ÉCOLE DOCTORALE DES SCIENCES EXACTES ET DE LEURS APPLICATIONS ED 211

Service-Oriented Integration of Component and Organizational MultiAgent Models

THÈSE

présentée et soutenue publiquement le 4 décembre 2012

pour l'obtention du

Doctorat de l'Université de Pau et des Pays de l'Adour
(spécialité informatique)

par

Nour Alhouda Aboud

Composition du jury

Président :	Mourad Oussalah	Professeur à l'université de Nantes
Rapporteurs :	Antoine Beugnard	Professeur à TELECOM Bretagne
	Jean-Paul Arcangeli	Maître de Conférences (HDR) à l'université Paul Sabatier
Examinateurs :	Philippe Aniorté	Professeur à l'université de Pau et des Pays de l'Adour
	Eric Cariou	Maître de Conférences à l'université de Pau et des Pays de l'Adour
	Eric Gouardères	Maître de Conférences à l'université de Pau et des Pays de l'Adour

Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour — EA 3000

Mis en page avec la classe thloria.

Remerciements

Ce mémoire de thèse représente l'achèvement d'un long travail qui n'aurait pu voir le jour sans la participation, l'aide, les conseils, ou encore la présence de nombreuses personnes. Je suis vraiment ravi d'être arrivé à mener à bien cette thèse. Je tiens à remercier toutes les personnes qui ont contribué de près ou de loin à cette réussite.

En premier lieu, je tiens à remercier l'ensemble des membres du jury. Je remercie Monsieur Antoine Beugnard et Monsieur Jean-Paul Arcangeli pour avoir accepté de rapporter cette thèse et d'avoir contribué à sa amélioration par leur remarques. Je remercie aussi Monsieur Mourad Oussalah pour avoir accepté de présider ce jury. Merci pour l'investissement que vous avez mis dans l'évaluation de mes écrits et de ma soutenance.

Je tiens à exprimer ma gratitude à mon directeur de thèse Monsieur Philippe Aniorté, pour m'avoir accordée sa confiance tout au long de ces quatre années. Mes remerciements vont également à Monsieur Eric Cariou et Monsieur Eric Gouardéres pour leurs encadrements, conseils et tous leur patiences pendant la période de cette thèse. Merci pour votre aide précieuse dans toutes les occasions. Je suis ravi d'avoir travaillé à vos côtés.

Je remercie mes camararades (les thésards au LIUPPA): Youssef, Damien, Eric, Cyril, Natacha, Ehssan, Mamour, Hui et Jihad. Merci également à tout les membre de LIUPPA et les membre de equipes MOVIES, particulièrement à Frank qui pense à tous ce qui peut être utile pour un thèsard.

Je n'oublie évidemment pas mes amies pour me soutenir: Meriem, Wissam, Louna, Tarek, Karmel, Youssef, Yousra, Zeina et Fady. Merci pour avoir toujours été là pour moi. Je souhaite aussi remercier tous mes amis en Syrie et en France qui m'ont supporté et encouragé pendant ce travail.

J'adresse aussi mes remerciements à Régine pour sa gentillesse. Merci Laurent, Christophe, Sophie, Annig, Congduc, Bruno, Nabil, Nicolas et tous les membres de département d'informatique à UPPA pour les divers échanges au niveau général, cours comme pour les très nombreuses pauses cafés.

Un grand merci à tous mes professeurs, dans toutes les étapes de mon éducation, qui m'ont motivée et guidée pendant mes études en Syrie et en France.

Merci particulièrement à Youssef, Jihad et Damien d'avoir accepté de relire ce manuscrit pour corriger les nombreuses fautes restantes.

Dans la vie, nous avons tous besoin d'un exemple, d'une personne à qui l'on aimerait ressembler, merci à Marianne Huchard pour être mon exemple. Le travaille avec elle, m'encourage de faire la thèse, comme elle est tellement travailleuse et active en plus de sa gentillesse.

Maman, Il me tient énormément à cœur de te dire MERCI. Merci d'être ma mère, de m'avoir donné des racines et des ailes. De m'avoir supporté et appuyé durant toutes ces années. De m'avoir inculqué de vraies valeurs et de m'avoir permis de devenir celle que je suis aujourd'hui. Grace à toi j'ai pu m'épanouir et m'ouvrir à la vie. Tu me manque. Amour infini et gratitude à mes sœurs et mon frère qui ont toujours cru en moi, toujours me soutenir et qui sommes toujours fiers de moi.

J'exprime également ma profonde gratitude à ma belle famille, qui m'a toujours encourager, croient en mon succès, et envoyez-moi son amour à travers les distances. Merci à mon beau-père pour me supporter des le debout de cette thèse.

Je ne saurais dire assez de mercis à Nidal, qui m'a toujours soutenu, ma donné l'énergie et la compréhension nécessaire pour finir ce travail. Merci Nidal de votre confiance continue en moi, même au cours du stress et les difficultés. Merci pour tous notre discussion du nuit au sujet de la programmation et de la mise en œuvre. Pardonnez-moi pour tous les périodes difficiles qui ont affecté notre vie.

Je sais que tu ne peux pas lire maintenant, mais je ne t'ai pas oubliée, ma petite papillonne qui puisent le sourire sur mon visage quand je suis déprimée avec tes mots et tes sourires, merci Yara.

Merci pour le pays qui m'a donné l'opportunité de faire cette thèse, merci la France pour m'aider à réaliser mon rêve. Merci pour mon pays qui me manque tellement "'time flies indeed but It goes slowly when I feel homesick'" à toi à ma Syrie, je dédie cette thèse, que je vous aime.

Cet exercice de style s'avère plus ardu que je ne l'aurais pensée… Je souhaiterais n'omettre personne mais cela me semble difficile. Aussi, je remercie par anticipation tous ceux qui liront ces lignes en ayant le sentiment d'avoir été oublié dans la liste précédente.

To my beloved country "Syria", my beloved family: Nidal, Yara, mother and to the memory of my two fathers.

Contents

List of Tables

List of Figures

Part I

Introduction

Every new beginning comes from
some other beginning's end

Seneca *Roman philosopher, mid-1st century AD*

Chapter 1

Introduction

Contents

1.1 Context and Problematic

Twenty years ago, information systems were homogeneous, monolithic and centralized, traditional mature approaches such as object-oriented software engineering were sufficient. Nowadays, information systems are distributed, large-scaled, heterogeneous, open and complex. This leads to the emergence of more high-level technologies that interoperate between each other and break the software's isolation [PCW98]. We can cite here MultiAgent Systems (MAS) [JSW98] in artificial intelligence domain, component-based approaches [Szy02] and Service Oriented Architecture (SOA) [PH07] in software engineering domain. These abstract approaches reduce both development time and complexity on one hand, and increase the quality, reliability and reusability of the developed systems, on the other hand. Figure 1.1 presents the evolution of software engineering inspired from [Som04], where we can see the progress from lines of code in structured programming to the current trends or approaches like service orientation and model based ones.

Our research is related to the development of distributed systems, for which, we aim providing more efficient paradigms. We share the same point of view of Braubach

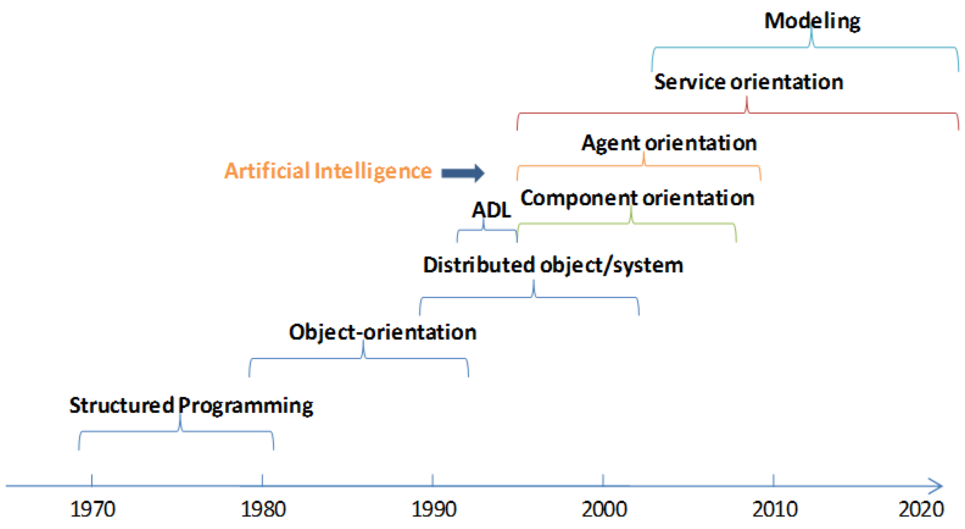


Figure 1.1: The software evolution.

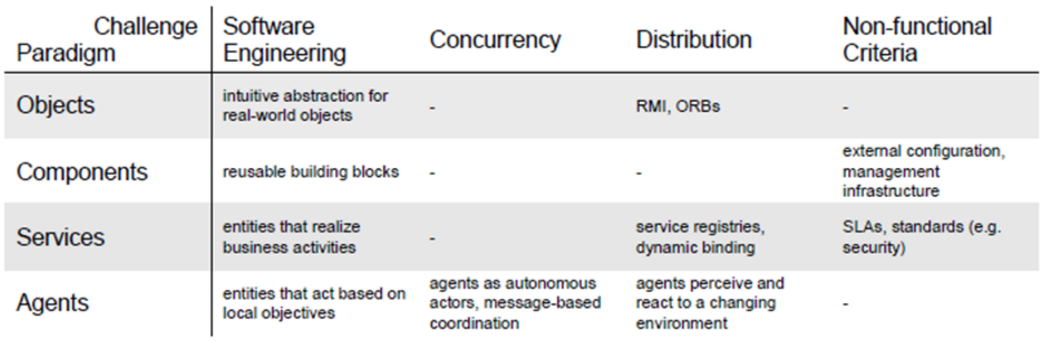
et al. [BP11] which states that the existing approaches like object, component, agent and service may not be surely sufficient to describe effectively all kinds of distributed systems. We must never forget that these approaches inherit conceptual limitations. Therefore, we need to integrate these approaches or to combine them, in order to raise their levels of efficiency. Figure 1.2 highlights the challenges of each paradigm. Objects represent the abstraction of real world ones using Remote Method Invocation (RMI)[1] to ensure their interactions in distributed systems. The component-based approach extends the object-oriented paradigm by focusing on the separation of concerns and reusability interests. A component provides or uses functionalities to (from) other components through well-defined interfaces and most components use the RMI for interactions in a distributed environment. Service Oriented Architecture (SOA) attempts to ensure the interoperability between distributed systems using registries to find and publish services. The autonomy, concurrency and ability to take decisions are key properties of the agent oriented approach relatively to the previous ones, where agents can coordinate and negotiate with each other in a distributed environment.

We start by reviewing briefly the history of Service Oriented Architectures, Component-based approaches, and MultiAgent approaches.

1.1.1 Service Oriented Architecture (SOA)

The SOA paradigm appeared at the late nineties. Nevertheless, its architecture style and the logic behind it have been existing since the eighties [Tow08]. SOA is based on service definitions and actors that use or provide them. The registry (where we find and publish services) is a key element in SOA approaches. Since it is related to

[1]http://docs.oracle.com/javase/1.4.2/docs/guide/rmi/



Challenge / Paradigm	Software Engineering	Concurrency	Distribution	Non-functional Criteria
Objects	intuitive abstraction for real-world objects	-	RMI, ORBs	-
Components	reusable building blocks	-	-	external configuration, management infrastructure
Services	entities that realize business activities	-	service registries, dynamic binding	SLAs, standards (e.g. security)
Agents	entities that act based on local objectives	agents as autonomous actors, message-based coordination	agents perceive and react to a changing environment	-

Figure 1.2: The challenges and contributions of current paradigms [BP11].

specific technologies, we consider it in a lower level of design (i.e., Universal Description, Discovery and Integration (UDDI)[2] for web services). The interactions between these actors to use or provide services is a key concept in this paradigm. This architecture responds to the needs of open, interoperable and complex information systems.

In other words, SOA views applications as sets of interacting services according to their roles and independently of their locations, in order to satisfy heterogeneous and loose-coupled software systems.

1.1.2 Component-based approach

The Architecture Description Language (ADL) is the historical base of the component-based approaches and appeared in the mid nineties [MDEK95]. This approach offers the main interest of reusing blocks of code, which implement certain services without knowing details about their implementations. These black boxes associated with well-specified interfaces present solutions to reduce the cost of development and redundancy of codes. It provides efficient solutions for defining well-structured and robust applications by composing and reusing existing components (following for instance the Commercial Off the Shelf (COTS) approach [CL00]). A component interface defines the services of this component. We can consider service-based approaches as a logical extension of component ones since both of them meet reusability and composition purposes.

1.1.3 MultiAgent Systems (MAS)

The agent approach appeared at the late seventies under the form of Distributed Artificial Intelligence (DAI) where Hewitt [Hew77] proposed the concept of *Actor* which is a self-contained, interactive and concurrently executing object. In the mid nineties, the collective MAS models appeared. In these models, agents interoperate to achieve common activities. Organizational MultiAgent Systems (OMAS) are among these new models. MAS is a paradigm for understanding and building distributed systems, where

[2]http://uddi.nic.go.th/uddipublic/help/1033/intro.whatisuddi.aspx

it is assumed that the computational elements, i.e., the agents are able to perform autonomous actions in some environment. The social abilities of cooperation, coordination, and negotiation between agents [Woo09] is one of their main characteristics. There are two main types of agents:

- Reactive agents wait until an action happens to respond to the changes in their environment.
- Proactive agents take the initiative and make decisions in their environment thanks to goal-directed behaviours characteristic.

Organizational MultiAgent Systems are effective paradigms for addressing the design challenges of large and complex MAS. Organizations are emergent whenever agents work together in a shared environment. Many similarities exist between OMAS and service oriented approaches. They both meet the flexibility and dynamicity features. Organizations are ways to makeup systems of collaborative services [SH05]. The nature of agents, as autonomous entities with auto-organized capabilities and high-level interactions, facilitates the automatic service discovery. For all these reasons, we restrict our research in OMAS models; then, whenever we refer to agent models in the rest of this document, we refer to OMAS ones implicitly.

1.1.4 Complementarities between components, agents and services

Component and agent approaches are complementary. They have common points where a reactive agent is equivalent to a component and they both represent service providers or consumers in SOA. However, at the same time, they own their key features, which are not shared between them.

The lack of reutilization is one of the limitation of the agent approach [SA04a, Lin01], in addition to the risk of losing control related to the autonomy property of agent. These limitations reflect the need of adding reutilization and robust properties to the models of agents (which are already key properties of the component approach). However, the component approach suffers from the lack of high-level interactions [BGZ06], in addition to its disability to make decisions for repeatable scenarios. Components need more open and abstract types of interactions since they depend on provided services of heterogeneous entities to accomplish composition requirements, they also need reasoning capabilities (which are already key properties of agent approaches).

Figure 1.3 presents the properties of service, agent and component approaches. Agents offer high-level interactions and behavioural features versus reusability and composition ones of components. One of the key features of service-oriented approach is the interoperability; then, services present pivots of interoperability between agents and components and it presents business abstraction of agents and components.

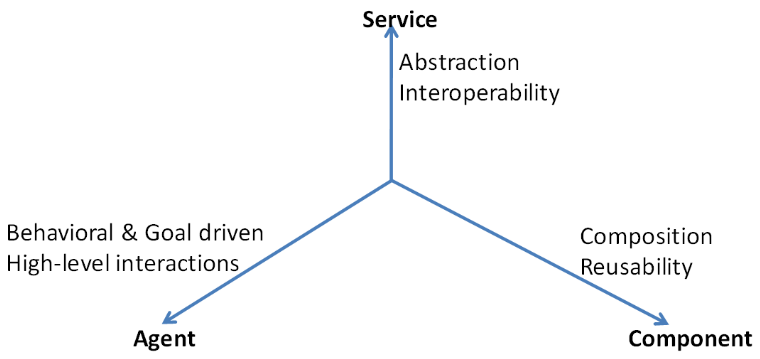


Figure 1.3: Properties of components, agents and services.

1.2 Objective

We already saw that agent and component approaches have their own features and drawbacks. There is a nice complement between the drawbacks of one approach and the interests of the other; therefore we aim to provide an approach that integrates these two paradigms. In other words, our goal is to implement the triangle presented in figure 1.3 to reach making interacting agents and components via services. Most of the current applications are designed according to a single paradigm, i.e., only agents, only components or only services. It would be interesting to use these approaches together to provide an effective paradigm for the development of more efficient distributed applications using interoperable agents and components (through services). The service is the business abstraction of a component or an agent, it represents a pivot to support their interaction in order to integrate their advantages. Therefore, we focus on the concepts of service and interaction as key points, notably regarding cooperation between agents and components.

We can say that the feasibility of specifying an application by interoperable agents and components via services is our main objective. However, we can always specify this application with only services, with only components or with only agents along this research according to the system requirements. A global view of basic elements in this research is provided in our proposed framework [ACG11]. This framework relates the three considered domains (service, agent and component) with the domain allowing the interoperability between components and agents via services (see figure 1.4).

We can recognize the following main elements in figure 1.4:

1. Each domain is represented by a general model, i.e., service, component and agent models. An additional model named **C**omponent **A**gent **S**ervice **O**riented **M**odel (CASOM) is also a part of this framework. This last model allows interoperable agents and components via services in the same application specification.

2. There is a kind of hierarchy between the four models (service, component, agent and CASOM). The service model belongs to a higher level of conception than the

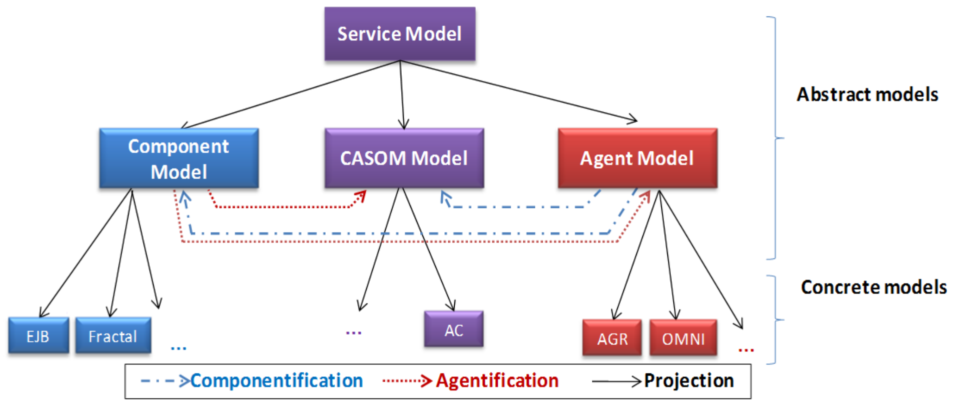


Figure 1.4: Integration framework of service, component and agent approaches.

other three models, although all of them are abstract models as they are not based on any technological platform.

3. The relations between models represented by arrows allow to transform an application specification according to a model to another one. Among these structural transformations, we introduce here the direct arrows between component and agent models *Agentification* and *Componentification* ones. They are inspired from the two methods of combining agents and components in [KMW03]. The *Agentification* (resp. *Componentification*) is the transformation of an application based on components (resp. agents) to an application based on the target agent (resp. component) model. They can also be indirect towards the intermediate model CASOM, by adding agents (resp. components) to the original components (resp. agents).

