the tabular simple notation we presented in the previous section. In this section we are going to show how we have represented this two models using a UML diagram. Figure 5.10 shows the UML Agent Model (partial and simplified) we have defined for the MASDIMA and Figure 5.11 shows the UML Service Model (partial and simplified) for the agent class *OpMonitor*.

To do that we have adopted the following mappings between the methodology abstractions and UML concepts:

Table 5.13 Services (partial) agent class OpMonitor

Service Description
Service: Monitor Crew Events. *Input*: current date, crew slack time, pairing slack time. *Output*: A list of dutyID, crewNumber, prngNumber, listOpenPositions, eventID. *Pre-condition*: Successful connection with CrewSignON and Pairing resources. *Post-condition*: A new crew event that has to be different from an existing unclosed one.
Service: Update crew event status. *Input*: eventID, eventStatus. *Output*: Number of records updated. *Pre-condition*: Successful connection with CrewEvents resource. *Post-condition*: Successful update of the CrewEvents resource.

Table 5.14 Service (partial) agent class OpSupervisor

Service Description
Service: Obtain crew solution authorization. *Input*: List of crew members to be assigned. *Output*: Authorization status (OK or NOT OK). *Pre-condition*: At least one crew solution found. *Post-condition*: User confirms or does not confirm authorization.
Service: Request crew solution application. *Input*: Authorized list of crew members to be assigned. *Output*: Request status (YES = solution can be applied. NO = solution cannot be applied). *Pre-condition*: Authorization status = OK. *Post-condition*: User sees status of the request on the screen.

- *Agent Class Abstraction*: We have mapped this abstraction to a class and created an *Agent Class* stereotype associated to the class metaclass in UML. To identify the roles that each agent class implements we have created a *role* stereotype associated to the property metaclass in UML. The instances of the agent class are represented using *Constraints*.
- *Services Abstraction*: We mapped the services abstraction to an *interface*. In UML an *interface* is a classifier that has declarations of properties and methods but no implementations. It provides a contract that a classifier that provides an implementation of the interface must obey. The inputs are represented as properties of the interface and the method represents the service that needs to be implemented. The outputs are what the method returns. For example, the interface *MonitorCrewEvents* represents the *service* with the same name and attributes *currentDate*, *crewSlackTime* and *pairSlackTime* are the inputs. *CheckNewCrewEvents* is the operation to be implemented. The agent class *OpMonitor* realizes that interface by providing an implementation for the operations and properties (the dashed line starting at the agent class to the interface, with a closed arrowhead at the end, shows this realization).
- *Pre and Post-conditions*: We present the pre/post-condition using constraints, associated to the specific interface. For example, the *MonitorCrewEvents* interface has the following pre-condition and post-condition, respectively:

$$connCrewSignON = true;\ connPairings = true$$

$$new\ \underline{event} <> existing\ \underline{event}$$

It is important to note that *Invariant Constraints*, that is, constraints applied to all instances of the class are not reflected in this diagram. It is possible to do it by applying domains, attribute types, attribute multiplicity and valid values of attributes.

