SEHL MELLOULI

FATMAS : A Methodology to Design Fault-tolerant Multi-agent Systems

Thèse présentée
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de doctorat en Informatique
pour l'obtention du grade de Philosophiae Doctor (Ph.D.)

Département d'informatique et de génie logiciel
Faculté des Sciences et Génie
UNIVERSITÉ LAVAL
QUÉBEC

Mai 2005

Résumé

Un système multi-agent (*SMA*) est un système dans lequel plusieurs agents opèrent et interagissent. Chaque agent a la responsabilité d'exécuter des tâches. Cependant, chaque agent, pour diverses raisons, peut rencontrer des problèmes pendant l'exécution de ses tâches ; ce qui peut induire un disfonctionnement du *SMA*. Cependant, le *SMA* doit être en mesure de détecter les sources de problèms (d'erreurs) afin de les contrôler et ainsi continuer son exécution correctement. Un tel *SMA* est appelé un *SMA* tolérant aux fautes.

Il existe deux types de sources d'erreurs pour un agent : les erreurs causées par son environnment et les erreurs dûes à sa programmation. Dans la littérature, il existe plusieurs techniques qui traitent des erreurs de programmation au niveau des agents. Cependant, ces techniques ne traitent pas des erreurs causées par l'environnement de l'agent. Tout d'abord, nous distinguons entre l'environnment d'un agent et l'environnement du SMA. L'environnement d'un agent représente toutes les composantes matérielles ou logicielles que l'agent ne peut contrôler mais avec lesquelles il interagit. Cependant, l'environnment du SMA représente toutes les composantes que le système ne contrôle pas mais avec lesquelles il interagit. Ainsi, le SMA peut contrôler certaines des composantes avec lesquelles un agent interagit. Ainsi, une composante peut appartenir à l'environnement d'un agent et ne pas appartenir à l'environnement du système. Dans ce travail, nous présentons une méthodologie de conception de SMA tolérants aux fautes, nommée FATMAS, qui permet au concepteur du SMA de détecter et de corriger, si possible, les erreurs causées par les environnements des agents. Cette méthodologie permettra ainsi de délimiter la frontière du SMA de son environnement avec lequel il interagit. La frontière du SMA est déterminée par les différentes composantes (matérielles ou logicielles) que le système contrôle. Ainsi, le SMA, à l'intérieur de sa frontière, peut corriger les erreurs provenant de ses composantes. Cependant, le SMA n'a aucun contrôle sur toutes les composantes opérant dans son environnement.

La méthodologie, que nous proposons, doit couvrir les trois premières phases d'un développement logiciel qui sont l'analyse, la conception et l'implémentation tout en in-

tégrant, dans son processus de développement, une technique permettant au concepteur du système de délimiter la frontière du SMA et ainsi détecter les sources d'erreurs et les contrôler afin que le système multi-agent soit tolérant aux fautes (SMATF). Cependant, les méthodologies de conception de *SMA*, référencées dans la littérature, n'intègrent pas une telle technique.

FATMAS offre au concepteur du *SMATF* quatre modèles pour décrire et développer le *SMA* ainsi qu'une technique de réorganisation du système qui lui permet de détecter et de contrôler ses sources d'erreurs, et ainsi définir la frontière du SMA. Chaque modèle est associé à un micro processus qui guide le concepteur lors du développement du modèle. **FATMAS** offre aussi un macro-processus, qui définit le cycle de développement de la méthodologie. FATMAS se base sur un développement itératif pour identifier et déterminer les tâches à ajouter au système afin de contrôler des sources d'erreurs. À chaque itération, le concepteur évalue, selon une fonction de coût/bénéfice s'il est opportun d'ajouter de nouvelles tâches de contrôle au système.

Le premier modèle est le *modèle de tâches-environnement*. Il est développé lors de la phase d'analyse. Il identifie les différentes tâches que les agents doivent exécuter, leurs préconditions et leurs ressources. Ce modèle permet d'identifier différentes sources de problèmes qui peuvent causer un disfonctionnement du système. Le deuxième modèle est le *modèle d'agents*. Il est développé lors de la phase de conception. Il décrit les agents, leurs relations, et spécifie pour chaque agent les ressources auxquelles il a le droit d'accéder. Chaque agent exécutera un ensemble de tâches identifiées dans le modèle de *tâches-environnement*. Le troisième modèle est le *modèle d'interaction d'agents*. Il est développé lors de la phase de conception. Il décrit les échanges de messages entre les agents. Le quatrième modèle est le *modèle d'implémentation*. Il est développé lors de la phase d'implémentation. Il décrit l'infrastructure matérielle sur laquelle le *SMA* va opérer ainsi que l'environnement de développement du *SMA*. La méthodologie inclut aussi une *technique de réorganisation*. Cette technique permet de délimiter la frontière du SMA et contrôler, si possible, ses sources d'erreurs. Cette technique doit intégrer trois techniques nécessaires à la conception d'un système tolérant aux fautes : une technique de prévention d'erreurs, une technique de recouvrement d'erreurs, et une technique de tolérance aux fautes. La technique de prévention d'erreurs permet de délimiter la frontière du SMA. La technique de recouvrement d'erreurs permet de proposer une architecture du SMA pour détecter les erreurs. La technique de tolérance aux fautes permet de définir une procédure de réplication d'agents et de tâches dans le SMA pour que le SMA soit tolérant aux fautes. Cette dernière technique, à l'inverse des techniques de tolérance aux fautes existantes, réplique les tâches et les agents et non seulement les agents. Elle permet ainsi de réduire la complexité du système en diminuant le nombre d'agents à répliquer.

De même, un agent peut ne pas être en erreur mais la composante matérielle sur laquelle il est exécuté peut ne plus être fonctionnelle. Ce qui constitue une source d'erreurs pour le SMA. Il faudrait alors que le SMA continue à s'exécuter correctement malgrè le disfonctionnement d'une composante. FATMAS fournit alors un support au concepteur du système pour tenir compte de ce type d'erreurs soit en contrôlant les composantes matérielles, soit en proposant une distribution possible des agents sur les composantes matérielles disponibles pour que le disfonctionnement d'une composante matérielle n'affecte pas le fonctionnement du SMA.

FATMAS permet d'identifier des sources d'erreurs lors de la phase de conception du système. Cependant, elle ne traite pas des sources d'erreurs de programmation. Ainsi, la technique de réorganization proposée dans ce travail sera validée par rapport aux sources d'erreurs identifiées lors de la phase de conception et provenant de la frontière du SMA. Nous démontrerons formellement que, si une erreur provient d'une composante que le SMA contrôle, le SMA devrait être opérationnel. Cependant, FATMAS ne certifie pas que le futur système sera toujours opérationnel car elle ne traîte pas des erreurs de programmation ou des erreurs causées par son environnement.

Abstract

A multi-agent system (*MAS*) consists of several agents interacting together. In a *MAS*, each agent performs several tasks. However, each agent is prone to individual failures so that it can no longer perform its tasks. This can lead the *MAS* to a failure. Ideally, the *MAS* should be able to identify the possible sources of failures and try to overcome them in order to continue operating correctly ; we say that it should be *fault-tolerant*.

There are two kinds of sources of failures to an agent : errors originating from the environment with which the agents interacts, and programming exceptions. There are several works on fault-tolerant systems which deals with programming exceptions. However, these techniques does not allow the MAS to identify errors originating from an agent's environment. In this thesis, we propose a design methodology, called FATMAS, which allows a MAS designer to identify errors originating from agents' environments. Doing so, the designer can determine the sources of failures it could be able to control and those it could not. Hence, it can determine the errors it can prevent and those it cannot. Consequently, this allows the designer to determine the system's boundary from its environment. The system boundary is the area within which the decision-taking process of the MAS has power to make things happen, or prevent them from happening. We distinguish between the system's environment and an agent's environment. An agent's environment is characterized by the components (hardware or software) that the agent does not control. However, the system may control some of the agent's environment components. Consequently, some of the agent's environment components may not be a part of the system's environment.

The development of a fault-tolerant *MAS* (*FTMAS*) requires the use of a methodology to design *FTMAS* and of a reorganization technique that will allow the *MAS* designer to identify and control, if possible, different sources of system failure. However, current MAS design methodologies do not integrate such a technique.

FATMAS provides four models used to design and implement the target system and

a reorganization technique to assist the designer in identifying and controlling different sources of system's failures. **FATMAS** also provides a macro process which covers the entire life cycle of the system development as well as several micro processes that guide the designer when developing each model. The macro-process is based on an iterative approach based on a cost/benefit evaluation to help the designer to decide whether to go from one iteration to another.

The methodology has three phases : analysis, design, and implementation. The analysis phase develops the *task-environment model*. This model identifies the different tasks the agents will perform, their *resources*, and their *preconditions*. It identifies several possible sources of system failures. The design phase develops *the agent model* and the *agent interaction model*. The agent model describes the agents and their resources. Each agent performs several tasks identified in the task-environment model. The agent interaction model describes the messages exchange between agents. The implementation phase develops the *implementation model*, and allows an automatic code generation of Java agents. The implementation model describes the infrastructure upon which the *MAS* will operate and the development environment to be used when developing the *MAS*. The *reorganization technique* includes three techniques required to design a fault-tolerant system : a fault-prevention technique, a fault-recovery technique, and a fault-tolerance technique. The fault-prevention technique assists the designer in delimiting the system's boundary. The fault-recovery technique proposes a MAS architecture allowing it to detect failures. The fault-tolerance technique is based on agent and task redundancy. Contrary to existing fault-tolerance techniques, this technique replicates tasks and agents and not only agents. Thus, it minimizes the system complexity by minimizing the number of agents operating in the system. Furthermore, FATMAS helps the designer to deal with possible physical component failures, on which the MAS will operate. It proposes a way to either control these components or to distribute the agents on these components in such a way that if a component is in failure, then the MAS could continue operating properly.

The **FATMAS** methodology presented in this dissertation assists a designer, in its development process, to build fault-tolerant systems. It has the following main contributions :

1. it allows to identify different sources of system failure ;
2. it proposes to introduce new tasks in a MAS to control the identified sources of failures ;
3. it proposes a mechanism which automatically determines which tasks (agents) should be replicated and in which other agents ;
4. it reduces the system complexity by minimizing the replication of agents ;

5. it proposes a *MAS reorganization* technique which is embedded within the designed *MAS* and assists the designer to determine the system's boundary. It proposes a MAS architecture to detect and recover from failures originating from the system boundary. Moreover, it proposes a way to distribute agents on the physical components so that the MAS could continue operating properly in case of a component failure. This could make the *MAS* more robust to fault prone environments.

FATMAS alows to determine different sources of failures of a MAS. The MAS controls the sources of failures situated in its boundary. It does not control the sources of failures situated in its environments. Consequently, the reorganization technique proposed in this dissertation will be proven valid only in the case where the sources of failures are controlled by the MAS. However, it cannot be proven that the future system is fault-tolerant since faults originating from the environment or from coding are not dealt with.

Avant-propos

First, I would like to warmly thank my thesis supervisor, Guy Mineau, for his invaluable support and patience, for his precious and never lacking enthusiasm for research and for his continuous assistance in preparing and writing this thesis. I also want to thank him for having so strongly believed in me and provided me with insights which helped me solve many of the problems I encountered during my research. I also acknowledge him for his generosity which gave me opportunities to attend several international conferences.

I would like to express my gratitude to my co-supervisor, Bernard Moulin, whose expertise, understanding and patience, added considerably to my graduate experience. I want to thank him for highly stimulating discussions that have led to the discovery of many results in this thesis.

I would like to thank Daniel Pascot for his valuable contributions in the development of this thesis.

I also want to thank Sylvain Delisle from Université de Trois-Rivières (Québec) for having accepted to evaluate the first version of this thesis. I want to thank Andrea Omicini from Universita di Bologna (Italy) and Marie-Pierre Gleizes from Université de Toulouse (France) for having accepted to be on the committee of the thesis.

My years as a Ph.D student would not have been as much fun without my friends, particularly my colleagues with whom I shared the laboratory and unforgettable moments.

Finally, I would like to thank my mother, all my family, and all my friends for the support they provided me through my entire life, without whose love, encouragement and editing assistance, I would not have finished this thesis.

[À la mémoire de mon père
À ma mère
À ma soeur et mon frère
À tous mes amis
À tous ceux qui m'ont soutenu]

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

A multi-agent system (MAS) is a system in which several agents operate and interact with one another [30]. Each agent is autonomous, reactive and proactive [58]. An agent is autonomous means that the agent is an independent entity that is able to function without direct programmer or user intervention [32]. An agent is reactive means that the behavior of the agent is directed by the goals it has to achieve [32]. An agent is proactive means that the agent can monitor its environments and respond quickly and effectively to changes in those environments [32]. The environment of a MAS is defined as what lies outside the MAS boundary [9]. The MAS boundary is the area within which the decision-making process of the MAS has power to make things happen, or prevent them from happening. Consequently, we can define the environment of an agent as what lies outside the agent boundary. The agent boundary is then defined as the area within which the decision-making process of the agent has power to make things happen, or prevent them from happening.

In a fault-tolerant system, any component acting in this system should be able to overcome any possible failure which could prevent it from achieving its goals and hence the system's goals. If an agent is in failure in a multi-agent system, and if this failure is not originating from the agent's environment, then the agent will be able to prevent this failure from happening since it has the power to prevent things happening if they occur in its boundary. Moreover, if the failure is originating from the agent's environment, then the agent will quickly and effectively respond to that failure since agents are reactive. Furthermore, since agents are proactive, their behavior is directed by their goals. So, if an agent is in failure, then it could not achieve its goals. Hence, the system could not achieve its goals. Consequently, there could be other agents which could not achieve their goals. So, each agent must quickly and effectively respond to these failures in order to overcome them. So multi-agent systems are appropriate solutions to build

fault-tolerant systems [36].

Consequently, fault-tolerance can be dealt with during either a methodological process in which the fault-tolerance aspects are taken into account when designing the future system, or when using a pre-defined structure of the system so that the system adapts itself when failure occurs, or even by applying fault-tolerance to the infrastructure on which the MAS will be deployed :

1. At the methodological level, any design methodology of fault-tolerant MAS should integrate a reorganization technique which allows the MAS to detect and recover from its failures ;
2. An adaptive MAS is defined in [49] as a *self-adaptive software which evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible*. An agent can change its behavior by changing the way it perceives its inputs and by the way it interacts with other agents. An adaptive MAS does not use redundancy so that the system recovers from its failures. It makes the agents adapt their behavior to overcome the encountered failures. An adaptive MAS has a pre-defined structure that must adapt its behavior to encountered failures. The adaptive MAS are used in a forward recovery approach since the system should operate, sometimes, in a degraded mode as in [18] ;
3. Infrastructure faults are dealt with by adopting self-healing MAS. The MAS automatically detects, diagnoses, and repairs localized software and hardware problems [52]. For example, solutions based on broker agents allow to manage the system and detect its component failures and to manage the infrastructure failures such as network failures or machine crashes [29].

The goal of this thesis is to provide a methodology to design *FTMAS*. In this introduction, we first describe the problem that we intend to address in the thesis. Then, we set the objectives of the thesis. Finally, we outline the methodology described in this thesis and give an overview of the remaining chapters of this thesis.

1.1 The Issue

We stated earlier that a MAS is not able to make things happening or prevent them from happening if they are originating from its environment [9]. Hence, the MAS does not control its environment, nevertheless it controls its boundary. Any agent operating in a MAS has two possible sources of failures : its environment or programming exceptions. The programming exceptions are dealt with when the system is programmed and

tested. However, failures coming from the environment can be dealt with when designing the system since they are identified before MAS implementation. Consequently, the development of a fault-tolerant *MAS* (*FTMAS*) requires the use of a *MAS* design methodology which assists the *MAS* designer in identifying the different sources of system failures and determining which of them can be controlled by the system and which could not be. Doing so, the methodology determines the future MAS boundary. We distinguish between the MAS' environment and an agent's environment. The MAS does not control its environment and so the sources of failures situated in the environment. However, the MAS may control an agent's environment and thus the sources of failures of this agent.

Current *MAS* design methodologies such as those described in [2][7][14][17][20][24] [27][31][43][44][57] do not integrate a fault-recovery technique to assist the designer in determining the MAS boundary. Furthermore, several fault-recovery techniques have been proposed in the literature to build *FTMAS* [8] [11] [12] [16] [18] [21] [22] [25] [28] [29] [39] [45] [47] [51]. All these techniques address programming exceptions. None of these techniques address the issue to identify the system boundary in order to determine the sources of failures situated in its environment that the system does not control, and those situated in its boundary that it controls. Consequently, it should be better for the development of a *FTMAS* to use an appropriate design methodology that integrates a reorganization technique [38] from the start. The reorganization technique includes three techniques required to design a fault-tolerant system : a fault-prevention technique, a fault-recovery technique, and a fault-tolerance technique. The fault-prevention technique assists the designer in delimiting the MAS boundary. The fault-recovery technique proposes a MAS architecture allowing it to detect failures. The fault-tolerance technique is based on agent and task redundancy. Contrary to existing fault-tolerance techniques, this technique replicates tasks and agents and not only agents. Thus, it minimizes the system complexity by minimizing the number of agents operating in the system.

1.2 The Objective ?

A software development method as defined in [4] *encompasses a notation whose purpose is to provide common means of expressing strategic and tactical decisions, ultimately manifesting themselves in a variety of artifacts and a process, responsible for specifying how and when certain artifacts should be produced. A notation serves as the language for communicating decisions that are obvious or cannot be inferred from the code itself, provides rich enough semantics, sufficient to capture all important strategic*

and tactical decisions and offers a concrete form for human to reason about decisions.

In this thesis, we aim at providing a fault-tolerant MAS design methodology, called **FATMAS**,. Hence, this methodology must propose notations and models to design the MAS. Furthermore, the methodology must integrate a reorganization technique as stated in [35] [38]. This reorganization technique includes techniques required to build a fault-tolerant system [47] :

1. *A Fault-prevention* technique to prevent the introduction and occurrence of faults ;
2. *A fault-recovery* technique to detect the existence of faults and eliminate them ;
3. A *fault-tolerance* technique to provide services complying with the specifications of the system in case of faults.

FATMAS takes its origins in two related domains : multi-agent systems software engineering and fault-tolerant systems. From a methodological perspective, there are several concepts which should be taken into account when defining the methodology. These concepts define the basis upon which **FATMAS** is built. Doing so, *FATMAS* is not built from scratch, it relies on well-accepted software engineering concepts as presented in Chapter 2.

From a fault-tolerant systems perspective, the reorganization technique includes three techniques for fault-prevention, fault-recovery, and fault-tolerance. The concepts related to these techniques must be integrated with the software engineering concepts. In Chapter 2, we formally define all the methodology concepts and their relations. Doing so, we integrate the reorganization technique within the methodology's models and hence ease the design of a fault-tolerant multi-agent system.

1.3 A MAS Design Methodology With Fault-tolerant Features

FATMAS is a software design methodology. It has three phases which are analysis, design, and implementation. Each phase proposes models to develop. They propose four models and a reorganization technique. The analysis phase proposes *the task-environment model* that determines the different tasks to be performed in the system and assist the designer to delimit the system's boundary. The design phase proposes

the *agent model* and the *agent interaction model*. The agent model that determines the agents that will perform the identified tasks, the agent interaction model that determines the interactions between agents when performing their tasks. The implementation phase proposes the *implementation model* and *code generation.* The implementation model determines the infrastructure upon which the system will operate, and the code generation determines the Java code of the different agents. Finally, the methodology proposes a *reorganization technique* which allows the MAS to control the sources of failures identified in its boundary and overcome them. This technique will be based on the information carried by these models.

Each model is associated with a micro-process which describes how to build it. The methodology is characterized by a macro-process that supports the designer when going through the design steps and the creation of the different models. FATMAS is based on an iterative approach to allow the designer discover sources of failures and introduce new tasks in order to control them. The designer goes from an iteration to another based on a cost/benefit evaluation.

FATMAS deals with faults which can be identified at design level and not on faults originating from coding. Hence, the reorganization technique proposed in this dissertation will be proven valid only in the case where the sources of failures are controlled by the MAS. However, it cannot be proven that the future system is fault-tolerant since faults originating from the MAS environment and from coding are not dealt with.

1.4 Case Studies

The methodology presented in this thesis has been applied to two different types of multi-agent systems :

- One which implements a supply chain that produces mechanical pieces. Multi-agent systems are well-suited for this task because of the complexity of the domain and its natural degree of distribution ;
- One which manages printers in a computer network. The system controls the printers and notifies the users of any problem that occurs in the system, preventing the printing of a document.

This dissertation has 7 chapters. In chapter 2, we present the different concepts which are used in the *FATMAS* methodology. Chapter 3 presents FATMAS, the design methodology that we propose to build fault-tolerant multi-agent systems. This chapter is based on the supply chain example. Chapter 4 presents further case studies to apply

FATMAS. Chapter 5 presents several agent-oriented design methodologies and shows how they can be integrated with FATMAS. Chapter 6 presents related work on fault-tolerant techniques. Chapter 7 concludes this dissertation and presents future research works.

Chapitre 2

FATMAS Basic Concepts

This thesis aims at defining a methodology, *FATMAS*, to design fault-tolerant multi-agent systems. This methodology is built upon different concepts related to multi-agent systems and fault-tolerant systems while it includes a reorganization technique. In this chapter, we first present an ontology that defines the different concepts on which FATMAS is based. These concepts are related to system theory, multi-agent systems, and fault-tolerant systems. Second, in order to formally define the reorganization technique of the methodology, we formalize the definitions and relations between these concepts. The formal definitions will be used to delimit the system's boundary, to propose a fault-recovery technique to deal with possible agents and physical component failures, and to establish a relation between the different models produced in FATMAS. FATMAS models will be defined in Chapter 3.

2.1 Key Concepts for Designing Fault-tolerant Multi-agent Systems

This section defines the key concepts on which our methodology is based, in the form of an ontology. An ontology is defined in [54] as "some sort of world view with respect to a given domain...often conceived as a set of concepts...their definitions and their inter-relationships". So, in this section, we present informal definitions of these concepts and their relationships. In the next section, we propose formal definitions of these key concepts in order to propose our reorganization technique.

Before introducing the concepts related to multi-agent systems and fault-tolerant

systems, we present concepts related to general system theory. This theory is described in [9][40][56]. As presented in Figure 2.1, a system is composed of interacting components that operate together in order to achieve some objectives or purposes. A system is intended to get inputs, process them in some way and produce outputs. The inputs are requests for services sent by the environment to the system. They are considered as *external stimuli* to the system. They are out of control of the system. The outputs are defined as services that the system provides to its environment. The system may require access to resources in order to deliver the services it is requested for.

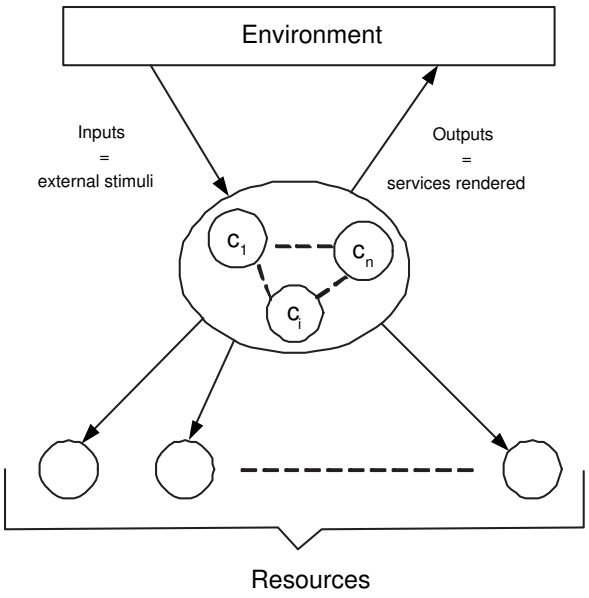

FIG. 2.1 – A system definition.

The external stimuli comes from the environment. Their combination characterizes the situations to which the system must react to. A situation characterizes a snapshot of the environment [34]. However, the external stimuli may be affected by the outputs delivered by the system. Hence, the system outputs have an influence on its environment : they may trigger environmental changes that may result in a situation change.

In a system, some of the components are activated by stimuli originating from the system's environment while others are activated by stimuli originating from system's components. The stimuli originating from the system's components are called the *internal stimuli* [10]. The internal stimuli characterize the observable state of the system. Furthermore, the system must have control on things happening in it. Hence, the internal stimuli must be controlled by the system. Consequently, they characterize the boundary of the system.

Each system's component can perform several tasks. Each task has a set of precon-

ditions which must be met so that the task is properly performed. Also, each task has post-conditions which are conditions that must be met after the execution of the task. If a precondition is not met, the task cannot be performed. Hence, preconditions may prevent tasks from execution. Consequently, preconditions could be possible sources of failures to the system. This will be discussed in the next sections.

A fault-tolerant system is a system which recovers from its failures. Any fault-tolerance technique used in the literature is based on redundancy [8] [11] [12] [16] [18] [21] [22] [25] [28] [29] [39] [45] [47] [51]. Consequently, a fault-tolerant multi-agent system should be based on agent replication. Hence, we introduce the concepts of an *original agent* and a *replica agent*. An *original agent* is an agent which is identified, when designing a MAS, before taking into account the fault-tolerance aspects. A *replica agent* is an agent which is added to the system in order to support fault-tolerance.

From the study above, the basic concepts on which a fault-tolerant MAS design methodology is built are summarized in what follows :

1. System : A system can be viewed as an organization in which agents have to perform tasks [42] ;
2. Component : a component is a software program which operates in the system.
3. Agents : As stated in [32], there is no consensus in the literature on the definition of an agent. However, there is a consensus on the properties of an agent. An agent has at least four properties [32] :
   - Autonomy : agents are able to function without direct programmer or user intervention ;
   - Reactiveness : agents can monitor their environments and respond quickly and effectively to changes in those environments ;
   - Proactiveness : agents have overarching goals that direct behavior over longer periods of time towards achieving complex tasks ;
   - Social ability : since agents operate in dynamic and open environments with many other agents, they must have the ability to interact and communicate with these others.
4. Objective : As defined in [9], and adapted to multi-agent systems, an objective is an end which a set of agents may seek to achieve ;
5. Input : As defined in [9], an input is something that is changed by a transformation process ;
6. Output : As defined in [9], an output is something that is produced by a transformation process ;

7. Resource : As stated in [46], a resource includes for example computational resources, network resources or data resources ;
8. Environment : As defined in [9], the environment is defined as what lies outside the system boundary. The system boundary is the area within which the decision-making process of the system has power to make things happen, or prevent them from happening ;
9. Internal Stimuli : The internal stimuli are the stimuli originating from the system's agents [10] ;
10. External Stimuli : The external stimuli are the stimuli originating from the system's environment [10] ;
11. Task : An abstract description of how the world needs to be transformed in order to achieve a desired behavior or functionality [55] ;
12. Precondition : A precondition is a condition to the execution of a task ;
13. Postcondition : A condition which must be met after the execution of a task ;
14. State : A state is the observable state of the system. It is characterized by the internal stimuli of the system ;
15. Failure : A failure is defined as a result of a task malfunctioning ;
16. Original agent : An original agent is an agent identified at system design before introducing fault-tolerance in the system ;
17. Replica agent : A replica agent is an agent which is added to the system to replicate an original agent in order to take into account fault-tolerance.

The described concepts and links between them are depicted in Figure 2.2. Subclasses are shown by entities having an extra small rectangle on the left. This notation is taken from [46].

2.2 Formal Definitions of Concepts Related to System Theory

At this stage, we identified the different concepts on which FATMAS is based. In this section, we formally define [1] these concepts and their relations in order to propose a reorganization technique which assists the designer in determining the system boundary

[1] We notice that the majority of the definitions provided in this chapter serves to define the reorganization technique. Some of them will be referred to when defining the methodology in Chapter 3.