4. A layer of concrete models exists in each domain (technological targets for each domain), like Enterprise JavaBeans (EJB) of component models, Agent Group Role (AGR) of agent models and AgentComponent (AC)[3] of works allowing mixing components and agents together.

We design our models with the aim of developing an application structured around the concepts of service and interaction implemented by agents and / or components. Therefore , this work addresses the analysis of concepts of the different domains and their mappings according to the structural dimension of an application. The concepts related to the behavioural dimension are present (e.g. interaction protocol, task, operation ...), but their study in behavioural terms is beyond the scope of this thesis which represents a first step towards the integration of service, agent and component paradigms.

[3]These specific models will be detailed lately in the dedicated chapters for each domain in the state of the art part.

1.3 Contribution

In order to realize our objective by implementing this framework, we provide the following contributions.

- We study firstly several existing models of components, agents and services. The originality of this study is by focusing on the two concepts of *service* and *interaction* (since the interoperability between components and agents via services is our goal). Then, we extract shared concepts between studied models in each domain and how the two key concepts of interaction and service exist. We also study several approaches that try to integrate and add values between possible permutation pair of component, agent and service domains or all of them together.

- We also defined the presented framework in figure 1.4. It groups the four studied domains (service, component, agent and the domain mixing the previous three domains) and defines the relation between them, this contribution was presented in [ACG11].

- From the two previous points, we decide to define our unified models of component and agent where the concepts of *service* and *interaction* are explicit and central. We also define our own abstract service model, where elements implementing services (participants) are not considered. Our service model specifies an application as sets of interacted services, this contribution was presented in [ACGA11].

- The third part of our contribution is related to the integration of these approaches through the definition of the intermediate model CASOM. CASOM allows the application specification by interacting components and agents using services.

- After the definitions of the four models, we define mapping rules between their concepts. These rules define the required transformations (arrows in figure 1.4) to move from one model towards another, this contribution except the mappings towards CASOM was presented in [ACGA12, ACG12].

- A design guide is proposed to help the designer in the cases of transformations or application specifications by CASOM requiring his intervention. It contains hints to help a designer in specifying applications by CASOM, in order to choose suitable entities. Moreover, it contains possible variants of transformations from one model to another.

- The implementation of our framework with its models is the last part of our contribution. We implement our models and the transformations between them in an environment supporting the Model Driven Engineering (MDE) [Ken02] principles using the Eclipse Modeling Framework (EMF) [SBPM09].

1.4 Thesis Outline

The second part of this document presents the state of the art of the existing models of each domain with focusing on the two concepts of service and interaction. Component models in chapter 2, agent models in chapter 3 and service models in chapter 4. Lately, we present the approaches that already mix these three domains or at least two of them in chapter 5. This chapter contains the works that consider service and component approaches simultaneously. Then, the approaches that consider service and the agent together before presenting works that consider component, agent and possibly service approaches simultaneously.

The third part of this dissertation proposes our contribution. We start by presenting our framework which defines and introduces the relations between the other parts of the contribution in chapter 6. Then, we give a short presentation of the MDE principles that we use along the contribution. This chapter introduces also our case study of a holiday reservation system as a running example to illustrate all our contribution. We propose our general unified models of service, component and agent in chapter 7. CASOM model that integrates the approaches of component, agent and service is proposed in chapter 8. The definitions of the mappings between the four models are presented in chapter 9. Chapter 10 presents our design guide to help the designer in choosing suitable concepts and mapping variants. The concrete implementation of these models and their transformations are provided in chapter 11.

At the end of this document, we provide a general conclusion and we propose sets of our main perspectives.

Part II

State of the Art and Related Work

The more things change, the more
they are the same

Alphonse Karr, French writer
(1808-1890)

Overview

Since our objective is the integration of component and agent approaches by considering services as a pivot of interoperability and interaction, we need to put the light on already existed models in each domain, with focusing on the two main concepts of interaction and service. We need also to study works that already integrate these approaches jointly or partially.

Many models exist for each domain, however we choose to consider the most representative ones, we extract common concepts between the models, subject of the study in each domain, and the form of the existence of service and interaction concepts (explicit or implicit).

This part starts by presenting some component models in chapter 2 then we browse organizational agent models in chapter 3 and we present service models in chapter 4. Finally, we put the light on the existing works that consider these three approaches jointly or partially in chapter 5.

Like a jigsaw puzzle: you have to make the pieces fit without getting out the scissors.

Dr. Karl Maurer

Chapter 2

Component-based Models

Contents

2.1 Introduction

The term component was proposed by McIlroy in [McI68] where he implemented an infrastructure of the component idea in UNIX using pipelines and filters. Component-based development appeared in the early nineties, where the object-oriented approach failed to cover reusability needs.

Reading and understanding an existing code is always an annoying task for developers, but reusing an existing code on the form of a component is of great interests. In this case, a developer just needs to know what a component does, not how it was developed (done). In component-based approach, we distinguish clearly between component and system developments, where a system development presents the assembly and composition of compatible components, and component development presents how this component is built.

In this chapter, we present some general definitions related to the component-based approaches and we study several component models focusing on the two key elements of interaction and service. The main aim of this chapter and the two following ones (chapters 3 and 4) is to extract shared concepts between the studied models and the form of the existence of the concepts of service and interaction (i.e., explicit or implicit).

We start this chapter by presenting some component definitions.

2.2 What is a component?

Components do not have yet a unified standard definition. Table 2.1 lists some of existing definitions.

Szyperski [Szy02]	*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*
Councill and Heinmann [CT01]	*A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*
Meyer [Mey03]	*A component is a software element (modular unit) satisfying the following conditions:* *1. It can be used by other software elements, its clients.* *2. It possesses an official usage description, which is sufficient for a client author to use it.* *3. It is not tied to any fixed set of clients.*
[OMG05]	*A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. A component conforms to the interfaces that it exposes, where the interfaces represent services provided by elements that reside on the component.*

Table 2.1: Component definitions

From the previous definitions, we define component as a black box with well-specified interfaces. These interfaces are its communication points with the other parts of the system. A component is a reusable and replaceable part in a system. The interfaces describe the services used and provided by a component. A well-defined component allows us to understand its functionality from the specification of its interfaces. Figure 2.1 presents an example of a simple component named *ContactFinder* to find someone contacts. We can use this component *ContactFinder* in different contexts like mobile phones, yellow pages, emails address book or Human Resource systems.

Component-based approach is looking ahead towards the Commercial Off The Shelf component (COTS) [DYC+05, CL00] and their atomic assembly, and atomic substi-

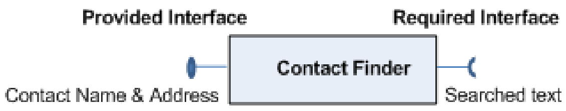


Figure 2.1: A contact finder component.

tutability. Here, the cost of the system and its quality depends on the cost and quality of its COTS.

2.3 What is a component model?

A **component model** *defines standards for defining properties those individual components must satisfy. It defines methods, and possible mechanisms for composing components* according to [CCSV07].

A component must conform to a component model according to [CT01]. The component model defines how to build a component itself and how to interact and communicate with other components in component-based systems. Figure 2.2 presents the component model in [Som04]. It contains three main sub models or three parts. The first part defines the interfaces and their compositions. Usage information and parameters are defined in the second part and the information related to the components deployment and packaging are defined in the third part. We notice that the concept of interaction does not appear explicitly in this model whereas it might be implicit under the composition concept of the interface part.

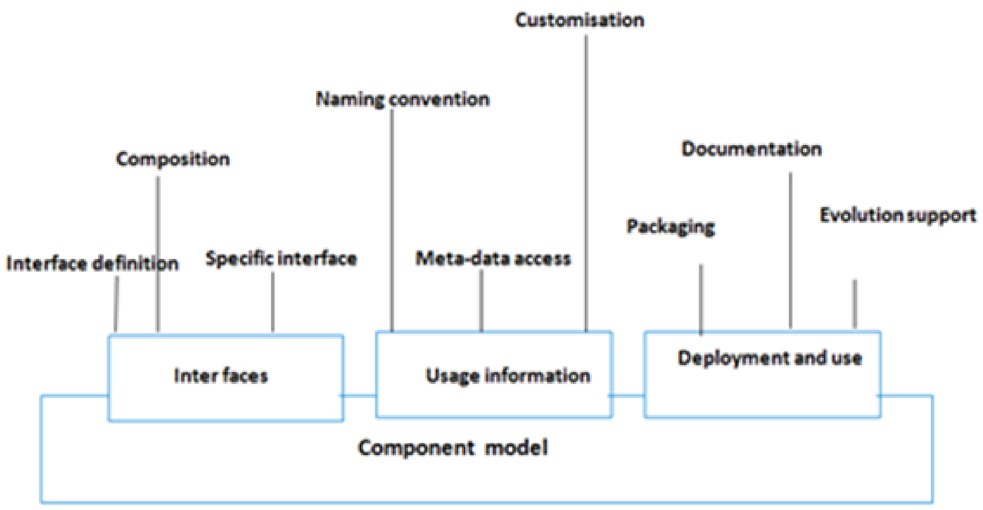


Figure 2.2: The component model in [Som04].

Component models provide ideal choice to build complete systems from the assemblage of reusable blocks. The reusability of component in a changed context is a key property, as components are black boxes and their only visible parts are their interfaces. This property implies the well-defined specification of the component interfaces. The interface describes services/operations provided by a component and needed ones to accomplish the component functionalities.

Components within their models are easy to maintain, support, and modify for future requirements. However, one of their weaknesses is the lack of dynamicity either in component's composition or in its substitution. Then, it is interesting to have different distributed access point for a component since its interfaces are sited in one location as proposed in [Car03]. The interaction between components exists in different ways: by Remote Method Invocation (RMI) which is an extension of Remote Procedure Calls (RPCs) [BN84] or by defining network protocols, which support communications between different components. We may find a connector entity in some component model like WRIGHT [AG97] to assemble the components. A well-defined component can be easily connected to other components to build a composite component that leads in its turn to complex systems. This guides us to another kind of interaction between the composite component and its internal ones that is named delegation. A component delegates task to another internal component. The sender component is responsible for the possible outcomes of the receiver component. Then, the encapsulation aspect is well respected in component-based approaches [BGZ06].

2.4 Component models

We introduce the component models through a simplified classification based on their types of applications. Many works presents component models through classification like [LW07] and [CCSV07]. In [LW07], the authors categorize component models according to their *semantic*, *syntax* and *component's composition during the component's life cycle*. In [CCSV07], the authors classify component models according to their *Lifecycle*, *Construction* and *Extra-Functional Properties*. Since these concepts are not essential in the concerns of our research; we choose to present the component models through a simplified classification related to their domains of applications. The kinds of service and types of interaction between components differ according to the component model domains of application.

Table 2.2 groups the component models according to the domains using it in addition to their types of application. We find four main categories: Architectural Description Languages (ADLs), industrial, conceptual and academicals models.

Category	Models
ADLs	WRIGHT, RAPIDE,...
Industrial	EJB, CCM,...
Conceptual	UML 2.0, EDOC,...
Academicals	Fractal, SOFA, ...

Table 2.2: Our component model categories

There are many other component models related to ADL and categorized in [LW07] as ADL-like languages such as Koala [OLKM00] and PECOS [WZ02]. There are also many models extending UML by defining specific profiles such as Kobra [ABB$^+$02] and Palladio [BKR07]. However, we choose to represent two different, most representative and well-defined models of each category to extract main concept from this category, then in ADLs and conceptual models, we consider implicitly the concepts of the ADL-like models and the UML profile ones respectively.

In the following section, we provide a simple summary of the main properties of our component model categories by studying two models of each category. We need to integrate these models to reach a unified component model. This model must meet our requirements (i.e., the concepts of service and interaction are explicit and central) to precede this research and it must represent all the models that were studied.

2.4.1 Architectural Description Language (ADL)

The early ADLs produced before 1990s [KM85]. ADLs explored ways to model different aspects of software architecture. There are many ADLs, each one is devoted to certain issues and has its ways to solve it. Nevertheless, there are definitely many common points between them. We provide here the most common definition of architecture in software engineering.

Software architecture *is a level of design that involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns* according to [SG96].

From this definition, we can see an ADL as a language used to model software systems conceptual architecture. According to Medvidovic and Taylor [MT00], the ADLs common concepts are: components, connectors, and architectural configurations or systems. In many ADLs, it is hard to distinguish between the specification and the architecture description (there is usually a direct relation with the code).

Different types of ADL already exist like Darwin [MDEK95], Acme [GMW97], C2 [MRT99], UniCon[4], WRIGHT [AG97] and RAPIDE [LKA$^+$95]. We choose to present the last two models because they represent the difference between ADLs, where some ADLs concentrate on the dynamic aspect just like RAPIDE and Darwin, other ADLs concentrate on the structural aspect, where there is no dynamic reconfiguration, such as WRIGHT, Acme, C2.

We present here WRIGHT and RAPIDE models respectively as examples for ADLs. They both belong to a high-level of abstraction and do not suppose a particular relationship between an architectural description and an implementation.

- WRIGHT is proposed by Allen and Garlan in [AG97]. It is an ADL specifies structure and formal behavioural specifications for interfaces between components and connectors. The three main concepts in WRIGHT are:

[4]http://www.cs.cmu.edu/ UniCon/referencemanual/Reference_Manual_1.html

  - Component - It represents an abstract localized, independent computation unit. A WRIGHT component has its port and its computation. The ports (interfaces) are the component interaction points. The complete description of what the component does is presented in its computation that clarifies the relationship between the ports.
  - Connectors - It signifies composition of behaviours for interaction among components. A component can behave differently in interactions according to its role (its responsibilities through an interaction), while the description of the interaction between roles is available in the glue of a connector. The glue itself describes the coordination between the roles activities.
  - Configuration - It views the whole system with its interacted components through connectors. It provides a general view of the system structure (instances of components and connectors and attachments between components ports and connector's roles).

  A main feature in WRIGHT is the use of Communicating Sequential Process (CSP) [Hoa78] notation to specify patterns of behaviour for the component, connector and system as it distinguishes between internal and external choice. An essential advantage of WRIGHT is the translation of the formal interface specifications automatically into CSP. WRIGHT has not direct mappings to implemented systems, it does not support the execution or the code generation.

  The interaction aspect is presented under the connectors to define interacted components in a system and to precise the role of each one. The concept of role exists explicitly, while the service one does not exist.

- RAPIDE is developed by David Luckham at Stanford university [LKA+95]. RAPIDE is designed with highlighting the simulation yielding Partially Ordered Sets of Events (POSET) [Pra86]. This ADL language does not have connectors explicitly. The components communicate through events by calling actions or functions in their interfaces in order to explore the dynamicity of the system. The main concepts in RAPIDE are:

  - Component - It specifies its internal behaviour by an abstraction of the code (the behaviour described by the formalism of POSET (event-based execution model)). The component in RAPIDE consists of two essential parts, an interface and a module. The interface is composed of sets of services that define interactions, a service can be provided or required by a component. The interface has also five elements: (a) type, (b) defined functions by synchronous communication, (c) defined actions by the asynchronous communication, (d) a behavioural part which represents the constraints on execution of the module and (e) constraints to check the validity and conformity of interfaces with the module through the execution. The second part in the component is the module that implements an interface and represents an executable prototype for the component.

- Events - It is generated by action calls. These calls can ask for services or information which specify application behaviour. An event pattern defines the feature of events round between components using types of connection patterns like causality, dependency or independency.
- Configuration - It contains the component instances with connection rules between these instances that allow the verification of events patterns.

A main advantage of RAPIDE is the dynamic reconfiguration. It represents a unique and expressive language for describing asynchronously communicating components.

The interaction is somehow implicit under the communication by events through action calls, at the same time, the concept of role does not exist explicitly since the pattern of events describes how events can be related then how components are related. The concept of service is also implicit in the provided or required interfaces.

Summary: as we saw previously, the main concepts in ADLs are components, connectors and configurations. The component concept in ADLs is treated from an internal view (its functionality) and from an external view (how does it interact with its environment?). ADLs define a component as an architectural unit that represents a primary computational element and data store of a system. The hierarchical composition exists explicitly in lots of ADLs except RAPIDE. The composition is achieved by correspondences between the POSETs in the system interfaces and the component interfaces in RAPIDE.

The connector concept is a first-class concept in most of ADLs but it disappears in some ADLs like RAPIDE and Darwin. The role concept has the same analysis of the connector one, since it is usually related to it or exists implicitly within it in most ADLs. However, it disappears in RAPIDE. The concept of service exists implicitly in the component interface in RAPIDE model but not in the port of WRIGHT model. The last concept of configuration or architecture exists in all ADLs where it allows the reutilization in complex systems.

2.4.2 Industrial component models

The industrial models allow the development, assembly and deployment of software products based on component approach. In this section, we present two industrial models, i.e., EJB (Enterprise Java Beans) [5] and CCM (CORBA Component Model)[6].

- Enterprise JavaBeans (EJB) is a server side component that simplifies the development of java applications based on a multi-tier distributed object architecture. This architecture defines the interactions between the entities (EJB clients, EJB servers, EJB containers and enterprise beans).

[5]Enterprise JavaBeans: http://www.oracle.com/technetwork/java/javaee/ejb/index.html

[6]http://www.omg.org/spec/CCM/3.0/PDF/

EJB allows the developer to concentrate on writing business logic where its architecture defines a standard model for java application servers to support portability (Write Once, Run Anywhere) “WORA”[7].

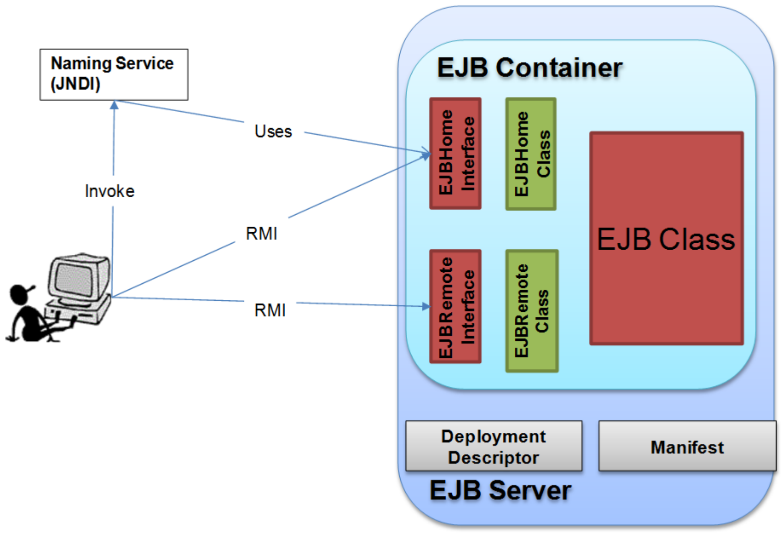


Figure 2.3: Enterprise JavaBeans.

Figure 2.3 presents the main concepts in EJB, which are the bean (component), the remote and home interfaces and the container. Here, we detail these concepts.

The **beans** are java classes that represent the component and they have three main types: *session beans*, *entity beans* and *message driven* beans. A *session bean* is created for each client to execute his tasks. It is the first to access, while the *entity beans* represents underlying data object or context like databases and many clients can share one. *Message driven bean* represents a business process that can be triggered only by receiving messages from other beans.