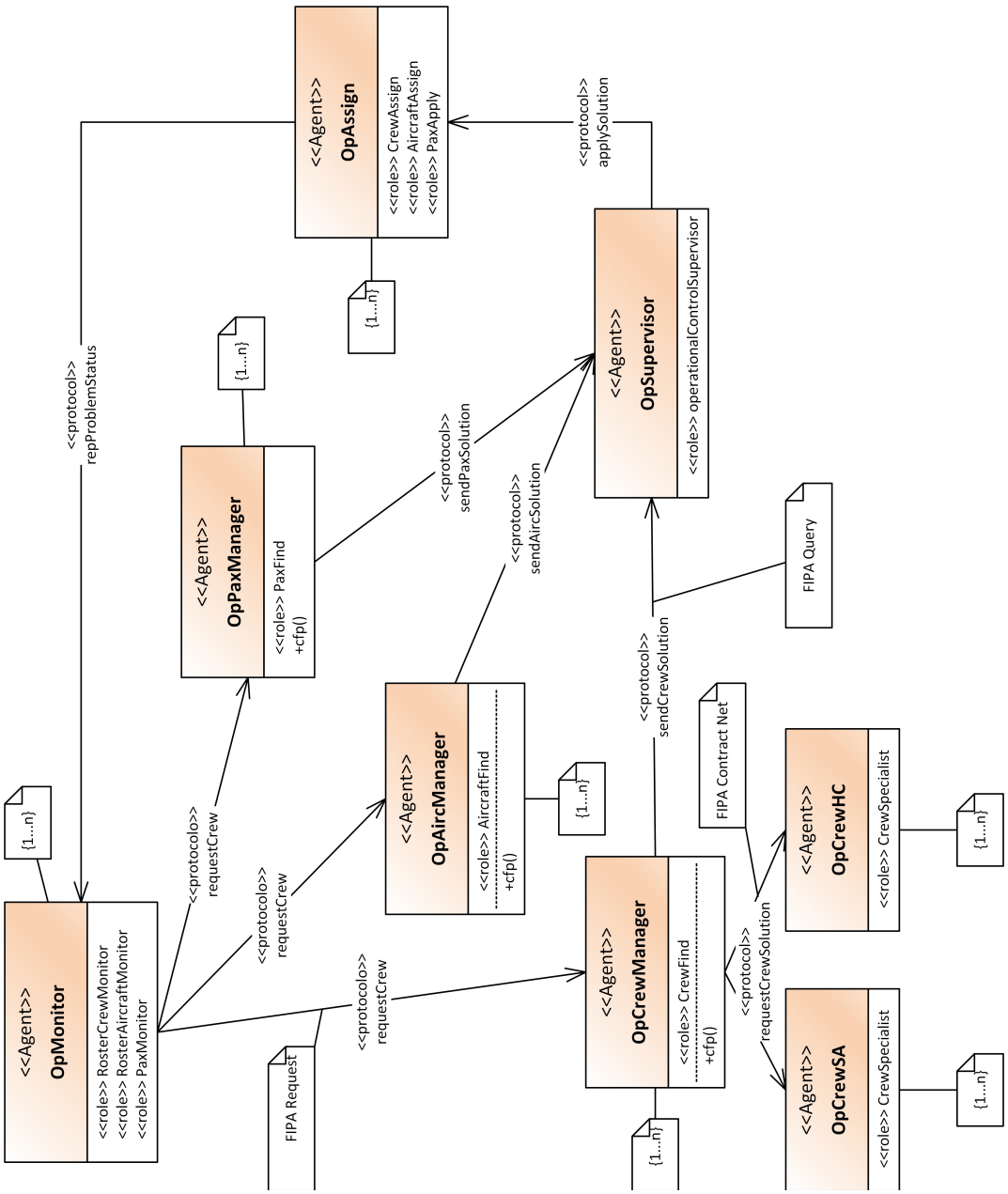

Fig. 5.10 Simplified UML Agent Model (partial)