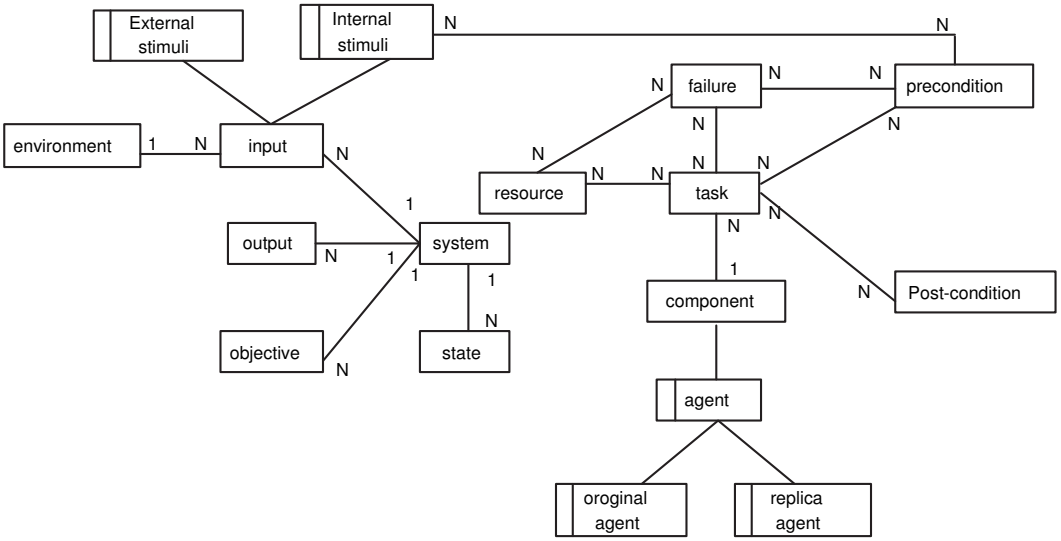

FIG. 2.2 – A diagram showing the concepts and the links between them, including the cardinalities of the links : many-to-many, many-to-one or one-to-many, and subclassing.

and propose a reorganization technique to overcome failures originating from the system boundary.

As presented in Figure 2.1, and stated earlier, a system is composed of interacting components that operate together in order to achieve some objectives or purposes. A system is intended to get inputs, process them in some way and produce outputs. The inputs are requests for services sent by the environment to the system. They are considered as *external stimuli* to the system. The outputs are defined as services that the system provides to its environment. The system may require access to resources in order to deliver the services it is requested for.

In Definition 1, we define the set of the external stimuli received by the system.

Definition 1. *We define* E_s *as the set of the external stimuli to the system.* $E_s = \{st_1, ..., st_n\}$, $|E_s|=n$.

The external stimuli are the elements of the environment that influence the system operation. The combination of the external stimuli characterizes the situations to which the system must react to. They are out of control of the system. They are situated outside the system boundary. Each external stimulus could take several values belonging to a definition domain. However, each external stimulus could condition the functioning of the future system since the change of value of an external stimulus may require that the system delivers a new service. Hence, these values could be used to determine the

services the system may deliver. Consequently, the system evaluates conditions, which truth values are evaluated based on the external stimuli, in order to determine which services it has to deliver. So, we will consider these preconditions in order to describe the external stimuli of the system. Hence, if there are n external stimuli to which the system must react to, then the system is confronted to 2^n different situations resulting from the combination of external stimuli generated by the environment.

Definition 2. *We define the set* $E = \{e_i = (st_{i_1}, .., st_{i_k}, .., st_{i_n}) \mid i \in [1..\mid E \mid], \forall j \in [1..n], st_{i_j} \in E_s \text{ and } |E_s|=n\}$. e_i *is a situation to which the system must react to.* $\mid E \mid = 2^n$.

When an external stimulus is perceived by the system, a set of outputs must be generated by the system as a response to this external stimulus. The system outputs have an influence on its environment : they may trigger environmental changes that may result in a situation change. The outputs of the system, which are the services the system has to deliver, can be thought of as the objectives that the system must achieve. Hence, the resulting environment situations, from the system's point of view, can be thought of as *goal situations*. We notice that not all environment situations are goal situations. However a goal situation is a particular situation of the environment. Consequently, the set of goal situations is a subset of the environment situations.

Definition 3. *We define* G *as the set of goal situations. They describe the environment after the delivery of the related services.* $G = \{g_i \mid g_i \in E, i \in IN\}$. g_i *is a goal situation.* $G \subseteq E$.

The outputs of the system are delivered by the different components operating in the system. In Definition 4, we define the set of these components.

Definition 4. *Let* $C = \{c_1,...,c_m\}$ *be the set of all* m *components operating in the system.* $|C|=m$.

In a system, some components are activated by stimuli originating from the system's environment while other components are activated by stimuli originating from the system's components. The stimuli originating from the system's components are called the *internal stimuli* [10]. The internal stimuli are components' outputs which stay within the system, i.e, messages from other components. They are not meant for the environment. In fact, they do not reach the outside environment. Hence, the system controls the internal stimuli. Consequently, the internal stimuli are defined in the

system boundary.

Definition 5. *We define I as the set of the internal stimuli of the system.* $I = \{is_1, ..., is_h\}$, $|I|=h$.

However, each internal stimulus could condition the functioning of an agent since the change of value of an internal stimulus may require from an agent to deliver a new output. Hence, these values could be used to determine the outputs an agent may deliver. Consequently, an agent evaluates conditions, which truth values are evaluated based on the internal stimuli, in order to determine which outputs it has to deliver. So, we will consider these preconditions in order to describe the internal stimuli of the system. Hence, if there are h internal stimuli in the system, then the system has at most 2^h different situations resulting from the combination of the internal stimuli generated by the agents. The internal stimuli characterize the observable state of the system. Hence, the system could have at most 2^h observable states.

Definition 6. *We define* $S = \{s_i = (is_{i_1}, .., is_{i_k}, .., is_{i_h}) \mid i \in [1..|S|], \forall\ j \in [1..h], is_{i_j} \in I$ *and* $|I|=h\}$. s_i *is an observable state of the system.* $|S| \leq 2^h$.

Since the environment is out of control of the system, the system could not prevent from failures originating from the environment. However, it should be able to prevent from failures originating from the system boundary ; That is these failures are caused by the system's internal stimuli. So, for the rest of this document, we only take into account the observable states of the system.

Each system's component can perform several tasks. Tasks are the activities required, or believed to be necessary for a component to achieve a goal in an interactive environment [3]. We define, in what follows, the set of all the tasks to be performed by the system's components.

Definition 7. *We define* τ *the set of all tasks to be performed by the system's components of . Let* $|\tau|=q$, *then* $\tau = \{t_i, i \in [1..q]\}$.

Of course, each component has tasks to perform. We define a function that determines the tasks which should be performed by a particular component.

Definition 8. *Let $i \in [1..|C|]$. We define the function task : $C \rightarrow \wp(\tau)$[2], $\forall\ c_i \in C$, $task(c_i) = \{t \mid t \in \tau\}$. $task(c_i)$ represents the set of tasks which will be performed by component c_i.*

Now, each component may have to perform a particular task when it receives a stimulus for that. Hence, the achievement of a task is related to some particular observable states of the system. We define a function that determines the tasks a component could perform in a particular observable state of the system.

Definition 9. *Let $i \in [1..|C|]$. Let $c_i \in C$ and $s \in S$. We define the function performableTask : $S \times C \rightarrow \wp(\tau)$, with $performableTask(s, c_i) \subseteq task(c_i)$. $performableTask(s, c_i)$ defines the set of tasks that c_i could perform in a particular observable state s.*

However, each task has a set of preconditions which must be met so that the task is properly performed. We define a function that determines the set of preconditions of a task t. These preconditions are based on the truth values of the external and internal stimuli. As stated in Definitions 2 and 6, these stimuli have boolean values.

Definition 10. *Let $t \in \tau$. Let $u = n+h$. Let $k \leq u$. We define the function $prec(t) = \{c = s_1 \wedge .. \wedge s_k \wedge .. \wedge s_u \mid s_k \in I_s$ or $s_k \in E_s\}$.*

Also, each task has post-conditions which are conditions that must be met after the execution of the task. We define a function that determines the post-conditions of a task t. These post-conditions may change the value of an internal or external stimuli. Consequently, they are expressed based on the internal and external stimulus.

Definition 11. *Let $t \in \tau$. Let $u = n+h$. Let $k \leq u$. We define the function $post(t) = \{c = s_1 \wedge .. \wedge s_k \wedge .. \wedge s_u \mid s_k \in I_s$ or $s_k \in E_s\}$.*

Fault-tolerance techniques are based on redundancy [47]. Redundancy is used in order to add fault-tolerance capabilities to the system. The redundancy can be a redundancy of hardware or software components. Redundancy is introduced in a system after the system was designed without taking into account fault-tolerance aspects. However, in a system, it is possible to have two identical components operating in the system and identified at system design. For example, a system in which two printers,

[2] $\wp$ is the partition set function

with the same characteristics, are used. In this case, it could be possible not to introduce redundancy to deal with fault-tolerance. In the case where software redundancy is used, it could be a redundancy of tasks. Hence, there could be identical tasks which must be performed in the system and which are identified at system design before adding fault-tolerance capabilities to the system. Two tasks are identical if under the same preconditions, the two tasks provide the same post-conditions.

Definition 12. *Let t and t' $\in \tau$. We define the function idTask : $\tau \times \tau \rightarrow$ {true, false}. idTask(t, t') = true $\Leftrightarrow$ (if prec(t) = prec(t') $\Rightarrow$ post(t) = post(t')).*

Furthermore, a task may be equivalent to another task. A task t is equivalent to a task t' if under the same preconditions, the post-conditions of t are a subset of the post-conditions of task t'. If task t is equivalent to task t', then it is not sure that t' is equivalent to t.

Definition 13. *Let t and t' $\in \tau$. We define the function equiTask : $\tau \times \tau \rightarrow$ {true, false}. equiTask(t, t') = true $\Leftrightarrow$ (if prec(t) = prec(t') $\Rightarrow$ post(t) $\subseteq$ post(t')). equiTask(t, t') $\neq$ equiTast(t', t).*

However, if task t is equivalent to task t' and task t' is equivalent to task t, then tasks t and t' are identical.

Definition 14. *Let t and t' $\in \tau$. idTask(t, t') = true $\Leftrightarrow$ equiTask(t, t') = true $\wedge$ equiTask(t', t) = true.*

As stated earlier, the system has control over its internal stimuli. Consequently, there must be tasks in the system to control them. Hence, the truth value of a precondition can be changed if the *MAS* has control over the stimuli from which the precondition depends. These stimuli are called the *effectors* of the condition. We define :

Definition 15. *Let $i \in [1..|I|]$. $\forall$ $is_i \in I$, effector : $\tau \times I \rightarrow$ {true, false} | $\exists$ $t \in \tau$, such that if effector(t, is_i) is true, then t controls the effector of is_i.*

Moreover, each task, which controls the effectors of an internal stimulus, has an effect on this stimulus by changing its value. We define, in what follows, the function *effect* :

Definition 16. *Let $i \in [1..|I|]$. $\forall$ $is_i \in I$, effect : $\tau \times I \rightarrow D \mid \exists\ t \in \tau$, such that effect(t, is_i) $\in D$ where D is the domain of is_i.*

Consequently, the system has control over its internal stimuli. Hence, the internal stimuli characterize the system boundary, and the external stimuli characterize the system environment. So, based on tasks' preconditions, we can determine which ones can be controlled by the system and which could not be. Doing so, we delimit the system's boundary.

Now, for the system to function properly, the different components of the system operate with each other in order to provide the requested services to the environment. Consequently, a component's task may request the execution of another component's task. Hence, the components must operate together [3].

Definition 17. *Let $t \in \tau$ and $t' \in \tau$. We define the function call : $\tau \times \tau \rightarrow$ {true, false}. call(t, t') = true means that task t will request the execution of task t'.*

This assumes that the various components need to communicate with each other.

Definition 18. *Let $i \in [1..|C|]$. Let $j \in [1..|C|]$. We define the function communication : $C \times C \rightarrow$ {true, false}. Let c_i and $c_j \in C$. communication(c_i, c_j) = true $\Rightarrow \exists\ t \in$ task(c_i), $\exists\ t' \in$ task(c_j) $\mid$ call(t, t') = true.*

Consequently, each component may interact with several other components.

Definition 19. *Let $i \in [1..|C|]$. Let $j \in [1..|C|]$. We define the function interaction : $C \rightarrow \wp(C)$. Let $c_i \in C$. interaction(c_i)=$\{c_j \in C \mid$ communication(c_i, c_j) = true, for $i \neq j\}$.*

For a system to operate correctly, its components may require to use resources. If a resource, required by the system, is not available, then the system may be in failure. Hence, the set of resources required by the system must be identified. We define, in what follows, the set of resources required by the system.

Definition 20. *We define $R = \{r_1, ..., r_l\}$ the set of resources. $\mid R \mid = l$.*

[3] This assumes full cooperation of the different components

More precisely, the resources may be required by tasks. Consequently, if a resource is not available, the tasks which require this resource may be in failure. Hence, when designing a fault-tolerant system, it is important to identify, for each task, its required resources.

Definition 21. *Let $t \in \tau$. We define the function resources : $\tau \rightarrow \wp(R)$. resources(t) determines the set of the resources required by t; resources(t) $\subseteq$ R.*

Nevertheless, each resource, for some reason, may be available or not in a particular situation. If a resource is not available, then the system may be in failure. Consequently, it is important to identify the resources which are not available and in which state of the system.

Definition 22. *Let $s \in S$ and $r \in R$. We define the function resourceAvailable : $S \times R \rightarrow$ {true, false}. resourceAvailable(s, r) = true if r is available in s, and false otherwise.*

Furthermore, each component will perform tasks. As stated earlier, each task requires resources to be properly performed. Hence, each component must have access to the required resources by its tasks.

Definition 23. *Let $c \in C$. We define resourceAccessibility : $C \rightarrow \wp(R)$. resourceAccessibility(c) = $\{r \in R \mid \exists\ t \in task(c), r \in resources(t)\}$.*

As stated earlier, redundancy could be a redundancy of hardware or software. Resources are a part of the hardware used by the system. Consequently, in order to provide fault-tolerance to the system, resource redundancy can be used. However, as for tasks, if there are identical or equivalent resources required by the system (before introducing redundancy to deal with fault-tolerance), then it could be unnecessary to replicate resources. A resource could be identical or equivalent to another resource for a particular task. Hence, we define a function that indicates whether two resources are equivalent or not for a particular task.

Definition 24. *Let $i \in [1..|R|]$. Let $j \in [1..|R|]$. Let $t \in \tau$, r_i and $r_j \in R$. We define the function resourceEquivalence : $\tau \times R \times R \rightarrow$ {true, false}. resourceEquivalence(t, r_i, r_j) = true if the two resources can be used by t independently.*

Consequently, we can define a function which determines, for each resource, its equivalent resources, for a particular task.

Definition 25. *Let $i \in [1..|R|]$. Let $j \in [1..|R|]$. Let $t \in \tau$ and $r_i \in R$. We define the function equivalentResources : $\tau \times R \rightarrow \wp(R)$. equivalentResources(t, r_i) = $\{r_j \in R \mid i \neq j$, resourceEquivalence(t, r_i, r_j) = true$\}$.*

Now that we have defined the different concepts composing a system, we provide a system characterization so that the links between these concepts are established :

Definition 26. *Let Sys be a system. Sys = $<\tau$, C, E, G, S, R$>$ in which τ is the set of tasks to be performed by the components, C is the set of components operating in the system, E is the set of situations to which the system must react, G is the set of goal situations, S is the set of observable internal states of the system and R is the set of required resources by the system.*

For each request of service, there is a plan which determines the tasks to be achieved in order to deliver the requested service. Hence, each component may have to achieve some of its tasks at particular observable states of the system.

Definition 27. *Let $s \in S$, $c \in C$, $g \in G$ and $t \in \tau$. We define the function perform : $S \times C \times G \rightarrow \wp(\tau)$. perform(s, c, g) = $\{t \in \tau \mid t$ is scheduled to be executed in s to deliver g$\}$. We have perform(s,c, g) $\subseteq$ performableTask(s, c).*

In each situation e of E that represents a request for services, there are several tasks that must be performed in order to provide these services. Hence, these tasks must be identified.

Definition 28. *Let $i \in [1..|S|]$. Let $j \in [1..|C|]$. Let $k \in [1..|G|]$. Let Sys be a system. Let $e \in E$. Let $\{g_1, ..., g_t\} \in \wp(G)$. We define the function service : Sys $\times$ E $\times$ $\wp(G)$ $\rightarrow \wp(\tau)$. service(Sys, e, $\{g_1, ..., g_t\}$) = $\{t \in \tau \mid \exists\ s_i \in S, \exists\ c_j \in C, \exists\ g_k \in \{g_1, ..., g_t\}$, t $\in$ perform(s_i, c_j, g_k)$\}$. The function service determines the plan that must be performed to deliver the requested services.*

At this stage, we have provided a formal definition of the different concepts characterizing a system. These definitions allowed us to delimit the system's boundary based

on the identification of tasks' preconditions. In the next subsection, we adapt these concepts to multi-agent systems. Then, we define a reorganization technique which will be integrated in FATMAS as it will be shown in Chapter 3.

2.3 Concepts From Multi-agent Systems

Multi-agent systems are systems in which agents operate. Hence, the components of these systems are agents. Agents are reactive and proactive components of some system ; components that are somewhat autonomous [57]. We define the set of agents operating in a multi-agent system.

Definition 29. *We define* Λ *the set of agents operating in a MAS. If there are m agents,* $\Lambda = \{a_i,\ i \in [1..m]\}$. $\mid \Lambda \mid = m$.

In a multi-agent system, the set of components is the set of agents operating in the system. Hence, in this case, C = Λ. Consequently, we can adapt the previous definitions to deal with multi-agent system :

Definition 30. *Let* $i \in [1..|\Lambda|]$. *Let* $task : \Lambda \rightarrow \wp(\tau)$, $\forall\ a_i \in \Lambda$, $task(a_i) = \{t \mid t \in \tau\}$, *so that* $task(a_i)$ *represents the tasks that agent* a_i *could perform.*

Definition 31. *Let* $i \in [1..|\Lambda|]$. *Let* $s \in S$. *Let* $a_i \in \Lambda$. *We define the function performableTask :* $S \times \Lambda \rightarrow \wp(\tau)$, *with* $performableTask(s,\ a_i) \subseteq task(a_i)$. $performableTask(s,\ a_i)$ *defines the set of tasks that* a_i *could perform in a particular system state.*

Definition 32. *Let* $i \in [1..|\Lambda|]$. *Let* $j \in [1..|\Lambda|]$. *We define the function call :* $\tau \times \tau \rightarrow$ *{true, false}. Let* $t \in \tau$, $t' \in \tau$. $call(t,\ t') = true$ *means that when task t requests the execution of the task t'.*

Definition 33. *Let* $i \in [1..|\Lambda|]$. *Let* $j \in [1..|\Lambda|]$. *Let* a_i *and* $a_j \in \Lambda$. *We define the function communication :* $\Lambda \times \Lambda \rightarrow$ *{true, false}.* $communication(a_i,\ a_j) = true \Rightarrow \exists\ t \in task(a_i), \exists\ t' \in task(a_j) \mid call(t,\ t') = true$.

Definition 34. *Let* $i \in [1..|\Lambda|]$. *Let* $j \in [1..|\Lambda|]$. $i \neq j$. *Let* $a_i \in \Lambda$. *We define the function interaction :* $\Lambda \rightarrow \wp(\Lambda)$. $interaction(a_i) = \{a_j \in \Lambda \mid communication(a_i,\ a_j) = true\}$.

Definition 35. *Let $a \in \Lambda$. We define resourceAccessibility : $\Lambda \rightarrow \wp(R)$. resourceAccessibility(a) = $\{r \in R \mid \exists\ t \in task(a),\ r \in resources(t)\}$.*

Definition 36. *Let Mas be a multi-agent system. Mas = $< \tau, \Lambda, E, G, S, R>$ in which τ is the set of tasks to be performed by agents, Λ is the set of agents, E is a set of situations to which the system must react, G is the set of goal situations, S is the set of observable states of the system, and R is a set of resources.*

Definition 37. *Let $s \in S$, $a \in \Lambda$, $g \in G$, and $t \in \tau$. We define the function perform : $S \times \Lambda \times G \rightarrow \wp(\tau)$. perform(s, a, g) = $\{t \in \tau \mid$ t is scheduled to be executed in s to deliver g$\}$. We have perform(s,a, g) $\subseteq$ performableTask(s, a).*

Definition 38. *Let $i \in [1..|S|]$. Let $j \in [1..|\Lambda|]$. Let $k \in [1..|G|]$. Let Mas be a multi-agent system. Let $e \in E$. Let $\{g_1, ..., g_t\} \in \wp(G)$. We define the function service : Mas $\times$ E $\times$ $\wp(G) \rightarrow \wp(\tau)$. service(Mas, e, $\{g_1, ..., g_t\}$) = $\{t \in \tau \mid \exists\ s_i \in S, \exists\ a_j \in \Lambda, \exists\ g_k \in \{g_1, ..., g_t\}, t \in perform(s_i, a_j, g_k)\}$. In the situation e, the system has to produce the set of services $\{g_1, ..., g_t\}$.*

In this section, we adapted some definitions introduced in Section 1 to multi-agent systems. In the next section, we provide more definitions to deal with fault-tolerant multi-agent systems.

2.4 Concepts Related to Fault-tolerant Systems

At this stage, we have determined the MAS boundary. In this section, we present the reorganization technique which will be a part of FATMAS and includes fault-prevention, faut-recovery, and fault-tolerance techniques. To this end, we continue providing definitions in order to build and formally valid our reorganization technique.

The second domain to which *FATMAS* is related is fault-tolerant systems. A fault-tolerant system is a system which recovers from its failures. A system is said to be in failure whenever it is not capable of performing correctly at least one of the tasks that it must perform in order to provide some desired service. A task cannot be performed if at least one of its preconditions is not met or if one of its required resources is not available. Hence, we do not deal with failures as programming languages exceptions but

as preconditions which are not met when a task is required to be performed.

Definition 39. *Let s in S and $t \in \tau$. We define the function taskFailure : $S \times \tau \rightarrow$ {true, false}. taskFailure(s, t) = true $\Leftrightarrow$ ($\exists$ c $\in$ prec(t) | c = false) $\vee$ ($\exists$ r $\in$ resources(t) | resourceAvailable(s, r) = false).*

An agent is in failure, if one of the tasks that it must perform in a particular observable system state s to deliver a service, cannot be performed.

Definition 40. *Let $s \in S$, $a \in \Lambda$, and $g \in G$. We define a function agentFailure : $S \times \Lambda \times G \rightarrow$ {true, false}. agentFailure(s, a, g) = true $\Leftrightarrow \exists$ t $\in$ perform(s, a, g) | taskFailure(s, t) = true.*

So, we define for each agent which tasks are in failure : the set of aborted tasks.

Definition 41. *Let $s \in S$, $a \in \Lambda$, and g in G. We define the function abortedTasks : $S \times \Lambda \times G \rightarrow \wp(\tau)$. abortedTasks(s, a, g) = {t $\in$ perform(s, a, g) | taskFailure(s, t) = true}.*

Nevertheless, an agent can be down. An agent is down, if all the tasks that it must perform in a particular observable state s in S in order to deliver a service, cannot be performed. Hence, if an agent is down, then it is in failure.

Definition 42. *Let $s \in S$, $a \in \Lambda$, and $g \in G$. We define a function agentDown : $S \times \Lambda \times G \rightarrow$ {true, false}. agentDown(s, a, g) = true $\Leftrightarrow \forall$ t $\in$ perform(s, a, g) | taskFailure(s, t) = true.*

Definition 43. *Let $s \in S$, $a \in \Lambda$, and $g \in G$. agentDown(s, a, g) $\Rightarrow$ agentFailure(s, a, g).*

A *MAS* is in failure, in a particular observable state s of S, if one of its agents is in failure in s. Hence, the system cannot reach a goal situation. If the MAS is in failure, and the system has no control on the effectors of the preconditions or cannot repair the resource in failure then the system is in a fatal failure. That is, the system has no control on the origin of the failure. So, the source of the failure is originated from the system's environment.

Definition 44. *Let MAS be a mutli-agent system. Let $s \in S$. Let $g \in G$. MAS is in a fatal failure if the origin of the failure comes from the MAS' environment. We define the function : FatalMasFailure : $MAS \times S \times G \rightarrow$ {true, false}. FatalMasFailure(Mas, s, g) = true iff $\exists$ $a \in \Lambda$, and $t \in \tau \mid t \in$ perform(s, a, g), taskFailure(s, t) = true $\wedge$ $\exists$ $c \in$ prec(t) $\mid$ c=false, $c = s_1 \wedge .. \wedge s_k \wedge .. \wedge s_u$ and $\exists$ s_k=false, k leq u, $\nexists$ t' $\in \tau$ effector(t', s_k) = true .*

If the MAS is in failure, and the system has control on the effector of the preconditions or can repair the resources in failure then, since the system has control over the sources of failure, it can overcome them. This kind of failure is not a fatal failure. The source of the failure is originating from the MAS boundary.

Definition 45. *Let Mas be a multi-agent system. Let $s \in S$, and $g \in G$. We define a predicate MasFailure : $Mas \times S \times G \rightarrow$ {true, false}. MasFailure(Mas, s, g)=true iff $\exists$ $a \in \Lambda$, $\exists$ $t \in \tau \mid t \in$ perform(s, a, g), taskFailure(s, t) = true $\wedge$ $\forall$ $c \in$ prec(t) $\mid$ $c =$ false, $c = s_1 \wedge .. \wedge s_k \wedge .. \wedge s_u$, k leq u, and $\exists$ t' $\in \tau$, effector(t', s_i) = true for all s_k=false.*

At this stage, we defined a multi-agent system failure. However, we aim at designing a fault-tolerant multi-agent system. In what follows, we define the techniques that must be applied to transform a multi-agent system into a fault-tolerant system. As presented in Chapter 1, at least the following techniques should be defined in order to build fault-tolerant systems according to [47] :

1. *Fault-prevention* technique : to prevent fault introduction and occurrence ;
2. *Fault-recovery* technique : to detect the existence of faults and eliminate them ;
3. *Fault-tolerance* technique : to provide services complying with the system's objectives in case of faults.

Consequently, these technique should be included in the *FATMAS* methodology. The first technique is for *fault-prevention*. It is achieved by using rigorous design techniques in order to eliminate the conditions that may trigger faults during execution [47]. The second technique is for *fault-recovery*. It leads the system from an erroneous state to an error-free state. It is achieved by [47] :

1. Detecting an error : identifying an erroneous state ;
2. Recovering from an error : substituting the erroneous state with an error-free state.

There are two ways to recover from errors [47] : backward recovery, and forward recovery. The backward recovery restores the system to a previously saved state which is supposed to be error-free. The system's states are saved at predetermined recovery

points. This approach requires significant resources (time and computation). Nevertheless, the backward approach is the most generally applicable recovery technique for fault-tolerance [47]. The forward recovery approach finds a new state from which the system can continue to operate. This state may be a degraded mode of the previous error-free state.

The last technique is *fault-tolerance*. It is achieved by using redundancy [16][47]. Redundancy provides the additional capabilities and resources needed to detect and tolerate faults. There could be a redundancy of either hardware or software. Hardware redundancy includes replicated and supplementary hardware added to the system to ensure fault tolerance. The software can reside on the redundant hardware to tolerate both hardware and software faults. Software redundancy includes the additional programs or modules used in the system to achieve fault tolerance.

In what follows, we present how to achieve these elements in order to include them in our design methodology.