The EJB components have two interfaces, *home* and *remote* ones. *Home* interface provides remote access to create, find, and remove beans. In other words, all operations related to the bean life cycle. *Remote* interface provides access to the tasks and business methods of a bean.

The EJB containers host the beans and provide services related to its life cycle, like Java Naming and Directory Interface (JNDI)[8], life cycle management, lookups, remote connectivity and security, etc. The major importance of the containers comes from supplying run-time environment for an EJB. The client usually accesses the EJB component by the use of Remote Method Invocation (RMI) in a special type of beans like session bean. This reflects implicitly the notion of interaction, as the first connection starts with the session beans, then the session bean access

[7]http://www.interhack.net/people/cmcurtin/rants/write-once-run-anywhere/
[8]http://docs.oracle.com/cd/B14099_19/web.1012/b14012/jndi.htm#i1084314

the entity bean. The containers provide some services to ensure the deployment of the bean. The concept of role does not exist since there are specific types of beans for specific responsibilities, while the bean itself does not define its services explicitly where they are presented implicitly within the bean's interfaces, there are no provided or required types for the interfaces but home and remote ones.

- CORBA Component Model (CCM)[9] is also a server side component model. It defines distributed components and their interactions. Its components are meta types according to [LW07] which are the extension of the object in Corba [BN95] provided by OMG[10]. The main concepts in CCM are the component, port and component's factory. A CCM component has four ports types (interfaces) (figure 2.4)

  - Facet: The provided port (interface) that exposes component services.
  - Receptacle: The required port and connection point that allows components to "hook" themselves together.
  - Event source: It is a connection point for event production, in other words, it works when a component declares its interest to publish an event.
  - Event sink: The components become consumers of the ones that produced events by declaring events sinks.

  CORBA components are assembled by method and event delegations by the matching between facets and receptacles from one side and event sources and sinks from the other side. We can see that CCMs and EJB are very much similar and they are interoperable between each other.

  Each component has a home or a factory which manages the component type's life cycle. It has its attributes, which are their configurable properties.

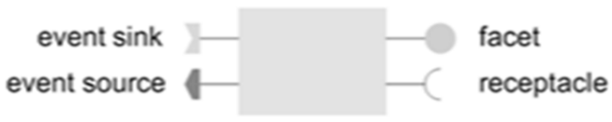


Figure 2.4: A CCM component.

In CCM, we find both deployment and execution models. Deployment model consists of four XML files: a software package descriptor, CORBA component descriptor, component assembly descriptor and property file descriptor. The executable component model consists of component containers, which provide the runtime environment for CCM component instances.

As presented above, the notion of interaction is implicit under the event calls of component's ports. The service concept is also implicit within the provided and required ports (Facets and receptacle).

[9]http://www.omg.org/spec/CCM/3.0/PDF/
[10]http://www.omg.org/

Summary: we browsed industrial component models through a briefly view of the two component models of EJB and CCM. There are other models under this category like COM/.NET [WSW+02]. Component Object Model (COM) have several input and output interfaces with both synchronous operations and asynchronous events. An essential property of COM is the ability of being implemented by different programming languages, in addition to the packaging ability for distributed purposes.

There are many common points between both EJB and CCM models. In fact, both of them are created and managed by homes, run in containers and hosted by component servers. The main concepts in the industrial component models are components, interfaces (ports), homes and containers. The notion of interaction is not explicit here, it exists in low-level technologies like RMI in EJB and method and event calls in CCM. The notion of service in EJB and CCM is implicit within their interfaces nevertheless, we can say that it is more obvious in the CCM model under the ports.

2.4.3 Conceptual component models

These models allow designing the system's architecture. Most of conceptual components are related to patterns [PLR+99] but they can be implemented in different technologies such as EJB, CCM or any programming language. UML 2.0 [11] [CD01] and EDOC [EDO04] are among these conceptual models.

- Unified Modelling Language (UML 2.0) is a modelling language, which respects the methodologies of Booch [Boo95], Rumbaugh [RBP+91] and Jacobson [JCJÖ92]. UML is widely spread because of the large use of its essential diagrams (use case, class, instance, sequential, collaboration, state, activity and deployment diagram) in design phases. A simple component model is defined in UML. A component defines its behaviour through their provided or required interfaces, which implement its services. The ports are the interaction points of the components. A connector is the notion that presents the communication between component instances in the design time. UML contains two types of connectors assembly and delegation. The assembly connector relates required to provided interfaces. The delegation connector forwards operations between interfaces of the same type (two required or two provided ones). The delegation exists usually between the composite components and their internal ones. The designed component in UML can be implemented by any other language.

  As a result of this short review, components (primitive or composite), ports, required or provided interfaces and connectors (assembly or delegation) are main concepts in UML component model. Here the service notion is implicit within the interface and the interaction notion is explicit in connectors. These connectors are generally the lines between two interfaces of the same type (delegation) or between two interfaces of different types (assembly).

[11] http://www.omg.org/cgi-bin/doc?ptc/2003-08-02

- Enterprise Distributed Object Computing (EDOC) is a UML profile proposed by the OMG [EDO04]. This profile simplifies the development of component-based distributed systems by means of a modelling framework. It embraces Model Driven Architectures (MDA) [OMG03], which provides design, infrastructure models and mappings. EDOC is composed of many sub profiles, such as *Component Collaboration Architecture (CCA)*[12] which allows specifying the hierarchical levels of composition, *Entity Profile* describes the concepts for domain application, *Event Profile* specifies the system events and *Process Profile* attaches system's functionality to a specified domain. The component is presented as a process in a business logic design, it is viewed as a *ProcessComponent* in the CCA. Each *ProcessComponent* owns its activities, the flow of activities is specified by the *choreography* of a protocol. The *choreography* defines a sequence of actions (nodes and transitions) in a state machine. The notion of composition is also attached to *choreography* to define ways to assemble the components in order to reach certain objectives.

  The port is an abstraction of component interface, it can be either simple *flow ports* or composites *protocol port* by attaching a role for each port, we can distinguish if this port is an initiator or a responder to business functions.

  The cooperation between components is realized by their connection. A connection connects the compatible interfaces to define a data flow as a logical canal of events. This connection is unidirectional for the simple port and bidirectional for the composite ones. Another type of communication between components is presented by the *PortConnector*, which is the connection point between components and it is defined by its role.

  As we saw above, components (ProcessComponents), ports (simple or composite), choreography, connections and portConnectors represented by roles are from the main concepts of EDOC. The notion interaction is explicit in the choreography, connection and the portConnector, while the notion of service is implicit within component ports.

Summary: from the previous overview, we can see that *UML* is interesting for a general design purposes for component based systems. The interaction concept (connection) between components was presented as assembly or delegating connectors between interfaces, which are general types of interaction. Contrary to EDOC model, the notion of interaction is more detailed with attached roles. The notion of service is implicit in the component interfaces in both *UML* and EDOC models.

2.4.4 Academicals component models

The academicals models provide an open structural frame for component model by providing a hierarchical composition and assuring the system control and dynamicity. They

[12]http://www.synsyta.com/readings/docs/dat/EDOC-CCA-summary.pdf

are generally used and resulted of academic researches in universities. We list here the two models of Fractal[13] [BCS04] and SOFA [PBJ98].

- Fractal is a component model resulted of the collaboration of INRIA and France Telecom in 2002 [BCS04]. It can be viewed as three different models: abstract, concrete and implemented models. In the abstract model, the main concepts are *content* and *controllers*. The component in its turn consists of a functional part named the content; it consists generally from the sub components and a non-functional part named the controller that provides external interfaces to (re)configure internal features. The component with no sub internal components is a base one and it behaves like an object.

  Fractal concrete model represents the condition of creating and binding the component instances such as Fractal ADL[14] where a component is a black box with well-defined *interfaces*. These interfaces can play *server* roles that provide services or *client* roles that describe the services needed for a component. Each interface has its signature which is a file written in the same programming language as the implemented component. A reference for the connection of these two types of interfaces is done by *binding* server and client interfaces. This type of connection can be considered as a simple implicit interaction. *Delegation* is the communication between two interfaces of the same type in a composite component. The import delegation is from an external interface towards an internal interface, the export delegation works in the reverse order.

  Fractal implemented model in Java is named Julia platform[15], it can be implemented also in C on Cecilia platform[16], it is constructed in a programming environment.

  In a general view, the main concepts of Fractal model are components (content and controller) and they can be primitive or composite, interfaces (server or client), communications (binding or delegating). The notion of interaction is implicit under the binding and delegation and there are no connectors. The service notion is also implicit in the components interface which exposes the service provided through the component (server interface) or service used by the component (client interface).

- SOFtware Appliance (SOFA) is a component model proposed as an implementation for distributed system at Charles university of Prague. We view applications realized in SOFA as a nested component hierarchy. A Dynamic Component UPdating (DCUP) is a key propriety in SOFA [PBJ98] that allows changes at run time. The main concepts in SOFA are components, connectors and protocols. A component is an entity that has an implementation and specifications. SOFA represents a component by a frame and architecture. The frame defines the component interfaces with their types provided or required for providing or using services.

[13]http://fractal.ow2.org
[14]http://fractal.ow2.org/fractaladl/index.html
[15]http://fractal.ow2.org/java.html
[16]http://fractal.ow2.org/cecilia-site/current/index.html.

The architecture describes how to make composite components and the component architecture.

SOFA components are defined in an ADL-like language named Component Description Language (CDL) derived from the Interface Description Languages (IDL)[17]. A connector is an entity that achieves the interaction between components, it is defined in the same way as a component, this means that it consists of two parts frame and architecture. The connector frame is defined by a set of roles, each role is a connector interface, which may be attached to a component interface. The connector architecture describes its internal structure (primitive or composite).

There are four connector types: binding between required-provided interfaces of two sub components, delegating between provided interfaces of a composite component and its sub components, subsuming from required sub component interface to the required component one (composite) and exempting an interface of a sub component from any type. SOFA has also a behavioural protocol that gathers all the traces (events) made by a component. The interface protocol generates a behaviour of an interface, when it generates behaviour of its frame called frame protocol. Another type of protocol is named architecture one which is defined by CDLs.

There are two ways to view the interaction in SOFA. The first one is within a connector where the interaction is presented in the same level as a component and the second one is a protocol. The notion of service is implicit in this model through the component's provided and required interfaces.

Summary: we can see that SOFA component model is more general than Fractal, because it describes the component from internal and external views. In addition to the explicit existence of connectors, SOFA helps to reach a good level of interaction between the primitive and composite components using its defined behavioural protocols. We can see that the notion of service is implicit in these two models. A high level of interoperability exists between Fractal and SOFA components, where SOFA application views Fractal components as primitive ones.

2.5 Comparison

Table 2.3 provides a comparison between the basic elements in component models, where the '-' symbol refers to not available concepts.

Components, interfaces (provided, required), compositions and low-level interactions are the most shared explicit concepts between the studied models. There are also many shared implicit concepts between these models, like services through provided or requested interfaces and roles. The separation of component types (primitive and composite) is not clearly defined in all component models, but it exists surely, as the composition

[17]OMG: OMG IDL Syntax and Semantics, ftp://www.omg.org/pub/docs/formal/98-02-08.ps

is a key property in all component models. It may have different forms like in ADLs where composition is presented in system architecture or configuration. The component life cycle is not a central concept in our research although this concept determines whether this component is just for design purposes or it goes further and it has its own implementation.

Figure 2.5 presents all types of interactions in the studied models depending on [CCSV07]. It shows the aggregation as an architectural style of interaction, it may be a horizontal (components assembly) or vertical (here we have a new component resulted from the component composition). These two types share the implicit / explicit binding between different interfaces. However, in the vertical interaction we find also the delegation between composite components and its internal one (import) or the reverse (export).

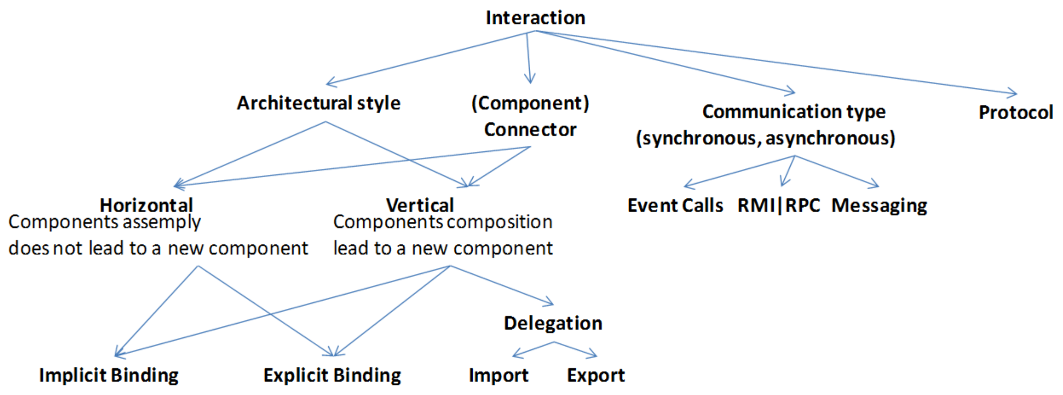


Figure 2.5: Interaction types in component models

Although many models consider exposing and requiring services through component interfaces, we can find in the table 2.3, that the concept of service is implicit in most of the studied models.

We find that the interaction between components is usually done through their interfaces. This interaction allows the exchange of services between components. We can see that the interaction is not presented at the same level of components (not a first-class concept) except in WRIGHT, EDOC and SOFA. In the last three models, we find connectors performing component interactions that can be considered as high-level ones.

Model		Concepts									
		Component		Interface			Composition	Interaction			Role
		Definition	Specif.	Form	Types	Service		Form	Type	Special	
ADL	WRIGHT	It is an independent computation unit. Composite, primitive	CSP	Port	-	-	Configuration, or architecture concept	High-level	Connector	Glue for interacted roles	Implicit under the interactional behaviour
	RAPIDE	It consists of two parts: module and interface. Composite, primitive	-	Interface	Required, provided	Implicit in the interfaces	Configuration, or architecture concepts	Low-level	Events' calling	synchronous, asynchronous and communication	-
Industrial	EJB	It can be a session, entity, or message driven beans. primitive	Java classes	Interface	Home, remote	Implicit in the interfaces	assembled by methods calls	Low-level	RMI	-	-
	CCM	It is a Corba meta types. Server side. primitive	IDL	Port	Required, provided, Event source and sink	Implicit in the interfaces	assembled by method, and event delegations	Low-level	Events' calling	-	-
Conceptual	UML2.0	It is a modular unit. Composite, primitive	-	Port& Interface	Required, provided	Implicit in the interfaces	Composite component by delegation connectors	Low-level , Interface binding and delegation connectors	Delegation and assembly	-	-
	EDOC	It is a process in a business logic design. Composite, primitive	-	Port	Simple, composite	-	Composite component in a CCA profile	High-level	Connector and port Connector	Protocol	Initiator or responder
Academicals	Fractal	It is a run-time entity, it comprises content and a controller. Composite, primitive	Fractal ADL	Interface	Client, server	Implicit in the interfaces	Composite component by delegation and binding	Low-level	Interface binding and delegation	-	Implicit under the interface role of client and server
	SOFA	It is a unit of design, it consists of a frame and architecture. Composite, primitive	CDL	Interface	Required, provided	Implicit in the interfaces	Composite component by different types of connectors	High-level	Binding, delegating, subsuming, exempting Connectors	Protocol	-

Table 2.3: A comparison to extract shared concepts between component models

In several component models, low-level interactions are presented using some technologies like events' calling, RMI or RPC. Other component models may specify the interaction by protocols.

Unfortunately, whenever the concept of interaction (resp. service) is considered as a first-class element in a component model, the definition of service (resp. interaction) is implicit and ambiguous. We take SOFA model for example, we find connectors with the same structure of components, in addition to protocols for interaction, but the concept of service is implicit under its interfaces. This justifies our need to define a general component model lately proposed in chapter 7.3 where the concepts of service and interaction are considered as first-class ones at the same time.

Next chapter browses background in MultiAgent system and main concepts in some studied models of organizational MultiAgent models.

An ideal agent should have a good mixture of autonomy and sociability; Autonomous enough to do things properly even without specific commands. Sociable enough to communicate properly with you and help you do arbitrary tasks.

Pr. Yasuo Kuniyoshi

Chapter 3

Organizational MultiAgent Models

Contents

3.1 Introduction

The agent technology started in the Artificial Intelligence (AI) domain, more precisely in Distributed Artificial Intelligence (DAI) [Nwa96]. The first form of agent was an object that encapsulates its internal states and receives (sends) messages from other objects of the same types named (actors) [Hew77]. In mid nineties, the collective MultiAgent models appeared where agents interoperate to achieve common activities. These models facilitate the build of distributed systems because of the agents' capabilities of perceiving their environments and responding autonomously to their environments changes. Since we focus on the concepts of interaction and service along this research, we choose to concentrate on a specific type of MultiAgent systems, which is the Organizational MultiAgent System (OMAS) [FGM04]. OMAS is viewed as an effective paradigm for addressing the design challenges of large and complex MAS, where organizations are emergent whenever agents work together in a shared environment. Many similarities exist between OMAS and service oriented approaches. They both meet the loose-coupled, flexibility and dynamicity features. Organizations are ways to makeup systems of collaborative services [SH05]. The agent itself can be from different types like interface, information, mobile, collaborative, reactive, etc.

In this chapter, we present some general definitions and models that deal specifically with the organization (models and methodologies) and interaction dimensions. We start by presenting some definitions of agents and MultiAgent systems.

3.2 What is an agent?

Agents like components do not have yet a unified standard definition. We list some of the existing definitions in table 3.1.

From our point of view, an agent is an autonomous entity, which owns certain capabilities that help to achieve its services or to use the services of another agent through an interaction. The agent interacts with other agents and plays certain roles in its organization to achieve a common or an individual goal autonomously or by executing defined tasks. Agents are characterized by the social ability to cooperate, coordinate, and negotiate with each other [Woo09].

The autonomy and high-level interactions are the main points of difference between agents and both component and object approaches. Components (resp. objects) interact between each other by calling methods (RMI) or by sending messages between interfaces via a protocol. These messages may be interpreted in basic ways while Agent Communication Languages (ACL)[18] enables agents to talk the same language with different interpretation.

[18]FIPA-ACL message structure specification. December 2002. http://www.fipa.org/specs/fipa00061/

Shoham [Sho93]	*An entity whose state is viewed consisting of mental component such as beliefs, capabilities, choices and commitment. What make any hardware or software an agent is precisely the fact that one has chosen analysis and control it in its mental state.*
Russell and Norvig [RNC+96]	*An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.*
Wooldridge and Jennings [WJ95, Woo09]	*An agent is a hardware or (more usually) software-based computer system that enjoys the following properties :* • ***autonomy*** *agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;* • ***social ability*** *agents interact with each other (and possibly humans) via some kind of agent-communication language.* • ***reactivity*** *agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it.* • ***pro-activeness*** *agents does not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative.*
Ferber in [Fer95]	*A real or virtual entity, operating in an environment capable to perceive and act on it, it can communicate with other agents, which exhibits an autonomous behaviour, which can be seen as the consequence of its knowledge, its interactions with other agents and goals it pursues.*
Luck et al. [LMSW05]	*An agent is a computer system that is capable of flexible autonomous action in dynamic, unpredictable, typically multi-agent domains.*

Table 3.1: Agent definitions

3.2.1 Typical agent view

The agent from an external point of view is an autonomous, reactive, pro-active and social entity that interacts with other agents (MAS) and with its environment. From an internal point of view, it is a software or hardware entity capable of taking decisions and managing its relationships with other agents. It is also able to control its actions and perceptions from its environment.