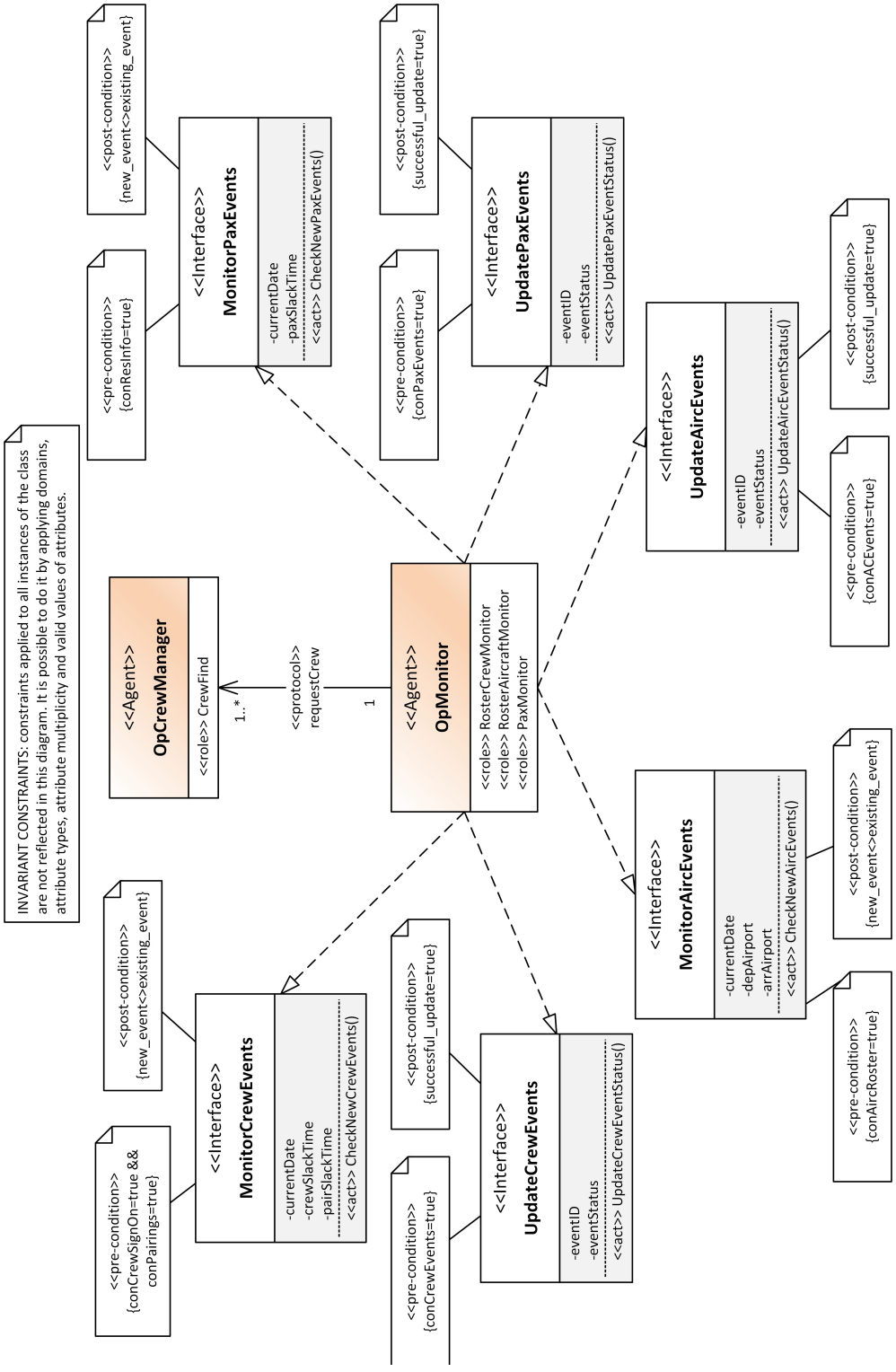

Fig. 5.11 Simplified UML Service Model (partial)

5.7 Implementation

GAIA methodology does not include tools for describing an Implementation phase. According to its authors ”after the successful completion of the design process, developers are provided with a well defined set of agent classes to implement and instantiate, according to the defined agent and services model”. This section is our approach to include an implementation phase in the methodology and is the result of the experience we had in developing the MASDIMA system.

The implementation phase includes three processes: (i) identification of concepts and actions; (ii) mapping of services to behaviours and (iii) development. The output of this phase will be the *system source code*. Although these processes are generic enough to be used independently of the programming language used, here we are going to use JAVA and JADE (Bellifemine *et al.*, 2004) as examples, because they were the language and framework we used to implement MASDIMA.

JADE is a software development framework written in Java language aimed at the development of MAS. JADE also works as a distributed agent platform across several hosts. Another important feature is that JADE is a FIPA[7] compliant agent platform and provides implementations of agent communication language (ACL) messages between agents as well as standard interaction protocols (such as FIPA-request, FIPA-query, etc.).

To start the implementation it is necessary to map between the detailed design obtained from *GAIA* and the language/middleware used. We have defined four tasks to be performed:

1. Model the interaction between the several agents (in terms of communications and how to represent the content of messages), identifying the proper concepts and actions and defining them as classes, deciding which of the methods (serialized objects or extensions of predefined classes) will be the ideal to use.
2. Define a notation to be used for the names of Agents, Services and Protocols according to the implementation language and their best practices.
3. Relate each one of the services in the service model to the possible behaviours to use, according to the necessary activities to be performed.
4. Define for each one of the interactions protocol in the model, the necessary performatives to use and why. It should also reflect the choice of using a standard protocol or the choice of building a new one.

The following sections explain how the above tasks were included in each of the processes defined.