2.5 The Reorganization Technique

The reorganization technique covers three techniques : fault-prevention technique, fault-recovery technique, and fault-tolerance technique. This reorganization technique will be a part of FATMAS. In what follows, we present these different techniques.

2.5.1 Fault-prevention Technique

The first technique to include in the reorganization technique is *fault-prevention* that identifies different sources of failure. We define a source of failure of a *MAS* as the cause which prevents the *MAS* from providing the environment's requested services. Hence, a system fails when it cannot reach any goal situation in G from a particular situation s in S. In this case, there are two possibilities :

1. The goals were all unrealistically set, and they must be revised. Hence, we define a new set of goals G' for the *MAS* ;
2. or something came up during the system operation and prevented it from reaching the goal situations in G, which, under other circumstances, would have been reached. A system failure comes from observing and responding to either

some external or internal stimuli. If the failure comes from an external stimulus, the *MAS* cannot recover from that failure since it has no control over such a stimuli.

As stated earlier, the sources of failure of a *MAS* that we will consider are the tasks' themselves. Hence, the first technique, *fault-prevention*, will be achieved by identifying all tasks' preconditions (to determine if they can be met) and all the tasks' required resources (to determine if they are available). Once the preconditions are predetermined and the resources are identified, new tasks could be added to the system in order to control some of the stimuli and consequently delimit the system's boundary. Consequently, we delimit the stimuli that the system will control and the stimuli that the system will not control. Hence, some sources of failures will be prevented by adding tasks to control them, and others will not be because it is not possible to control them.

The fault-prevention technique can be achieved in a linear time since if there are n tasks in the system, the number of preconditions of each task and the number of resources which will be used by the different tasks is limited and is independent from the number of tasks.

2.5.2 The Fault-tolerance Technique

The second element to consider in a design methodology of fault-tolerant systems is *fault-tolerance* itself, which is based on redundancy. Particularly, fault-tolerant multi-agent systems [18] [21] [22] are based on agent redundancy. Consequently, designers may overload the *MAS* with a large number of agents which increases the *MAS* complexity. In what follows, we propose an approach that minimizes the number of agents to be replicated and thus the overall system complexity. The idea is to replicate tasks first whenever possible, and if not, then to replicate agents.

In a MAS, there are agents determined at system design which are called *original agent*, and there are agents added to the system which replicate original agents in order to acquire the system with fault-tolerance capabilities. These latter agents are called *replica agents.* Consequently, a fault-tolerant multi-agent system will be composed of the original agents and of the replica of some agents. We make the distinction between these two kinds of agents.

Definition 46. *Original agents are the agents which compose the* MAS *before introdu-*

cing agent replication for the purpose of fault-recovery. Let O_a be the set of the original agents of a MAS. $| O_a | = p$.

Definition 47. *A replica agent is an agent which is created to prevent the failure of an* original agent*. Let R_a be the set of the replica agents.* $| R_a | = v$.

The set of agents Λ is defined as the union of O_a and R_a :

Definition 48. $\Lambda = O_a \cup R_a$. *Hence,* $n = p + v$.

Our approach is based on the following principle. If an agent is in failure, then there should be another agent in the *MAS* that could act in its place so that the *MAS* continues operating correctly. Of course, in order to perform a task, an agent needs to access to the task's required resources. Consequently, if a task t is in failure, only the agents that have access to the appropriate resources in *resource(t)* would be candidates to perform t.

Definition 49. *Let $i \in [1..|\Lambda|]$. Let $t \in \tau$ and $a_i \in \Lambda$. We define the function ability :* $\Lambda \times \tau \rightarrow$ *{true, false}, such that ability(a_i, t)=true if resource(t)* $\subseteq$ *resourceAccessibility(a_i), and false otherwise. t can be performed by a_i if resource(t)* $\subseteq$ *resourceAccessibility(a_i).*

The fault-tolerance approach that we propose aims at minimizing the number of agents to replicate. Hence, only a certain number of agents will be replicated. To determine which agents will be replicated, we introduce the notions of *critical* and *non-critical agents*. A *critical agent* is an agent which performs at least one task that cannot be performed by any other agent in the system.

Definition 50. *Let $i \in [1..|\Lambda|]$. Let $j \in [1..|\Lambda|]$. Let C_a be the set of the critical agents.* $C_a = \{a_i \in \Lambda \mid \exists\ t \in task(a_i), \nexists\ a_j \in \Lambda$ *with* $i \neq j$ *such that ability(a_j, t)=true* $\vee\ \nexists$ *t' $\in$ task(a_j) such that equiTask(t, t') = true or idTask(t, t') = true}.*

A *non-critical agent* is an agent for which each task can be performed by at least another agent.

Definition 51. *Let $i \in [1..|\Lambda|]$. Let $j \in [1..|\Lambda|]$. Let NC_a be the set of the non-critical agents.* $NC_a = \{a_i \in \Lambda \mid \forall\ t \in task(a_i), \exists\ a_j \in \Lambda$ *with* $i \neq j$ *such that ability(a_j, t)=true*

$\vee$ $\exists$ *t'* $\in$ *task*(a_j) *such that equiTask(t, t') = true or idTask(t, t')=true}. Hence,* $NC_a \cap C_a = \varnothing$.

Furthermore, if an agent is in failure, then this failure can either be originating from the system's boundary or from the system's environment. In the first case, the system can prevent this failure from happening. In the second case, it cannot. Consequently, if each agent is replicated at least once, and if the failure is originating from the system's environment, then all the replica agents will be confronted to the same environment situation, and so to the same failure. Hence, the system will not be able to recover from its failure. Thus, it is not necessary to have several replicas per agent. Each agent must be replicated once. So, we propose that each task of a non-critical agent be replicated in another agent. Hence, implicitly, the agent is replicated within several other agents. We also propose to replicate each critical agent only once. Doing so, we can minimize the overall system complexity.

At this stage, we claim that each agent in a fault-tolerant *MAS* can be either an *original agent* or a *replica agent* (see Figure 2.3). Moreover, when the *MAS* is designed, each *original agent* can be either a *critical agent* that must be replicated or a *non-critical agent* that should not be replicated.

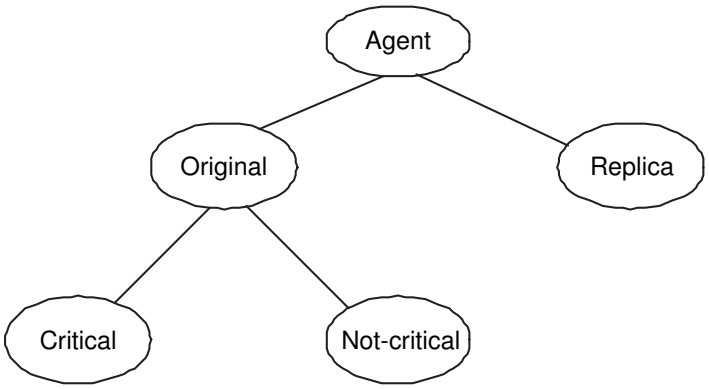

FIG. 2.3 – The different agent sets.

Furthermore, each task is replicated in only one agent. If there are two possible agents in which a given task can be replicated, then the MAS' designer may use several criteria in order to choose the agent in which to replicate the tasks. These criteria can depend, for example, on the cost to replicate a task in another agent or to replicate a task into the agent which has a minimal number of tasks to perform.

Now, when the system will operate, agents must identify the agents in which their tasks are replicated in order to coordinate their activities. To this end, we couple the agents on a task basis so that we avoid the overhead associated with task reallocation

after a task failure.

Definition 52. *Let $i \in [1..|\Lambda|]$. Let $j \in [1..|\Lambda|]$. Let $t \in \tau$, $a_i \in \Lambda$, $a_j \in \Lambda$ with $i \neq j$ and $t \in task(a_i)$. We define the function coupling : $\tau \rightarrow \Lambda \times \Lambda$ | coupling(t)=(a_i, a_j), if $\exists$ t' $\in task(a_j)$ such that equiTask(t, t')=true or idTask(t, t')=true.*

Moreover, each agent can have its tasks replicated in different agents such that each task is replicated in only one agent. Hence, we define a function *duplicate* to determine, for each agent, the agents in which its tasks are replicated :

Definition 53. *Let $i \in [1..|\Lambda|]$. Let $j \in [1..|\Lambda|]$. We define the function duplicate : $\Lambda \rightarrow \wp(\Lambda)$ | $\forall$ $a_i \in \Lambda$, duplicate(a_i)=$\{a_j \in \Lambda$ with $i \neq j$ | $\exists$ $t \in task(a_i)$, coupling(t)=(a_i, a_j)$\}$. duplicate(a_i) is the set of agents in which a_i's tasks are replicated.*

To conclude, the *fault-tolerance* technique allows the system to replicate either agents or tasks. Each *original agent* can be either a *critical agent* or a *non-critical agent*. If it is a *non-critical agent*, then it is not replicated and its tasks are replicated in other *agents*. If it is a *critical agent*, then it must be replicated. Hence, the *fault-tolerance* technique minimizes the system complexity by minimizing the overall number of agents in the system.

2.5.3 The Fault-recovery Technique

The third element to take into account when defining a fault-tolerant system is *fault-recovery*. This technique should allow the *MAS* to continue operating correctly despite agent failures. The fault-recovery technique should allow the system to first detect failures and then recover from them. In this section, we provide techniques to detect and recover from failures. The validity of these techniques will be demonstrated in Section 4.

There are five possible kinds of failures for an agent :

1. The communication with the agent is down but :
   (a) the agent can perform all of its tasks ;
   (b) the agent can only perform some of its tasks ;
   (c) the agent cannot perform any of its tasks.

2. The communication with the agent is up but :
   (a) the agent can only perform some of its tasks ;
   (b) the agent cannot perform any of its tasks.

For each of these kinds of failure, we present a technique that allows the *MAS* to detect the failure and recover from it. We notice that the two kinds of failures can be dealt with in the same way since, in both cases, the agent is not able to perform its tasks to deliver the requested services. To illustrate our techniques, we consider an agent a_i in failure such that $|\ duplicate(a_i)\ | \geq 1$ and $i \in [1..|\Lambda|]$.

Fault Detection Techniques

As stated earlier, any fault recovery technique is based on redundancy (agent or task redundancy). So, if an agent is in failure, there should be other agents in the *MAS* to replace it. Hence, each agent should be able to detect other agents' failure and help to recover from it. Let a_j be an agent such that a_i has some of its tasks replicated in a_j, i.e, $a_j \in duplicate(a_i)$. a_i can be down or some (and not all) of its tasks could be in failure. In order for the agents in *duplicate(a_i)* to take over the failed tasks, they have to know whether a_i is down or in failure. To that effect we propose to use a handshake protocol in order to ensure that coupled agents know whether their counterparts are down or not.

Definition 54. *Let $i \in [1..|\Lambda|]$. Let $j \in [1..|\Lambda|]$. Let $a_i \in \Lambda$. Let $a_j \in \Lambda$. Let $s \in S$. We define the function handShake : $S \times \Lambda \times \Lambda \rightarrow$ {true, false}. handShake(s, a_i, a_j) = true if a_j receives a handshake from a_i in state s. If handShake(s, a_i, a_j) = false, then a_i knows that a_j is down.*

If a_i is down, then each of the agents in *duplicate(a_i)* will not receive a handshake from a_i, and will assume that they have to take over.

Nevertheless, all these agents must agree that a_i is down in order to take over. We propose to use a counter on the number of agents agreeing that a_i is down. All the agents which are in *duplicate(a_i)* can modify the value of this counter. Each agent which detects that a_i is down, increments the value of this counter. If the value of the counter is equal to $|$ *duplicate(a_i)* $|$, then the agents agree that a_i is down. Otherwise, the agents that have incremented the value of the counter deduce that a_i is not able to communicate with them, but a_i is not down.

Definition 55. *Let $i \in [1..|\Lambda|]$. Let $a_i \in \Lambda$. Let $s \in S$. We define the function downDetected : $S \times \Lambda \rightarrow$ {true, false}. downDetected(s, a_i) = true if $\forall\ a_j \in$ duplicate(a_i), handshake(s, a_j, a_i) = false.*

In the same way, if a_i is in failure and cannot perform a task t, then it should notify a counterpart, an other agent a_j acting as a backup agent for task t, i.e, a_j in *coupling(t)=(a_i, a_j)*, since each task is replicated in one agent as stated earlier.

Definition 56. *Let $i \in [1..|\Lambda|]$. Let $j \in [1..|\Lambda|]$. Let $a_i \in \Lambda$. Let $a_j \in \Lambda$. Let $t \in \tau$. Let $s \in S$. We define the function informFailure : $S \times \Lambda \times \tau \times \Lambda \rightarrow$ {true, false}. informFailure(s, a_i, t, a_j) = true means that in system's state s, agent a_i informs agent a_j that it has a task t in failure ; with $a_j \in$ duplicate(a_i) and coupling(t) = (a_i, a_j).*

The following algorithm summarizes the *fault-detection* step.

Let $g_i \in$ G. –g_i is one of the services the system has to provide.
– we verify that a_i has some of its tasks replicated in a_j.
if *agentDown(s, a_i, g_i) = true* **then**
  for $\forall\ a_j \in$ *duplicate(a_i)* **do**
    if *handShake(s, a_i, a_j) = false* **then**
      counter$_{a_i}$:= counter$_{a_i}$ + 1 ; – we increment the counter associated to a$_i$.
    end if
  end for
end if
– we verify whether the agents agree over a_i's failure.
if counter = | *duplicate(a_i)* | **then**
  downDetected(s, a_i) = true ;
end if
– if the agent is not down but in failure.
if (*agentDown(s, a_i, g_i) = false*) and (*agentFailure(s, a_i, g_i)= true*) **then**
  – we inform the agents in which t is replicated that t is in failure.
  for $\forall\ t \in$ *abortedTasks(s, a_i)* **do**
    for $\forall\ a_j \in$ *duplicate(a_i)* **do**
      if *coupling(t) = (a_i, a_j)* **then**
        informFailure(s, a_i, t) := a_j ;
      end if
    end for
  end for
end if

Now that a failure is detected, the system should recover from it, whenever possible.

Fault Recovery Techniques

In what follows, we present how the system recovers from its failures. As stated earlier, an agent can be in failure or down. In both cases, coupled agents will perform the tasks in failure.

In both cases of a_i failures, there could be at least one task of a_i that could not be performed. Each task t in failure will be performed by agent a_j in *duplicate(a_i)* such that *coupling(t)*=(a_i, a_j). Nevertheless, a_j should have a copy of a_i's knowledge so that it can continue operating from the last non-faulty point, if possible. a_j can have a copy of a_i's knowledge by either receiving a_i's knowledge from it or by sharing a common memory area in which a_i's knowledge pertaining to task t is stored. In the first case, the network may be overloaded by messages allowing agents to exchange their knowledge. The more there are agents in the system, the more there are messages exchanged between agents. This could slow down the system tremendously. Consequently, it is expected that for many applications a partially shared memory between agents would be best in order to minimize the quantity of information exchanged by agents. The access to the shared memory areas must be synchronized.

One of the problems that can occur when using a shared memory is that the agent acting as a replica cannot access to this memory. Hence, in the case of an agent failure, the replica agent will not have the necessary information to back-up the failed agent. So, we propose that each agent maintains a copy of each agent's shared memory to which it acts as a replica. Doing so, if an agent is in failure, and if its replica has no access to the shared memory, then the replica could use its copy of the shared memory to back-up the failed agent.

For each task, each memory area can be accessed by the two agents that are coupled with regard to this task. This may change the communication links between agents, as calls between agents will need to be redirected. The following algorithm summarizes the *fault-recovery* technique.

for $\forall$ $t \in$ *abortedTasks(s, a_i)* **do**
  for $\forall$ $a_j \in$ *duplicate(a_i)* **do**
    if *coupling(t) = (a_i, a_j)* **then**
      t' $\in$ perform(s, a_j, g_i) ; –with equiTask(t, t') = true or idTask(t, t') = true.
    end if

– the interactions between agents change since t is performed by another agent.

for $\forall$ $a_k \in$ *interact*(a_i) **do**
for $\forall$ *t"* $\in$ task(a_k) **do**
if *call(t, t") = true* **then**
call(t, t") := false ;
call(t', t") := true ;
end if
end for
communicate(a_i, a_k) := *false* ;
communicate(a_j, a_k) := *true* ;
end for
end for
end for

Consequently, even if several agents will be able to modify the content of a_i's shared memory, only one agent is allowed to modify the content of this memory at a given time.

To summarize, the *fault-recovery* technique is based on :
– fault detection : faults are detected based on a presence notification mechanism that relies on :
  – a) a direct handshake protocol between coupled agents ;
  – b) a request for the counterpart agent whenever a task failure is detected by an agent.
– fault recovery : the system can recover from a one failure at a time based on the shared memory areas between agents.

2.5.4 The Validation of the Technique

As stated earlier, a fault-tolerant multi-agent system is not able to recover from its failures if these failures are originating from sources of failures situated in the system environment since the system has no control over these sources. Hence, the system is able to recover from its failures, if they are originating from the system's boundary. Consequently, as proved in theorem 1, if the system is in failure, and this failure is originating from the system's boundary, then it will recover from it. This theorem is true only in the case that there exists a task in the system that controls the effector of the internal stimulus which originated the failure. Otherwise, the system will not be

able to recover from its failure if there is no task which controls the internal stimulus that originated the failure. In the next section, we will propose an approach to deploy the system in such a way that there always exists a task which controls the internal stimulus that originated the failure.

Theorem 1 : A fault-tolerant multi-agent system to which we apply the fault-recovery techniques presented above will recover from an agent failure :

1. if the effectors of the preconditions of the tasks in failure are controlled by system's tasks ;
2. if their resources are available to the agent that takes over.

To prove the validity of our technique, we demonstrate that the fault-tolerant multi-agent system behaves in the same way when it operates under non-faulty system's states as under faulty system's states. We demonstrate that the plan the system performs, when there is a request for services from the environment, is the same whether the system goes into a failure state or not.

Let us consider $e \in$ E as a request of services to the system. The system has to reach a set of goal situations to provide these services. Let $\{g_1, ..., g_t\}$ be this set of services to be provided. The system has to achieve tasks to provide these services. These tasks belong to the system's plan P, i.e, $P = service(Mas, e, \{g_1, ..., g_t\})$. During the execution of the plan, the MAS will go through several intermediate states. Let s be an intermediate state in which the MAS could not perform one of the tasks scheduled to be achieved in s. Consequently, the agent which has to achieve t in s is in failure. Let a_i refer to this agent.

A task is in failure if either its preconditions are not met or its required resources are not available. The truth value of a precondition of t is evaluated based on the values of the external and internal stimuli of the system. Each internal stimulus is controlled by a task of the system. Hence, the values of these internal stimulus can be changed to make the precondition true.

Moreover, if one of the resources, r_j, of the task t is not available, then r_j cannot be used by a_i. If there exists another resource r_k, in the system, which is equivalent to r_j for task t, and if a_i has access to r_k, and r_k is available in s, then a_i will perform task t in failure using r_k. Otherwise, the task t must be achieved by an agent a_p such that $coupling(t) = (a_i, a_p)$. a_p will perform either an identical task or an equivalent task. In this thesis, we study the case in which a_p performs an identical task. If a_p has access to r_k, and r_k is available in s, then a_p will perform task t' that is identical to task t using

r_k. Consequently, the system will continue performing its plan as if task t would not be in failure. Hence, the system will not be in failure. This proof is expressed as follows :

Proof :

Let $e \in$ E a request of services to the system ;
Let g = $\{g_1, ..., g_t\} \in \wp$(G) the set of services to provide in e ;
Let P be a plan such that $P = service(Mas, e, \{g_1, ..., g_t\})$;
In the case of an agent is in failure or down :
Let $s \in$ S a failure situation | $\exists\ a_i \in \Lambda$ | $\exists\ g_j \in$ g | $agentFailure(s, a_i, g_j) = true$;
$\exists\ t \in P$ | $t \in perform(s, a_i, g_j)$, $taskFailure(s, t) =$ true ;
$taskFailure(s, t) =$ true $\Rightarrow \exists\ c \in prec(t)$ | $c = false \vee \exists\ r \in resources(t)$ | $resourceAvailable(s, r) = false$;
Nevertheless, $c = s_1 \wedge .. \wedge s_u$ | with $s_k \in I_s$ or $s_k \in E_s$;
If all $s_k \in E_s$ are false, then it will not be guaranteed that the system will recover from its faults since the truth value of s_k is not controlled by the system.
Let us suppose that only the s_k belonging to I_s are false ;
Since for each of the internal stimuli s_k, $\exists\ t' \in \tau$ | $effector(t', s_j) = true$, consequently we ensure that the precondition $c = true$;
Moreover, if a resource $r \in resources(t)$ is not available $\Rightarrow$ $resourceAvailable(s, r) =$ false ;
if $\exists\ r_k$ | $resourceEquivalence(t, r, r_k) = true \wedge$ resourceAvailable(s, r_k) $\wedge$ resourceAccessible(a_i)
then
agentFailure(a_i) = false ;
$MasFailure(Mas, s) = false$;
else
$\exists\ t_1 \in perform(s, a_p, g_j)$ | $idTask(t, t_1) =$ true and $a_p \in \Lambda$ | $coupling(t) = (a_i, a_p)$;
end if
if $r_k \in resourceAccessibility(a_p)$ and $resourceAvailable(s, r_k) = true$ **then**
$t_1 \in$ P ;
taskFailure(s, t_1) = false ;
end if
This demonstration is true for any task in failure in s. Consequently, each failure task can be either recovered by using replicated resources or by performing identical tasks ;
$MasFailure(Mas, s) = false$;

At this stage, we demonstrated that if the sources of failures are originating from the system's boundary, then the system will recover from its failure. However, each agent will be deployed on a physical component, and each physical component is prone to

failure too. Consequently, we need to determine how the system could be distributed over the physical components so that if a physical component is in failure, then the system will not be in failure. This is the scope of the next section.

2.6 Physical Components Failure

The physical components on which the fault-tolerant MAS will operate must function without any failure. Meanwhile, the physical components are prone to failure. Since FATMAS proposes to not replicate all agents, it will also propose to not replicate all the physical components. If all the physical components are replicated, then all the agents could be replicated. However, it is not always possible to replicate all the physical components because replication is expensive. Furthermore, FATMAS tries to minimize the number of agents to be replicated. So, we will propose a technique in order to determine the number of physical components on which the system will operate so that the failure of a component will not alter the MAS functioning. We notice that, since FATMAS is applied to design fault-tolerant multi-agent systems running on a limited number of physical component, the probability that two physical components are in failure at the same time, and the probability that a resource is in failure is very low. Otherwise, there must be more components on which agents will be running.

The idea is to define groups of agents and deploy each group on a physical component. Nevertheless, these groups must be defined such that if a physical component is in failure, which means that the agents running on this component are in failure, then there exists agents in the system to backup the failed ones. Hence, these groups have a characteristic that there cannot be two agents a and a' in the same group such that a' $\in$ duplicate(a). To this end, we propose to define a group of agents as follows :

Definition 57. *Let $i \in N$, the set of natural numbers. Let $j \in N$. Gp is a group a agent such that $Gp = \{a_i \in \Lambda \mid \forall\ a_j \in Gp,\ j \neq i,\ a_j \notin duplicate(a_i))\}$.*

Doing so, each agent will not be running on the same physical component with its replica. Hence, if a physical component is in failure, then the group of agents running on that component may be in failure. Since the agents of a group are independent from each other, each of the agents of a group in failure has its tasks replicated in other agents running on other physical components which are not in failure. So, by an adaptive behavior of the agents, the remaining agents on which the tasks are replicated can organize themselves so that the system continues operating. Hence, if there are n agents in the MAS, then they can be grouped in k groups such that $k \leq n$. So, there

should be k physical components. We notice that each physical component on which a group of agents is running must offer access to the required resources of all agents of the group.

The way agents are assigned to physical components could be compared to works done in agents resource allocation [5] [15] . However, the works done in resource allocation are about how a resource can be shared between several agents which requires access to that resource. In our case, while agents are assigned to physical components, we only determine how agents should be distributed over the physical components. We do not address the issue on how the processors of the physical components are shared between agents.

2.7 Summary

In this chapter we presented and defined the different concepts on which the *FATMAS* methodology is based. They deal with the different concepts that describe a multi-agent system, and they also cover three techniques required to build a fault-tolerant system, i.e, *fault-prevention*, *fault-recovery*, and *fault-tolerance*. We also presented how a multi-agent must be distributed over physical components so that if a physical component is in failure then the system will not be in failure. In the next chapter, we detail the process development of *FATMAS* based on a case study and present a general system design approach required to use *FATMAS*.

Chapitre 3

FATMAS : an Agent-Oriented Methodology for Fault-Tolerant Multi-Agent Systems

A Multi-Agent System (*MAS*) operates in and interacts with an environment (as stated in Chapter 2). This environment is characterized by the services it requests from the *MAS*. Moreover, the *MAS* may go through several observable states which may cause system failure. However, the *MAS* should be able to continue properly operating in case of failure. Designing a fault-tolerant multi-agent system should require the identification of possible causes of failure in order to overcome them. However, existing *MAS* design methodologies, such as those described in [2] [7] [14] [17] [20] [24] [27] [31] [43] [44] [57], are not well suited for the design of fault-tolerant multi-agent systems since they do neither allow to identify probable sources of faults nor integrate a fault recovery technique. In this chapter, we propose an agent methodology, FATMAS, to design fault-tolerant multi-agent systems. This methodology is based on the development of several models to support a designer in order to define the software architecture of a fault-tolerant multi-agent system. Each model is associated with a micro-process providing guidelines on how to build the model. The methodology also has a macro-process that provides guidelines on how to proceed from one model to the next. This methodology includes the reorganization technique presented in Chapter 2.

This methodology, as presented in this chapter, is original in the way that it designs fault-tolerant multi-agent systems, which is not the case of existing methodologies. It guides the designer through an iterative process to design its fault-tolerant system while proposing criteria based on a cost/benefit approach to stop iterating over the models. Moreover, the methodology integrates the reorganization technique defined in

Chapter 2. Consequently, it helps the designer to determine the system's boundary, to use a replication algorithm which reduces the system complexity, and to determine the required physical components on which the future system will operate. Furthermore, based on definitions of Chapter 2, the methodology provides a model validation when moving from one model to another.

In this chapter, we first present an overview of *FATMAS*. Then, we present the different models of *FATMAS* and their micro-processes. Finally, we present the development life-cycle of *FATMAS* that represents its macro-process.

3.1 Overview of FATMAS Models

In this section, we first identify the different models of *FATMAS*. Then, we present the relations between these models.

3.1.1 The Identification of FATMAS Models

As presented in Figure 3.1, a *MAS* is a system composed of several agents interacting with each other. Each agent performs several tasks. Each task is performed under conditions and could require resources to be correctly performed.