The typical agent elements are presented in figure 3.1, where an agent consists of action, perception, communication and reasoning elements. Some elements disappear in certain types of agents. for example, in reactive agent, there are no beliefs neither goals, whereas in Believe Desire and Intention (BDI) agents [RG95], all the elements exist.

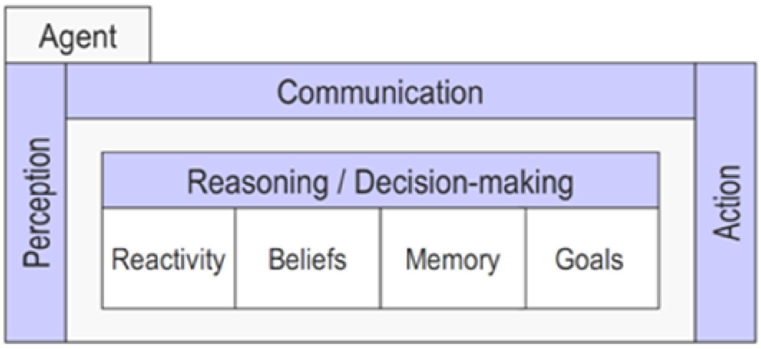


Figure 3.1: Typical agent elements [MT07].

3.2.2 An example of agent architecture

The BDI type of agent views system as a rational agent with certain mental attitude representing the information, motivation and deliberative states of this agent. The system behaviour or performance is determined by the agent mental state [Bra87,RG95,KG91]. This model can be viewed as an agent centered approach, where a system is an agent who owns its believes, desires and intentions. Beliefs can be viewed as the informative block in the state of the system. Desires can be viewed as the motivational block of the system state. It represents information about the objectives to be accomplished or more generally, what priorities or payoffs are associated with the various current objectives. Intentions can be viewed as the deliberative part of the system state. The BDI agent model is represented in formal ways where it consists of five essential sets (believe, desire, intention, events with plan libraries). Figure 3.2 represents a part of BDI meta-model presented in [HMMF+06], where the agent (abstract type of team), role, plan and event are the main concepts. There is a platform dedicated for the BDI model named Jack [HRHL01]. We can see that the BDI model is far from our interests as it focuses on the internal structure of the agent and not on its communication with others to provide or use services. However, we present it since it is considered in the interaction dimension between agents and in a related work considering agents (this model) and services jointly [HJR10].

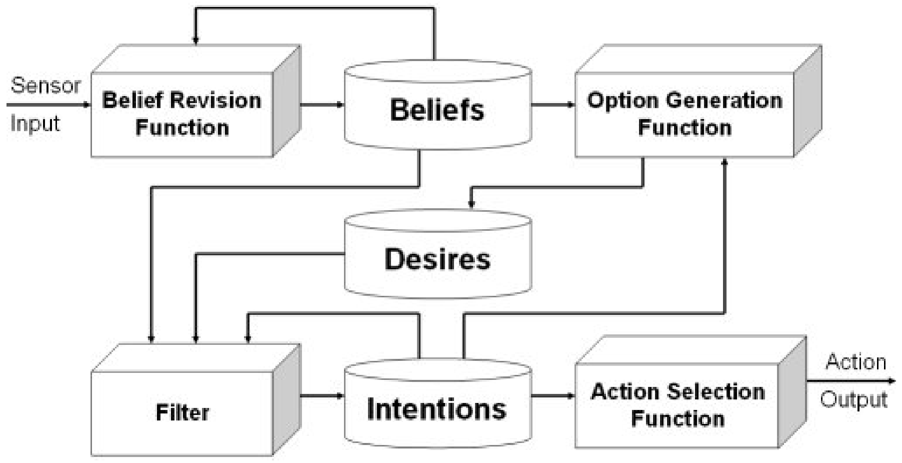


Figure 3.2: The BDI agent architecture [Woo00].

3.3 What is a MultiAgent System?

MultiAgent System (MAS) is resulted from the need of new technologies to build complex distributed cooperative systems in the artificial intelligence domain [Fer95]. MAS contains sets of agents interacting with each other in a common environment possibly forming an organization. A typical view of MultiAgent System is presented in figure 3.3 where we can see, agents perceive and act on their environment and interact with each other. Here is another interesting definition of MASs.

A **MAS** *is a software system made up of multiple independent and encapsulated loci of control (i.e., the agents) interacting with each other in the context of a specific application viewpoint* according to [Cos05].

Ferber defines MultiAgent system as a population of interacting autonomous agents. He focuses on the two aspects of organization and interaction [Fer95].

Demazeau finds that a MAS approach consists of four essential modules Agent, Environment, Interaction, Organization (AEIO) representing the "Vowels approach" in [Dem95, HSB02a].

- **A** defines agent with its internal reasoning capabilities as the core of the MAS.
- **E** is for the environment where the agent interacts and preserves with and from it.
- **I** is for the interaction, it exists between agents and between agents with their environment.
- **O** is for the organization that provides a structure for MAS based on functionality, behaviour and interaction between agents.

In OMASs, the notion of role is also important because it clarifies the participants in an interaction from one side and it specifies agent behaviours and capabilities from the

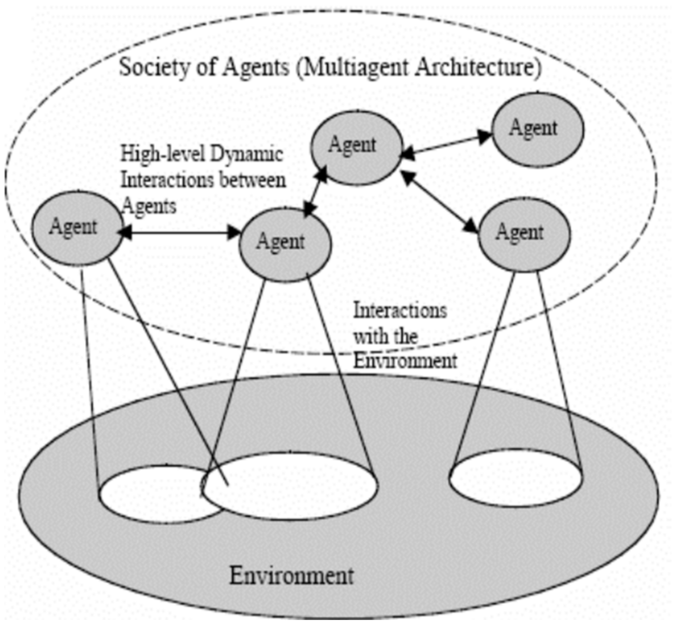

Society of Agents (Multiagent Architecture)
High-level Dynamic Interactions between Agents
Agent
Agent
Agent
Agent
Agent
Interactions with the Environment
Environment

Figure 3.3: The typical structure of MultiAgent system [Woo09].

other side. An agent with the A and E modules would be called an autonomous agent, with the A, E, I modules is a communicative agent and the agent, with the four modules is a social one.

Organization MultiAgent systems are collective where a problem is divided and distributed between many collective agents. However, there are types of MAS called centered agent. This kind of individual agents can take their decisions independently when solving certain problems.

These two types of MAS can be viewed from internal and external views according to Ferber[19]. We have four essential types of MultiAgent systems (see table 3.2): The individual intern that considers the mental states and the architecture of an agent. The individual extern view defines the agent behaviour. Collective MASs from the internal view defines their ontology's, common knowledge and social norms and it represents the organization from the external view.

MASs respond to many questions like how to allocate, decompose defended problems, and how to gather the divided solution? How to reach a high level of connectivity and interoperability between agents? How to maintain coherence in different agent behaviours? How to allow individual agents to represent actions of other agents? How to manage the sharing of limited resource?

Along this research, we consider the two dimensions of interaction and organization in the studied MultiAgent models.

[19]Introduction aux systèmes multiagents: un point de vue intégral, Jacques Ferber, cours/LIRMM, 2005.

	Intern	Extern
Individual	Mental states and agent architecture	Agent's behaviour
Collective	Ontology, common knowledge and social norms	Organization and institutions

Table 3.2: Internal and external views of MASs

3.4 Categories of agent technologies

There is a lack of standardization in MASs classification, it may change according to the context. However, we can always distinguish the two main categories individual and collective ones. It seems to be difficult to define the boundaries between MAS types, there will be always cognitive and reactive agents in each category, which can be viewed by functional or decisional views.

Then, we chose to introduce the studied agent models through a classification that respects our objective of focusing on service and interaction concepts along this research in the organizational and interactional dimensions of MultiAgent systems. This classification contains an organization or structural dimension of MAS that allows building systems of collective services. This dimension contains organizational models and abstract level of OMAS in the meta-models of the methodologies categories. The interaction dimension lists existing types of interactions between agents, (see table 3.3).

Category	Models
Organizational	AGR, MOCA , MOISE/-MOISE+,GORMAS, OMNI,...
Methodologies and General models	Gaia, PASSI, Tropos, FAML,...
Interaction	ACL, Speech act, KQML,...

Table 3.3: Agent technologies

We start by presenting the models of MASs that belong to the organizational category.

3.5 Organizational models

Agents need to be grouped in social groups or grids in order to gain advantages in their current environments to deliver or to use services. The organization notion, that is presented in collective MAS and in the individual ones under the form of self-organization, helps to decide how, when and where to modify the system's structure and functionality. The three modules of the Vowels approach "AIO" exist in the Organization MultiAgent

System (OMAS), while the E module exists in different ways.

Many models exist for OMAS like the Model for Organizational and Componential for MultiAgent (MOCA), this name stands for the name of the model in French (Modèle Organisationnel et Componentiel pour les systémes multi-Agents) [Ami03]. This model separates clearly between the design and implementation levels like the work in [BLHS+09] (that will be present lately), where there are correspondences between the concepts of these two levels such as organization/ group, type of Agent/Agent, capability description/ capability, etc.

OMG and Foundation for Intelligent Physical Agents (FIPA)[20] collaborate to extend UML static models to define dynamic model named Agent UML (AUML) [BMO01]. AUML specifies agent interaction protocols, and represents the agent internal behaviours, but this research was stopped in 2008 due to changes in UML. In this section, we present models that consider agent, group and role concepts as first-class elements in [FG98] and [ONL05], Model of Organization for MultiAgent SystEms [HBSS00] and Organizational Model for Normative Institutions [VSDD05] and Guidelines for ORganizational Multi-Agent Systems [ABJ09], as they are general and well-developed ones.

3.5.1 Agent Group Role (AGR)

AGR is a general model based on the three primitive concepts of agent, group, and role. A group contains defined roles, which can be played by agents. Ferber provides this model [FGM04] as a formal semantic for the Aalaadin model proposed in [FG98]. The core model of AGR is presented in figure 3.4. We can see that an agent can belong to different groups and play different roles.

Here, We detail some characteristics of each concept.

- ***Agent:*** there are no constraints on agent internal and external architectures or on its capabilities in AGR model. This enables the designer to adopt it according to the application requirements. An agent plays different roles within any group and it may be either cognitive or reactive.

- ***Group:*** the group is an atomic set of agents with common properties that interact with each other according to their assigned roles. An agent is a group member when it plays a role within the group. An agent can belong to many groups simultaneously. Groups can overlap and two agents can interact with each other if they belong to the same group [FG98].

- ***Role:*** the role is an abstraction of agent functions or services or even its identification within a group. The relation between an agent and a group is defined by its roles. A role can be handled by one agent while an agent may play many roles. According to [FMBB04], a role is an instance of a role type which is a part of the group description structure and it describes agent behaviour. The role type

[20]The Foundation for Intelligent Physical Agents (FIPA) is a non-profit association concerned with specifying standards for heterogeneous, interoperating MAS.

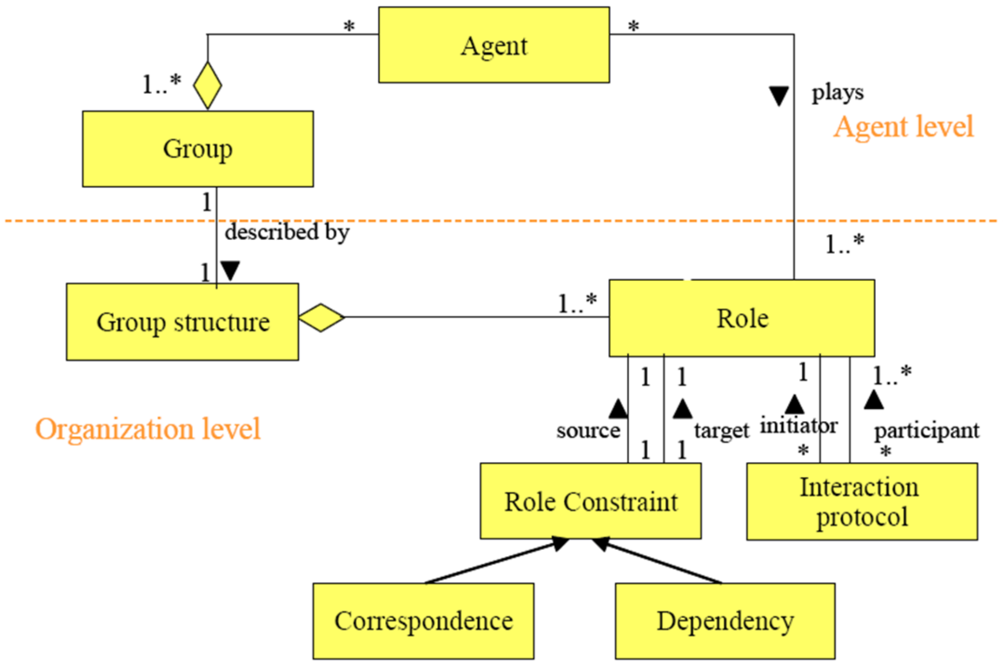


Figure 3.4: The AGR core model [FGM04].

is defined either by the attributes or by the interaction protocols description and structural constraints between roles. The roles ensure the build of social systems based on agents by providing the requirements for agent interactions.

The AGR model has its independent platform named Madkit [GF01]. This platform is quite interesting because of the wide space provided for the designer to determine the agent internal architecture as there is no constraints on it to be violated. This platform is also efficient in distributed MultiAgent systems development [GF01]. At the organizational level in MAS, the designer describes the "what" not the "how", because of the activities patterns structure that the model provides without specifying the agent's behaviour. In other words, the organizational level provides the specification limits of the agent behaviour based on specific rules; it does not define the agent mental state. The organization provides a way to portion the system into groups (parts), which is an organizational unit where its entire members can interact without constraints.

The concept of service exists in implicit way under the role concept, where an agent with a specific role can be a provider or consumer to certain services. The interaction concept is also related to the role, which describes the patterns of interactions and the interaction protocols that relate the elements in an organization.

The role is seen as a border agents performing and using services in Agent Group Role Service (AGRS) in [MF07].

3.5.2 A meta-model of Agents, Roles and Groups

A meta-model of MAS is proposed in [ONL05] where the concepts of agent, group and role are key ones like in [FG98]. Many meta-types exist in this meta-model, for example an *agent classifier* class appears in addition to the concrete agent one. This classifier classifies agents with common features together. The importance of the classification comes from the ability of defining a set of entities that share one or more capabilities. The agent concept contains features that are not common between most of agents and specific provided services.

In this meta-model, a group can be viewed as an agent, there are two types of groups: *agentified* if a group is addressable and act as an agent and *non agentified*. The group represents a certain level of an organization that defines the roles, communication languages and the norms that may be applicable to it. The agentified group contains interaction points to be like agents.

Each agent is linked to other agents by the roles it plays in the application functional requirements. The notion of role appears under the *agent role classifier* which is a specialization of *agent classifier*, this classifier groups the agents according to their roles. Figure 3.5 presents the meta-model of agent, group and role proposed in [ONL05] where we can find the three concepts of group and agent, which are instances of their classifier, and we can see that a group can be a specialization of an agent.

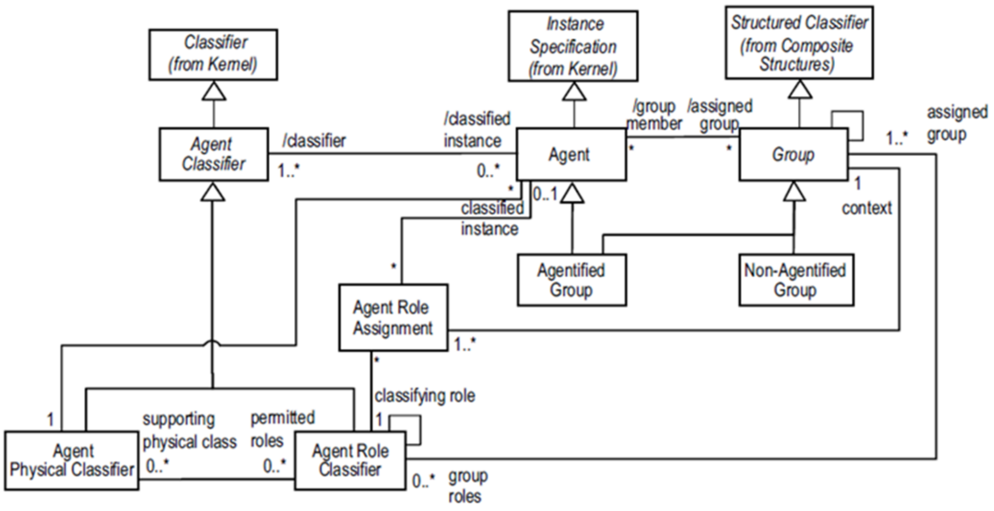


Figure 3.5: The Agent Group Role abstract syntax proposed in [ONL05].

The provided service is implicit under the agent concept, the interaction concept is also implicit and related to the role of agents within a group or the role of agentified group.

3.5.3 Model of Organization for multIagent SystEms(MOISE/MOISE+)

MOISE [HBSS00] is an organizational model that combines architectures that focus on agents with ones that focus on organizations.

MOISE model is formed of two important levels: Organizational Structure (OS) and Organizational Entity (OE). An OS is a set of roles, groups and all the lines that participate in defining the system structure independently of its included agent. Agents' population that function under the OS constraints are the OE. These two levels facilitate the representation from the design point of view, even if the dynamic control of the organization or the exact semantics of a role instance or group is not defined, but we can see the OE as an instance of an OS for a set of agents.

Figure 3.6 presents both of OS and OE, we find in the OS types of groups with their roles related to schemas. A schema is a tree of goals and missions to be reached. In the OE level, we find the instances of group and schema, with the missions, role players related to agents.