5.7.1 Identification of Concepts and Actions

In this process the first task is to define the vocabulary and semantics for the content of the messages that will be exchanged by the agents in the system. This can be different according to the programming language and middleware used for implementation. In the case of JADE and Java, it provides three ways to implement communication between agents regarding the content of the messages:

1. The use of strings.
2. Transmission of serialized Java objects.
3. Ontology classes taking advantage of the standard FIPA format.

[7] http://www.fipa.org

Before deciding which of the methods to use and after reviewing the interaction, environment, agent and service models, we can identify the necessary concepts and actions. Some of the concepts and actions, taken from the MASDIMA, are represented in Table 5.15. In MASDIMA we chose to pass the content of messages as objects. However, if we are modeling an *open* MAS, where the agents need to interoperate with agents designed and implemented by different system designers, the best choice might be to use ontology classes.

Table 5.15 Some concepts and actions from MASDIMA

Concepts	Description
CrewEvent	Characterizes a crew event that initiates the process of finding a crew solution.
CrewSolutionList	Characterizes a list of crew solutions proposed by the agents that are specialists in crew problems to the *OpCrewManager* corresponding to the *CFP* initiated after a crew event has been detected.
CrewSolution	Characterizes the crew solution chosen by agent *OpCrewManager*, that will be presented to the *OpSupervisor* for authorization.
Actions	**Description**
ApplyCrewSolution	Action of applying the crew solution after it has been authorized.
UpdateEventStatus	Action of making the status update of a crew/aircraft or passenger event.

The second task to be performed in this process is to define the implementation notation to use for agents, protocols and services. During the design phase, the notation used is defined according to the modeling tool used (in the case of *PORTO* is UML) and the names used for the concepts reflect that choice. However, for the implementation, it is important to follow the programming language and middleware guidelines. So, in this task, besides defining the notation we also map between the names used in the design and the new names defined for implementation. Table 5.16 presents a partial list of the notations and mappings defined for the MASDIMA system.

At the end of this process we get the following:

1. A list of concepts and actions that, using the chosen programming language, need to be implemented.
2. An implementation notation for the concepts, agents, protocols and services according to the programming language and middleware best practices and guidelines.
3. A mapping between the names used in design time and the names chosen for implementation.

5.7.2 Mapping Services to Behaviours

The goal of this process is threefold:

1. Map the services that need to be implemented to the behaviours provided by the programming language and middleware that are more suitable to be used.
2. Choose between using a standard protocol and/or implement new ones.
3. Considering the choice made regarding the protocols, define the performatives to be used.

Table 5.16 Agents, Protocols and Services notations (partial) from MASDIMA

Design Name	Implementation Name
Agents	
OpMonitor	MonitorAgent
OpAircManager	AircraftManagerAgent
OpCrewManager	CrewManagerAgent
OpPaxManager	PaxManagerAgent
OpCrewSA	CrewSASpecialistAgent
OpCrewHC	CrewHCSpecialistAgent
OpAssign	SolutionAssignAgent
OpSupervisor	SupervisorAgent
Protocols	
requestSolution	request-solution
sendAircSolution	send-aircraft-solution
sendCrewSolution	send-crew-solution
sendPaxSolution	send-pax-solution
requestCrewSolution	crew-solution-negotiation
applySolution	request-apply-solution
Services	
MonitorCrewEvents	MonitorCrewEvents
UpdateCrewEvents	UpdateCrewEvents
RequestSolutionApplication	RequestApplySolution

Regarding the first and second, the choice of behaviours (or other similar concept) and standard protocols is limited by the programming language and middleware used. In the case of MASDIMA all services will be implemented with JADE behaviours that will *run* inside or extend a JADE *CyclicBehaviour*. This is necessary because all agents will be running indefinitely, as it is possible to infer from the liveness expressions of the roles that each agent represents. The agents will perform indefinitely some services (for example, monitoring) and/or waiting for a message to act (for example, messages that initiate interactions protocols that they need to be part of). Regarding the standard protocols and as stated before, JADE is FIPA compliant and, as such, the FIPA standard protocols are available. Table 5.17 shows a partial list of the mappings between services and JADE behaviours and FIPA protocols.