Considering Figure 3.1, we notice that the definition of a *FTMAS* requires at least the following steps :

1. The identification of the inputs and outputs of the environment.
2. The identification of the tasks which will be performed by agents ;
3. The identification of the agents which will perform the tasks ;
4. The identification of the interactions between agents ;

Each task is performed under certain conditions and may require a set of *resources*. Moreover, it may require to interact with the environment. According to *Definition 2* of Chapter 2, the environment is characterized by its stimuli to which the *MAS* must react. Consequently, the environment will be defined during the task identification phase of the *MAS* design. We propose a task-environment model which identifies the different tasks that the agents will perform. Each task is described by its *preconditions* and *resources*. The *task-environment* model allows us to determine the preconditions under which the tasks operate, and their required resources. Hence, it allows to determine the

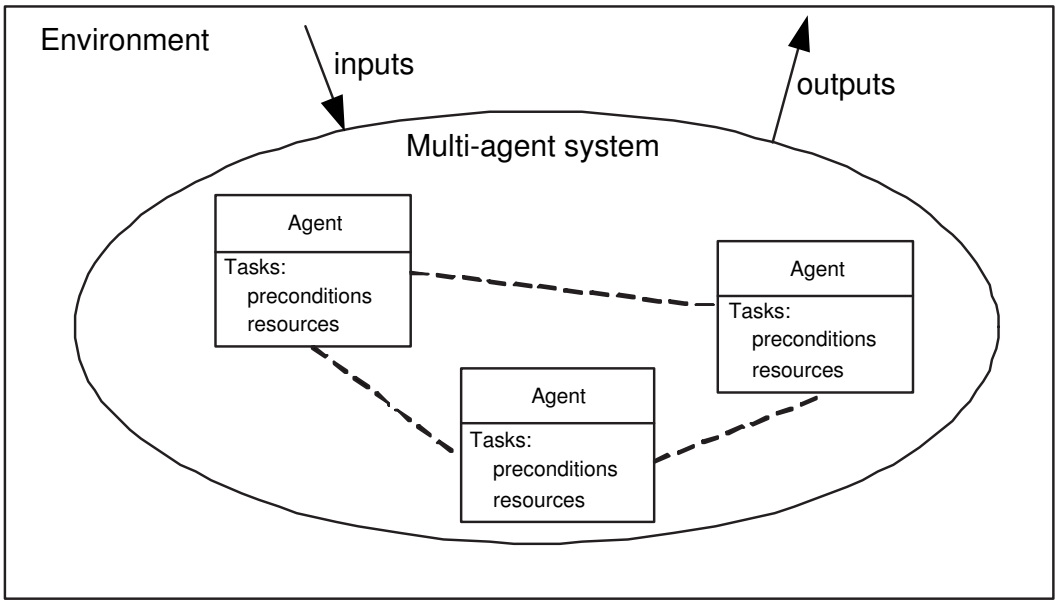

FIG. 3.1 – A multi-agent system interacting with its environment.

system's boundary, as stated in Chapter 2. Consequently, the *task-environment* model integrates the *fault-prevention* technique in *FATMAS*.

The different tasks identified in the *task-environment model* will be performed by several agents. Each agent must have access to the resources required by its tasks. An agent model is proposed to specify task assignment according to the *task-environment* model. Each agent may have relations with other agents, such as communication or control relations as defined in [37]. These relations must be explicitly represented in the *agent model* as part of the specification of the system. Moreover, relations defined between agents in *AUML* [24][41][57] can be considered per se in the *agent model*.

Furthermore, agents, in a *MAS*, interact with each other. An interaction, as defined in [41], is the ongoing two-way or multi-way exchange of data among computational entities. The exchange of information between agents is materialized by a sequence of messages exchanged between them [41]. The *agent model* does not represent this sequence of messages. To this end, we propose an agent interaction model to model agent interactions. The agent interactions define the communication protocols between agents. Hence, since the sequence or collaboration diagrams of *AUML* [41] represent these protocols, these diagrams can be directly used to represent the *agent interaction model*.

Each agent is prone to failure. If there is an agent in failure, some other agent will take over as we saw in Chapter 2. Consequently, the sequence of messages exchanged

between agents would change. Hence, a MAS reorganization must take place since the agents and their interactions have been modified. Therefore, in order to build a fault-tolerant system, the *MAS* methodology should provide a reorganization technique which represents the possible changes that may affect the agents and their interactions in case of failures. Hence, this *reorganization technique* includes the *fault-tolerance* and *fault-recovery* techniques. It was presented in Chapter 2 ; it is based on the replication of tasks and agents. As a result, the interaction model may need to change.

Finally, the *FTMAS*, as any software system, will be implemented. The infrastructure and the physical components on which the *MAS* will run must be determined. So, we propose an implementation model that defines the physical components on which the *FTMAS* will operate.

To summarize, the *FATMAS* methodology contains the following deliverables :

1. A *task-environment model* that specifies the different tasks that will be performed by the agents. For each task, we identify its preconditions in order to insure that the task will be properly performed. These preconditions are used to identify different sources of failure. They characterize either the environment in which the fault-tolerant multi-agent system operates or its observable states. It determines the system's boundary. This model includes the *fault-prevention* technique ;
2. An *agent model* which shows the agents' tasks and relationships. In the agent model, we focus on assigning and determining agent tasks and relations ;
3. An *agent interaction model* which describes the interactions between agents ; it is inferred from the *task-environment model*, at least partially ;
4. A *reorganization technique* which specifies the new system reorganization in case of a failure. The reorganization technique specifies the different techniques used to recover from a fault : *fault recovering* and *fault tolerance*. As a result, the interaction model may change ;
5. An *implementation model* which determines the physical components upon which the fault-tolerant multi-agent system will operate.

3.1.2 The Relations Between FATMAS Models

As presented in the previous section, *FATMAS* proposes to build several models in order to provide a fault-tolerant multi-agent system. These models should be related

to one another so that a designer can build each model from the previous ones. Hence, *FATMAS* should support the main activities of software engineering which are [33] :

1. *Requirements' analysis and specification* : The requirements' analysis and specification are the permanent recording of a set of requirements. They produce the basis on which to implement a computer-based software system to 'deliver' the required features and functions ;
2. *Design of computer-based software solution* : The design of a computer-based software solution is the process of conceptualizing and implementing an artefact that supplies the requirements. The result of software design should be a blueprint of what can be made. Software design provides the highest level specification of the software to be developed, to meet the specification ;
3. *Implementation of the design as programs* : The implementation of the designed programs allows to code the program and to determine the support environment.

Each model of *FATMAS* can be associated to one of these phases (see Figure 3.2). The *task-environment* model allows to define the services required from the system. The services are expressed as the outputs of the system on its environment. These outputs are provided by the tasks of the system. Hence, this model is developed during the requirements' analysis and specification phase.

The *agent model*, the *agent interaction model* and the *reorganization technique* provide the highest level specification of the software to be developed. The *agent model* provides the static structure of the system by identifying the agents and their relationships. The *agent interaction model* provides the dynamic interactions that will take place in the future system to be developed. The *reorganization technique* provides a mechanism specifying how the system will behave in case of failure. Hence, these models are developed during the system's design phase.

The *implementation model* allows code generation of the agents and the determination of the hardware infrastructure upon which the *MAS* will operate. This model is developed during the implementation phase.

Now that we have determined the different phases of *FATMAS*, we present in the next sections these phases in detail using a case study as an illustration.

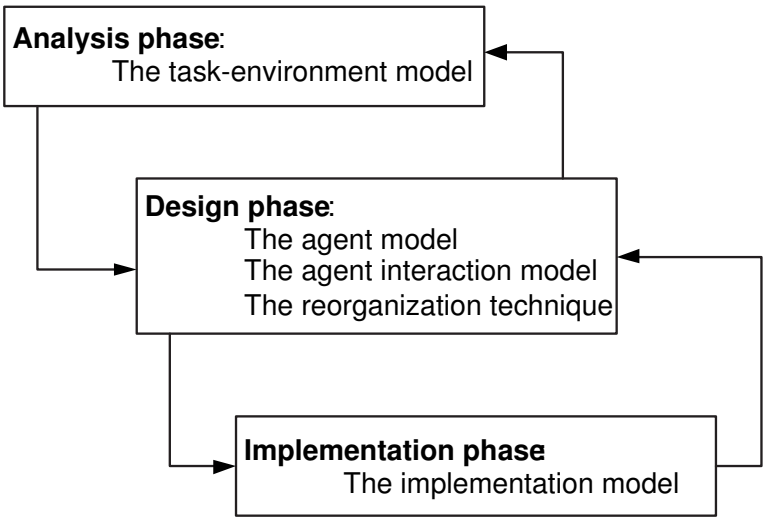

FIG. 3.2 – The different phases of FATMAS.

3.2 Case Study

In order to illustrate the *FATMAS* methodology, we use the example of a flexible manufacturing cell (see Figure 3.3). The manufacturing cell produces small mechanical pieces like nails. It is composed of a :

- machining unit ;
- lathe which is a machine tool for producing circular and cylindrical forms, in which the workpiece rotates in contact with a stationary tool or cutter ;
- robot which stores material in the lathe ;
- carrousel : a conveyor on which objects are placed and carried around a complete circuit on a horizontal plane ;
- mobile carriage : there is a storage unit for each machine-tool (the machining unit or the lathe) in which the carriage puts the pieces or delivers them to another machine-tool.

The manufacturing cell operates as follows. The information of each piece to be produced is stored in a *manufacturing order*. It represents the different instructions needed to produce this piece. The manufacturing orders are stored in a database. To follow the production process, an electronic identification tag is associated with each piece. The instructions which are in the manufacturing order are transcribed on the tag associated with the piece to be produced. Each time, the piece under production is in front of a machine-tool, the agent controlling the machine-tool reads the instructions on the tag and performs them. Before starting the production process, the carrousel

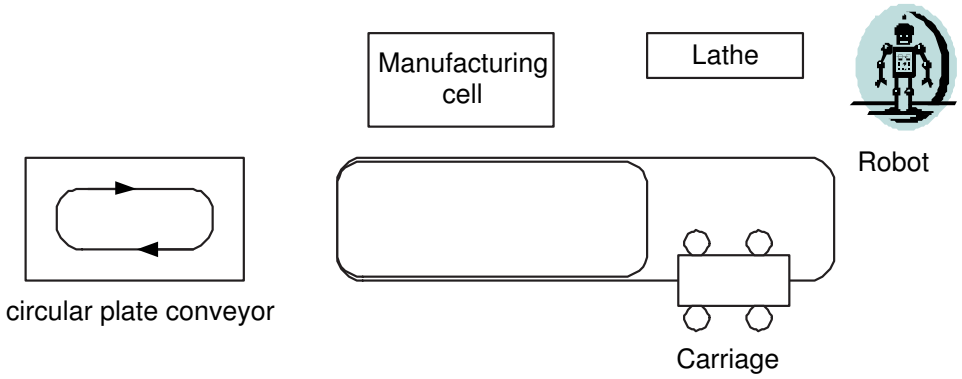

FIG. 3.3 – Manufacturing cell.

is started so that the pieces to be produced are carried around all the machine-tools. Each machine-tool has a local storage unit from which or in which the mobile carriage takes or puts the pieces under production.

On the basis of this example, we present the different models of the *FATMAS* methodology.

3.3 The Analysis Phase

The analysis phase provides the basis on which to implement a computer-based software system to 'deliver' the features and functions required. The analysis phase includes the production of the *task-environment* model which is presented in what follows.

3.3.1 The Task-environment Model

As stated in Section 1, the first model to be developed in the *FATMAS* methodology is the *task-environment model*. It helps the designer to determine the different tasks that the future agents will perform, the *required resources*, and the *preconditions* under which each task is to be performed. The *task-environment model* allows to support the *fault-prevention* technique, the first technique necessary to build a fault-tolerant system. *Fault-prevention* is ensured by identifying the tasks' preconditions and their resources. These preconditions or resources, as stated in Chapter 2, when they are not

met or available, are the possible sources of a *MAS* failure which can be identified when designing the system.

First, the tasks which will be performed in the* MAS *are determined. These tasks can be identified by analyzing the problem requirements. The model will go through a refinement process by performing a functional decomposition of the system's task into several sub-tasks and by applying this stepwise refinement process recursively to these sub-tasks. The decomposition leads to a task hierarchy that captures the fundamental functional aspects of the system.

For the manufacturing cell example, the system's overall task is to produce small mechanical pieces. Hence, the objective is to produce, at each production cycle, a mechanical piece. We can define a condition to state that the mechanical piece is correctly produced. This condition is referred to as *pieceProduced*. It has a boolean value and it belongs to set G defined in Definition 3. The tasks, which the system must perform in order to reach its global objective, are identified from the presentation of the case study introduced in Section 2 as follows (see Figure 3.4) :

1. The manufacturing cell is started at the beginning of the production process. We introduce the task *startMC* to start the manufacturing cell ;
2. When the manufacturing cell starts, the manufacturing order must be read from the database and the carrousel is started. Hence, *startMC* can be decomposed into two sub-tasks which are : *readInst* ; i.e, read the instructions from the manufacturing order, and *startC* which starts the carrousel ;
3. There are two machine-tools in the cell. Each machine-tool must be controlled. Hence, we introduce two tasks to manage the machine-tools : *manageLathe* which controls the lathe, and *manageMUnit* which controls the manufacturing unit ;
4. When a mechanical piece arrives at the manufacturing unit, the instructions on the electronic tag of the piece must be read. Moreover, when the manufacturing unit starts working on the piece, it must be controlled so that the manufacturing process works correctly. Hence, the management of the manufacturing unit includes : the *readInstM* task which reads the instruction on the electronic tag, and the *controlM* task which controls the manufacturing unit ;
5. When a piece arrives at the lathe, the instructions on the electronic tag of the piece must be read. Moreover, when the lathe starts working on the piece, it must be controlled so that the process works correctly. Hence, the management of the lathe includes : the *readInstL* task which reads the instruction on the electronic tag, and the *controlL* task which controls the lathe ;
6. During the manufacturing process, the mobile carriage moves between the manu-

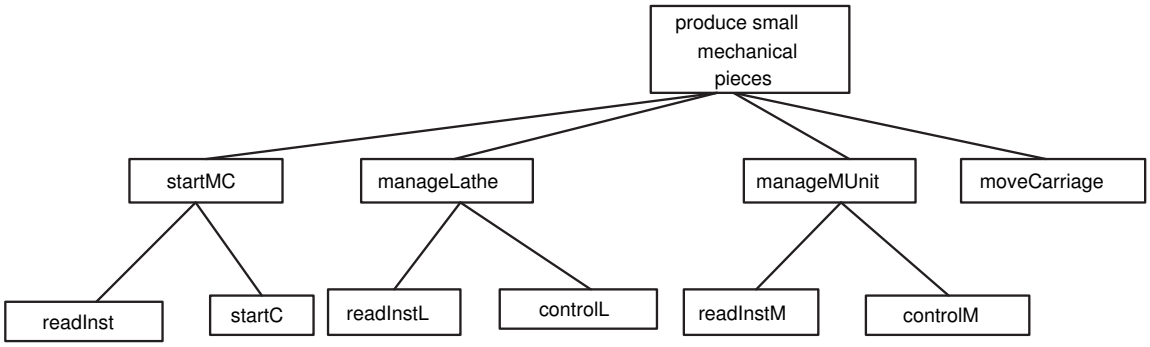

FIG. 3.4 – The task hierarchy of the manufacturing cell.

facturing unit and the lathe. Hence, the system should have a task, *moveCarriage*, which controls the mobile carriage ;

The different tasks identified in FATMAS will be performed by agents which are determined in the design phase.

Second, the requested resources of each task are determined. In the manufacturing cell example, we can identify two resources : the electronic tag and the database in which the orders are stored. We refer to these resources by *elecTag* and *dBase*. Each of these resources can be available or not. The parameter *elecTagAvailable* has two possible values : *0* when it is not possible, for any reason, to read the electronic tag and *1* otherwise. The database can be either accessible or not accessible. Hence, the parameter *dBaseAvailable* has two possible values : *0* when the database, for any reason, is not accessible and *1* otherwise. The task *readInst* has access to the database to read the manufacturing order and to the electronic tag to transcribe the instructions of the manufacturing order in order to perform them. However, the task *readInstL* has access to the electronic tag to read the instructions to perform, but it need not access the database.

Third, the designer determines for each task its* preconditions *so that it will be properly performed. These preconditions are considered as the possible sources of the *MAS*'s failures. They will characterize either the environment with which the system interacts or the system's boundary. As stated in Chapter 2, the preconditions of the tasks are evaluated based on the inputs of the agents. Each agent could respond to inputs either from the environment or from other agents. We also stated that we only focus on failures originating from inputs of the system's boundary since the MAS has no control on the failures originating from its environment. Consequently, in FATMAS, we make the difference between the environment in which the future MAS operates and

the implementation environment of the MAS. The environment in which the system operates is characterized in the task-environment model. However, the environment in which the MAS will be implemented is characterized at the implementation phase. This is not the case of other methodologies which only consider the environment as the implementation environment of the MAS as in SODA [43] and Gaia [57].

In the manufacturing cell example, the mobile carriage moves only if there is a piece to put or take from a machine-tool, or it could break and therefore be unable to move [1]. Hence, one of the preconditions of the task *carriageMove* is that the carriage has pieces to put in or take from the machine-tools. We refer to this precondition by *pieceAvailable* ; *pieceAvailable* $\in$ *prec(carriageMove)*. In addition, the task *readInst* must have access to the database to read the manufacturing order. Hence, one of the resources required by the task *readInst* is that the database is accessible. We refer to this resource by the parameter *dBase* ; *dBase* $\in$ R and *dBase* $\in$ *resources(readInst)*.

Consequently, the micro-process for creating a *task model* is :

- *Identify the different tasks needed to operate the future system by building a task hierarchy according to a top-down approach. The granularity of the decomposition depends on the specific problem. The task hierarchy only provides a general picture of the activities that will be present in the target system* ;
- *Determine, for each task, its name, and its required resources*. Each task, whether there exists some equivalent, requires resources or not. If not, enquire whether it should and could be replicated ;
- *Identify the different tasks' preconditions so that the possible causes of failure associated with each of them can be identified*. For each precondition, enquire whether and how it can be remedied.

The *task-environment model* can be formally defined using the sets E_s, G, I, τ, R, and the functions prec and post respectively defined in Definitions 2, 3, 5, 7, 20, 10 and 11.

Definition 58. *Let TEM refer to the task-environment model. We define TEM = {τ, E_s, G, I, R, prec, post}. τ is the set of tasks. R is the set of resources. E_s is the set of the external stimuli of the system. I is the set of the internal stimuli of the system. G is the set of goal situations. prec determines the preconditions of each task. post determines the post-conditions of each task.*

[1] In which case the manufacturing unit could be paralysed

The task-environment model micro-process provides the designer with three steps to allow him to identify the tasks, their preconditions, and their resources. Nevertheless, it is on the designer discretion to determine which tasks to take into account in the system, their preconditions, and their resources. Hence, there could not be an automatic way to build this model. However, the formal definition of this model will allow FATMAS to verify, at the design step, that all these tasks, their preconditions, and their required resources are taken into account in the future system.

3.3.2 Iterating the task-environment model

A failure occurs if the system has no control on the source of the stimuli that originated the failure or if the required resources are in failure. Hence, before pursuing the development of the other models, the designer should continue introducing new tasks or resources to the system in order to control the sources of failures, or to prevent from a resource failure. The designer may have to replicate resources. In this case, it has to identify the resources which cannot be replicated.

Furthermore, the designer may iterate over the task-environment model to add new tasks that would either control sources of failures, or control the resources (if they cannot be replicated) by preventing, detecting, or repairing their failures, whenever possible. If there are new tasks that need to be added to the system as a result of that process, then the designer must modify the *task-environment model* to include them. Hence, new preconditions or resources will be added to the description of the environment. The designer can iterate this process so that the *MAS* is as autonomous as possible with regard to tasks and resource failure. However, the designer cannot indefinitely reiterate.

At each iteration, the designer may add new tasks or resources to the system. Each task is added with the objective to either control a source of failure or to control a resource. For each added task t_i, we can associate a cost c_i. Moreover, for each added resource to the system r_j, we associate a cost c_j. The MAS can be in failure or in fatal failure. If a fatal failure occurs, then the MAS must be restarted. Hence, we associate a cost c to restart the system in case of system fatal failure. The designer has to evaluate the total cost of adding new tasks and resources to the system with the cost of restarting the system in case of fatal failure. We propose that the designer stops iterating if he finds that the total cost of adding tasks and resources to the system is greater than the cost of restarting the system in case of fatal failure. Consequently, FATMAS is based on a cost/benefit approach to support the designer when iterating over the task-environment model.

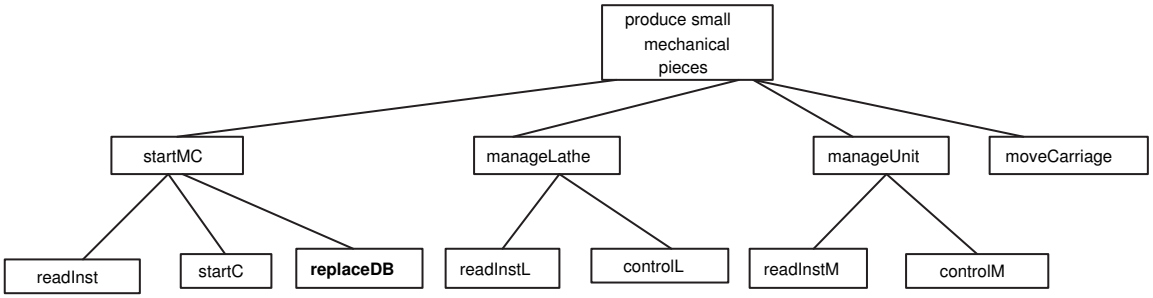

FIG. 3.5 – Modification of the task hierarchy of the manufacturing cell.

In the manufacturing cell example, if the machine-tools are no longer able to read the instructions from the database then the system should be able to connect to a backup database which duplicates the orders since the cost to replicate the database is less than the cost to restart the manufacturing cell. Of course, a backup database is a replicated resource. We refer to this resource as *dBase1*. Hence, dBase1 $\in$ R, and using the function resourceEquivalence (see Definition 24), we have resourceEquivalence(readInst, dBase, dBase1) = true. In this case, a new task, called *replaceDB*, is added to the system's tasks to allow machine-tools to connect to dBase1. This task is under the control of the task which starts the manufacturing cell.

If dBase is in failure, then the MAS uses dBase1. Consequently, the MAS evaluates whether dBase is in failure. Let us use the precondition *dBaseAvailable* to check whether dBase is available. If dBase is not available, then the truth value of dBaseAvailable is false. However, the MAS will run with dBase1. Hence, it changes the truth value of dBaseAvailable to true. Consequently, replaceDB controls the precondition dBaseAvailable. Using the function effector (see Definition 15), we have effector(replaceDB, dBaseAvailable) = true ; dBaseAvailable $\in$ I (the set I is defined in Definition 5). Furthermore, the designer evaluates the cost to replicate all the machine tools. However, the machines cannot be replicated since it is very expensive to replicate a lathe or a carriage. Consequently, the preconditions describing the state of each machine will characterize the system's environment. The new task model is presented in Figure 3.5.

At this point, we identified the different tasks that should be performed in the system. In the next model, we will specify the agents that will operate them.

3.4 The Design Phase

The design phase provides the highest level specification of the software to be developed, to meet the specification. In FATMAS, the design phase provides two models and a technique : the *agent model*, the *interaction model*, and the *reorganization technique*. We present each of these models in the following sub-sections.

3.4.1 The Agent Model

The *task-environment model* has identified the different tasks that the *MAS* may perform. We propose to specify the agents by considering *groups* of tasks, each group being a set of logically related tasks, bound by the same constraints or goals. For instance, each different output of the system could determine different group of tasks. The complementarity of tasks could be dealt with by different groups. This grouping of tasks, i.e, of functionalities is left to the expertise of the system designer. The agents, identified in the *agent model*, represent the types of agents that will operate in the future system. This model does not determine the agent instances that will really operate in the system.

In our example, the tasks can be grouped in four groups in order to deliver mechanical pieces. The first group has the control of the manufacturing unit. The second group has the control of the lathe. The third group has the control of the carriage. The fourth group corresponds to the activities which should be performed when the manufacturing unit starts operating. Hence, we could assign an agent to each of these groups. Thus, there could be four types of agents in the manufacturing cell example :

- *ManufacturingUnitAgent* : an agent which controls the manufacturing unit ;
- *LatheAgent* : an agent which controls the lathe ;
- *ManufacturingAgent* : an agent which starts the manufacturing cell ;
- *CarriageAgent* : an agent which controls the carriage.

In this example, the set Λ (see Definition 29) of agents is : Λ=$\{$*ManufacturingUnitAgent*, *LatheAgent*, *ManufacturingAgent*, *CarriageAgent*$\}$.

Each agent must have access to the resources required by its tasks. In the manufacturing cell example, the *required resources* of the *ManufacturingAgent* is the database in which the orders are stored and the tag in which it writes instructions ; $dBase \in resourceAccessibility(ManufacturingAgent)$. For both the *LatheAgent* and the *ManufacturingUnitAgent*, the required resource is the tag from which they read instructions.

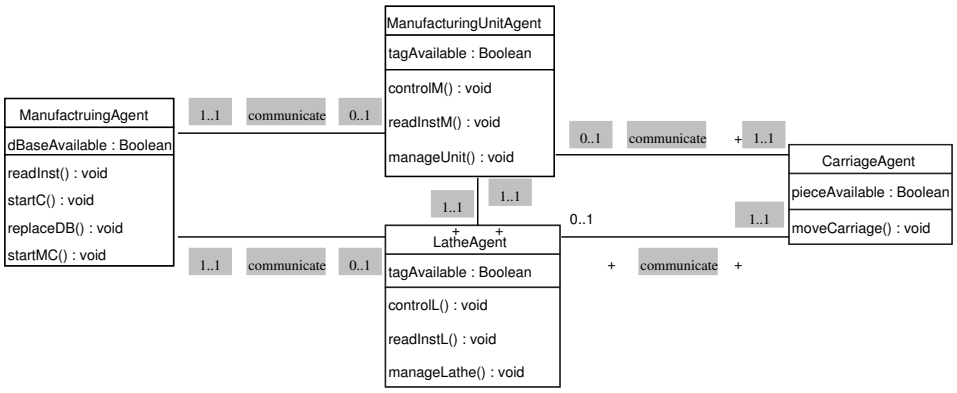

FIG. 3.6 – The agent model of the manufacturing cell.

The required resource of the *CarriageAgent* is the carriage.

Each agent can be related to other agents according to their interactions. In the agent model of the manufacturing cell (see Figure 3.6), agents are related by association relations as defined in [41]. The *ManufacturingAgent* can either communicate with the *ManufacturingUnitAgent* or not. Hence, the cardinality of this relation is (0..1) on the *ManufacturingUnitAgent* side. Nevertheless, the *ManufacturingUnitAgent* must communicate with the *ManufacturingAgent*. So, the cardinality of this relation is (1..1) on the *ManufacturingAgent* side. The associations between the *LatheAgent* and the *ManufacturingAgent*, the *LatheAgent* and the *CarriageAgent*, and the *ManufacturingUnitAgent* and the *CarriageAgent* are similar to the association between *ManufacturingAgent* and *ManufacturingUnitAgent*.

The micro-process used to create the *agent model* is :

- *Define groups of logically related tasks that share complementary, common goals or are bound by the same constraints* ;
- *Associate a type of agent to each grouping* ;
- *For each type of agent, define its required resources as the union of the resources of all the tasks it will perform* ;
- *Define the different agent relations.*

The agent model can be formally defined based on Λ, τ, and the function task which are respectively defined in definitions 7, 29, and 30 of Chapter 2.

Definition 59. *Let AM be the agent model. AM = {Λ, τ, Rel, task}. A is the set of*

agents. τ is the set of tasks to be performed by agents. Rel :$A \times A \times$ Label $\rightarrow$ {0, 1} which relates two agents and in which Label is a kind of relation which can be observed between two agents. The function task *determines the tasks which will be performed by each type of agent.*

From the formal definition of the agent model, it is possible to verify, for a designer using FATMAS, that the set of tasks taken into account in the task-environment model is equal to the set of tasks taken into account in the agent model. Doing so, tasks' preconditions and resources must be declared in the agents' attributes. Moreover, the third item of the agent model micro-process makes that each agent has access to the required resources of its tasks. Consequently, at this stage, we ensure that the different tasks are taken into account in the future system and that each agent has access to the required resources of its tasks.

In the *agent model*, the different tasks have been assigned to types of agents. In the next sections, we present the *agent interaction* models to model the dynamic behavior of the *MAS*.