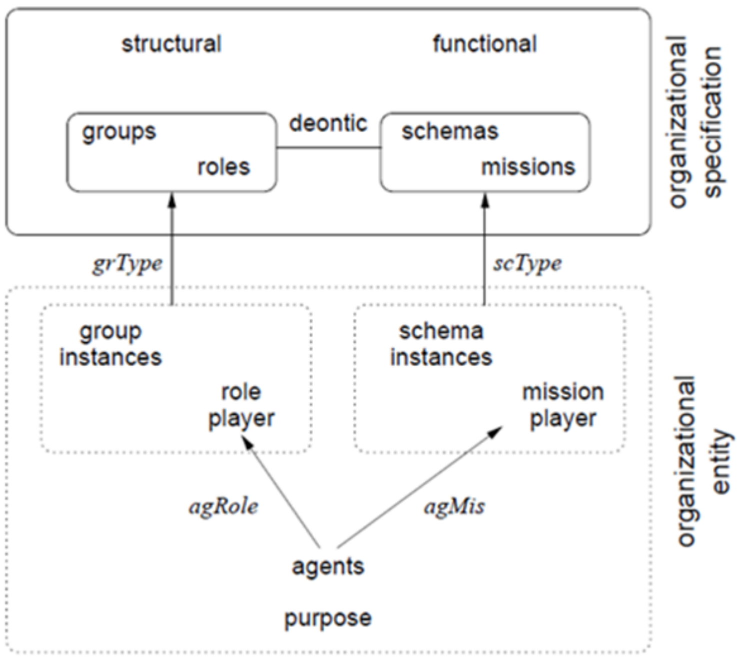


Figure 3.6: The organizational specification and entity [HBSS00].

MOISE+ [HSB02b] is an extension of MOISE model with many contributions, such as the reorganization process and the appearance of role inheritance. The main concepts in MOISE/MOISE+ are: roles, organizational lines which represent the relation between roles and groups. Here, we detail some of these concepts.

- ***Organization*** acts as a system of rules which constrain the activity of agents, or more precisely their individual possible actions. The expression of the rules is made by the concept of role, which defines expected behaviours of an agent in its social structure.

- ***Group*** is simply defined as an aggregation of roles, i.e., a set of consistent rules. This consistency is provided by specified links between roles, which express possible communications, expected acquaintances or relations of authority between roles.

- ***Agent*** is specified as a responsible for a part of the whole task of the application and it owns its resources to achieve some actions.

- ***Role*** defines agent's behaviour and the provided or required services of an agent. The originality of MOISE model for this concept is by the ability of considering the role as a set of missions, and the agent that plays a role must follow its missions.

- ***Mission*** is a sub concept related to the role one, an agent that plays certain roles must archive certain missions. These missions define the constraints and behaviours to achieve a task. They contain the authorization for the four element of any behaviour (objective, plan, action and resource)

- ***Organizational line*** structures social exchanges between roles. This relational line has its source and target roles and it is labeled by N elements (set of constraints, set of missions of the source and the target roles, which define the context of a mission). In MOISE, the organizational line can have one of these types: communication lines, authority lines and acquaintance lines. The communication lines structure information exchanges; it defines also the interaction protocols, while the authority line defines the control and the sequence from one role to another, the acquaintance line defines the agent view for the other agents in the same organization to use it in its justification.

We find the concept of interaction explicitly in the organizational lines between the roles that may define protocols while the service concept exists implicitly under the role, agent behaviour and mission concepts.

3.5.4 Organizational Model for Normative Institutions (OMNI)

This model balances the organization needs and the agent autonomy.

There are three essential dimensions for this model: organizational, normative and ontological [VSDD05]. Figure 3.7 represents the different levels and dimensions of OMNI.

- ***Organizational dimension*** contains the organizational model, which specifies the organizational characteristics of agents society, in terms of social structures (roles) and interaction ones. It contains two models social and interaction models. It has the same main concepts of group, role, agent and interaction like previously presented models. It contains additional concepts like norms.

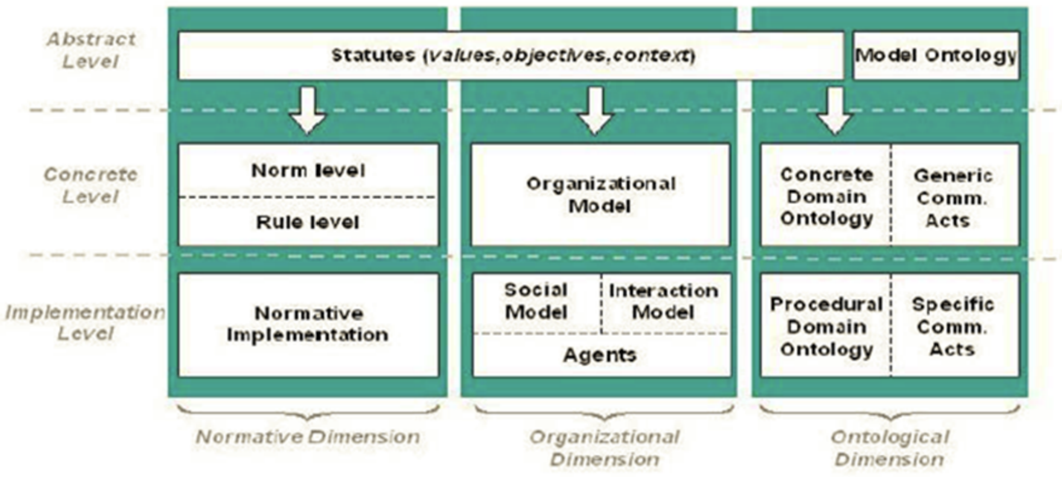


Figure 3.7: Levels and dimensions in the OMNI framework [VSDD05].

The agreements between agents are described in an interaction contract. The contract defines the activities of agents which play roles in the society, the period of the contract validation, the condition and what are the sequences in case of violation.

- ***Normative dimension*** specifies the mechanisms of social order, in terms of common norms and rules that the members are expected to adhere to. The standards set at the organizational rules are transformed into interpretable by agents through mechanisms of interpretation. There are two ways to define the interpreter: either we create an interpreter that all agents should possess, or we transform the rules into protocols which will be part of contracts of interaction.

- ***Ontological dimension*** defines environmental and contextual relations and communication aspects in organizations. By defining the entire concepts according to the norms: rules, roles, groups, violations and penalties. It defines the act of the languages in ACL in order to clarify the content of the interaction.

Our interests is more restrictive in the organizational dimension, we can see from the figure that this last dimension contains objectives in its abstract level, the organizational model in its concrete level and the agent with the interaction model in its implementation level. The concept of service exists implicitly in the organization dimension under the role concept. The interaction has a dedicated model with explicit contract concept, this contract contains all what we need to deal with other peers by providing or using services over agent interactions. The ACL language is also used in this model and it is more clarified in the ontological dimension.

3.5.5 Guidelines for ORganizational Multi-Agent Systems (GORMAS)

GORMAS [ABJ09] extends the ANEMONA meta-model [BG08] in order to contain the concepts of organization unit, service and norm. The ***organization meta-model*** of this model defines the entities of the system (agents, organizational units, roles, norms, resources, applications) in its structural view. In this view, we find the static elements of an organization like the "A-Agent" concept (abstract agent). This concept is extended by the new concept named "Organizational Unit" in addition to be specialized by an agent. The organization meta-model defines also how the previous static entities are related (roles and A-Agent social relationships) in its social view. This meta-model defines in its functional view, the internal and external behaviour of the organizational unit. The last view of this model is the dynamic one, where it defines the services of the organizational units that manage all structural and dynamic elements.

In addition to the organizational meta-model, there are five other meta-models, the ***activity*** meta-model focuses on the system functionality by defining service profiles. The interactions for the usage of the services are detailed in the ***interaction meta-model***. The resources and the permission of their usage are detailed in the ***environment meta-model***. The concrete agents, their responsibilities, objectives, services, tasks and played roles are detailed in the ***agent meta-model***. The normative objectives that agents and roles must follow are defined in the last ***normative meta-model*** including the permission, prohibition, obligation and rewards. A dedicated meta-model to describe the interactions and service exist in GORMAS.

Figure 3.8 presents a unified and simplified view of GORMAS meta-model. We can see the concept of service explicitly where each service owns its profiles and ports. The service is specified by tasks and service implementation. The interaction concept is also explicit and it contains an attribute named protocol.

After the short presentation of various organizational models, we present main concepts of meta-models of some agent-oriented methodologies.

3.6 Methodologies and general models

We browse here some meta-models of well-known methodologies. Most of these meta-models contain the same main concepts as the previously presented organizational models. We start by defining the methodologies.

A **methodology** *is a collection of methods covering and connecting different stages in a process. The purpose of a methodology is to prescribe a certain coherent approach to solve a problem in the context of software process by preselecting and putting in relation a number of methods* according to [GJM91].

A **method** *prescribes a way of performing some kind of activity within a process, in order to properly produce a specific output, i.e., an artefact or a document starting from a specific input* according to [CCZ05].

A methodology has two important components: the first one describes the process elements of the approach, and the second one focuses on the work products and their

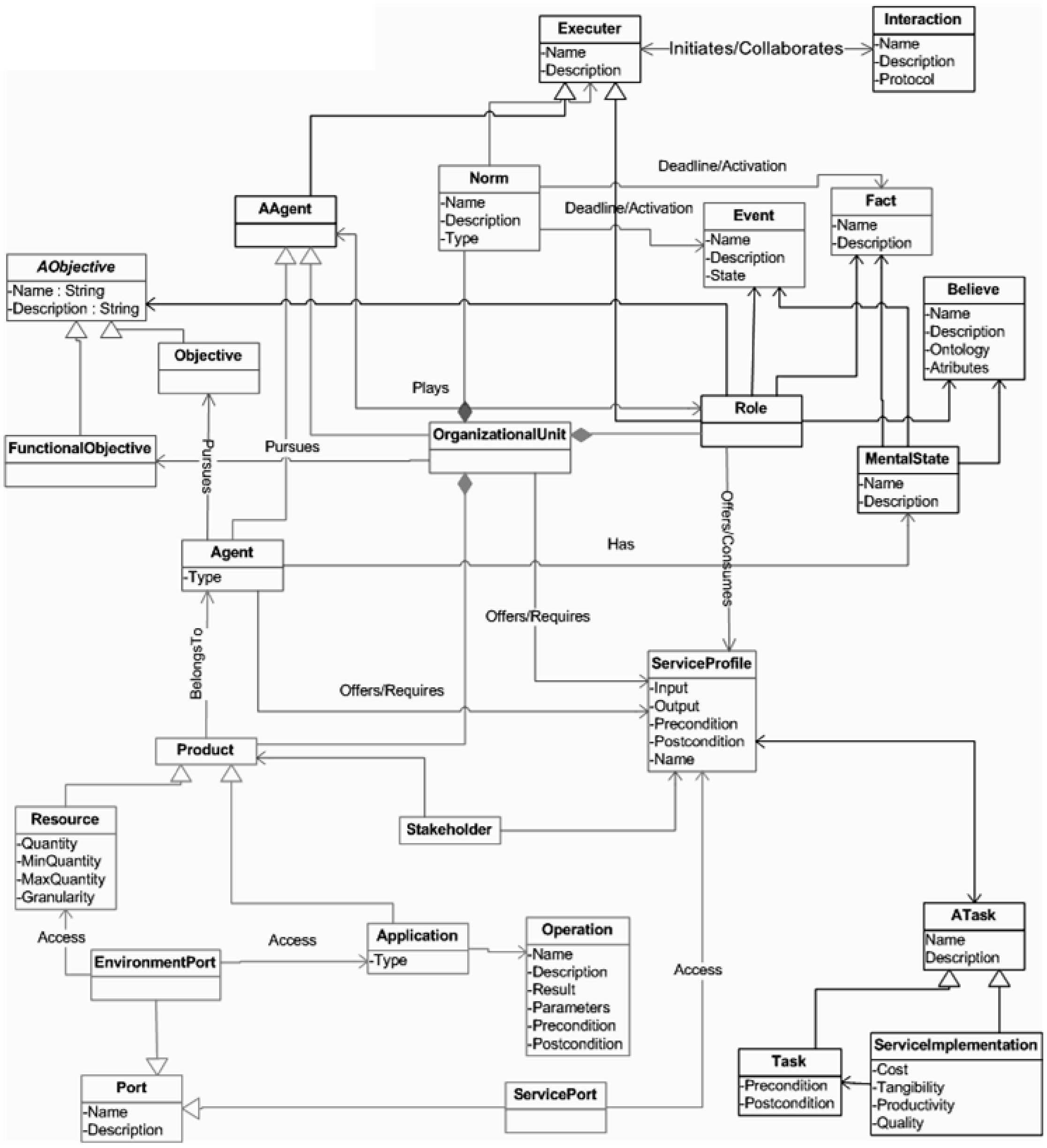


Figure 3.8: The unified and simplified meta-model of GORMAS.

documentation.

Most methodologies define meta-models to clarify and unify different abstraction in MultiAgent system. Here, we browse briefly some well-known methodologies meta-models of Gaia [ZJW03], Passi [Cos05], Tropos [BCP05] and INGENIAS [GSP02] before presenting a general FAML model [BLHS+09] that is based on these methodologies.

3.6.1 Gaia

Gaia methodology appeared in [WJK00] for the analysis and design of closed MultiAgent systems. However, it is extended in [ZJW03] to consider open MultiAgent systems. The Gaia methodology builds the MAS of organization/society that consists of set of roles that are later assigned to agents. These roles define patterns of interactions.

The organization defines the organizational abstraction like the environment in all its entities and resources that are used whenever agents interact to achieve the organizational goals. Roles and interactions are detailed lately in their dedicated models. The organization defines also the organizational rules that define the functionalities and capabilities required by an organization.

It relies on two modeling levels from abstract to increasingly concrete concepts, in the more abstract level, corresponding to the analysis step that allows the specification of **role** and **interaction** models. The concrete level corresponds to the design time, which defines the models of **agent**, **service** and **acquaintance**.

- The ***role model*** describes different roles of a system. A role in Gaia is an abstract description of a function. It is defined by four elements:
  - Responsibilities that define the role and its functionalities;
  - Permissions that are the affected rights to a role;
  - Activities that perform the need calculations to realize the responsibilities;
  - Protocols that symbolize the relationships between the roles. They are defined in a model of interaction.
- The ***interaction model*** is composed of protocol with inter-role interaction definitions. The protocols account for patterns of interaction. The protocols are defined by initiators, interlocutors, the inflows, outflows and a textual description to ensure an interaction.
- The ***agent model*** consists of identifying the types of agents and the instances of agents that will realize these types in the execution phases. The type of agent is the set of roles that can be played by this agent.
- The ***service model*** describes the services provided by each type of agents, by describing the activity associated to each agent role. A service corresponds to a function in Gaia.
- The ***acquaintance model*** defines simply the acquaintance of the existing communication links between types of agents in the form of directed graph.

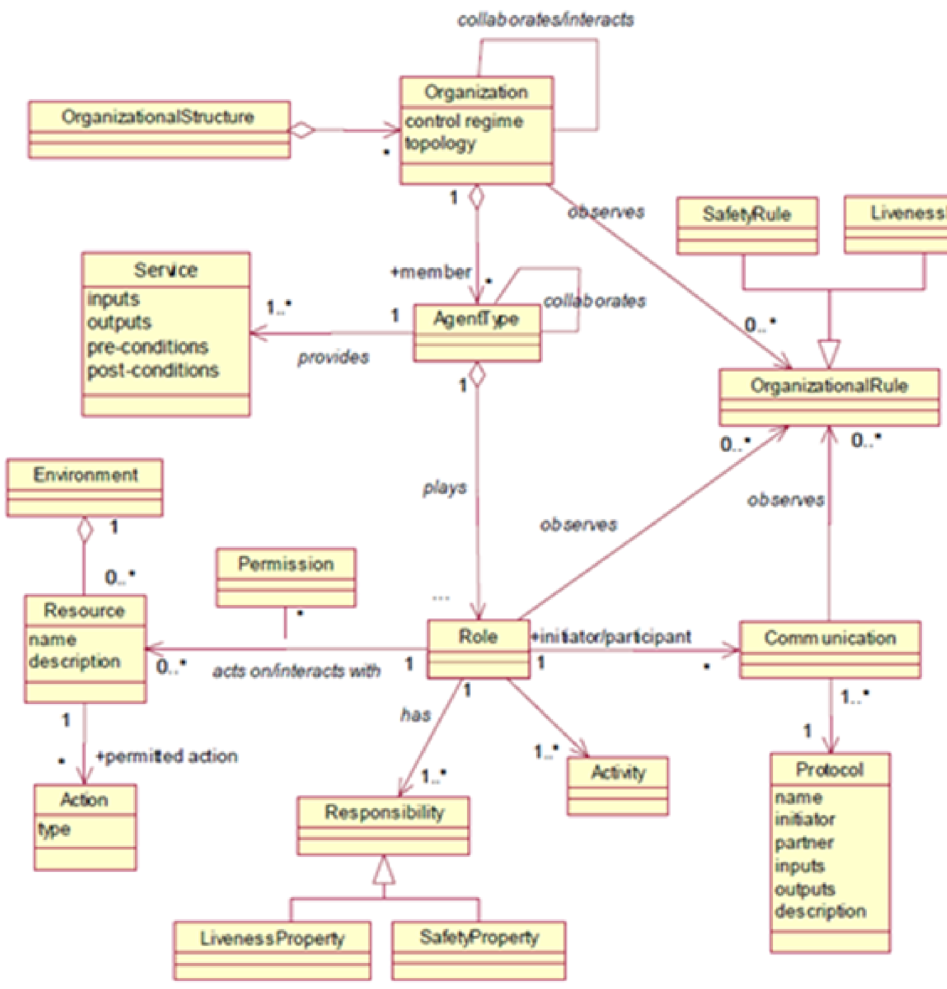


Figure 3.9: Gaia meta-model [BCP05].

Figure 3.9 shows that an organization in Gaia is an aggregation of agents and it is a part of an organizational structure. The agent provides services and plays roles through its interactions. The service concept is explicit in this methodology and it is a single and coherent block of activity in which an agent may engage. The interaction concept is also explicit by defining the dependency and relationships between roles in terms of protocol definitions. This methodology does not deal with implementation level at all, it can be implemented by any agent platform.