Table 5.17 Mapping (partial) of JADE behaviours and Services in MASDIMA

Service: MonitorCrewEvents *JADE Behavior:* Ticker *FIPA/JADE IP:* fipa-request *Protocol implementation name:* request-solution
Service: FindCrew *JADE Behavior:* Simpler *FIPA/JADE IP:* fipa-request; fipa-contract-net *Protocol implementation name:* request-solution; crew-solution-negotiation

Regarding the third, the performatives are related to the interaction protocols used. So, if the designer chooses to follow a standard (like FIPA-ACL performatives) there is no need to create

new performatives. As an example, in the MASDIMA system we used the FIPA-ACL performatives and in the case of the *crew-solution-negotiation* protocol we used the *FIPA contract net*. In this case, the performatives are: cfp, refuse, propose, *reject_proposal*, *accept_proposal*, failure, inform (done), and inform (result).

5.7.3 Development

The last process in the implementation phase is the development of the software system. Having all the information collected from the previous steps it is possible to use a suitable development environment to start coding. It is important to point out that it is in this phase, that the *unit tests* are performed and not in the Test and Validation phase. *Unit Testing* is a method by which individual units of source code are tested to determine if they work as expected. As such, they are part of the development phase. Developers should follow the best practices and guidelines of the development tool and programming language used to implement the system.

In Appendix B, Section B.1 we present some examples of how we have implemented in the MASDIMA system, some of the concepts defined previously, i.e:

- The *CrewEvent* concept.
- The agent *MonitorAgent* that implements the *RosterCrewMonitor*, *RosterAircraftMonitor* and *PaxMonitor* as well as the *Monitor* and *Update* services associated, implemented using JADE behaviours *Ticker* and *OneShot*.
- The *MonitorCrewEvents* through the JADE *TickerBehaviour*.

The output of this process is the System Source Code that, after compilation, can be tested and validated as we propose in the Test and Validation phase.

5.8 Test and Validation

The Test and Validation phase (or Verification and Validation) is the process of checking that a software system meets specifications and that it fulfills its intended purpose. In some literature it may also be referred to as Software Quality Control. Although it seems similar, *verification* and *validation* are different concepts. The definition of these two concepts, according to the *Capability Maturity Model* (Paulk *et al.*, 1993), is as follows:

- *Verification*: The process of evaluating software to determine whether the products of a given development phase satisfy requirements imposed at the start of that phase.
- *Validation*: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.

The goal of *verification* is to ensure that the system has been built according to the requirements and design specifications, i.e., to ensure that the *right thing was built*. *Validation* has the objective of ensuring that the system meets the needs of the users and stakeholders, and that the specifications were correct, i.e., to ensure that *it was built right*. *Validation* confirms that the system, as provided, will fulfill its intended use.

In this phase we have two processes: definition of the test cases and execution of the test cases. The outputs will be the test plan for the first process and the test results for the second one. This phase ends when the test results are positive for each and every test case.

5.8.1 Define Test Cases

Test cases are prepared for *verification*, i.e, to determine if the process that was followed to develop the final system is right. To fully test that all requirements of a system are met, it is necessary to define test cases that cover all requirements, testing not only the return defined according to the requirements but, also, return that is not defined on the requirements.

As one might expect, it is important to keep track of the link between the requirement and the tests. One way of doing that is to use a *traceability matrix* (Carlos, 2012). A traceability matrix is a document, usually in the form of a table, that relates each requirement to the test cases.

The written test cases should include the following (an example is presented in Table 5.18):

- A description of the functionality to be tested.
- The preparation required to ensure that the test can be conducted.
- Known input (tests a precondition).
- The test step number if applicable.
- The related requirement or requirements.
- The name of the person that performed the test.
- Expected Result, i.e., the result that should be obtained according to the requirements (tests a post-condition).
- Actual Result, i.e., the result after the test has been performed.
- Test result, i.e., pass or fail.
- Remarks about the execution of the test.

As we stated in the Implementation phase, the test cases designed here are not the *Unit Tests* designed and performed during implementation. The goal of the *Unit Tests* is to test specific parts of the code and the test cases aim at testing the response of the system according to the requirements specification.

The output of the *Define Test Cases* process is *Test Plan*, i.e., the set of all test cases designed to test the system. As one might expect, the most time consuming part is the creation of the tests and modifying them when the system changes. The *Test Plan* will be used by the system testers to validate the system.

5.8.2 Perform Tests

After getting the *Test Plan*, it is possible to perform the final process in this *Test and Validation* phase, i.e., validation. As we stated before, the objective is to validate if the system is built according to the requirements of the user.