3.4.2 The Agent Interaction Model

In this section, we present the *agent interaction model* which details *how* the system operates to reach its objectives. This model is defined based on the definition of interaction [41] that is the ongoing two-way or multiway exchange of data among computational entities, such that the output of one entity may influence the later outputs of the other entity.

The agent interactions are defined by the information received or sent by the agents' tasks. Agent interactions can be represented by either a sequence diagram or a collaboration diagram as defined in [53]. Agents' interactions can be deduced from the agent model and the relations between the different tasks.

For the manufacturing cell example, we determine the different calls between the tasks. For example, the task controlM can ask the task moveCarriage to move the carriage to the manufacturing unit. Hence, the controlM() task calls the moveCarriage() task. Consequently, based on the definition of function call (see Definition 32), call(controlM(), moveCarriage())=true. We can determine the different calls between the tasks.

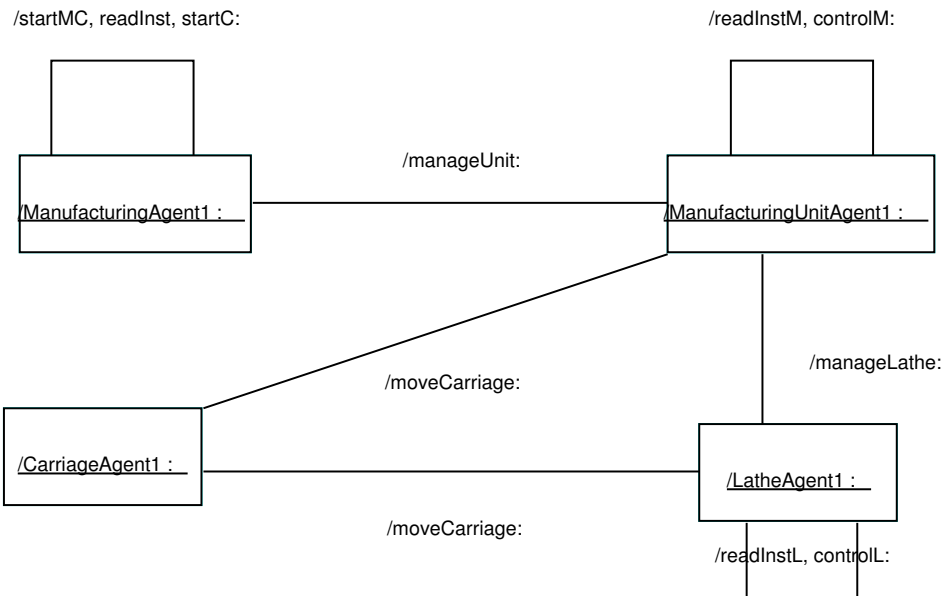

FIG. 3.7 – The collaboration diagram of the manufacturing cell.

Based on this relation, ManufacturingUnitAgent communicates with the CarriageAgent. Consequently, communication(ManufacturingUnitAgent, CarriageAgent) = true. This communication can be verified in the agent model of Figure 3.6. The collaboration diagram is presented in Figure 3.7.

The micro-process of the *agent interaction model* is :

- *Determine the set* Λ *of agents that will operate in the system* ;
- *Determine the information sent or received by tasks ; mainly from the task-environment model, task hierarchy, and the agent model. If a task is a sub-task of another task, then the two tasks communicate with each other. Moreover, if an agent is in relation with another agent in the agent model, then these two agents interact with each other.* Hence, the agent interaction model can be deduced based on the task-environment model and the agent model. Consequently, a continuity in building the models is provided to the designer which could ease its process in designing the MAS ;
- *Deduce the agent interaction model.*

We formally define the interaction model based on the definition of Λ and of the function interaction (see Definition 34).

Definition 60. *Let IM be the agent interaction model. IM = {Λ, interaction}. Λ is the set of agents. The function interaction determines the different agents with which an agent interacts.*

At this stage, we have presented the *task-environment model*, the *agent model*, and the *interaction model*. In the next section, we present the *reorganization technique* that defines how the system will recover from its failures.

3.4.3 The Reorganization Technique

The reorganization technique described in Chapter 2 includes a fault-prevention technique, a fault-recovery technique, and a fault-tolerance technique. The fault-prevention technique is dealt with in the task-environment model. We proved the validity of the fault-recovery technique in Chapter 2. In this section, we determine, based on the fault-tolerance technique, the agents which should be replicated. In the fault-tolerance technique, only the *critical agents* (see Definition 50) are replicated. The non-critical agents (see Definition 51) will have their tasks replicated in other *original agents*.

In the manufacturing cell example, we determine the set of critical agents in order to determine which agents and tasks need to be replicated. There are two *critical agents* which are *CarriageAgent* and *ManufacturingAgent*, since the other agents do not have access to the resources that these two agents can access. So, each instance of these two agent types will be replicated. The *ManufacturingUnitAgent* and the *LatheAgent* are two *non-critical agents* since each of them has access to the required resources of the other agent. Hence, each instance of one of the agent types will have its tasks replicated in another instance of the other agent type. Consequently, instead of replicating four agent types, we only replicate two. Furthermore, the two tasks readInstM and readInstL are identical since they read information from an electronic tag. readInstM is performed by ManufacturingUnitAgent, and readInstL is performed by LatheAgent. Based on the function idTask (see Definition 12), we have idTask(readInstM, readInstL) = true. Consequently, it is not necessary to replicate each of these tasks in the other agent. Then, we have, based on Definition 53, and Definition 52 : LatheAgent ∈ duplicate(ManufacturingUnitAgent), and coupling(readInstM) = (ManufacturingUnitAgent, LatheAgent).

Now, that the agents and their replica were determined, the agent model must be updated. Each agent replica or each replicated task will have its name preceded by the tag *repl* followed by the name of the original agent. Hence, the updated agent model of

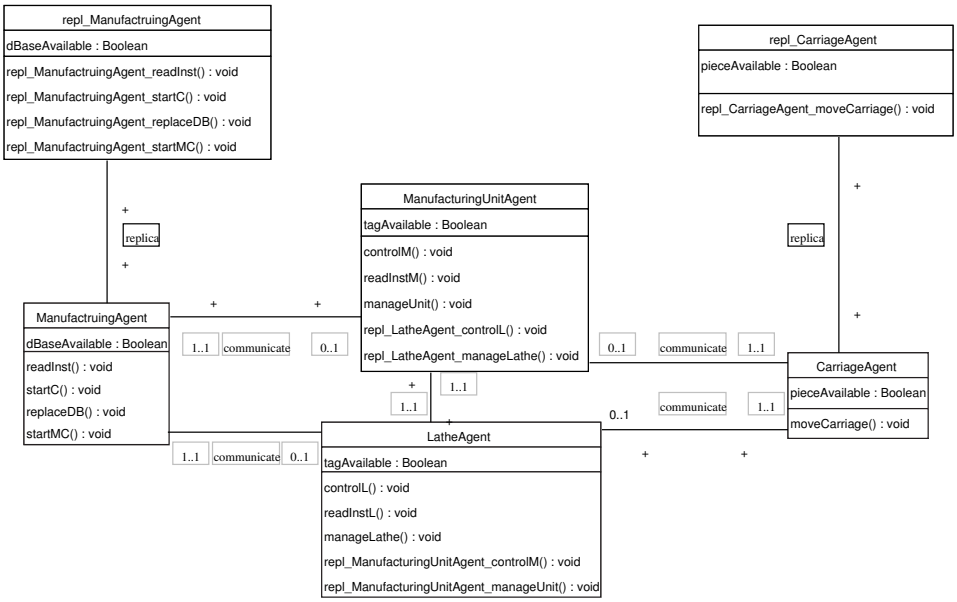

FIG. 3.8 – The updated agent model of the manufacturing cell.

the manufacturing cell example is presented in Figure3.8.

In the next section, we present the *implementation model*.

3.5 The implementation Phase

The implementation phase allows to code the programs and to characterize the support environment. FATMAS provides an implementation model to develop the future system.

3.5.1 The Implementation Model

The *implementation model* determines the *development environment* of the *MAS*. The *development environment* specifies the physical architecture of the system, and the rules used to automatically generate a skeleton code for the system.

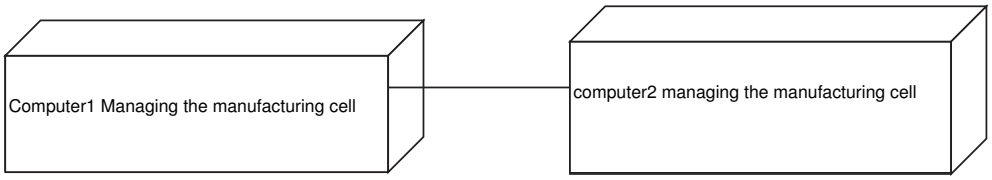

FIG. 3.9 – The deployment diagram of the manufacturing cell.

MAS Physical Architecture

The MAS physical architecture first determines the number of physical components on which the MAS will run, and then their descriptions. In Chapter 2, we proposed, based on agents grouping, a way to determine the number of physical components on which the system should run. From the manufacturing cell agent model, the agents can be grouped into two groups as defined in Definition 57 : g_1 = {ManufacturingAgent, ManufacturingUnitAgent, CarriageAgent} and g_2 = {LatheAgent, repl_ManufacturingAgent, repl_CarriageAgent}. Hence, there should be two computers in the MAS to manage the physical components (lathe, manufacturing unit, carriage, manufacturing cell). Each computer requires access to the available machines and resources. If a computer is out of service, there will be another computer to replace it.

The physical architecture deals with the detailed description of the hardware on which the system will run [53]. In the context of *FATMAS*, there is no hardware specification in addition to UML's hardware specifications [53]. Hence, we propose to use the physical architecture of UML materialized by the *deployment diagram*.

The *deployment diagram* of the manufacturing cell example is presented in Figure 3.9.

Code Generation

The second step in the *implementation model* is *code generation*. The code generation allows us to generate the skeleton code of the application. To automatically generate the code, in the context of object-oriented programming, we propose the following micro-process :

- *the agent name is the class name* ;
- *each agent's task is represented by a method in the class representing the agent* ;
- *each task precondition is declared as a variable* ;
- *each information exchanged by two tasks is translated into a variable declared in*

```
public class LatheAgent extends ManufactruingAgent {
  /* {src_lang=Java}*/


    public repl_ManufacturingUnitAgent  1..1;
    public repl_ManufacturingUnitAgent  0..1;
    public CarriageAgent myCarriageAgent;
    public ManufacturingUnitAgent myManufacturingUnitAgent;

  public void controlL() {
  }

  public void readInstL() {
  }

  public void manageLathe() {
  }

  public void repl_ManufacturingUnitAgent_controlM() {
  }

  public void repl_ManufacturingUnitAgent_readInstM() {
  }

  public void repl_ManufacturingUnitAgent_manageUnit() {
  }
}
```

FIG. 3.10 – The code of LatheAgent.

the two classes participating in this interaction.

For the manufacturing cell example, the skeleton code of the *LatheAgent* is presented in Figure 3.10.

At this point, we have presented the different models that *FATMAS* will produce, together with their micro-processes. In the next section, we present *FATMAS* macro-process that guides the designer through the different models.

3.6 The FATMAS Macro-process

There are two kinds of approaches to design a system : a task-driven approach or a goal-driven approach. In FATMAS, each agent performs several tasks. Each task is performed under conditions and could require resources to be correctly performed. Hence, each agent in the MAS has to react to inputs, and check whether it can perform tasks. So, FATMAS uses a task-driven process since agents have a reactive architecture based on event-condition-action rules. Moreover, FATMAS includes a reorganization

technique. It provides a way for the system to overcome agent failures in order to perform its required tasks and consequently to reach its objectives. As stated in [13] [48], a goal-driven process is appropriate to deal with system failure since it allows the system to adapt its plan in order to overcome a failure. Consequently, FATMAS includes by the mean of its reorganization technique a goal-driven process to design the system. Consequently, FATMAS uses a hybrid process, a macro-process, based on task-driven and goal-driven processes.

The *FATMAS* macro-process model defines the different steps to follow to go from one model to another. These steps allow a designer to establish a link between the different models.

As seen before, the additional control over effectors of preconditions and over the resources, may require that new tasks be added to the *task-environment* model. Also an agent replication may require that additional agents be added to the agent model. Hence, the macro-process of our methodology is summarized as follows and presented in Figure 3.11 :

Step 1 : Develop the task-environment model ;
Step 2 : For each identified task precondition, determine whether it is possible to add new tasks to the system to control these preconditions ;
if new tasks are added to the system **then**
  go to Step 1 ;
end if
Step 3 : Develop the agent model ;
Step 4 : Develop the agent interaction models ;
if new tasks are added to the system **then**
  go to Step 1 ;
end if
Step 5 : Develop the implementation model ;
if new tasks are added to the system **then**
  go to Step 1 ;
end if
Step 6 : Implement the reorganization technique ;
if new agents are added to the system after agent replication **then**
  go to step 3 ;
end if

Now, we have defined all the models that should be developed using the *FATMAS* approach. *FATMAS* determines different sources of failures other than those related to code bugs. This methodology should allow to build *FTMAS* which, under certain

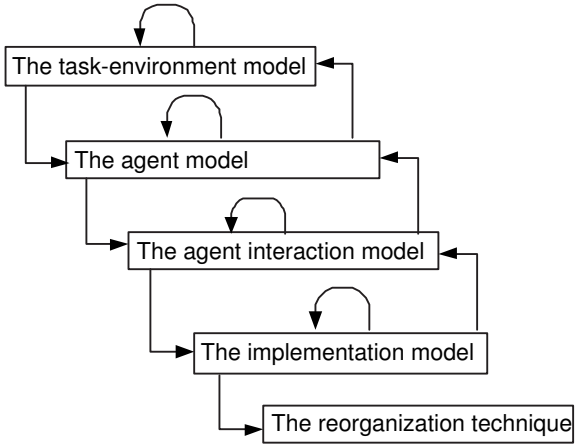

FIG. 3.11 – The macro-model of the *FATMAS* methodology.

conditions, can recover from its failures.

3.7 Application Domains of FATMAS

As any methodology, FATMAS has restrictions on its application domains. First, FATMAS determines the different tasks and agents of the system which do not evolve over time. Hence, tasks and agents are known before implementing the system. Consequently, FATMAS can be applied to design MAS in which the set of agents is known in advance. Hence, FATMAS cannot be applied to design an open MAS in which agents come from different locations, and are not known in advance.

Furthermore, FATMAS can be applied to build large-scale multi-agent systems. In [30], scalability refers to how well the capacity of a system to do useful work increases as the size of the system increases. Thus, the scalability of a MAS requires that we take a society view of the performance of the system at different sizes. The variables that affect the performance of MAS include, but not exclusively, the number of agents, the number of tasks/goals the agent are carrying out, and the type of the coordination protocols employed [30]. In FATMAS, we do not specify a particular coordination protocol to be used by agents. However, FATMAS minimizes the number of agents to operate in a fault-tolerant MAS, and allows the agents to operate a high number of tasks. Nevertheless, at each iteration, new tasks may be added and hence new preconditions must be dealt with. If the number of tasks increases, so is the number of preconditions, and so is the number of iterations. This may render the system development complex. Hence, we can advance that FATMAS could be applied to build large-scale multi-agent systems.

Meanwhile, there should be a limit to the number of agents which must operate in a MAS designed using FATMAS, that beyond it the performance of the system degrades.

3.8 Summary

In this chapter, we proposed **FATMAS** a design methodology to build fault-tolerant multi-agent systems. This methodology uses an incremental strategy, based on a cost/benefit approach, to build the system. It has the advantage to provide techniques for *fault-prevention, fault-recovery, and fault-tolerance* right from the design phase, which is not the case of existing methodologies used to design multi-agent systems. It has the following main contributions :

1. It helps the designer to determine the system boundary and the system's environment ;
2. It proposes a task reallocation mechanism which automatically determines which tasks (agents) should be replicated and in which other agents ;
3. It reduces the system complexity by minimizing the replication of agents ;
4. It proposes an approach to distribute the system over physical components ;
5. It proposes a *MAS fault-recovery* technique that is embedded within the designed *MAS* which allows it to detect certain faults and recover from them, making the *MAS* more robust to fault prone (unstable) environments.

Chapitre 4

Further Case Study : A Fault-Tolerant Multi-Agent System For Document Printing

In the previous chapter, we applied FATMAS to design a fault-tolerant MAS to manage a manufacturing cell. In this chapter, we apply *FATMAS* to design a fault-tolerant multi-agent system to manage a company's printers. By the use of this example, we highlight the iterative approach of *FATMAS* which is useful to add fault-tolerant capabilities to a *MAS*, right from the design phase of the system. The different sets and functions referred to in this chapter were defined in Chapter 2. Furthermore, in Chapter 3, we specified for each model the definitions and functions it refers to in Chapter 2.

Let us consider a company which has several employees and three printers. Each employee has access to the three printers (see Figure 4.1). One of the printer prints documents in the format $8^{1/2}/11$, another printer prints documents in the format $8^{1/2}/14$, and the other one prints documents in the format $8^{1/2}/17$. Each employee needs that his/her documents be printed despite any problem that could occur.

Our proposed example is a distributed problem since there are several distributed users and printers. In order to exemplify the usefulness of *FATMAS*, we propose a multi-agent system to manage the various printing tasks. This *MAS* must be fault-tolerant since any document should be printed despite any failure that could occur with the printers. Moreover, any problem that appears in a printer must be repaired since every printer is dedicated to a specific printing task. As described above, *FATMAS* is based on an incremental development. As will be shown below, it has taken four iterations to design the proposed system. In the following sections, we describe these iterations.

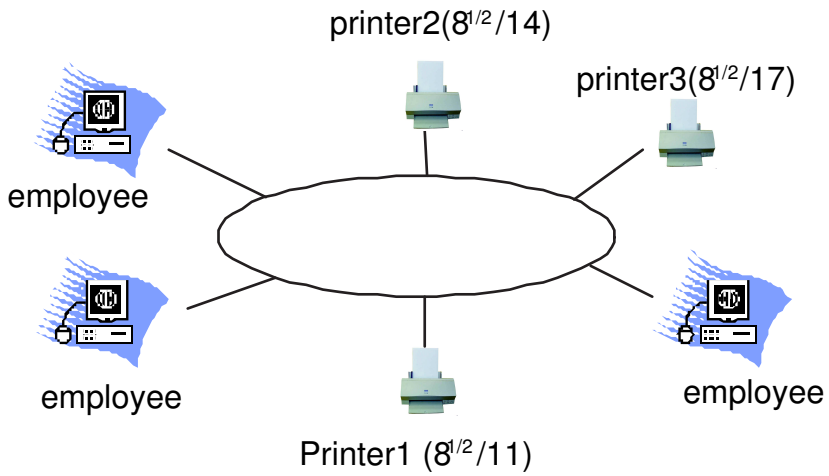

FIG. 4.1 – The company local network.

Before presenting the different iterations, we provide the conventions used to choose the names of the different tasks, preconditions, and agents. The names of the tasks are given by the concatenation of a verb describing the task and the object on which the task is applied (e.g, if there is a task to manage a printer, then this task is called *managePrinter*). The names of the preconditions are given by either the concatenation of the name of a resource with the term *available* to state whether the resource is available (e.g, if, for a particular task to be correctly performed, a printer must be available, then one of the preconditions of the tasks is *printerAvailable*) , or the concatenation of the name of the task and the term *receptive* to state whether it is possible to call this task (e.g, if there exists a task called *sendMessageAvailability* which calls the task *notifyRepairStatus*, then one of the preconditions of *sendMessageAvailability* is *notifyRepairStatusReceptive*). Finally, the names of the agents are given by the concatenation of the semantic of the grouped tasks and the term *agent* (e.g, if there is an agent that manages a printer, then this agent is called *PrinterManagerAgent*).

4.1 The First Iteration (Analysis Phase)

As presented in Chapter 3,*FATMAS* has three phases : analysis, design, and implementation. In the analysis phase, we iteratively create the *task-environment model* that aims at determining the system boundary. In the task-environment model, we determine the tasks that will be performed by the future agents of the system, the preconditions of these tasks, and their required resources. The case study refers to different sets or relations defined in Chapter 2.

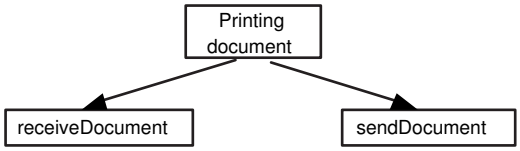

FIG. 4.2 – Task hierarchy at iteration 1.

4.1.1 The Task-environment Model

The system's objective is to guarantee that a document sent to a printer is printed as requested. To reach this objective, the document is received by the application that requests its printing. Hence, the system needs at least to perform the following tasks :

1. *receiveDocument* : a task that receives from an employee the document to be printed. It does not require any resource to be properly performed [1] ;
2. *sendDocument* : a task that sends the document to the printer to which the employee has access. This task requires an access to a printer.

The task hierarchy associated to the first iteration is presented in Figure 4.2. Hence, $\tau = \{receiveDocument, sendDocument\}$.

Each task may require resources to properly operate. Hence, for each task, we determine its required resources.

1. The *receiveDocument* task does not require any resource to correctly operate.
2. The *sendDocument* task requires access to a printer, considered as its needed resource ;
   resources(sendDocument) = {printer1, printer2, printer3}, since the page size of the document could be any of the three available.

Hence, the set of resources R = {printer1, printer2, printer3}.

Each task has preconditions that must be met before it can be performed. The tasks' preconditions are :

[1] To simplify the example, we assume no limit on its input buffer

1. The *receiveDocument* task does not have any precondition ; *prec(receiveDocument)* $= \varnothing$;
2. The *sendDocument* task has a precondition that the printer to which the document must be sent is not defective (*printerNotDefective*) ; *prec(sendDocument)* $=$ {*printerNotDefective*}

Finally, if a printer is defective, then the system must look for another printer while trying to repair the defective one so that the *MAS* continues to operate properly. Hence, the system should have some control over its printers in order to detect their defective components and to repair them, if possible. The designer evaluates the cost to add new tasks to detect and repair problems in the printers. This will consist of developing agents to manage the printers and try to repair them. Doing so, this could make employee not waste their time in waiting for any person to repair the printer. Consequently, new tasks should be added to the system, as presented in the next iteration.

4.2 The Second Iteration

In the first iteration, we concluded that there must be new tasks in the system to manage the printers in order to detect defective components and repair them, if possible. Hence, we iterate over the design of the *task-environment model*.

4.2.1 The Task-environment Model

In order to manage a printer, a printer manager task (*managePrinter*) is added to the system. This task identifies the problems that could occur in a printer, repair them and then report them to the end-user. Hence, this task is decomposed into three subtasks : *detectPrinterProblems* that identifies the problems in the printer, *repairPrinter* that repairs the printer if needed, and *reportProblems* that reports the problems to the end-user. The end-user must be able to receive the reported problems as information sent to him. Hence, there must be a task that receives and presents these problems to the end-user (*reportProblems*).

So, the system must now have the following tasks :

1. *receiveDocument* : a task that receives the document to print. It does not need to access any resource ;

2. *sendDocument* : a task that sends the document to the printer. It requires to access one of the three printers ; *resources(sendDocument)*={printer1, printer2, printer3} ;
3. *managePrinter* : a task that manages a printer. It requires to access to the printer ; *resources(managePrinter)*={printer1, printer2, printer3} ;
4. *detectPrinterProblems* : a task that identifies the problems that occur in the printer. It requires the access a the printer ; *resources(detectPrinterProblems)* = {printer1, printer2, printer3} ;
5. *repairPrinter* : a task that repairs, if possible, the problems that occur in the printer. This task can be decomposed into the following ones :
   (a) *updatePrinterDriver* : a task that updates a printer driver if there is a problem with it. It requires the access to the printer ; *resources(updatePrinterDriver)* = {printer1, printer2, printer3} ;
   (b) *detectPrinterCPUProblems* : a task that detects whether there is a problem with the printer CPU. It requires the access to a printer ; *resources(detectPrinterCPUProblems)* = {printer1, printer2, printer3} ;
   (c) *detectHardwareProblems* : a task that detects hardware problems. It requires the access to a printer ; *resources(detectHardwareProblems)* = {printer1, printer2, printer3} ;
   (d) *evaluatePaperQuantity* : a task that determines whether a printer must be fed with paper. It requires the access to the printer ; *resources(evaluatePaperQuantity)* = {printer1, printer2, printer3} ;
   (e) *evaluateInkQuantity* : a task that determines whether there is a sufficient quantity of ink in a printer. It requires the access to the printer ; *resources(evaluateInkQuantity)* = {printer1, printer2, printer3} ;
   (f) *detectPrinterConnectionProblems* : a task that detects whether an end-user has connection problems with its printer. It requires the access to the printer ; *resources(detectPrinterConnectionProblems)* = {printer1, printer2, printer3} ;
   (g) *notifyRepairStatus* : a task that notifies whether the repair task was performed properly. It does not require any resource to be properly performed.
6. *reportProblems* : reports any problem if the printing task is not achieved, or sends an acknowledgement to the end-user to inform him that the printing task was properly done. It does not require any resource ;

The task hierarchy associated to the second iteration is presented in Figure 4.3. The set τ = {*receiveDocument*, *sendDocument*, *managePrinter*, *detectPrinterProblems*, *repairPrinter*, *updatePrinterDriver*, *detectPrinterCPUProblems*, *detectHardwareProblems*, *evaluatePaperQuantity*, *evaluatePaperQuantity*, *detectPrinterConnectionProblems*, *notifyRepairStatus*, *reportProblems*}.

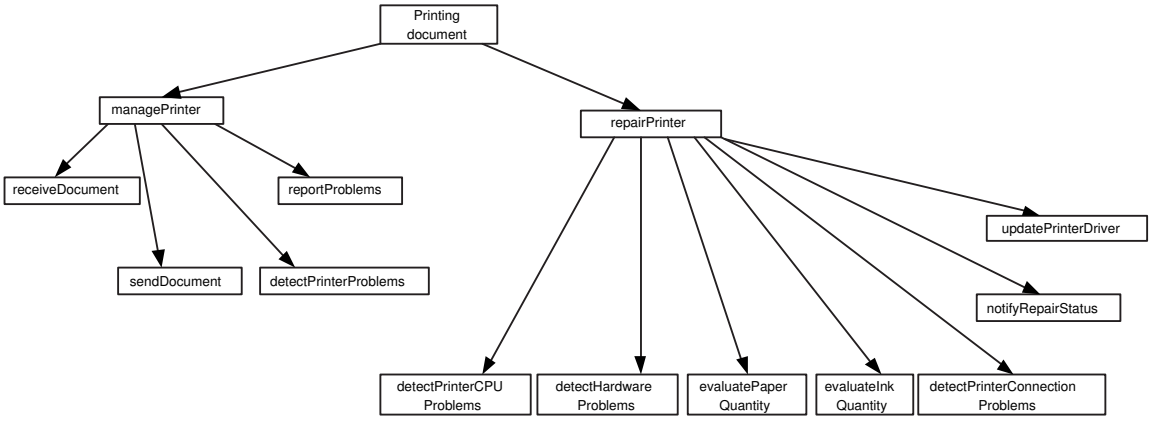

FIG. 4.3 – Task hierarchy at iteration 2.

– For each task, we identify the preconditions that enable it to properly perform.