3.6.2 PASSI

Process for Agent Societies Specification and Implementation (PASSI) [Cos05] covers all the phases of an application development from requirement-to-code and testing methodology. It extends UML concepts to fit to agent designs, and it uses FIPA platform for

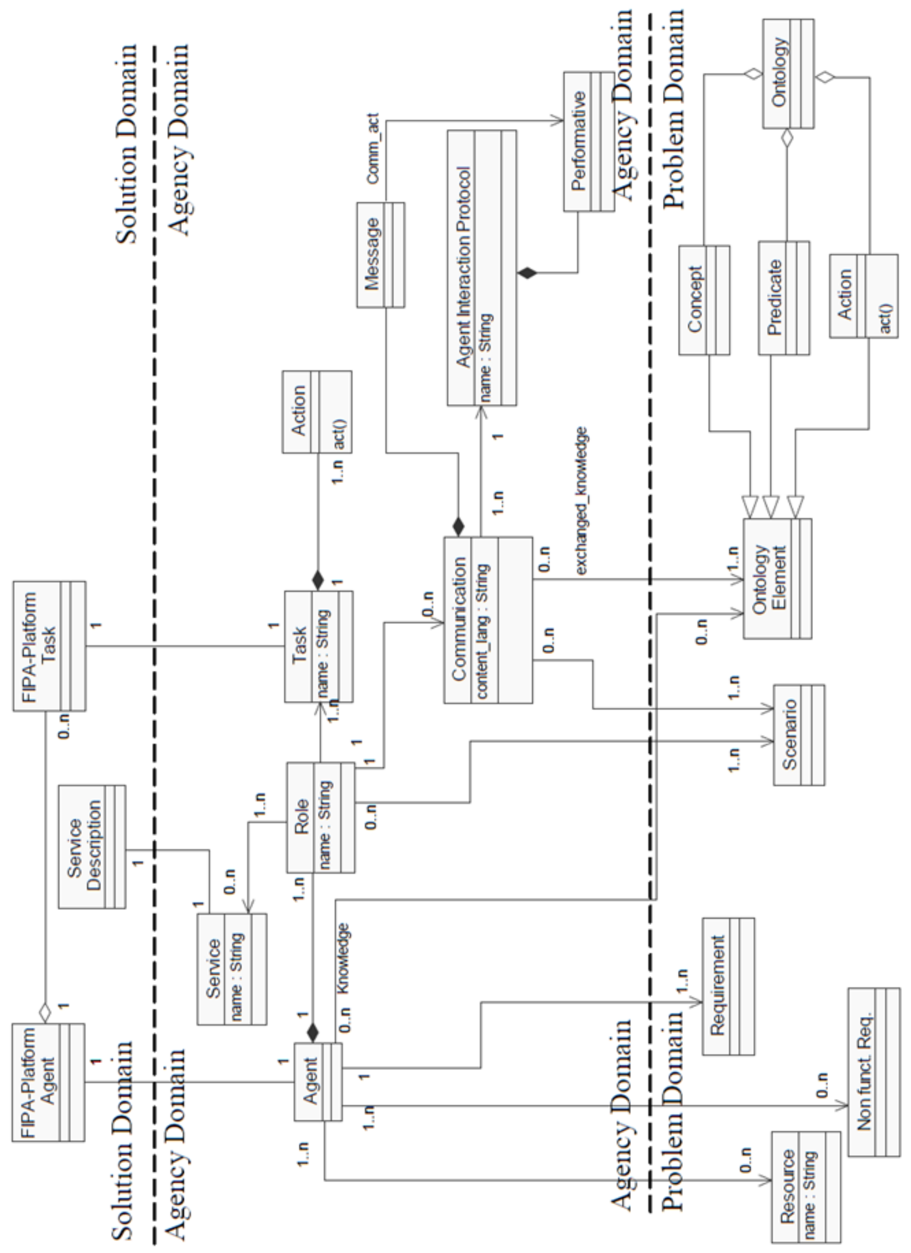


Figure 3.10: PASSI meta-model [BCP05].

the implementation of the agents.

This methodology contains three domains (see figure 3.10).

- The problem domain, which is related to the requirement and analysis phase. In this domain we find the context related information like scenarios, resources, requirements and ontology;

- the agency domain that defines the concepts of the agent model related to the requirement which is related to concepts in the solution domain;

- the solution domain or the implementation in FIPA agent platform.

The core of the meta-model of this methodology exists in the agency domain. We can find here the concepts of agent, role, service, goal, task, communication and agent interaction protocol.

The service in this model has the same signification of service in Gaia, it is associated with each agent role. The interaction is a pattern of conversations used to perform some general useful tasks. It is a dialogue patterns and networking protocol to refer to underlying transport mechanisms such as TCP/IP by sending messages.

3.6.3 Tropos

Tropos [BPG+04] methodology covers also all phases of a software development process. It focuses on the interactions of agents and their environments.

The originality of this methodology is the use of *actor* concept as a generalization of the *agent* [BPG+04]. An actor can be physical or a software agent as well as a *role* or a *position*. The position concept reflects the earlier identified positions as agents occupy the same one. An actor can reach its *goals* by agreeing a *plan* and/or by the use of the *resources* in an environment. There is a *dependency* relation between agents in order to satisfy their own goal or to access a resource (see figure 3.11).

The concept of service does not exist explicitly in this methodology, while it contains social patterns in considering the social and intentional aspects.

Another interesting methodology named INGENIAS is proposed in [GSP02].The meta-model of this methodology contains the most shared concepts in the previously presented models and methodologies like organizations, groups, agents, tasks, interactions and goals. It also contains other concepts like workflow, application and resource [BCP05]. The main particularity of this model from our point of view is the ability of having groups containing other groups, which represents the composition of groups. We did not find such ability of groups' compositions in the earlier studied models.

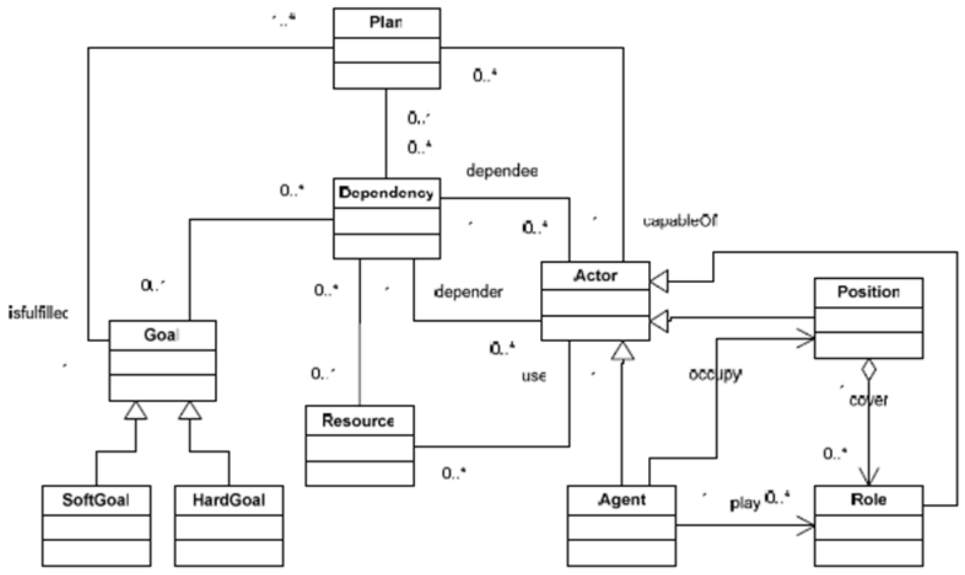


Figure 3.11: Tropos meta-model [BCP05].

3.6.4 FAME Agent-oriented Modeling Language (FAML)

The main objective of [BLHS+09] is to provide a unified meta-model from a combination of existing meta-models in FAML[21] (see figure 3.12).

The FAML meta-model is built in incremental way after studying several methodologies. There was a first version without the concept of service, then this meta-model is refined by considering Gaia and PASSI to include the service concept explicitly in its current version. This meta-model considers both of design and run times. In the design time, we find the organization concept that is a collection of the autonomous rational agents. These agents play certain roles that specify their behaviours. These behaviours are more detailed through defined tasks. The service concept is explicit in this model. It has also the same signification in Gaia as a coherent block of activity of an agent.

The interaction concept exists explicitly in this model in the design time under the *Interaction Protocol* concept which defines the possible pattern of communication in the system. Then the two key concepts of service and interaction exist explicitly in the design time model.

3.7 Interactional dimension

Agents need to speak a unified language to increase their social ability and to interact between each others. The interaction between agents is always a benefit for this approach.

[21]Framework for Agent-Oriented Method Engineering (FAME) is the project name under which FAML has been developed.

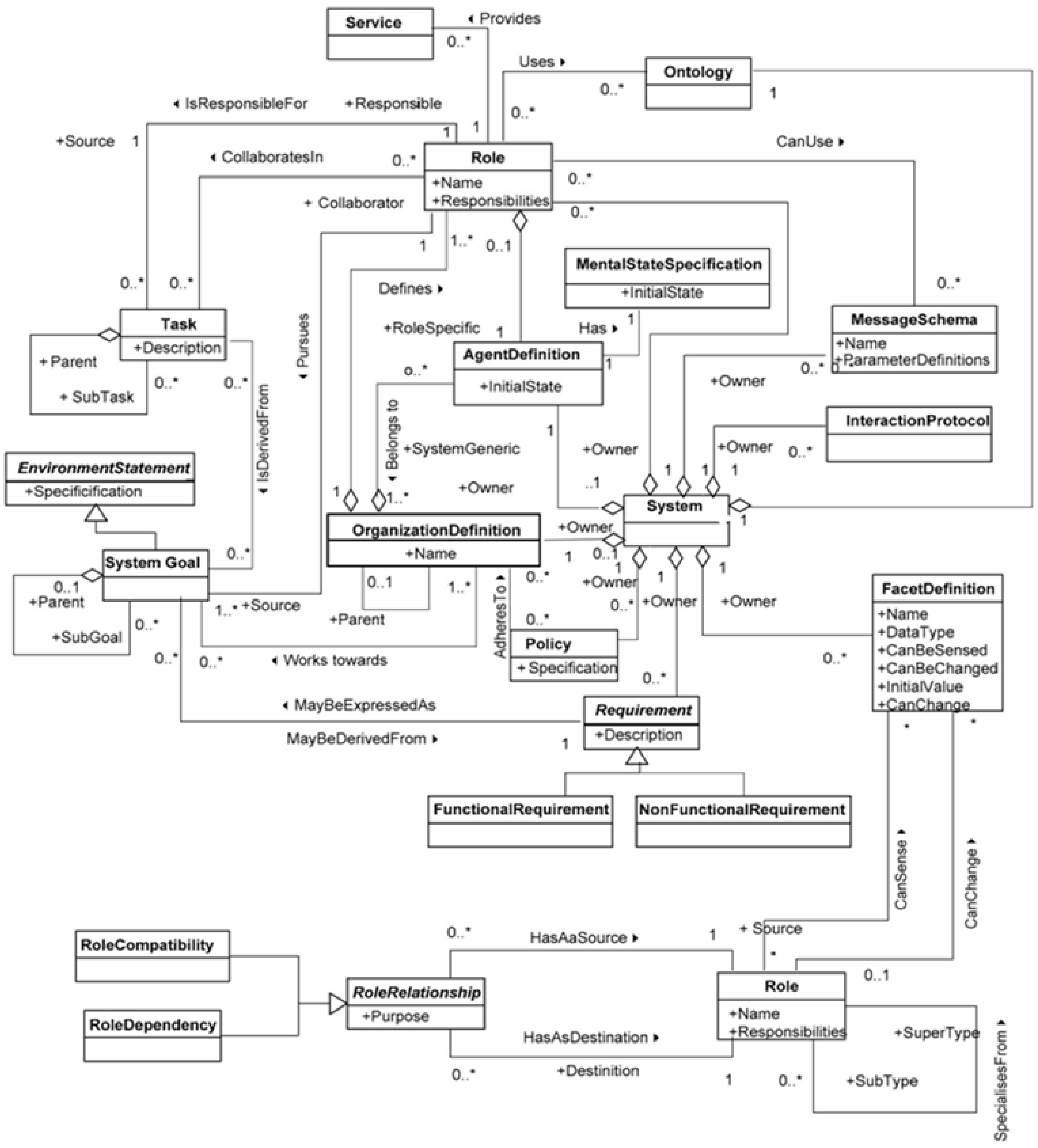


Figure 3.12: Design-time agent-external classes of FAML meta-model [BLHS+09].

Figure 3.13 represents the essential types of interaction in MAS, which are communication, like MTP, Speech act, KQML, FIPA ACL. Coordinations are like contracting, planning and organization structure and negotiations are like dialogue and market place protocols.

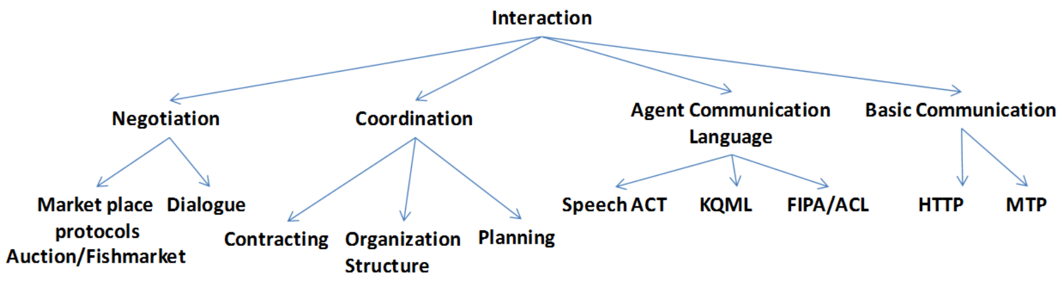


Figure 3.13: Interactions types in agent models.

3.7.1 Communication

The most basic type of communication is proposed by FIPA that specifies Message Transport Protocol (MTP) [22] over HTTP[23] protocol. In this protocol, the sender agent sends HTTP request, which contains agent message including the message envelope. The receiver agent sends HTTP response after parsing the message according to the information in the message envelope.

However, the ideal mode of interaction between entities is reached when these entities talk the same language. Any language consists of three main parts: syntax, semantics and pragmatics. Agent communication language has passed through different forms to reach its final stage. Here, we represent a brief view of Speech act, KQML and FIPA ACL languages.

3.7.1.1 Speech act

Speech act theory is the pragmatic theory of language that views human natural language as actions like requests, suggestions, commitments, and replies. This theory is also related to the theory of Austin in [Aus62] where the communication is an action, and everything we utter is in order to satisfy certain goals. It is the base for almost all agent communication languages and it depends on primitives. Speech act message is a tuple like the following:

```
<i, act (r, C)>
```

[22]http://www.fipa.org/specs/fipa00084/SC00084F.html
[23]http://www.w3.org/Protocols/rfc2616/rfc2616.html

Where i is the initiator of the speech act, act is the name of the act (like confirm, send, receive, etc.), r is the receiver and C is the semantic content. The Speech act considers three aspects in its message:

- ***Locution*** statement with context and reference of the sound of the speaker, i.e., who said what to who.
- ***Illocution*** act of conveying intentions, i.e., what the speaker wants from the listener.
- ***Perlocution*** actions that occur as a result of the illocution. It is not always clear to recognize the intention of the speaker.

There is a strong relation between the BDI model of agent and Speech acts where all the acts are uttered to achieve a goal. They respect some expression rules and they reflect a certain mental or psychological state of the speaker [RL90]. The Speech act defines the type of message by using the illocutionary force but the message content may be ambiguous.

3.7.1.2 Knowledge Query and Manipulation Language (KQML)

The KQML is the first try to standardize Agent Communication Language (ACL) [LF97] and it is based on Speech act. It is developed by the Defence Advanced Research Projects Agency (DARPA) for knowledge sharing initiative in 1990s. It is a language and we can say that it is a high-level communication message-driven protocol. It is independent of the semantics (ontology) and of the content (there are many languages dedicated for expressing the message content such as: FIPA SL[24], KIF[25],. . .).

KQML defines different acceptable performatives. Each performative represents a speech act associated to semantics, protocols and a list of attributes. These performatives can be classified in seven basic categories: basic query performatives (evaluate), multi response query performatives (stream-in), response performatives (reply), generic information performatives (tell), generator performatives (standby), capability-definition performatives (advertise), networking performatives (register).

KQML is written in Lisp syntax. The key property of KQML is that all information for understanding the content of a message are included in the communication itself. Here is an example of a general KQML message.

[24]http://www.fipa.org/specs/fipa00008/SC00008I.html
[25]http://www.fipa.org/specs/fipa00010/XC00010C.pdf

```
(KQML-performative
:sender <word>
:receiver <word>
:language <word>
:ontology <word>
:content <expression>
...)
```

This example can be instanced like,
(stream-about
:sender agent1
:receiver agent2
:language KIF
:ontology motors
:reply-to q1
:content m1)
where *reply-to* is a label to show that the message *m1* is the response of agent *agent1* to *agent2*. This message is written by *KIF* language in an ontology related to *motors*.

3.7.1.3 FIPA ACL

It is the standardization of FIPA for Agent Communication Language [fip02b,fip02a]. It has almost the same syntax of KQML. It can be considered as an extension of KQML as it contains more performatives. The FIPA defined a Semantic Language **FIPA SL**, but there is no commitment with it (in other words, it may use KIF or any semantic dedicated language for the message content). The message in FIPA ACL separates between the messages and envelopes. Each FIPA ACL message can be mapped to a formula of "Semantic Language" which defines constraints that the sender must satisfy and purposes of the action. Here is a simple example of an ACL message in an auction, agent1 proposes 150 dollars as a price for the product good02 in the fourth round of the auction.

```
(inform
:sender agent1
:receiver auction-server
:content (price (bid good02) 150)
:in-reply-to round-4
:language SL
:ontology auction)
```

After presenting communication types of interactions, we present in the following section, the coordination types.

3.7.2 Coordination

The coordination is a kind of interaction between agents performing some activity in a shared state to ensure the coherency of their community. The degree of coordination is the extent to which they avoid irrelevant activity like deadlocks and conflicts. Nwana et al., in [NLJ96] overview the coordination in three main categories: contracting, organizational structure and planning. Here, we present these kinds of coordination.

3.7.2.1 Contracting

It determines the tasks allocation for agents. Contract Net is a well-specified protocol that represents this type of coordination.

Contract Net Protocol (CNP) is defined for the coordination and negotiation between agents in [DS83]. FIPA specified this protocol [fip02c], where it contains two essential roles for agent (manager and contractor). The manager requests persistently from other agents by sending call for proposals (cfp) associated to a defined deadline in case of no answers received. These calls contain the conditions and the task specification by the manager.

The participant agents that receive these proposals can be considered as potential contractors, which are able to achieve the proposed task as acts. As the agents are autonomous object, the potential contractor may refuse to propose a proposal or it may define its own precondition to achieve the specified task.

The manager receives back and evaluates proposals of entire contractors. Then, it chooses the agent(s), which are going to achieve this task by sending them an acceptance message, while a rejection message is also sent to the non-chosen agents.

The accepted agents ask for a commitment by contract to perform the task, and after accomplishing it, they send a message to the manager to inform it. Figure 3.14 represents the principle steps in the interaction using this protocol. Huhns and Singh [HS94] note that the contract net is a high-level coordination strategy, which also provides a way of distributing tasks and a mean for self-organizing groups of agents.