The first thing to do, is to assign software testers to each test case, so that they can perform the tests. It is important to point out that someone on the team (typically the quality control or software test manager) should keep track of the tests, the person who performed it and the result.

Table 5.18 Test case example from MASDIMA

Test Case ID:	MON-023
Tester:	Pedro Rica
Summary Functionality:	Detect flight problem and ask request solution.
Requirement(s):	RQ-017 and RQ-021
Preparation:	Flights in operation should appear in *Flight Monitoring* window.
Step number:	01
Step Input:	An event should create a flight departure delay.
Step description:	Look at the *Flight Problem* window.
Step Expected Result:	The flight number, schedule departure time, expected delay, number of violations and unsolved status, should appear.
Step Actual Result:	The same as the expected result.
Step number:	02
Step Input:	An event should create a flight departure delay.
Step description:	Look at the *Flight Map* window.
Step Expected Result:	The flight affected should have a red circle blinking.
Step Actual Result:	The same as the expected result.
Step number:	03
Step Input:	The flight with the unsolved problem should appear in the *Flight Problem* window.
Step description:	a) Click in the flight number. b) On the *Solution* window click on the *Supervisor Default Values* tab.
Step Expected Result:	The correct supervisor default values should appear for each dimension.
Step Actual Result:	The same as the expected result.
Step number:	04
Step Input:	The flight with solved status should appear in the *Flight Problem* window.
Step description:	a) Click in the flight number. b) On the *Solution* window click on the *Solution Proposal* tab. c) On the *Solution* window click on the *Solution Plan* tab.
Step Expected Result:	a) Should appear values for delays and cost for each dimension as well as the solution utility. b) The actions to be applied in the operational plan should appear for the dimensions.
Step Actual Result:	The same as the expected result.
Test Result:	PASS
Remarks:	In step 02 the blinking should stay for, at least, 10 seconds.

The second thing to do is to perform each test case. The software tester should assign *PASS* or *FAIL* in the *Test Result* as well as any remarks on the *Remarks* field. Finally, having all the tests performed, the *software test manager* should review the test results with the rest of the team and pass the information so that the tests that failed can be corrected.

5.9 Chapter Summary

In this chapter we have presented an *AOSE* methodology called *PORTO*, that results from complementing the *GAIA* (Zambonelli *et al.*, 2003) methodology. As we stated in the introduction section, the main goal of our work was not about *AOSE*. However, the contributions in this area appear due to the need we had to model the MASDIMA system, a complex and realistic MAS.

When compared to *GAIA* our approach has the following main differences:

- *Requirements Analysis*: *GAIA* does not have a requirements analysis phase. We have adapted the early requirements analysis of the *TROPOS* (Bresciani *et al.*, 2004) methodology to be included here (Section 5.3). The advantages that emerged from including this phase are presented in Section 5.3.1.
- *Notation*: We have used *UML* to replace or complement the notation proposed by *GAIA's* authors. This has the advantage of replacing an unfamiliar notation by a standard and more familiar notation used by software developers. Specifically, we have replaced or complemented the following:
  - Replace the table notation for the protocol definition by UML Interaction Diagram as presented in Figure 5.8 (Section 5.5).
  - Replace the formal notation representing the organizational structure by a UML Class Diagram as presented in Figure 5.7 (Section 5.5.1).
  - Replace the table representation of the agent model by a UML Class Diagram as in Figure 5.10 (Section 5.6.1).
  - Replace the table representation of the service model by a UML Class Diagram as in Figure 5.11 (Section 5.6.2).
  - A new diagram, called *UML Combined Diagram* that includes the environment, interaction and role model as well as the organization structure (Section 5.4.3). This diagram has the advantage of helping to better visualize the organization with their roles, activities and protocols.
- *Implementation*: *GAIA* does not have an implementation phase. Using our experience in implementing the MASDIMA system we added this phase to the methodology (Section 5.7).
- *Test and Validation*: Like with the previous one, *GAIA* does not have this phase. Again, using the experience obtained in developing the MASDIMA system, we propose to include this phase in the methodology (Section 5.8).

In the next chapter we will present another of the main contributions of our work: *Generic Q-Negotiation Protocol*, a negotiation protocol with adaptive characteristics, that will be used later in our work.