1. The *receiveDocument* task does not have any precondition ;
2. The precondition of the *reportProblems* task is that there exists a problem in a printer to report to the end-user (*problemDetected*). *prec(reportProblems)* = {*problemDetected*} ;
3. The precondition of the *sendDocument* task is that the *managePrinter* task is receptive (*managePrinterReceptive*) ; *prec(sendDocument)* = {*managePrinterReceptive*} ;
4. The precondition of the *managePrinter* task is that the printer is connected (*printerAvailable*) ; *prec(managePrinter)* = {*printerAvailable*} ;
5. The precondition of the *detectPrinterProblems* task is that the printer is connected (*printerAvailable*) ; *prec(detectPrinterProblems)* = {*printerAvailable*} ;
6. The precondition of the *evaluatePaperQuantity* task is that the printer is connected (*printerAvailable*) ; *prec(evaluatePaperQuantity)* = {*printerAvailable*} ;
7. The precondition of the *evaluatePaperQuantity* task is that the printer is connected (*printerAvailable*) ; *prec(evaluatePaperQuantity)* = {*printerAvailable*} ;
8. The precondition of the *detectPrinterCPUProblems* task is that the printer is connected (*printerAvailable*) ; *prec(detectPrinterCPUProblems)* = {*printerAvailable*} ;
9. The precondition of the *detectHardwareProblems* task is that the printer is connected (*printerAvailable*) ; *prec(detectHardwareProblems)* = {*printerAvailable*} ;

10. The precondition of the *detectPrinterConnectionProblems* task is that the printer is connected (*printerAvailable*) ; *prec(detectPrinterConnectionProblems)* = {*printerAvailable*} ;
11. The precondition of the *updatePrinterDriver* task is that the printer is connected (*printerAvailable*) and it must have the needed software to update the driver (*softwareAvailable*) ; *prec(updatePrinterDriver)* = {*printerAvailable*, *softwareAvailable*} ;
12. The precondition of the *notifyRepairStatus* task is that the *reportProblems* task is receptive (*reportProblemsReceptive*) ; *prec(notifyRepairStatus)* = {*reportProblemsReceptive*}.

Finally, if the software needed to update a printer is not provided, then the system is in failure. Also, if there is not enough paper available, or there is not any toner cartridge available, or there is not an available CPU to replace the defective one, then the system is also in failure. At this stage, we determine that if a problem occurs while printing, then it is only possible to repair it in case of software update. Otherwise, the MAS will not be able to overcome the failure. However, the MAS could be able to identify the problem, and check whether the required materiel to make the printer functioning is available in the company's warehouse. If the answer is affirmative, then the person who will repair the printer will be informed that the material is available in the company's warehouse. Hence, this person will not have to order the required material from suppliers.

Based on the cost/benefit evaluation, the cost to add new tasks to check whether the required material is available in the company's warehouse is not expensive. Furthermore, this makes the company save time since the person who will repair the printer is informed that the material is available. In this case, the system should not be in failure for a long time.

So, we propose to add new tasks to the system in order to check whether the needed materials are available so that the printer can be repaired. This is done in the third iteration.

4.3 The Third Iteration

In the second iteration, we concluded that the system could not be able to achieve its objective in several cases. Hence, new tasks should be added to handle these cases,

as much as possible.

4.3.1 The Task-environment Model

The system must now be able to check whether there is the needed quantity of paper, toner cartridge, CPU, and hardware material available in the company's warehouse. Hence, the following tasks are added to the system :

- *determineMaterial* : this task determines the needed material to repair a printer. It is decomposed into the following subtasks :
  1. *getPaper* : this task determines the quantity of paper available ; It requires an access to a data base that has information about available stocks (*DbAvailable*) ; *resources(getPaper)* = {DbAvailable} ;
  2. *getToner* : this task determines the quantity of toner cartridges available ;It requires an access to a data base that has information about available stocks (*DbAvailable*) ; *resources(getToner)* = {DbAvailable} ;
  3. *getCPU* : this task determines the quantity of CPU available ;It requires an access to a data base that has information about available stocks (*DbAvailable*) ; *resources(getCPU)* = {DbAvailable} ;
  4. *getSoftware* : this task determines whether the needed software is available on particular servers ; It requires an access to a data base that has information about available stocks (*DbAvailable*) ; *resources(getSoftware)* = {DbAvailable} ;
  5. *repairConnection* : this task determines which repairs are needed to reestablish the connection with a printer ; It requires an access to a data base that has information about available stocks (*DbAvailable*) ; *resources(repairConnection)* = {DbAvailable} ;
  6. *sendMessageAvailability* : this task sends a message about the availability of the needed material.

Thus, the set of resources R = {printer1, printer2, printer3, DbAvailable}. The task hierarchy associated to the third iteration is presented in Figure 4.4. The set τ = {*receiveDocument*, *sendDocument*, *managePrinter*, *detectPrinterProblems*, *repairPrinter*,*updatePrinterDriver*, *detectPrinterCPUProblems*, *detectHardwareProblems*, *evaluatePaperQuantity*, *evaluatePaperQuantity*, *detectPrinterConnectionProblems*, *notifyRepairStatus*, *reportProblems*, *receiveMessageAvailability*, *sendMessageAvailability*, *repairConnection*, *getSoftware*, *getCPU*, *getToner*, *getPaper*, *determineMaterial*}.

For each task, we identify the preconditions that enable it to be properly performed.

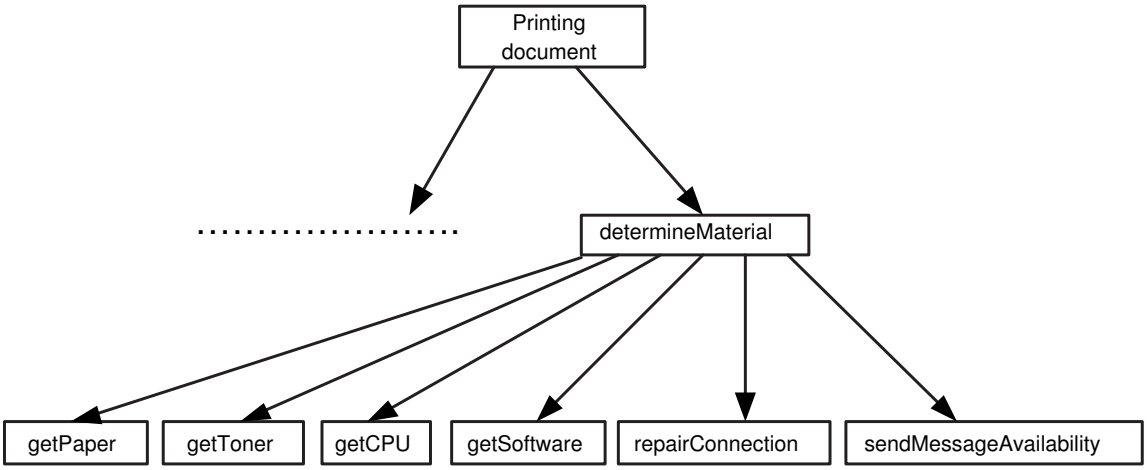

FIG. 4.4 – Task hierarchy at iteration 3.

1. The *getPaper*, *getToner*, *getCPU*, *bringHardware*, *repairConnection*, and *getSoftware* tasks have no preconditions ;
2. The precondition of the *sendMessageAvailability* task is that the *notifyRepairStatus* task is receptive (*notifyRepairStatusReceptive*) ; *prec(sendMessageAvailability)* = {*notifyRepairStatusReceptive*}.

There are several printers in the system. However, if there is a printer out of service, then the system should be able to repair it. Hence, if the system finds that there are not enough stocks in the company's warehouse to help in repairing a printer, then the printer could not be repaired. Hence, the system must be able to order the material for which there is not enough stocks from the company's suppliers. Hence, new tasks must be added to the system to order materials from suppliers. Hence, the person, who will repair the printer, will not have to order the required material from suppliers.

Based on the cost/benefit evaluation, the cost to add new tasks to order the required materials from suppliers is not expensive. Furthermore, this makes the company save time since the person who will repair the printer will not order the required material.

4.4 The Fourth Iteration

In the third iteration, we concluded that the system could not be able to achieve its objective. Hence, we have to modify our previous system design so that it can order the needed materials.

The system is now able to order from suppliers the needed materials such as paper, toner cartridge, CPU, or any hardware material. So, new tasks are added to the system accordingly, and some modifications need to be made to the previous *task-environment* model so that the system can order materials from its suppliers.

4.4.1 The Task-environment Model

– *orderMaterial* : this task orders the needed material to a supplier. In order to be able to order paper, toner cartridge, CPU, and hardware, this task is decomposed into the following subtasks :

1. *orderPaper* : this task orders papers to a supplier ;
2. *orderToner* : this task orders toner cartridges to a supplier ;
3. *orderCPU* : this task orders CPU to a supplier ;
4. *orderHardware* : this task orders hardware to a supplier ;
5. *orderSoftware* : this task orders software to a supplier ;
6. *receiveOrderStatusSupplier* : this task receives the answer from the supplier on its ability to fulfill the order.

The set of resources R = {printer1, printer2, printer3, DbAvailable, Supplier}. The task hierarchy associated to the fourth iteration is presented in Figure 4.5. The set τ = {*receiveDocument*, *sendDocument*, *managePrinter*, *detectPrinterProblems*, *repairPrinter*,*updatePrinterDriver*, *detectPrinterCPUProblems*, *detectHardwareProblems*, *evaluatePaperQuantity*, *evaluatePaperQuantity*, *detectPrinterConnectionProblems*, *notifyRepairStatus*, *reportProblems*, *receiveMessageAvailability*, *sendMessageAvailability*, *repairConnection*, *getSoftware*, *getCPU*, *getToner*, *getPaper*, *determineMaterial*, *orderPaper*, *orderToner*, *orderCPU*, *orderHardware*, *receiveOrderStatusSupplier*}. ;

– For each task, we identify the preconditions to be performed.

1. The precondition of the *orderPaper*, *orderToner*, *orderCPU*, *orderHardware*, *orderSoftware*, and *orderMaterial* tasks is that the supplier is connected (*supplierConnected*) ;
2. The *receiveOrderStatusSupplier* and *receiveOrderStatus* tasks, have no preconditions to be verified.

At this stage, the only possible source of failure that could make the system fails is that the supplier is not connected or could not deliver the ordered goods. In order

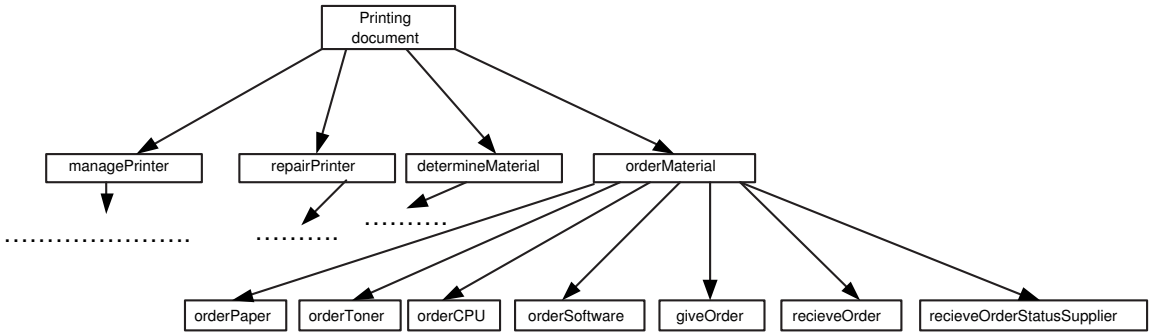

FIG. 4.5 – Task hierarchy at iteration 4.

to overcome this source of failure, there are no other tasks that can be added to the system to control this source since the supplier is out of control of the system. Hence, the company should have several suppliers to overcome this source of failure.

Now, there are no new tasks that can be added to the system, there is no possible iterations over the *task-environment* model. Consequently, we move on to the design phase.

4.4.2 The Design Phase

In the design phase, we develop the agent model and the agent interaction model, and we apply the reorganization technique, described in Chapter 2, to determine which tasks and agents should be replicated

The Agent Model

As stated in Chapter 3, we identify the types of agents by considering *groups* of tasks, each group being a set of logically related tasks, bound by the same constraints or goals. In this example, we identify four groups of tasks that can be logically grouped together. The first group has to manage the printer, rendering the services and maintaining its status. The second group has to repair a printer. The third group has to provide the required materials to repair a printer. The fourth group has to ask a supplier to provide the missing materials. Hence, we could define a type of agent for each of these groups. Thus, there will be four types of agents in the our example :

1. – Agent name : *PrinterManagerAgent*

- Description : this agent prints documents by sending them to a printer, and monitors the printer to detect any default in it
- Resources : printer1, printer2, printer3.

2. – Agent name : *PrinterRepairAgent*
   - Description : this agent repairs the problems identified by the *PrinterManagerAgent*
   - Resources : printer1, printer2, printer3.

3. – Agent name : *OrderPickerAgent*
   - Description : this agent checks whether the material needed to repair a printer is available
   - Resources : DbAvailable.

4. – Agent name : *OrderPlaceAgent*
   - Description : this agent orders the needed material from the company's suppliers
   - Resources : Supplier.

The set Λ of agents is : Λ={*PrinterManagerAgent*, *PrinterRepairAgent*, *OrderPickerAgent*, *OrderPlaceAgent*}.

Each agent can be related to other agents according to their interactions. In the agent model of the document printing example (see Figure 4.6), agents are related by association relations as defined in [41]. Each *PrinterManagerAgent* can collaborate with another *PrinterManagerAgent*. Hence, the cardinality of that relation is (0..*) on both sides oof *PrinterManagerAgent*. The *PrinterRepairAgent* can communicate with several *PrinterManagerAgent* or not. Hence, the cardinality of that relation is (0..*) on the *PrinterManagerAgent* side. Nevertheless, the *PrinterManagerAgent* can communicate with the *PrinterRepairAgent*. So, the cardinality of that relation is (0..1) in the *PrinterRepairAgent* side. The associations between the *PrinterRepairAgent* and the *OrderPickerAgent*, and the *OrderPickerAgent* and the *OrderPlaceAgent* are similar to the association between *PrinterManagerAgent* and *PrinterRepairAgent*. The agent model is presented in Figure 4.6.

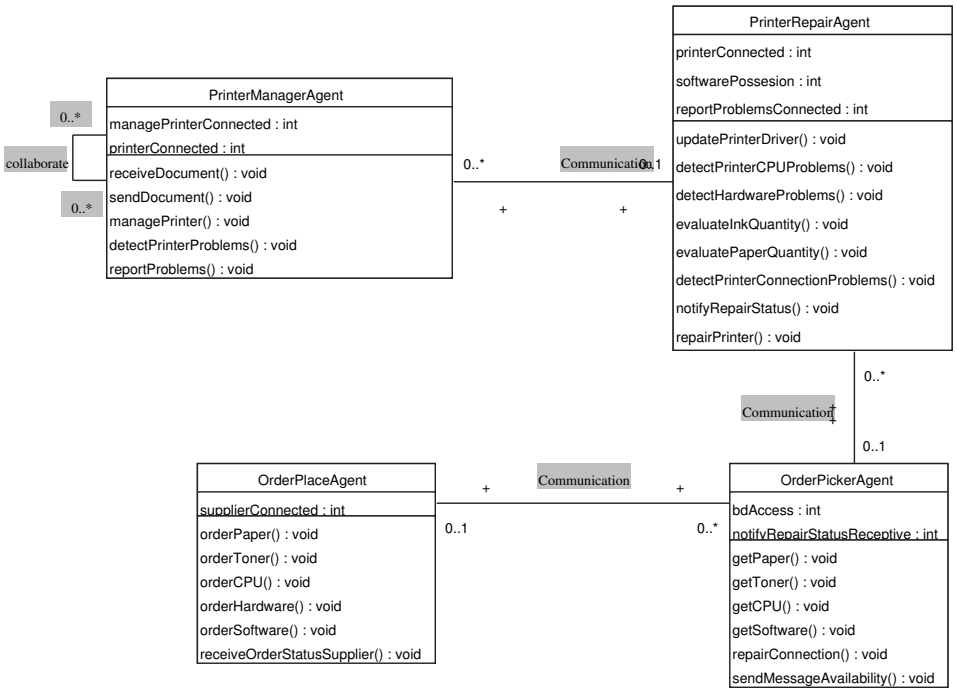

FIG. 4.6 – Agent model at Iteration 4

The Agent Interaction Model

We determine the different calls between the tasks. For example, in case the PrinterRepairAgent identifies a failure in the printer, it asks the OrderPickerAgent to check for the availability of the required material in order to repair the printer. Hence, the evaluateInkQuantity() task calls the bringToner() task. Consequently, call(evaluateInkQuantity(), bringToner())=true. We can determine the different calls between the tasks.

Based on the call function, the PrinterRepairAgent communicates with the OrderPickerAgent. Consequently, communication(PrinterRepairAgent, OrderPickerAgent) = true. In the same manner, communication (PrinterManagerAgent, PrinterRepairAgent) = true, and communication(OrderPickerAgent, OrderPlaceAgent) = true. Based on the communication and the call functions, we can deduce the following collaboration diagram of Figure 4.7.

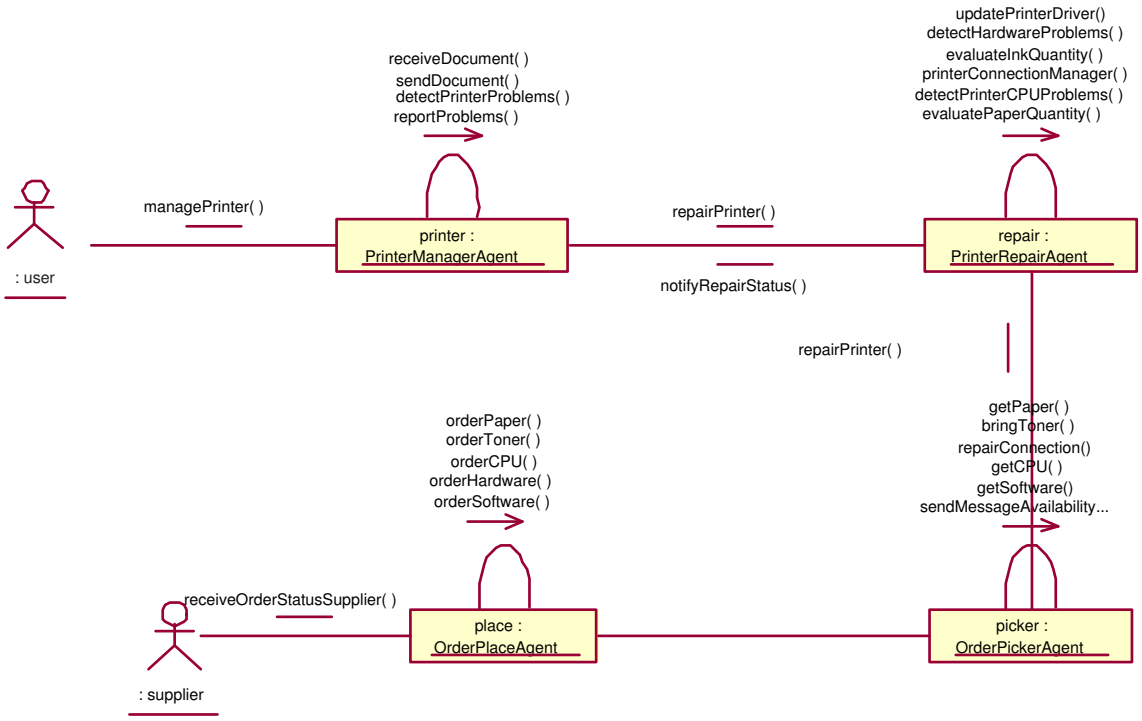

FIG. 4.7 – Collaboration model at Iteration 4.

The Reorganization Technique

As stated in Chapter 3, the reorganization technique described in Chapter 2 includes a fault-prevention technique, a fault-recovery technique, and a fault-tolerance technique. The fault-prevention technique is dealt with in the task-environment model. The fault-recovery technique has been proofed valid in Chapter 2. In this section, we determine, based on the fault-tolerance technique, the agents which should be replicated. Based on the agents' *required* resources, we deduce that :

- Any task of the *PrinterManagerAgent* can be replicated in the *PrinterRepairAgent*, and vice-versa, since the two agents have the same required resources ;
- The tasks of the *OrderPickerAgent* cannot be replicated in any other agent of the system since none of these agents have access to the database. So, the *OrderPickerAgent* should be replicated.
- The tasks of the *OrderPlaceAgent* cannot be replicated in any other agent of the system since none of these agents have access to the suppliers. So, the *OrderPlaceAgent* should be replicated.

In that case, the shared memory of *PrinterManagerAgent* is shared with *PrinterRepairAgent*. The shared memory of *PrinterRepairAgent* is shared with *PrinterManagerAgent*. The shared memory of *OrderPickerAgent* is shared with its replica. The shared memory of *OrderPlaceAgent* is shared with its replica.

Only two types of agents will be replicated instead of four, which reduces the number of agent instances that will operate in the system. Hence, the system will not be loaded with replicated agents. The new agent model is presented in Figure 4.8.

4.4.3 The Implementation Phase

The implementation phase has only one model to develop which is the implementation model.

The Implementation Model

The *implementation model* determines the physical architecture of the system and assists the code generation phase of the system. The physical architecture is provided as a deployment diagram as defined in [53]. There are four types of agents in the system. The *PrinterManagerAgent* which operates in the end-user machine since it receives the document to print from the end-user. The *PrinterRepairAgent* which operates in the server to which the printer to repair is connected since the end-user has not to repair the printer. The *OrderPickerAgent* which operates also in the server to which the printer is connected since it has to look if the needed material to the printer is available. Finally, the *OrderPlaceAgent* which operates in the server connected to the suppliers since it orders the material to one of the available suppliers.

From this distribution, we know that PrinterManagerAgent and PrinterRepairAgent are not running on the same physical components. However, repl_OrderPickerAgent and repl_OrderPlaceAgent are not assigned to a physical component. These two agents and PrinterRepairAgent can be grouped together since they do not replicate each other. Consequently, they can run on the same physical component. In this case, the server to which the printer to repair is connected must have access to database warehouse stocks and to the suppliers which are required by these two replica agents.

The deployment diagram is presented in Figure 4.9.

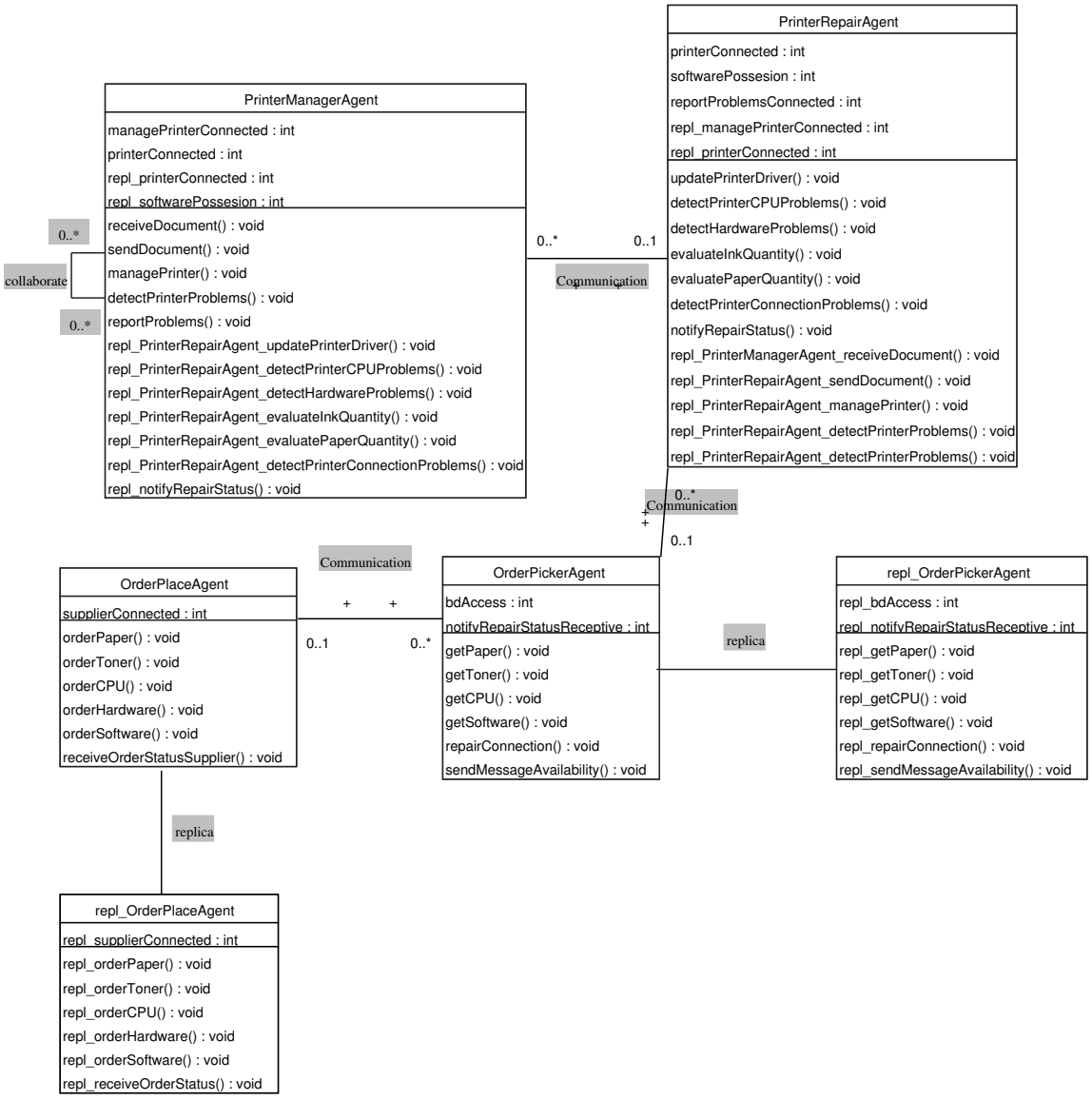

FIG. 4.8 – Agent model after introducing redundancy.

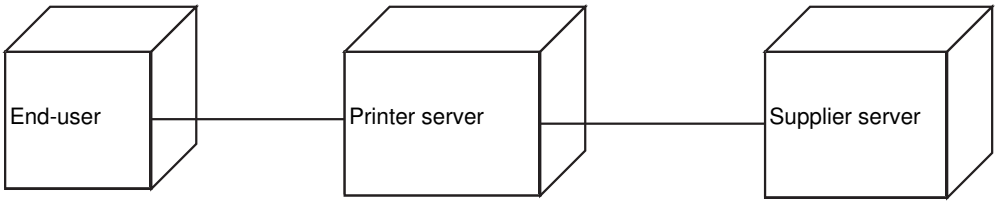

FIG. 4.9 – The deployment diagram of the printing system.

The second step in the *implementation model* is the code generation. The code of PrinterManagerAgent is presented in Figure 4.10. The codes of the remaining agents is presented in Appendix A.

4.5 Conclusion

In this chapter, we applied *FATMAS* to design a fault-tolerant multi-agent system which manages different printers used in a company. We have demonstrated how the iterations of *FATMAS* has allowed us to identify different tasks, and thus agents, which will operate in the system and add fault-tolerant capabilities to the system. These identified tasks and agents would not be identified by other existing methodologies since they do not integrate a fault-recovery approach.

```
public class PrinterManagerAgent {
  /* {src_lang=Java}*/


  public int managePrinterConnected;
  /* {transient=false, volatile=false}*/

  public int printerConnected;
  /* {transient=false, volatile=false}*/

  public int repl_printerConnected;
  /* {transient=false, volatile=false}*/

  public int repl_softwarePossesion;
  /* {transient=false, volatile=false}*/

    public PrinterRepairAgent
myPrinterRepairAgent;

  public void receiveDocument() {
  }

  public void sendDocument() {
  }

  public void managePrinter() {
  }

  public void detectPrinterProblems() {
  }

  public void reportProblems() {
  }

  public void
repl_PrinterRepairAgent_updatePrinterDriver() {
  }

  public void
repl_PrinterRepairAgent_detectPrinterCPUProblem
s() {
  }

  public void
repl_PrinterRepairAgent_detectHardwareProblems(
) {
  }

  public void
repl_PrinterRepairAgent_evaluateInkQuantity() {
  }

  public void
repl_PrinterRepairAgent_evaluatePaperQuantity()
{
  }

  public void
repl_PrinterRepairAgent_detectPrinterConnection
Problems() {
  }

  public void repl_notifyRepairStatus() {
  }
}
```

FIG. 4.10 – The code of PrinterManagerAgent.