3.7.2.2 Organizational structure

The definition of agent responsibilities and capabilities exist implicitly in the organization, where the role concept is a key one in defining the activities and interactions between agents. The organization identifies the authority between the roles of its elements. The authors in [NLJ96] discuss the coordination through organizational structure in two main organizations. The hierarchical structure defines master/slave relationships between its agents for resource allocation. Master agents are fully autonomous contrary to slave agents which are partially autonomous, this is against the principle of autonomous agents where the slave agents are controlled. The second organization structure is based

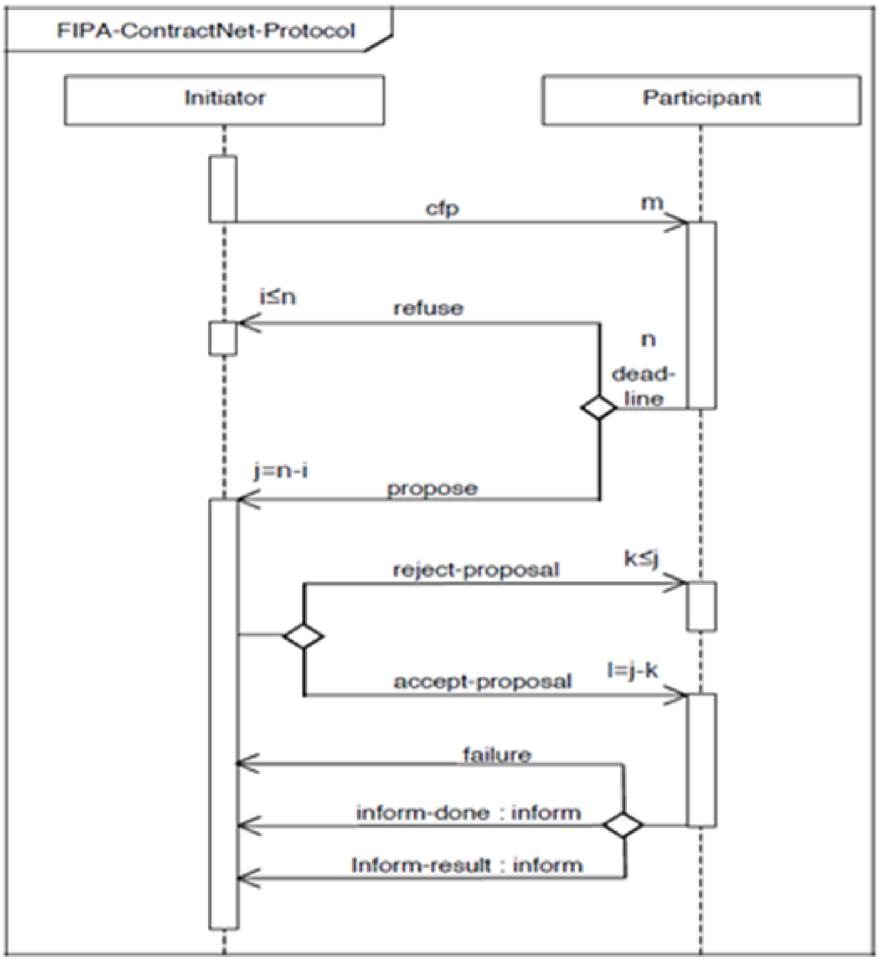


Figure 3.14: FIPA Contract Net Interaction Protocol [fip02c].

on blackboard architectures [HR85], where there are scheduling agents to schedule the write or read processes to / from the blackboard for resources or tasks allocation.

This kind of coordination exists explicitly or implicitly in most of the studied models since they belong to the organizational MultiAgent systems. The organization defines the authority between agents within groups and between groups.

3.7.2.3 Agent planning

Agents in MAS can design a detailed plan that may covers the possible action and required interaction to achieve certain goals. There are two essential types of planning:

- Centralized planning where agents have one goal with several partial plans, then there is a coordinating agent that tries to modify the partial plans and combine them to resolve conflicts.

- Distributed planning where agents have individual plans with models of other agents plans, and here the agents communicate to avoid conflicting where they can update and rebuild their plans during the communications.

Contrary to the previous types of coordination, the coordination by planning is restricted in the model of agents where the concept of plan is explicit and central such as in the BDI models and Tropos methodology meta-model. However, we cannot say agents can coordinate by planning when the concept of plan is implicit under the goal or intention.

3.7.3 Negotiation

Conflicts may occur whenever autonomous entities share resources to achieve their own objectives; looking towards an agreement to resolve these conflicts and convince the participants is needed.

In our case, these autonomous entities are agents that share the same environment resource. They have their own intentions and goals, which may be contradictory with the other agent goals. These agents have to negotiate to reach an agreement that makes them satisfied.

Many famous ways exist for negotiation, for example, there are different defined protocols of negotiation in market places (agents simulate customers and vendors behaviours), we can take for example Fish market[26] protocol and auction [fip01].

We detail here FIPA auction protocol, where two principal roles of agents exist in its specification, the auctioneer and the participant (see figure 3.15). The auctioneer or the initiator of the auction starts the auction by proposing a price, this act of proposition is named a call for proposal (cfp) that is broadcast for all participants. At the same time, any acceptance of a bid is simultaneously broadcast to all participants and not just the auctioneer. The bidder agents know if their bid (propose) has been accepted. Hence this protocol contains the two acts of accept-proposal and reject-proposal. In order to complete the auction transaction, this protocol ends by sending a request of the auctioneer for the winning bidder to reach this end.

The dialogue based on the exchange of messages is another way to reach agreements by negotiation inspired from human-like dialogue [BM93, JC05]. Cerri and Jonquet [JC05] propose STROBE model (STReams of messages exchanged by agents represented as OBjects and interpreted in multiple Environments) for negotiation between agents in the context of Dynamic Service Generation (DSG). The principal idea of this model is the interpretation of messages between agents according to the environment. Messages interpretation in STROBE is done in a given environment and with a given interpreter where both elements are dedicated to the interlocutor and they can be changed.

As a summary of this section, two essential types of interactions between agents exist. The first one is when agents communicate with each other to exchange messages to achieve certain goal or to resolve certain problem. The second type is when agents share certain resources and they need to coordinate and negotiate to avoid conflicts. We

[26]http //www.iiia.csic.esProjects_shmarket 04011999

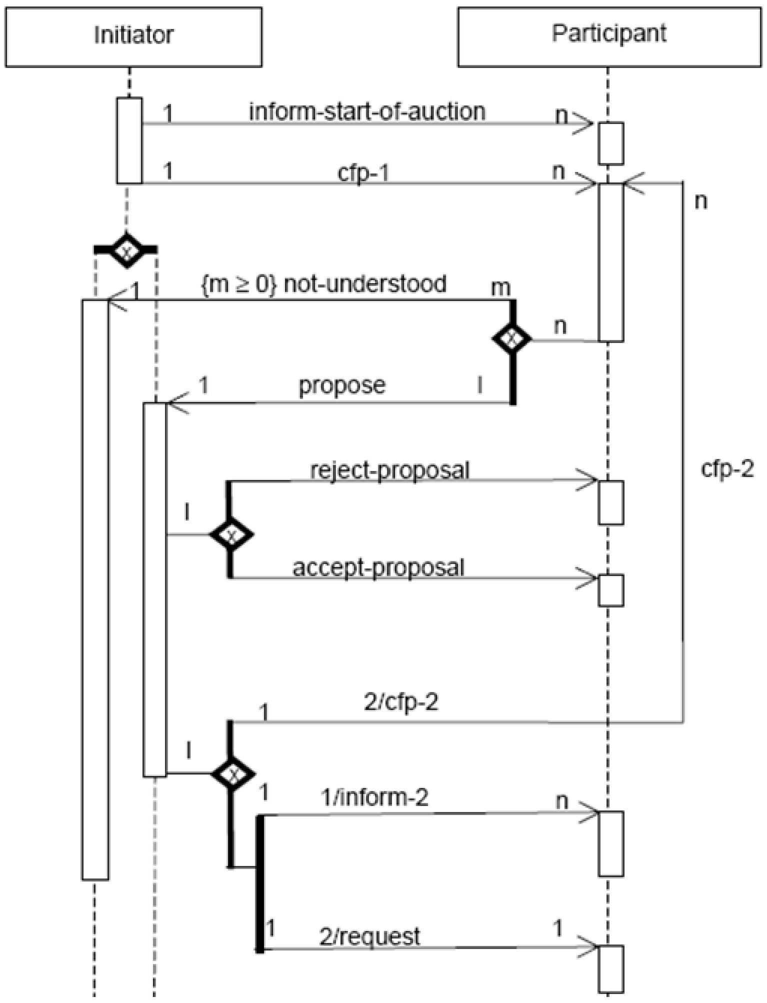


Figure 3.15: FIPA English Auction Interaction Protocol [fip01].

can see that we cannot define borders between these types of interactions because they can overlap.

3.8 Comparison

Here, we provide a short comparison between the presented agent and methodologies models with their interaction types in table 3.4. The '-' symbol refers to a non available concept, while "ok" refers to an existing explicit one. We may add text to clarify specific types and issues in table cells that need to precise a defined type of many other possible ones (e.g., the coordination cell of interaction can be from planning, organizational structure and contracting types).

The most shared explicit concepts between the studied models are agents, roles, organizations and ACL interactions. Other concepts like group, service, goal, task, resource and capability are explicit in some models and implicit or not available in others. Many concepts related to agent's autonomy property like plan, goal and intention exist in the studied models. However, we choose to use the concept of goal as a representation for the others in this table, since we do not consider the internal architecture of agents and we are more interested in their external view and how they interact with each others.

Model	Concepts												
	Agent	Group	Role	Organization	Environment	Interaction				Service	Goal	Capability	Task
						General	ACL	Coordin.	Negoti.				
AGR [FG98]	ok	ok	ok	implicit	implicit	interaction protocols	ok	organizational structure	-	implicit under role	-	-	-
A.R.G. meta-model [ONL05]	ok	ok	ok	-	implicit	implicit under role	ok	organizational structure	-	implicit under agent	-	-	-
MOISE [HBSS00]	ok	ok	ok	ok	-	protocols	ok	organizational lines	-	implicit under role	ok	-	-
OMNI [VSDD05]	ok	ok	ok	ok	-	dedicated model	ok	contracting	ok	implicit under role	-	-	-
Gormas [ABJ09]	ok	-	ok	ok	implicit under resource	dedicated meta model+protocol	ok	organizational structure	-	ok with service profile& port	ok	-	ok
BDI [RG95]	ok	-	-	-	implicit	events	ok	coordinations exist in general ways	-	-	ok	implicit	-
Gaia [ZJW03]	ok under agent type	-	ok	ok	ok with resource	protocol definitions	ok	organizational structure	-	ok	ok	-	activity
PASSI [Cos05]	ok	-	ok	-	implicit under resource	dialogue patterns and networking protocol	ok	-	-	ok	ok	-	ok
TROPOS [BPG+04]	actor	-	ok	-	implicit under resource	social pattern	ok	-	-	-	ok	-	ok under Plan
FAML [BLHS+09]	ok	-	-	ok	ok	Interaction Protocol	ok	organizational structure	-	ok	system goal	implicit	ok

Table 3.4: A comparison to extract shared concepts between agent models

Our objective is the integration of the two approaches of component and agent to overcome their weaknesses points by focusing on the interoperability via the two concepts of interaction and service. Then, we need to focus on these concepts in advance in the studied models.

We detail the types of interactions in table 3.4 according to their related concepts. Contract concept represents coordination by contracting and negotiation, like in OMNI model. We have coordination by organizational structure whenever the concept of organization is explicit. ACL communication language can be used in all agent models and methodologies, although it may not be explicitly stated in the studied models. Although, the interaction concept is a first-class element and a key property of agent models, it is not detailed sufficiently from our point of view in the studied model (i.e., we cannot find the three main types of interaction of communication, coordination and negotiation simultaneously in most of the studied models). We may find interactions by protocols to avoid details.

The concept of service is explicit in recently defined models like FAML [BLHS+09] and GORMAS [ABJ09]. It exits also explicitly in Gaia [WJK00, ZJW03] and PASSI [Cos05] methodologies, the service defines business activity that an agent is engaged to do. Service concept is implicit in other models under the role concept that defines the agent behaviour. GORMAS model is the only model that details the concept related to a service like port and service profile concepts.

Unfortunately, in the studied models, whenever the interaction concept is well detailed, the service one is implicit, for example, the model OMNI contains all types of interactions but it does not contain the concept of service explicitly. Moreover, when the concept of service is explicit there is an absence of detailed types of interaction like in PASSI meta-model. The two concepts of service and interaction are explicit in FAML, GORMAS, Gaia and PASSI models, but the interaction types are not detailed enough from our point of view. These models consider also other aspects like focusing on differences and relations between design and runtime concepts. However, our research does not consider models lifecycles, where we consider all the concepts in the design time except the concept of interaction, which may be related directly to low-level concepts. This reason (absence of the existence of detailed types of interaction and service concept simultaneously) justifies our need to define our general agent model lately proposed in chapter 7.4. In the defined model, the concepts of service and interaction will be detailed and considered as first-class concepts.

Next chapter provides a study of some service-oriented models with focusing on their level of abstraction.

Chapter 4

Service Oriented Architecture models

Contents

4.1 Introduction

Lately, companies realized the interests of exposing and exchanging their services with each other. Then, their relationships become automated, flexible, fast and secure. This increases the business specification of each enterprise in addition to the augmentation of the cooperation level between them. A totally connected, loosely coupled systems will be the result, and the service is a key concept to reach such properties. Then, we have to understand the used architecture to realize such systems that is, the Service Oriented Architecture (SOA). This chapter presents some key definitions in service-oriented approaches and browse some already defined models of service oriented architecture. We focus on the level of abstraction of these models (i.e., if the concepts implementing the services exist or not) and the existence of the two concepts of service and interaction.

The following section presents interesting definitions of the service concept.

4.2 What is a service?

Service orientation is an emergency to overcome the continual changes in systems because of business requirement changes. It enables loosely coupling feature between distributed systems and it covers also the heterogeneity of these systems. There is no agreed definition for service yet as presented in [Jon05]. We list some of service existing definitions in table 4.1.

WESOA [Bar]	*A service is a system function, which is well-defined, self-contained, and does not depend on the context or state of other services.*
OMG [OMG05]	*A service is a capability provided by entities through ports, these entities named Participant which may be person, organization or system.*
Papazoglou et al. [PTDL08]	*Services are autonomous, platform-independent computational entities that can be used in a platform in independent ways. It can be described, published, discovered, and dynamically assembled for developing massively distributed, interoperable, evolvable systems.*
SSOA [CLG+12]	*A service is a unit of work to be performed on behalf of some computing entity, such as a human user or another program.*
W3C [HB02]	*A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of provider's entities and requester's entities. To be used, a service must be realized by a concrete provider agent.*

Table 4.1: Service definitions

From the previously presented definitions, we define a service as an independent abstract business activity. It facilitates interoperability between distributed heterogeneous systems, since their elements provide or use services to reach cooperative and loosely coupled software systems.

4.3 What are SOA and SOC about?

We already viewed the general definition of architecture provided by Shaw and Garlan [SG96] in section 2.4.1. According to this definition the service is the basic element in building the SOA based systems. SOA views a distributed system as a collection of interactive services. In other words, SOA architectural style organizes software applications as a set of interoperable services [Har07, Blo03].

Service Oriented Architecture Reference Model (SOARM)[27] defines SOA as a paradigm for organizing and utilizing distributed capabilities that may be under the control of different domain ownerships. SOA can be viewed as a framework where enterprises build, deploy, and manage services. SOA is not a brand-new revolution while it is an evolution in computer science [Tow08].

The conceptual model of SOA consists of two main actors, service provider and consumer. An additional actor may be added to this model to facilitate the service discovery which is the repository (see figure 4.1). However, we believe that the registry is more related to technologies than to the conceptual model of SOA, i.e., Universal Description Discovery & Integration (UDDI)[28] registry allows to discover web services defined by Web Service Description Language (WSDL).

Service Oriented Computing (SOC) is a new computing paradigm that depends on SOA, where the service is the basic constructs to support fast, low-cost and easy composition features in the improvement of distributed applications [PTDL08]. It was somehow difficult to distinguish between these two terms (SOA and SOC), since they are not well separated.

As a summary for different points of view, SOA is about basic entities (services) and their interactions in software architecture. SOC is about service composition, where in services **orchestration**, the recursive composition of services defines a new service that has central control over the whole composition. However, there is no such control service in services aggregation known as **choreograph**.

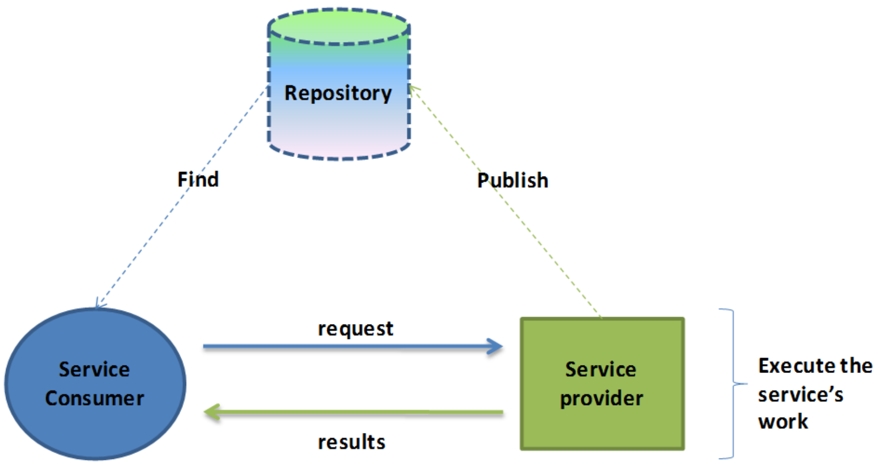


Figure 4.1: The conceptual model of service oriented architecture.

SOA provides many benefits on both business and technical levels, where it focuses on business domain solutions to cover frequently changes in business requirements. Next section browses some existing service models.

[27]http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf
[28]http://uddi.org/pubs/uddi-v3.0.2-20041019.htm

4.4 Service oriented models

Many models exist for service oriented architectures or for modeling the service itself. Differently from the component and agent approaches, we do not have categories to classify these models; for this reason, we list the studied models here and we detail them lately. Although the web services belong to a low-level of conception, we start by presenting the model of Web Service Description Language. The OMG proposed the SoaML [OMG09] for Service Oriented Architectures Modeling Languages. OASIS proposed SOARM for designing any service oriented application. Many other models are proposed by researchers like works that use the Model-Driven Engineering (MDE) [Ken02] for designing service oriented architecture models in [EKLM07,RRSA06] and the SOA models based on UML profiles like [EKLM07, BHTV03, Joh05].

4.4.1 The model of Web Service Description Language

Web services are the most used technology for implementing SOA. We present here the general model of the Web Service Description Language (WSDL). Figure 4.2 shows the basic elements in WSDL 1.1 (a portType class becomes interface in WSDL 2.0). This model focuses on the service itself regardless of the service provider or consumer [CBZ+04]. The *service* aggregates the *ports* which itself aggregates the service's *operations*. These operations contain *messages* which are the parameters of service *bindings*. Each message contains an envelope and it has a header and a body. The message header may contain specific aspects of the message, which can be related to its role. The message parts are the parameters with their types.

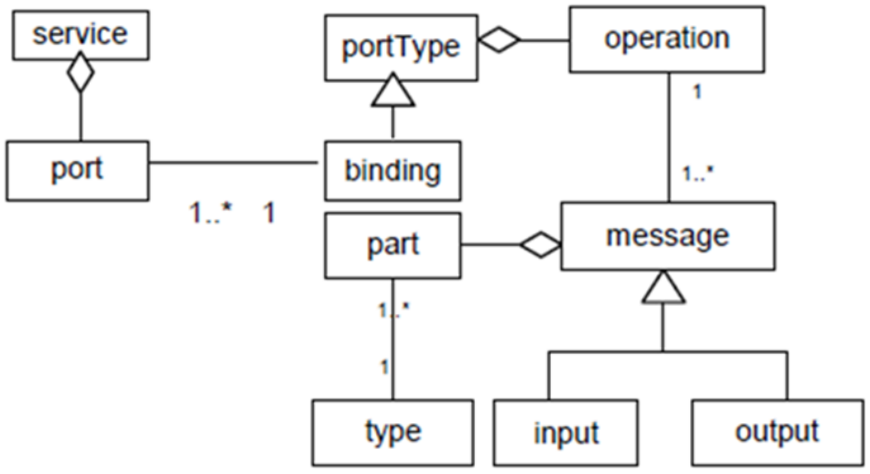


Figure 4.2: The Web Service Description Language model [CBZ+04].