Chapitre 5

Related Work On Agent-Oriented Software Engineering Techniques

In this chapter, we present an overview of several *MAS* design methodologies : (*MAS-CommonKADS* [24], *GAIA* [57], *SODA* [43], *AALAADIN* [17], *Adelf* [2], *SABPO* [14], *MESSAGE/UML* [7], *BDI agents* [27], *Tropos*, and Prometheus [44]. For each methodology, we first present its delivered proposed models. Then, we discuss which of these models can be added to FATMAS and their impacts on the fault-tolerance aspects of FATMAS.

5.1 The MAS-CommonKADS Methodology

MAS-CommonKADS[24] is a multi-agent system design methodology based on the *CommonKADS* [23][50] methodology and uses techniques and models borrowed from object-oriented methodologies. It has three phases : the conceptualization phase, the analysis phase and the design phase. *MAS-CommonKADS* produces the same models as *CommonKADS*, but adapted in order to deal with multi-agent systems.

5.1.1 The Conceptualization Phase

The conceptualization phase helps developers to understand the problems to be solved. The main outputs of this phase is a *use case* diagrams [53].

5.1.2 The Analysis Phase

The second phase is analysis. It carries out a requirement specification of the *MAS* through the development of the following five models :

1. The *agent model* consists in identifying and describing the agents ;
2. The *task model* consists in a task decomposition. For each task, it specifies the task's name, task's inputs and outputs, and task's preconditions ;
3. The *coordination model* consists in describing the interactions and coordination protocols for the agents. It shows the dynamic relationships between the agents ;
4. The *expertise model* is used to model the agents' reasoning capabilities to carry out their tasks and achieve their goals. It consists in determining the application knowledge model and the problem solving knowledge. The application knowledge model consists of the domain knowledge, the inference knowledge and the task knowledge. The domain knowledge represents the declarative knowledge of the problem modelled as concepts, properties, expressions and relationships. The inference knowledge represents the inference steps performed to carry out a task. The task knowledge represents the plans to perform the inference rules. The problem solving knowledge specifies how the inference is carried out ;
5. The *organization model* represents the organization in which the *MAS* will be introduced and the software organization of the *MAS*. It shows the static or structural relationships between the agents. This model is the specification of the structural relationships between human and/or software agents, and the relationships with the environment.

5.1.3 The Design Phase

The third phase is design. It carries out the *agent network design*, the *agent design* and the *platform design*.

1. The *agent network design* determines the infrastructure upon which the *MAS* will be deployed. It determines the network structure that agents will use to communicate ;
2. The *agent design* consists of determining the most suitable architecture for each agent ;
3. The *platform design* is the selection of the needed software and hardware to implement the *MAS*.

5.1.4 Comparison With FATMAS

There are several commonalities and differences between FATMAS and Mas-CommonKADS models. The use case and the expertise models can be considered as the main difference between FATMAS and MAS-CommonKADS. The expertise model details the agents structure. It can be used in FATMAS without affecting the fault-recovery technique. Nevertheless, MAS-CommonKADS uses a use-case driven approach to understand the problem to be solved by the system. In FATMAS, we use rather an event-driven approach to delimit the problem to be solved. A use-case driven approach allows the designer to determine the actors of the future system and the functionalities that will be provided. It does not support the designer to determine the system's boundary. However, in FATMAS, the sources of failures are identified within the events that trigger the system functioning. Consequently, the event-driven approach used by FATMAS is more appropriate to design a fault-tolerant system than using a use-case driven approach.

Looking at the remaining models of MAS-CommonKADS, we can map them to models produced by FATMAS. In fact the design phase of MAS-CommonKADS is equivalent to the implementation phase of FATMAS since the two phases determine the infrastructure upon which the MAS will be deployed. Moreover, the coordination model of MAS-CommonKADS is equivalent to the agent interaction mode of FATMAS in the sense that they model agents interactions. The agent model of MAS-CommonKADS is equivalent to the agent model of FATMAS. And finally, the task model of MAS-CommonKADS is included in the task-environment model of FATMAS since in FATMAS the task-environment model determines the resources required by the different tasks and also determines the future system's boundary.

5.2 The Gaia Methodology

The *Gaia*[57] methodology is applicable to a wide range of multi-agent systems in which agents are cooperative and the system is open. It is composed of two main phases : the analysis phase and the design phase.

5.2.1 The Analysis Phase

The objective of the analysis phase is to understand the system and its structure. To this end, four models are proposed : the *environment model*, the *preliminary roles model*,

the *preliminary interaction model*, and a *set of organization rules*. The *environment model* represents the environment in which the *MAS* will be located. The *preliminary role model* identifies the basic functionalities required to define the *MAS*. The *preliminary interaction model* identifies the basic interactions between roles required to perform the preliminary roles. The *set of organization rules* contains the rules that the organization should respect and enforce in its global behaviour.

5.2.2 The Design Phase

The objective of the design phase is to transform the analysis models into sufficiently low abstraction levels in order to implement the *MAS*. The design phase includes an *architectural design step* and a *detailed design step*. The *architectural design step* includes the definition of the system's organizational structure, and the completion of the preliminary roles and interaction models.

In the *detailed design step*, the designer specifies the *agent model* in which the agent classes are identified. The agent classes will make up the system and the agent instances that will be instantiated from these classes. This *detailed design step* also covers the definition of the *service model* which identifies the main services (tasks) required to achieve the agent's roles.

5.2.3 Comparison With FATMAS

FATMAS and Gaia have similarities and differences in the models they produce. The main difference between Gaia and FATMAS is the starting point to design the MAS. Gaia provides an organization view of the MAS. It focuses on the roles the agents will enroll and then determines the tasks to be enacted by agents. FATMAS does not focus on the role concept. However, the different tasks identified in the task-environment model of FATMAS can be grouped to define roles. Consequently, the role concept can be added to FATMAS. However, there should be new definitions to be added in FATMAS in order to integrate the role concept with the different other concepts of FATMAS. This could influence the reorganization technique since this technique refers to the agent and task concepts. If the role concept is added to FATMAS, then the reorganization technique must take into account this concept.

However, there are similarities between Gaia and FATMAS models. Precisely, the services model of Gaia is a sub-model of the FATMAS' task-environment model since

the service model only determines the tasks to be performed by agents but do not determine the preconditions or the requested resources of the tasks. The interaction model of Gaia can be used in FATMAS to allow a more detailed representation of the interaction protocols between agents.

5.3 SODA

The *SODA*[43] methodology is suited for the development of internet-based systems. It consists of two phases, the analysis phase and the design phase. During the analysis phase, the application domain is studied and modeled, the available resources and the technological constraints are listed, and the fundamental application goals and targets are pointed out. The design phase deals with the representation of the abstract models obtained during the analysis phase.

5.3.1 The Analysis Phase

The analysis phase produces three models that are the *role model*, the *resource model*, and the *interaction model*. In the *role model*, the tasks are expressed according to the responsibilities they involve, the competencies they require, and the resources they depend upon. The tasks are classified as either individual or social. Each individual task is associated with an individual role. A role is defined according to the responsibilities. Social tasks are assigned to groups. A social role describes the role played by an individual in a group.

In the *resource model*, the services express the functionalities provided by the agent environment to the multi-agent system such as querying a sensor and verifying an identity. Each service is associated with an abstract resource which is defined according to the services it provides. Each resource defines abstract access modes (permissions), modeling the different ways in which the corresponding service can be exploited by agents.

The *interaction model* presents interactions involving roles, groups and resources according to the interaction protocols. An interaction protocol associated with a role is defined according to the information required and provided by the role in order to perform its individual task. An interaction protocol associated with a resource is defined according to the information required to invoke the service provided by the resource

itself, and by the information returned when the invoked service has been brought to an end, either successfully or not. An interaction protocol associated with a group governs the interactions between social roles and resources in order to enable the group to perform its social tasks.

5.3.2 The Design Phase

The design phase is based on three models : the *agent model*, the *society model*, and the *environment model*. In the *agent model*, an agent class is defined as a set of one or several roles. It is characterized by the tasks, the set of permissions, and the interaction protocols associated with its roles. In the *society model*, each group is mapped onto a society of agents. An agent society is characterized by the social tasks, the set of permissions, the participating social roles, and the interaction rules associated with its groups. In *the environment model*, resources are mapped onto infrastructures classes.

5.3.3 Comparison With FATMAS

As with Gaia, SODA focusses on the social structure of a multi-agent system. The main differences between SODA and FATMAS are the concepts of role, group, or society which are central in SODA. But, as stated for Gaia, these concepts can be integrated in FATMAS if they are formally defined and well integrated with the other concepts of FATMAS. Furthermore, the concepts should be taken into account in FATMAS reorganization technique since they add new abstraction levels to the system.

However, there are similarities between SODA and FATMAS. The role model and the resource model of SODA can be compared to the task-environment model of FATMAS. Omitting the fact that the role model refers to the role concept, it includes a task identification as in the FATMAS' task-environment model. Furthermore, the resource model identifies the different resources to be used in the system which is done in the task-environment model of FATMAS. However, the interaction model in SODA includes the interaction between groups and roles. In the case of FATMAS, the notion of group is used only to group agents and not to form a social entity. And the interaction model only represents the interactions between agents.

The society model can be used in FATMAS if the different concepts of role, group, and society are formally defined and integrated with the different other concepts of FATMAS. The agent model of SODA is equivalent to the agent model of FATMAS.

Finally, the environment model of SODA is equivalent to the implementation model of FATMAS.

5.4 The *AALAADIN* Meta-methodology

AALAADIN[17] is a generic meta-model for multi-agent systems. The core concepts of *AALAADIN* are roles and groups. A group is defined as a set of agents. A role is defined as an abstract representation of an agent function, a service or an identification within a group. In *AALAADIN*, the agents are defined by their functions in an organization, that is by their roles and the set of constraints which they must accept in order to be able to play these roles. Agents can play different roles in different groups. *AALAADIN*'s methodological approach consists in determining first the group structure by identifying all the roles and interactions that can appear in a group, and secondly the *MAS* organizational structure, that is the set of group structures expressing the design of a multi-agent organization scheme.

5.4.1 Comparison With FATMAS

As presented, AALAADIN provides the concepts upon which a multi-agent system is built. It does not propose a set of diagrams. Since AALAADIN is based on the role and group concepts, it cannot be easily integrated in FATMAS. As state earlier, these concepts can be integrated in FATMAS if they are formally defined and integrated with the other concepts of FATMAS. Furthermore, the concepts should be taken into account in FATMAS reorganization technique since they add new abstraction levels to the system.

5.5 *ADELFE*, a Methodology for Adaptive Multi-agent Systems Engineering

ADELFE[2] is suited to adaptive multi-agent systems in which the environment is unpredictable and the system is open. A strong adaptation is the ability that the system must possess in order to take into account unpredictable events and to react to evolutionary environments. The *MAS* is developed to react to agent cooperation failures.

It is developed to consider new inputs from the environment that could make agent interactions change. In adaptive multi-agent systems, the agents are involved in cooperative interactions. ADELFE proposes three workflows : the requirements workflow, the analysis workflow and the design workflow.

5.5.1 The Requirements Workflow

In the requirements workflow, *ADELFE* provides a model composed of the target system, and the *system environment*. This workflow focuses on what may be in interaction with the studied system in terms of passive or active entities or constraints. It requires a characterization of data flows and interactions between passive or active entities and the system. These interactions are expressed by *UML's collaboration and sequence diagrams*[53].

5.5.2 The Analysis Workflow

In the analysis workflow, *ADELF* proposes to first identify the agents by performing a domain analysis to produce a *preliminary class diagram*. Each agent has to be analyzed as a system. Secondly, it proposes to study the interactions between the different entities as a set of *sequence* (like in *AUML*[41]) and *activity diagrams* which explain the possible interactions between the different entities within each level of the system.

5.5.3 The Design Workflow

In the design workflow, *ADELFE* defines the *agent model* and the *Non-Cooperative Situations model* (NCS) which can be thought of as exceptions in classical programs. The *agent model* represents the relationships between agents. The *NCS model* deals with the non-cooperative situations in which the multi-agent system cannot reach its objectives. In addition, the design phase produces the architecture of the system in terms of blocks, classes, agents and interactions.

ADELFE considers only NCS in which agent interactions may change. It does not deal with NCS in which agents are no longer available or in which tasks cannot be performed.

5.5.4 Comparison With FATMAS

ADELFE is a methodology used to design adaptive multi-agent systems. As stated earlier, adaptive multi-agents are systems in which agents change their behavior or interactions in order to overcome a non-common situation. ADELFE builds cooperative multi-agent systems. In case of a non-cooperative situation, agents change their interactions. It is another way to deal with fault-tolerance.

The requirements workflow of ADELFE characterizes the environment in which the system will be placed by identifying the active or passive entities and the constraints under which the system operates. These constraints can be seen as the delimitation of the environment of the MAS as done in FATMAS. Moreover, the passive or active entities of the environment can be seen as the resources that the agents will use. Hence, the requirements workflow of ADELFE can be used as a complementary model to produce the task-environment model of FATMAS.

The preliminary class diagram and the agent model of ADELFE are equivalent to the agent model of FATMAS. The sequence diagram and the activity diagram are equivalent to the agent interaction model of FATMAS. ADELFE does not have an implementation workflow to design the infrastructure upon which the system will operate. Moreover, it does not include a fault-recovery technique due to component failures.

5.6 *SABPO* : a Standard Based and Pattern Oriented Multi-Agent Development Methodology

A *MAS* behaves like a social organization in which each agent plays a specific role. The *Foundation for Intelligent Physical Agents* (*FIPA*) [19] standards define the services required to build MASs working in open environments and define interaction patterns in order to build robust organizational structures. *SABPO* takes the *FIPA* standards as a basis[14].

In *FIPA*-based agent systems, agent interactions are specified using the pre-defined *FIPA* interaction protocols. *SABPO*[14] tries to identify the required interaction protocols based on the system requirements during the analysis phase. The approach is composed of an analysis phase and a design phase.

5.6.1 The Analysis Phase

In the analysis phase, the following models are developed : the *role model* and the *interaction model*. The *role model* identifies the roles that should be enrolled by the *MAS*' future agents and their responsibilities in order to satisfy the organization's global goals. *SABPO* introduces two roles to comply with the *FIPA*'s abstract architecture. These roles are 'Directory Service Provider' and 'Ontology Service Provider'. The *interaction model* defines the interaction protocols between agents. These interactions are documented using *AUML*[41].

5.6.2 The Design Phase

In the design phase, three models are developed : the *ontology model*, the *agent model* and the *detailed interaction model*. The *ontology model* extends the knowledge obtained during the analysis phase. The *agent model* specifies the agent types and assigns to the agent types the roles defined in the analysis phase. The *detailed interaction model* maps the interaction protocols identified in the analysis phase to the FIPA specifications. In fact, the *FIPA* specifications define an interaction protocol which allows one agent to request another to perform some action.

5.6.3 Comparison With FATMAS

SABPO is a methodology that tries to determine the interaction protocols between agents. The core of SABPO is its analysis phase which determines the role model and the interaction model. Hence, the core concept of SABPO is the role. As stated earlier, this concept can be integrated in FATMAS if it is formally defined and integrated with the other concepts of FATMAS. Furthermore, this concept should be taken into account in FATMAS reorganization technique since it adds new abstraction levels to the system. However, SABPO can be integrated with FATMAS in order to defined FIPA compliant protocols to deal with agent interactions.

5.7 Agent Oriented Analysis Using Message/UML

MESSAGE[7] stands for : Methodology for Engineering Systems of Software AGEnts. It proposes five model views : organization view, goal/task view, agent/role view, interaction view and domain view.

The organization view (OV) shows concrete entities (agents, organizations, roles, resources) of the system and its environment and coarse grained relationships between them (aggregation, power, and acquaintance relationships). The goal/task view (GTV) shows the goals, tasks, situations and dependencies between them. The agent/role view (ARV) focuses on the individual agents and roles. In the interaction view (IV) a designer must, for each interaction between agents/roles, show the initiator, the collaborators, the motivator, the relevant information supplied/achieved by each participant, the events that trigger the interaction, and other relevant effects of the interaction. The domain view (DV) shows the domain specific concepts and relations that are relevant for the system under development.

5.7.1 Comparison With FATMAS

Message/UML is a methodology that produces five model views to design a multi-agent system. This methodology views the MAS as an organization which is not the view of FATMAS. Hence, this view cannot be easily integrated in FATMAS. The goal/task view allows to build the MAS by a goal-driven and task-driven approach, which is the same optic of FATMAS. Hence, this view could be integrated in FATMAS within the task-environment model. The interaction view is equivalent to the agent interaction model of FATMAS and could be integrated in FATMAS. Each of the models that can be integrated in FATMAS has no influence on the performance of the reorganization technique.

5.8 Agent Modelling Techniques for Systems of BDI Agents

This methodology develops BDI agents and is based on object-oriented technologies [26] [27]. This methodology distinguishes between the external aspects of the system and its internal aspects. The external aspects of the system determines the different

components which will operate in the system and the internal aspects of the system describes the architecture of the different components operating in the system.

The external aspects are characterized by two models : the agent model and the interaction model. The agent model determines the agent classes and their instances. The agents are identified after a refinement process on role identification and description. The interaction model describes the interactions between agents. The internal aspects are characterized by three models which represent the mental structure of the agents : the belief model, the goal model, and the plan model. The belief model describes the information about the environment and the internal state that an agent may hold, and the actions it may perform. The goal model describes the goal that an agent may possibly adopt and the event to which it can respond. The plan model describes the plans that an agent may possibly employ to achieve its goals.

5.8.1 Comparison With FATMAS

While this methodology is based on role concept to identify its agents, it proposes models which could be integrated within FATMAS without affecting its fault recovery approach. The agent model of FATMAS can be augmented by the identification and description of the instances which will operate in the system. Moreover, the interaction model is equivalent to the interaction model of FATMAS. Furthermore, the description of agent architectures can improve the agent description. The reorganization technique presented in FATMAS is independent from the agent architecture. Consequently, this methodology based on BDI agents can be easily integrated with FATMAS.

5.9 Tropos

The Tropos methodology[20] is based on key features that are agents, goals, and plans. The phases of the methodology are early requirements, late requirements, architectural design, detailed design, and implementation. The early requirement phase identifies actors and their goals. The late requirements introduces the system-to-be as an actor that interacts with other actors. In the architectural design more system actors are introduced and are assigned sub-goals. The detailed design defines the system actors in further details, including specifications of communication and coordination protocols. The implementation transforms the system to the JACK platform[6].

Tropos proposes fours models : the actor and dependency model, the goal and plans model, the capability diagram, and the agent interaction model. The actor and dependency model results from the analysis of social and system actors. An actor represents a physical agent, a software agent, a role, or a set of roles. The goal and plans model determine the differen goals the actor may achieve and the plans to achieve them. The capability diagram is an UML activity diagram from an agent's point of view. It represents the ability of an actor to define, choose, and excecute a plan to fulfill a goal. The interaction diagram is an AUML sequence diagram [41].

5.9.1 Comparison With FATMAS

Tropos is a methodology based on the extension of object-oriented methodologies. While Tropos refers to the role concept, it offers a set of models which can be easily integrated in FATMAS. The actor and dependency model can be integrated in FATMAS. As stated earlier, these concepts can be integrated in FATMAS if they are formally defined and integrated with the other concepts of FATMAS. Furthermore, the concepts should be taken into account in FATMAS reorganization technique since they add new abstraction levels to the system. Moreover, the goal and plan models provide a more detailed description of the agent's architecture. This can be a valuable addition to the agent diagram of FATMAS and improves its agents' descriptions. Also, the capability diagram can be integrated in FATMAS to detail how the agents will achieve their plans according to the request of services they receive from the environment. Finally, the agent interaction diagrams of Tropos are equivalent to the agent interaction diagram of FATMAS.

5.10 Prometheus

Prometheus [44] is an iterative methodology which aims at developing intelligent agents using goals, beliefs, plans, and events. Prometheus covers three phases which are the system specification, the architectural design, and the detailed design. The system specification identifies the basic functions of the system, along with inputs, outputs, and their processing. Prometheus proposes to adopt a use case view in order to determine these three elements. The architectural design determines which agents the system will contain and how they will interact. The architectural design produces two diagrams which are the system overview diagram (equivalent to the agent diagram of FATMAS) and the interaction protocols which are specified as interactions in AUML [41]. The

detailed design describes the internals of each agent and they way it will achieve its tasks. This phase outputs the agent overview diagram providing the agent's top level capabilities, the capabilities diagrams, the detailed plan descriptions so that agents perform their tasks, and the data descriptions describing the beliefs with which agents will operate.

5.10.1 Comparison With FATMAS

Prometheus can be easily integrated with FATMAS. In fact, the architectural design of Prometheus determines which agents the system will contain and how they will interact. This architectural design is equivalent to the design phase of FATMAS. Moreover, the system overview diagram is equivalent to the agent diagram of FATMAS. Furthermore, the agent overview diagram, the capabilities diagrams, the detailed plan descriptions, and the data descriptions can be integrated in FATMAS since they enrich the agent descriptions while they do not affect the reorganization technique of FATMAS.

5.11 Conclusion

In this chapter, we presented several methodologies to design multi-agent systems. First, we notice that none of these methodologies allows to design fault-tolerant multi-agent system since they do not integrate any of the techniques required to build fault-tolerant system. Nevertheless, each of these methodologies propose to develop a set of models or diagrams to design multi-agent system. This chapter, identified for each of the studied methodologies, their models or diagrams which can be integrated in FATMAS.

FATMAS is not a closed methodology. It is built in such a way that it can integrate other models or diagrams taken from other methodologies. FATMAS also is not built upon a large number of concepts and its reorganization technique is described in such a way that it does not refer to any system or agent architectures. Consequently, any other MAS design methodology can easily have some of its model or diagrams be integrated in FATMAS. This makes FATMAS open to integrate other methodologies' diagram or models.

Chapitre 6

Related Work on Fault-tolerant Systems

In Chapter 3, we proposed a methodology to design fault-tolerant multi-agent systems. This methodology integrates a reorganization technique based on task and agent replication which was described in Chapter 2. As demonstrated in Chapter 2, this technique allows the system to detect certain failures and to overcome some, while minimizing the system complexity. It focuses on failures originating from the system boundary or the system' environment. It does not focus on failures originating from programming exceptions.

In this chapter, we present several fault-tolerant techniques developed in multi-agent systems, and compare them to our technique. This comparison will not be based on a performance criteria since the presented techniques and the reorganization technique of FATMAS do not address the same kind of failures. However, this comparison will allow us to have an idea on the capacity of each technique to take into account criteria which are important to build fault-tolerant systems.

6.1 The Comparison Criteria

In this section, we provide qualitative criteria on which we compare our recovery technique with others. The other techniques are either based on redundancy or using adaptive MAS or using a fault-recovery approach at the infrastructure level. These criteria are defined based on the basic concepts of fault-recovery. A fault-recovery tech-

nique allows to prevent from faults by identifying different sources of failures. Hence, the first comparison criteria is whether a fault identification is provided. Moreover, a fault-recovery technique allows a system to detect and recover from its failures. Hence, two comparison criteria can be taken into account which are whether the technique allows the system to detect its faults, and whether it allows the system to recover from them.

Also, fault-tolerance is based on redundancy. Hence, the third comparison criterion takes into account is the kind of redundancy used by each fault-recovery technique. In addition, each element is replicated several times. Thus, the fourth comparison criterion is whether the number of replica per elements to be replicated is limited. Finally, each fault-recovery technique can use either a backward recovery or a forward recovery mechanism (as explained in Section 3 of Chapter 2). Hence, the final comparison criteria makes that distinction.

To summarize, the comparison criteria are :

1. fault identification : to state whether there are sources of faults identified while designing the system ;
2. fault-detection : to state whether the system is able to detect its faults ;
3. fault-recovery : to state whether the system can recover from its failures ;
4. kind of redundancy : to state which kind of redundancy is applied ;
5. number of replica : to determine the number of replica per replicated elements ;
6. kind of recovery : to determine whether the system uses a backward or a forward recovery technique or a mix of both.

In what follows, we present different approaches to build fault-tolerant multi-agent systems. These approaches are used in adaptive MAS [21], broker systems [29], mobile agents [39], autonomous robots [18] and computational grid [45]. These approaches will be compared with our approach using the criteria that we selected.

6.2 DARX Technique

The first technique we present is the *DARX* framework [21] used to develop fault-tolerant multi-agent systems. It is based on data and/or computation replication. *DARX* allows to automatically and dynamically apply replication mechanisms to agents. It only replicates *critical agents*. An agent is considered as *critical* during the *MAS* design phase in which the designer determines which agents are critical and should be replicated by the programmer before runtime, or at runtime by identifying, for each agent, the

criticality of the role it is enacting. The roles and their criticality are assessed during system design. Since each agent can change roles during the *MAS* operation, the *DARX* framework uses a role recognition method to determine which role each agent is enacting. Hence, as soon as an agent enacts a critical role, it is replicated (the number of replica is not determined by *DARX*). If an agent is no longer considered as critical, then all its replicas are removed from the system. Each replicated agent acts as a group leader for its replicas. The replicated agent is responsible for broadcasting its state to its replicas. When the agent fails, it is replaced by one replica, which is chosen according to a predefined strategy. *DARX* has the advantage of dynamically adapting the number of replicas to reduce the system complexity.

The *DARX* framework uses a backward recovering strategy since the replica agent starts performing from the last broadcasted state. However, if the last agent's state is not an error-free state, then the replica agent will also be in failure. Hence, the *DARX* framework does not provide a control mechanism to ensure that the replica will perform correctly. Moreover, the *DARX* framework provides a strategy to reduce the number of agents to be replicated. However, each agent can be replicated an unlimited number of times, which is not always possible due to resource limitation.

Based on the comparison criteria, we find that :

- Fault identification : there are no sources of faults identified while designing the system ;
- fault-detection : known ;
- fault-recovery : known ;
- kind of redundancy : agent replication ;
- number of replica : the number of replica is not provided ;
- kind of recovery : backward recovery.

6.3 Brokered Multi-agent Systems

Multi-agent systems often require brokers to accept requests, route requests and responses, manage the system, and for various other facilitation tasks [29]. However, these systems are prone to broker failures. In fact, a multi-agent system depending on brokers, can become unavailable if one or more system's brokers are inaccessible due to failures such as machine crashes, network breakdown, etc. In [29], a recovering broker failure technique is presented. The technique is based on broker replication. It is applied when there are several broker agents in a multi-agent system [29]. These broker agents

may be able to substitute for any broker agent that becomes unavailable. Hence, the multi-agent system can continue to operate as long as there is at least one broker agent remaining in the broker team. The agents, that were communicating with the failed broker, will subscribe with a new broker, and will restart their communication.

This technique uses a forward recovery approach. The system is moved from a failed state to an error-free state. The system does not continue operating from the last error-free state. In case of failure, agents restart their communications. This technique covers neither fault prevention nor fault forecasting. It does not restrain either the number of brokers to be replicated, or the number of replicas per broker.

Based on the comparison criteria, we find that :

- Fault identification : there are no sources of faults identified while designing the system ;
- fault-detection : unknown ;
- fault-recovery : known ;
- kind of redundancy : agent replication ;
- number of replica : unlimited since brokers can create other brokers ;
- kind of recovery : forward recovery.

6.4 Mobile Agents

Mobile agents can travel from one server to another to look for information or to perform tasks in the visited servers. A multi-agent system, composed of mobile agents, can be prone to two kinds of failures [39] : server failures and agent failures. We present, in the next subsections, techniques that have been proposed for dealing with both types of failures.

6.4.1 SG-ARP : a Server Recovering Approach

The Server Group based Agent Recovery Protocol (*SG-ARP*) approach is described in [39]. It enables mobile agents to perform properly despite server failures.

In order to overcome a server failure, each server is replicated several times. The server and its replicas define a server group. The members of a server group divide among

themselves the load brought upon by the visiting agents. The different servers share a storage area in which they store their states. When a server is down, all the agents that were running on the failed server are distributed to the remaining group members. Since the system uses a backward recovery approach, there could be information lost when the agents are distributed to the remaining servers.

This approach uses backward recovery since servers share their states. When a server fails, the system is backed to a free-error state so that agents can perform on other existing servers. However, it does not specify how many times a server is replicated. In addition, this technique does not cover fault prevention and fault forecasting.

Based on the comparison criteria, we find that :

- Fault identification : there are no sources of faults identified while designing the system ;
- fault-detection : unknown ;
- fault-recovery : known ;
- kind of redundancy : server replication ;
- number of replica : unlimited ;
- kind of recovery : backward recovery.

6.4.2 Recovering Approaches from Agent Failures

In the literature, several approaches are proposed to recover from agent failures in mobile agent systems [51]. An agent can be confronted to three sources of failures :

1. The failure of the component in which the agent is performing ;
2. Failure of other agents with which the agent is cooperating ;
3. Failure of the agent itself.

One of the proposed approaches [51] is the Meta-agent approach : each agent is associated with a meta-agent which is responsible for the fault-tolerant aspect. The meta-agent enables the agent to handle exceptions. In this approach, the meta-agent needs another meta-agent since it is prone to failure too. Hence, fault tolerance is not guaranteed. This approach does not cover fault prevention nor fault forecasting.

Based on the comparison criteria, we find that :

- Fault identification : there are no sources of faults identified while designing the

system ;
- fault-detection : known ;
- fault-recovery : known ;
- kind of redundancy : agent replication ;
- number of replica : there is a limit in the number of replica per agent. Nevertheless, this limit is not provided ;
- kind of recovery : forward recovery.

6.5 Autonomous Robot

Fault tolerance techniques are also developed for robot systems. In [18], an approach is proposed to overcome the physical failures of a robot. The robot on which these experiences are made is *Hannibal* [1] which has 19 actuators and over 60 sensors. Physical failure can be attributed to either mechanical failure or sensor failure. *Hannibal*'s task is to move in a rough and hazardous environment.

The approach used to overcome a robot failure is forward recovery. In fact, the robot can see its functionalities degrading when failures occur. If the robot has a physical failure or if one of its sensors is out of service, it is not possible to replace either the defected sensor or the defected physical component. Hence, the robot state cannot back up to an earlier free-error state. However, the robot can be developed so that it adapts to failures. For example, if the robot detects a sensor failure, it must no longer consider any information provided by the failed sensor. Nevertheless, it has to continue to operate properly, if possible, depending on the type of failure.

Based on the comparison criteria, we find that :

- Fault identification : there are no sources of faults identified while designing the system ;
- fault-detection : known ;
- fault-recovery : known ;
- kind of redundancy : no redundancy ;
- number of replica : 0 ;
- kind of recovery : forward recovery.

6.6 Computational Grid

Another domain in which fault-tolerant techniques can be applied is computational grids. Computational grids are computing environments with massive resources such as servers for data processing and storage [45]. In [45], a computational grid, *NetSolve*, is proposed to overcome server failures. Each agent can interact with a server which processes its requests. In order to overcome a server failure, the agent can ask another server to perform its requests. The new server needs information about the failed server state. Hence, *NetSolve* uses a backward recovery technique.

NetSolve provides storage servers that will store checkpoints of the computations running on other servers. If an agent detects that a server has failed, then it selects a new server to resume the computation from the most recent checkpoint in the storage server. As in [21], if the last server state is not an error-free state, then the replica server will be in failure. Hence, it is not guaranteed that the system will behave according to its specifications.

Based on the comparison criteria, we find that :

- Fault identification : there are no sources of faults identified while designing the system ;
- fault-detection : known ;
- fault-recovery : known ;
- kind of redundancy : server replication ;
- number of replica : unknown since there are no information on the number of replica per server ;
- kind of recovery : backward recovery.

This comparison is summarized in the table 6.1 below :

6.7 Summary

In this chapter, we have reviewed some related works on fault-tolerant systems applied to multi-agent systems. Each of the presented systems was described briefly and compared with the fault-recovering technique used in the **FATMAS** methodology. None of the described techniques provides all together the major contributions of **FATMAS** that are :

	fault identification	fault detection	fault recovery	kind of redundancy	number of replica	kind of recovery	
DARX	not identified	known	known	agent	unlimited	backward	
Brokered *MAS*	development	unknown	known	agent	unlimited	forward	
SG-ARP	not identified	known	known	server	unlimited	backward	
Meta-agent approach	not identified	known	known	agent	limited	forward	
Robots	not identified	known	known	-	0	forward	
NetSolve	not identified	known	known	server	unlimited	backward	
FATMAS	identified	known	known	agent and task	limited	backward	

TAB. 6.1 – The comparison between different techniques.

1. the use of task replication in existing agents as much as possible ;
2. the design of a *MAS* allowing *MAS* reorganization when a fault occurs in a manner consistent with its specification, design, and implementation ;
3. the limitation of the number of replicated components to minimize the impact of added fault-tolerant based features on the overall system complexity.

Chapitre 7

Conclusion

This thesis aimed at defining a methodology to design fault-tolerant multi-agent systems. This methodology was built on the basis of concepts that support the design of multi-agent systems. The methodology integrates a reorganization technique which allows the system to detect and recover from some failures.

In this chapter, we first present the achievements of this thesis. Then, we outline the future directions of the research and development of the *FATMAS* methodology.

7.1 Achievements

In this thesis, we provided a fault-tolerant multi-agent systems design methodology. It has three phases : analysis, design, and implementation. The analysis phase produces the task-environment model. The design phase produces the agent model and the agent interaction model. The implementation model produces the deployment model and allows a code generation of the agents. Furthermore, FATMAS integrates a reorganization technique which allows the MAS to be fault-tolerant against failures originating from the MAS boundary. These models are produced following a macro-process and four micro-processes. The macro-process guides the designer on how to proceed from one model to the next. Each model is associated with a micro-process which guides the designer to produce the model. FATMAS uses an iterative approach based on a cost/benefit evaluation to decide whether the designer go from one iteration to another.

This methodology has the advantage to integrate a reorganization technique, which

is not the case of existing *MAS* design methodologies [2][14][17][24][31][57]. Hence, a designer finds him/herself in a process allowing him/her to anticipate possible failures to which the system may be confronted. Consequently, *FATMAS* reduces the implementation complexity of the system by identifying possible sources of failures from the design phase, which is therefore built to easily adapt itself to these new arising situations.

The proposed technique has the advantage to include the techniques required to build a fault-tolerant system which are : fault-prevention, fault-recovery, and fault-tolerance. The fault-prevention technique allows to delimit the system boundary from its environment. The fault-recovery technique allows the agents to detect and overcome failures originating from the system's boundary. The fault-tolerance technique determines the critical agents of the system which must be replicated. The other agents of the system will only see their tasks being replicated in other agents. Hence, this technique minimizes the system complexity by reducing the number of agents that will be replicated.

The *FATMAS* methodology has the advantage to be open in that it does not rely on any programming language. The development of the different models does not refer to any particular technology. Furthermore, FATMAS is built upon a minimal number of concepts and provides a reorganization technique which is independent from any agent or MAS architecture. This allows it to integrate several models from existing methodologies.

Now that we have presented the achievements of this thesis which are the fault-detection and fault-recovery in the design phase, the automatic reassignment of failed tasks, and the decrease of the resulting system complexity by reducing the number of agents to replicate. We outline, in the next section, the limits of *FATMAS* and how research on the *FATMAS* methodology could proceed in the future.

7.2 Limits of FATMAS and Future Work

In spite of its achievements, *FATMAS* has limits. First, *FATMAS* does not address the issue of code bugs. Second, *FATMAS* is built for closed systems in which the agents are identified during the design phase. *FATMAS* cannot be applied to the development of an open system in which agents are not determined in advance. Third, *FATMAS* does not support all the processes of software engineering such as documentation of the model or the support of the test phases. Finally, but not last, *FATMAS* has not been used to develop real size projects in order to test its robustness. Meanwhile, there are

several challenges in the development of *FATMAS*. They can be represented by three research directions presented in what follows.

The first direction in the development of *FATMAS* is to develop a case tool which models the macro-process and the micro-processes. The case tool would assist the designer in the development of the different models and in the management of the projects that use *FATMAS*. It should also coordinate the activities of the different developers working on the design of a common system. The different models of FATMAS are defined. Furthermore, the formal definitions will allow to validate one model according to the previous ones.

The second direction in the development of *FATMAS* is to include a forward recovery approach. In fact, we stated that a *MAS* is in failure when one of its tasks is in failure. This assumption means that the *MAS* cannot operate in a degraded mode. Nevertheless, there are cases where a *MAS* could operate in such a mode. This could be a challenge in the future to model a system which can operate in a degraded mode, and thus to identify this degraded mode during the design level.

The third direction in the development of *FATMAS* is to evaluate this methodology on real industrial projects. This will allow to validate and overcome some of the weaknesses of the methodology. This methodology must also be compared to other methodologies used to develop fault-tolerant systems by applying them on the same projects.

FATMAS provides a new methodology to develop fault-tolerant multi-agent system. It integrates a reorganization technique in the design phase which allows to delimit the system boundary, and to anticipate on particular faults which should be taken into account in the design phase due for example to the high cost to remedy them during system execution. The methodology determines how the failed tasks can be performed by other agents so that the way the system overcome its failure is planned at design which avoids the system to reorganize itself. Finally the methodology decreases the resulting system complexity by minimizing the number of agents to replicate. Consequently, if it is costly to replace an agent by another one, then this methodology allows to minimize such a cost.

Bibliographie

[1] C. Angle and R. Brooks. Small planetary rovers. In *Proceedings of IEEE International Workshop on Intelligent Robots and Systems. Ibaraki, Japan, pp. 383-388*, 1990.

[2] C. Bernon, M. Gleizes, S. Peyruqueou, and G. Picard. Adelfe, a methodology for adaptive multi-agent systems engineering. In *Engineering Societies in the Agents World III : Third International Workshop, ESAW 2002, Madrid, Spain, September 16-17, 2002. Revised Papers, Lecture Notes in Artificial Intelligence , Vol. 2577 P. Petta, R. Tolksdorf, F. Zambonelli, pp. 156-169*, 2003.

[3] D. Bolchini and J. Mylopoulos. From task-oriented to goal-oriented web requirements analysis. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISEŠ03), pp. 166-175*, 2003.

[4] G. Booch. Object solutions : Managing the object-oriented project. In *Addison-Wesley, Paperback, Published October, 323 pages, ISBN 0805305947*, 1995.

[5] J. Brediny, R. T. Maheswaranz, and C. Imerz. A game-theoretic formulation of multi-agent resource allocation. In *Proceedings of the 4th International Conference on Autonomous Agents, pp. 349-356, Barcelona, Spain*, 2000.

[6] P. Busetta, R.R. onnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java. In *AgentLink News Letter, January*, 1999.

[7] G. Caire, W. Coulier, F.J. Garijo, J. Gomez, J. Pavon, F. Leal, P. Chainho, P.E. Kearney, J. Stark, R. Evans, and P. Massonet. Agent oriented analysis using message/UML. In *Agent-Oriented Software Engineering, pp. 119-135*, 2001.

[8] J.C. Campelo, F. Rodriguez, and J.J. Serrano. Dependability evaluation of fault tolerant distributed industrial control systems. In *Workshop on Parallel and Distributed Real-Time Systems, pp 384-388*, 1999.

[9] P. Checkland. Systems thinking, systems practice. In *John Wiley and Sons, 344 pages, ISBN : 0471279110*, 1981.

[10] S. Combettes and A. Nkesta. A virtual system approach for distributed simulation of complex systems. In *2002 IEEE International Conference on Systems, Man and Cybernetics*, 2002.

[11] W. T Comfort. A fault-tolerant system architecture for navy applications. In *IBM Journal Resource Development, vol.27, No.3, pp 219-236*, 1983.

[12] L. Cragg, P.W. Tsui, and H. Hu. Building a fault tolerant architecture for internet robots using mobile agents. In *Proceedings of 1st Workshop on Internet and Online Robots, University of Reading, May*, 2003.

[13] J. Debenham. Who does what in a multiagent system for emergent process management. In *Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS.02), pp. 35-40*, 2002.

[14] O. Dikenelli and R.C. Erdur. Sabpo : A standards-based and pattern-oriented multi-agent development methodology. In *Engineering Societies in the Agents World III : Third International Workshop, ESAW 2002, Madrid, Spain, September 16-17, 2002. Revised Papers, Lecture Notes in Artificial Intelligence , Vol. 2577 P. Petta, R. Tolksdorf, F. Zambonelli, pp. 213-226*, 2003.

[15] S. S. Fatima and M. Wooldridge. Adaptive task and resource allocation in multi-agent systems. In *Agents 2001 : Proceedings of the Fifth International Conference on Autonomous Agents. , Montreal, Canada*, 2001.

[16] A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy, pp. 737-744*, 2002.

[17] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS98) , pp 128–135, Paris, France*, 1998.

[18] C. Ferrell. Failure recognition and fault tolerance of an autonomous robot. In *Adaptive Behavior, 2 (4), pp. 375-398*, 1994.

[19] FIPA. www.fipa.org.

[20] F. Giunchiglia, J. Mylopoulos, and A. Perini. The tropos software development methodology : processes, models and diagrams. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems, pp.35-36*, 2002.

[21] Z. Guessoum, J.P. Briot, and S. Charpentier. A fault-tolerant multi-agent framework. In *Proceedings of the First International Joint Conference on Autonomous agents and multiagent systems, Bologna, Italy, pp. 672-673*, 2002.

[22] S. Hagg. A sentinel approach to fault handling in multi-agent systems. In *Proceedings of the Second Australian Workshop on Distributed AI, in conjunction with*

the Fourth Pacific Rim International Conference on Artificial Intelligence (PRICAI'96), Cairns, Australia, August 27, 1996.

[23] R. De Hoog, R. Martil, B. Wielinga, R. Taylor, C. Bright, and W. Van de Velde. The commonkads model set. In *Esprit Project P5248 KADS-II/M1/DM..1b/Uva/018/5.0, University of Amsterdam, Lloyd's Register, Touche Ross Management Consultants and Free University of Brussels*, 1993.

[24] C.A. Iglesias, M. Garijo, J. Centeno-Gonzalez, and J.R. Velasco. Analysis and design of multiagent systems using MAS-common KADS. In *Agent Theories, Architectures, and Languages, pp. 313-327*, 1997.

[25] S. Kim and A. K. Somani. Ssd : An affordable fault tolerant architecture for superscalar processors. In *2001 Pacific Rim International Symposium on Dependable Computing December 17 - 19, Seoul, Korea, pp. 27-34*, 2001.

[26] D. Kinny and M. Georgeff. A methodology and modelling technique of systems of bdi agents. In *Agents breaking away : Proceedings of the seventh European Workshop on Modelling Autonomous Agents in a Multi-agent World, Y. Demazeau and J.-P.Müller, (eds.) Volume 1038 of LNCS, New York : Springer, pp. 56-71*, 1996.

[27] D. Kinny and M. Georgeff. Modelling and design of multi-agent systems. In *Intelligent Agents III : Proceedings of the third international workshop on agent Theories, Architectures and Languages (ATAL-96), J. Müller, M. Wooldridge and N. R. Jennings, (eds.), Volume 1193 of LNCS, New York : Springer, pp. 1-20*, 1996.

[28] G. Kola, T. Kosar, and M. Livny. A fully automated fault-tolerant system for distributed video processing and off-site replication. In *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video, pp.122–126, Cork, Ireland*, 2004.

[29] S. Kumar and P.R. Cohen. Towards a fault-tolerant multi-agent system architecture. In *Proceedings of the fourth international conference on Autonomous agents, pp. 459-466*, 2000.

[30] L. C. Lee, H. S. Nwana, D. T. Ndumu, and P. De Wilde. The stability, scalability and performance of multi-agent systems. In *BT Technology Journal Vol 16 No 3, pp. 94-103*, July 1998.

[31] J. Lind. Massive : Software engineering for multi-agent systems. In *Phd Thesis., University of saarlandes*, 1999.

[32] M. Luck, R. Ashri, and M. D'inverno. Agent-based software development. In *Artech House Eds. 208 pages, ISBN : 1-58053-605-0*, 2004.

[33] A. Macro. Software engineering, concepts and management. In *Prentice Hall, 544 pages, ISBN : 0138202672*, 1990.

[34] J. McCarthy. Situation calculus with concurrent events and narrative. In *http ://www-formal.stanford.edu/jmc/narrative/narrative.html*, 2000.

[35] S. Mellouli, G. Mineau, and B. Moulin. Laying down the foundations of an agent modelling methodology for fault-tolerant multi-agent systems. In *Engineering Societies in the Agents World IV 4th International Workshop, ESAW 2003, London, UK, October 29-31, 2003, Revised Selected and Invited Papers, Lecture Notes in Artificial Intelligence , Vol. 3071 Omicini, Andrea ; Petta, Paolo ; Pitt, Jeremy (Eds.)*, 2004.

[36] S. Mellouli, G. Mineau, and B. Moulin. Towards an agent modelling methodology for fault-tolerant multi-agent systems. In *Informatica Journal. Ed. Shahram Rahimi, Marcin Paprzycki and Gabriel Ciobanu, pp. 31-40.*, 2004.

[37] S. Mellouli, G. Mineau, and D. Pascot. The integrated modeling of multi-agent systems and their environment. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems, pp. 507-508*, 2002.

[38] S. Mellouli, B. Moulin, and G. Mineau. Laying the foundations for an agent modelling methodology for fault-tolerant multi-agent systems. In *The Fourth International Workshop Engineering Societies in the Agents World. October 29-31, London, UK*, 2003.

[39] S. Mishra. Agent fault tolerance using group communication. In *Proceedings of the ISCA 8th International Conference on Intelligent Systems, Denver, CO*, 1999.

[40] J.L. Le Moigne. La modélisation des systèmes complexes. In *Bordas, 178 pages, ISBN : 2-04-019704-4*, 1990.

[41] J. Odell, H. Parunak, and B. Bauer. Extending uml for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence.*, 2000.

[42] J. Odell, V. D. Parunak, and M. Fleischer. The role of roles. In *Journal of Object Technology, January-February, Vol.2, No.1 39-51*, 2003.

[43] A. Omicini. SODA : Societies and infrastructures in the analysis and design of agent-based systems. In *Agent-Oriented Software Engineering, Ciancarini, Paolo AND Wooldridge, Michael J. (eds.), LNCS volume 1957, pp. 185-193*, 2000.

[44] L. Padgham and M. Winikoff. Prometheus : A methodology for developing intelligent agents. In *Agent-Oriented Software Engineering III, F. Giunchglia, J. Odell, and G.Weiss, (eds.) Volume 2585 of LNCS, New York : Springer, pp. 174-185*, 2003.

[45] J.S. Plank, H. Casanova, M. Beck, and J.J. Dongarra. Deploying fault tolerance and task migration with netsolve. In *Journal of Future Generation Computer Systems (15), pp. 745-755*, 1999.

[46] R. Price, S. Krishnaswamy, S. W. Loke, and M. B. Chhetri. Towards an ontology for agent mobility. In *Lecture Notes in Computer Science, Volume 3289, pp. 484 - 495*, 2004.

[47] L.L. Pullum. Software fault tolerance techniques and implementation. In *Artech House eds, 360 pages, ISBN 1580531377*, 2001.

[48] A.S Rao and M. Georgeff. Bdi agents : from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems, San Francisco, CA, USA. pp. 312-319.*, 1995.

[49] P. Robertson, R. Laddaga, and H. Shrobe. Introduction : the first international workshop on self-adaptive software. In *In Proceedings of the 1st IWSAS edited by P. Robertson, R. Laddaga and H. Shrobe in LNCS 1936, pp 1-10*, 2000.

[50] A.T. Schreiber, B.J. Wielinga, J.M. Akkermans, and W. Van de Velde. A comprehensive methodology for kbs development. In *Deliverable DM1.2a KADS-II/ M1/RR/UvA/70/1.1, University of Amsterdam, Netherlands Energy Research Foundation ECN and Free University of Brussels*, 1994.

[51] G. Di Marzo. Serugendo and A. Romanovsky. Designing fault-tolerant mobile systems. In *Scientific Engineering for Distributed Java Applications, International Workshop, FIDJI 2002, Luxembourg-Kirchberg, Luxembourg, November 28-29, pp. 185-201*, 2002.

[52] F. T. Sheldon, S. G. Batsell, S. J. Prowell, and M. A. Langston. Position statement : Methodology to support dependable survivable cyber-secure infrastructures. In *In Proceedings of the 38th Hawaii International Conference on System Sciences*, 2005.

[53] UML. Unified modelling lnaguage. In *www.uml.org*.

[54] M. Uschold and M. Gruninger. Ontologies : Principles, methods, and applications. In *Knowledge Engineering Review, Vol.11, No.2, pp.93-136*, 1996.

[55] G. Weiss. Multiagent systems - a modern approach to distributed artificial intelligence. In *MIT Press, 643 pages, ISBN : 0262232030*, 1999.

[56] B. Wilson. Systems : Concepts, methodologies, and applications. In *John Wiley and Sons, 410 pages, ISBN : 0-471-92716-3*, 1984.

[57] M. Wooldridge, N. R. Jennings, and D. Kinny. Developing multiagent systems : The gaia methodology. In *ACM Transactions on Software Engineering and Methodology 12(3) :pp. 317-370*, 2004.

[58] M. Wooldridge and N.R. Jennings. Intelligent agents : Theory and practice. In *The Knowledge Engineering Review, 10(2), pp 115-152*, 1995.

Annexe A

Printer's Manager Agents

The agents operating in the printers manager system described in Chapter 4.

```
import java.util.Vector;

class PrinterRepairAgent {
  /* {src_lang=Java}*/


  public int printerConnected;
  /* {transient=false, volatile=false}*/

  public int softwarePossesion;
  /* {transient=false, volatile=false}*/

  public int reportProblemsConnected;
  /* {transient=false, volatile=false}*/

  public int repl_managePrinterConnected;
  /* {transient=false, volatile=false}*/

  public int repl_printerConnected;
  /* {transient=false, volatile=false}*/

    /**
   *
   * @element-type PrinterManagerAgent
   */
  public Vector  myPrinterManagerAgent;
    public OrderPickerAgent myOrderPickerAgent;

  public void updatePrinterDriver() {
  }

  public void detectPrinterCPUProblems() {
  }

  public void detectHardwareProblems() {
  }

  public void evaluateInkQuantity() {
  }

  public void evaluatePaperQuantity() {
  }

  public void detectPrinterConnectionProblems()
{
  }

  public void notifyRepairStatus() {
  }

  public void
repl_PrinterManagerAgent_receiveDocument() {
  }

  public void
repl_PrinterRepairAgent_sendDocument() {
  }

  public void
repl_PrinterRepairAgent_managePrinter() {
  }

  public void
repl_PrinterRepairAgent_detectPrinterProblems()
{
  }

  public void
repl_PrinterRepairAgent_detectPrinterProblems()
{
  }
```

```
import java.util.Vector;

class OrderPickerAgent {
  /* {src_lang=Java}*/


  public int bdAccess;
  /* {transient=false, volatile=false}*/

  public int notifyRepairStatusReceptive;
  /* {transient=false, volatile=false}*/

    /**
   *
   * @element-type PrinterRepairAgent
   */
  public Vector  myPrinterRepairAgent;
    public OrderPlaceAgent myOrderPlaceAgent;

  public void getPaper() {
  }

  public void getToner() {
  }

  public void getCPU() {
  }

  public void getSoftware() {
  }

  public void repairConnection() {
  }

  public void sendMessageAvailability() {
  }
}
```

FIG. A.2 – The code of OrderPickerAgent.

```
import java.util.Vector;

class OrderPlaceAgent {
  /* {src_lang=Java}*/


  public int supplierConnected;
  /* {transient=false, volatile=false}*/

    /**
   *
   * @element-type OrderPickerAgent
   */
  public Vector  myOrderPickerAgent;

  public void orderPaper() {
  }

  public void orderToner() {
  }

  public void orderCPU() {
  }

  public void orderHardware() {
  }

  public void orderSoftware() {
  }

  public void receiveOrderStatusSupplier() {
  }
}
```

FIG. A.3 – The code of OrderPlaceAgent.

```
public class repl_OrderPickerAgent {

  public int repl_bdAccess;
  /* {transient=false, volatile=false}*/

  public int repl_notifyRepairStatusReceptive;
  /* {transient=false, volatile=false}*/

  public void repl_getPaper() {
  }

  public void repl_getToner() {
  }

  public void repl_getCPU() {
  }

  public void repl_getSoftware() {
  }

  public void repl_repairConnection() {
  }

  public void repl_sendMessageAvailability() {
  }
}
```

FIG. A.4 – The code of OrderPickerAgent replica.

```
public class repl_OrderPlaceAgent {

  public int repl_supplierConnected;
  /* {transient=false, volatile=false}*/

  public void repl_orderPaper() {
  }

  public void repl_orderToner() {
  }

  public void repl_orderCPU() {
  }

  public void repl_orderHardware() {
  }

  public void repl_orderSoftware() {
  }

  public void repl_receiveOrderStatus() {
  }
}
```

FIG. A.5 – The code of OrderPlaceAgent replica.

Annexe B

Publications

[1] **Mellouli., S.**, Mineau., G. and Moulin., B. Towards an Agent Modelling Methodology for Fault-Tolerant Multi-Agent Systems. *Informatica Journal. Ed. Shahram Rahimi, Marcin Paprzycki and Gabriel Ciobanu.* pp 31-40. 2004.

[2] **Mellouli., S.**, Moulin., B. and Mineau., G. Laying Down the Foundations of an Agent Modelling Methodology for Fault-Tolerant Multi-Agent Systems *Engineering Societies in the Agents World IV 4th International Workshop, ESAW 2003, London, UK, October 29-31, 2003*, Revised Selected and Invited Papers, Lecture Notes in Artificial Intelligence , Vol. 3071 Omicini, Andrea ; Petta, Paolo ; Pitt, Jeremy (Eds.) 2004.

[2] **Mellouli., S.**, Mineau., G., and Moulin., B. Towards an Agent Modelling Methodology for Fault-Tolerant Multi-Agent Systems. *Fourth International Workshop Engineering Societies in the Agents World.*, London, United Kingdom, October 28-29, 2003.

[3] **Mellouli., S.**, Moulin., B., and Mineau., G. Towards an Agent Unified Modelling Methodology (AUMM). *The 7th World Multiconference on Systemics, Cybernetics and Informatics.* Orlando, Florida, USA, July 27-30, 2003.

[4] **Mellouli., S.**, Moulin., B. and Mineau., G. Situation Event Logic for Early Validation of Multi-Agent Systems. *The Sixteenth Canadian Confe-*

rence on Artificial Intelligence. Halifax, Nova Scotia, Canada, June 11-13, 2003.

[5] Mellouli., S., Mineau., G., and Moulin., B. Modelling a Multi-Agent System Environment. *Engineering Societies in the Agents World III*. LNCS 2577. Springer-Verlag, April 2003.

[6] Mellouli., S., Mineau., G. and Pascot., D. Multi-Agent System Design. *Third International Workshop Engineering Societies in The Agents World (ESAW'02)*, Madrid, Spain, Septembre 16-17, 2002.

[7] Mellouli., S., Mineau., G. and Pascot., D. The Integrated Modeling of Multi-Agent Systems and their Environment. *First International Conference on Autonomous Agents and Multi-Agent Systems 2002 (AAMAS 2002)*, Bologne, Italie, July 15-19, 2002.