4.4.2 Service Oriented Architecture Modeling Language

The OMG asked for Request For Proposal (RFP) and solicited submissions for a UML Profile and Meta-model for Service (UPMS) [OMG09]. Enabling the interoperability, integration at the model level and the flexibility of platform choices are among the main objectives of this meta-model.

SoaML previously named SOA-pro is based on UML2.0 meta-model, it meets the mandatory requirements of the UPMS RFP. It supports bottom-up and top-down architecture design of services. In SoaML, a *service* is a capability provided by entities through *ports*, these entities named *participants* which may be person, organization, system or agent. The *ports* can be a simple UML interface or a *ServiceInterface*. The *ServiceInterface* is defined from a service provider viewpoint; it may expose the participant capabilities. The composition of service can be achieved through their *serviceInterface* where it may have itself *service points* or *request points* that define more granular services. The *capability* specifies the capacities of a participant and represents the provided operations. All the information about the service, choreography and any other terms and conditions are specified in the *ServiceContract* which specifies the collaboration between the participants (see figure 4.3) since it extends the collaboration meta-class of UML. The concept of *role* appears also within this contract. The behaviour in SoaML can be achieved by activity diagrams of UML2.0 or by a state machine. The communication between request and service points is done through *service channels* which extend connectors in UML2.0. *Protocols* are used whenever a conversation is involved between a service provider and consumer instead of method calls.

Many concepts exist in this model, but we are interested in following main ones: service provider, service requester that are represented in the participant concept, service points (provide service) and request point (require service) which are the interfaces and the interaction by contract between service providers and requesters, the contract contains roles of participants.

4.4.3 SOARM

The Organization for the Advancement of Structured Information Standards (OASIS)[29] proposed a reference model for service oriented architecture[30]. This general SOARM model defines main concepts that should be considered in the design of any system adopting a SOA approach. The main concepts in this model are presented in figure 4.4:

- Service that allows to access or use the capabilities through the service interface. There are some concepts in the SOARM model that describe the involved concepts for dynamic perspectives of a service, such as the visibility, interaction and real world effects.

[29]http://www.oasis-open.org/

[30]http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

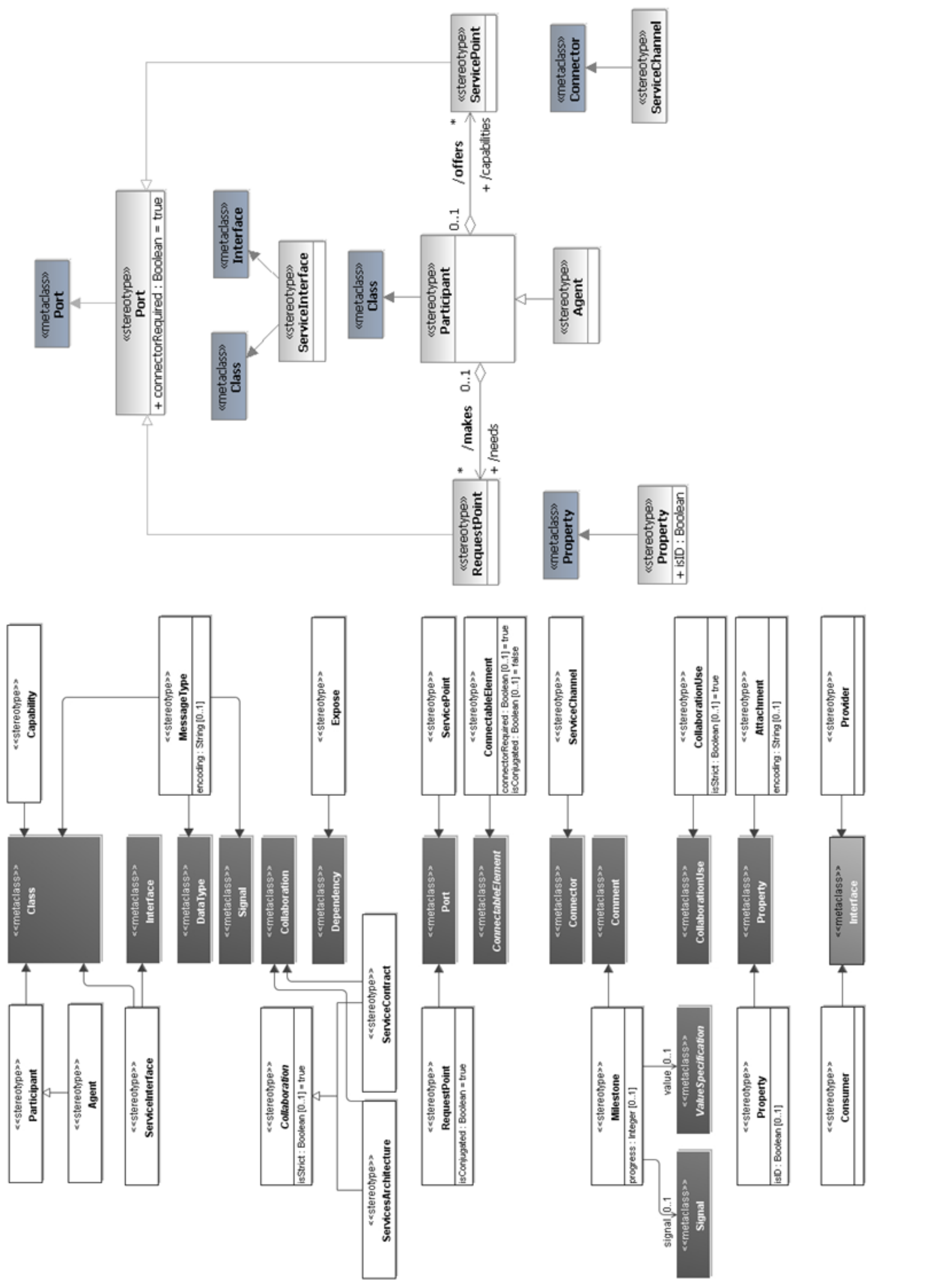


Figure 4.3: The SoaML profile.

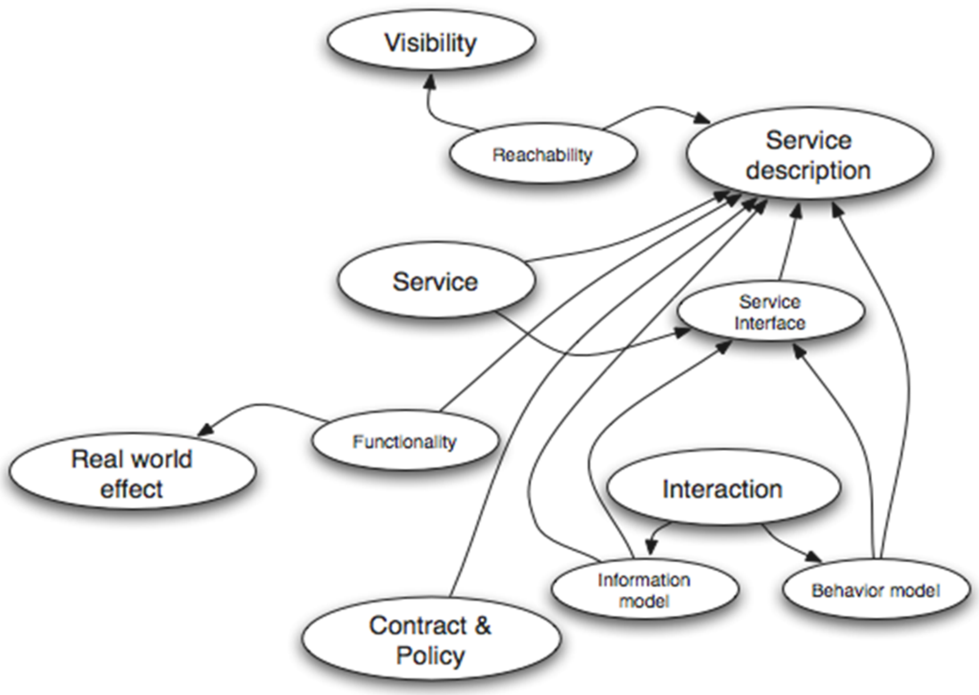


Figure 4.4: Main concepts and relations in the SOARM.

- Visibility which enables the service provider/consumer to see the service. It must satisfy the awareness, readiness and reachability between the provider and the consumer.

- Interaction that shows the exchanges between services either by sending, receiving messages or by changing the state of a shared resource. The format of the exchanged information is defined by an information model. A behavioural model is defined to ensure a successful interaction through the definition of the sequences of actions and process involved in a service by a protocol.

- Real world effects which are many for using a service. They affect the way of using provided services by a consumer.

- Service description defines the needed elements to be respected for using a service. It contains the service functionality and its interfaces. The service interface includes the protocols, commands, and information exchange to allow the interactions with other services.

- Execution concept which permits to differentiate services (different instances of the same service). According to this concept we can interpret the exchanged date of a service interaction.

- Contract & Policy where a contract defines the agreement between two participants (parties). A policy defines how to use, deploy or describe a service by identifying the constraints or the conditions to a participant.

The SOARM model defined the norms and aspects that may be considered in the development of any service-based application. It does not specify the concepts literary but it mentions the needed issues that must be respected like the description of the services, the visibility, etc. We mention that the concept of service, service interface and interaction should be explicitly available.

4.4.4 Model Driven Architecture (MDA) and SOA models

Emig et al. [EKLM07] proposed the SOA model presented in figure 4.5. This model consists of two essential parts: conceptual and the deployable parts according to the MDA principles with a formal definition using UML profile. The main concept of *service* extends the port concept in UML, however the *service interface* extends the interface concept and the *service providing component* extends the component concept in UML.

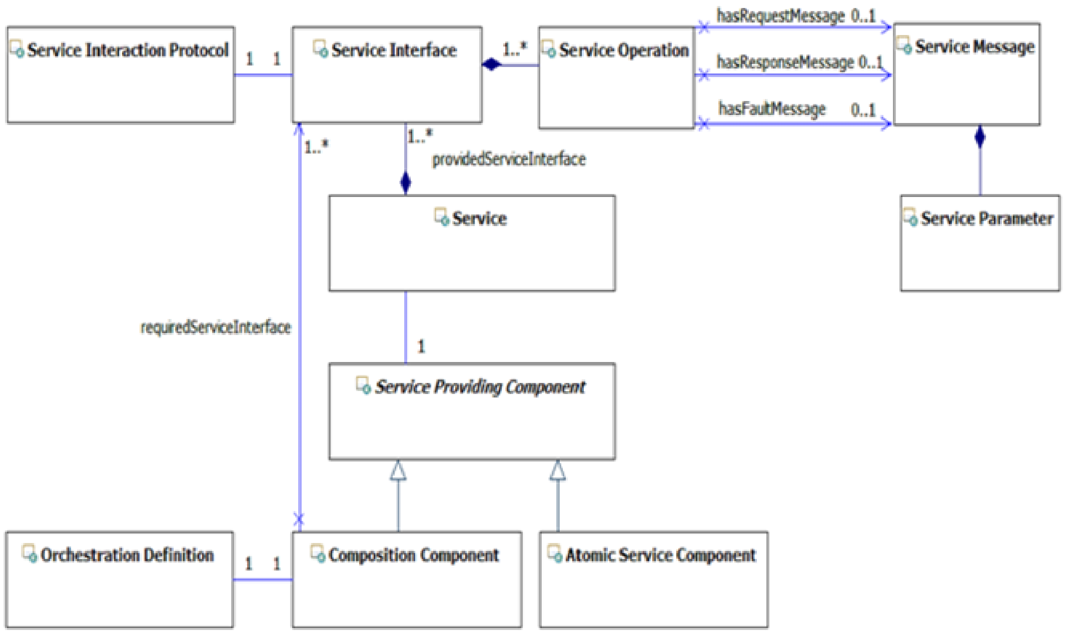


Figure 4.5: The conceptual part of the SOA service meta-model [EKLM07].

In the conceptual model the *Service* represents the central element of the meta-model. It provides a set of *Service Interfaces* each of them consists of *Service Operations*. This containment relation is strictly enforced. The provision of *Service Interfaces* is modeled using the association *Provided Service Interface*. The usage view on a service is defined via the signatures of its *Service Operations* and the corresponding *Service*

Messages which can be of the *Request Message* type (Incoming) or the *Response Message* (outgoing). The service parameters correspond to the service messages part in WSDL.

We can find the composition on the *Service Providing Component* not on the service itself (only primitive services), the *service providing Component* can be a *Composition Component* or *Atomic Service Component.* Then, the orchestration is related to the composite provider, it is achieved by the *Service Interaction Protocol* which defines also the order of the operations calls. The deployable service model extends the conceptual one by the deployment of specific information, like for instance the actual service endpoints.

Another work that respects MDA principles for modeling SOA, is proposed by Rahmani et al. in [RRSA06]. The main principle of MDA [Sol05] is to separate the Platform Independent Model (PIM) from the Platform Specific Model (PSM) and to use tools to make transformations between them. The proposed approach contains two PIMs and one PSM. The first PIM is designed from system components without the need to identify the services, they use here standard UML to define the class diagram. Then, they define a SOA based PIM (which is a PIM of next level) developed using UML to model the system based on SOA. The SOA based PIM is generated by adding interfaces on previous defined component which represent the services and manage all the attached components lifecycles; then they generate the PSM for specific technologies like web services.

4.4.5 UML Profiles and SOA models

Similarly to SoaML, the proposed SOA model in [BHTV03] is actually a UML profile.

It defines many stereotypes presented in (figure 4.6), we can see that a *service* is considered as a special type of component. It is published and found through *ports* and it has its *interfaces* which can be provided or required. These interfaces contain *operations.* We can see also that the service providers or consumers are components. The interaction concept is defined through the UML collaboration diagram by assigning messages.

Another service modeling approach based on UML profile is proposed in [Joh05]. We can see in figure 4.7 that there are new concepts such as: *partition*, which represents the logical or physical boundary in the system. These boundaries are strict where there must be a *gateway* to access each partition. A *gateway* represents a service proxy where it may be used to arbitrate protocols to allow access to a partition, *service channel* extends the connector class of the UML 2.0 profile just as in the case of SoaML, it represents the communication between services. The central concepts of service, service provider, service consumer, interface, protocol, message and operation are explicit with the same semantics as in the previously presented models. An additional concept related to the service and the protocol is named service specification that acts like contract between providers and clients.

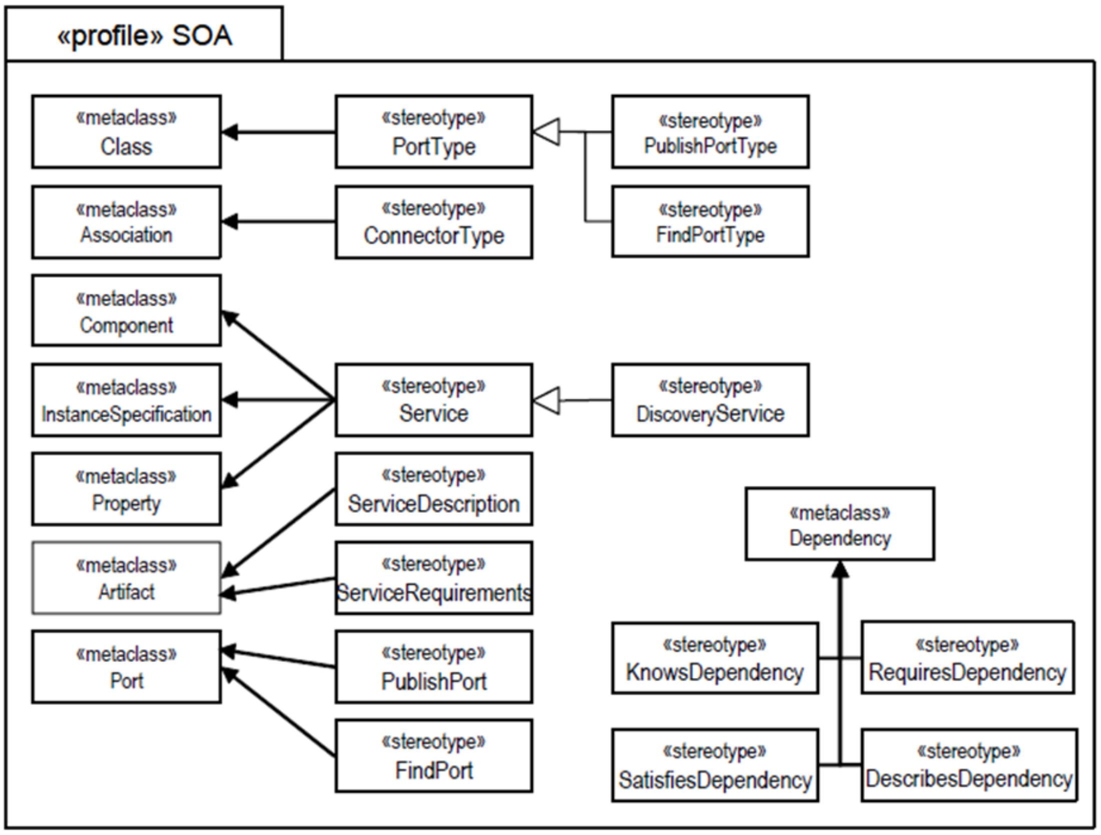

«profile» SOA
«metaclass» Class
«metaclass» Association
«metaclass» Component
«metaclass» InstanceSpecification
«metaclass» Property
«metaclass» Artifact
«metaclass» Port
«stereotype» PortType
«stereotype» ConnectorType
«stereotype» Service
«stereotype» ServiceDescription
«stereotype» ServiceRequirements
«stereotype» PublishPort
«stereotype» FindPort
«stereotype» PublishPortType
«stereotype» FindPortType
«stereotype» DiscoveryService
«metaclass» Dependency
«stereotype» KnowsDependency
«stereotype» RequiresDependency
«stereotype» SatisfiesDependency
«stereotype» DescribesDependency

Figure 4.6: Stereotypes in a SOA model [BHTV03].

4.5 Comparison

Because of the lack in the standardization of service oriented model, we can find many proposed models, with different aspects, like considering the composition on the service level or on the element that provides it. Most of these works have agreed on the presence of the elements providing or using the services. However, in WSDL meta-model, this element is not considered as it models the service itself. Here, we provide a short comparison between the presented service-oriented models (see table 4.2). The "-" symbol refers to non available or ambiguous concepts while "ok" refers to existing explicit ones. We may add text to clarify specific issues.

Primitive services or services in general (some models do not distinguish between primitive and composite), protocols, participants, interfaces, operations and messages are of the most common explicit concepts between these models. The type of interface (required or provided) is not distinguished in all these models. Also, we can see that the composition of service is not well-defined in most of these models. It may be defined on participants level like in [EKLM07], it may be done through protocols for choreography or orchestration like in [OMG09] or it may be ignored. The concept of role in service-

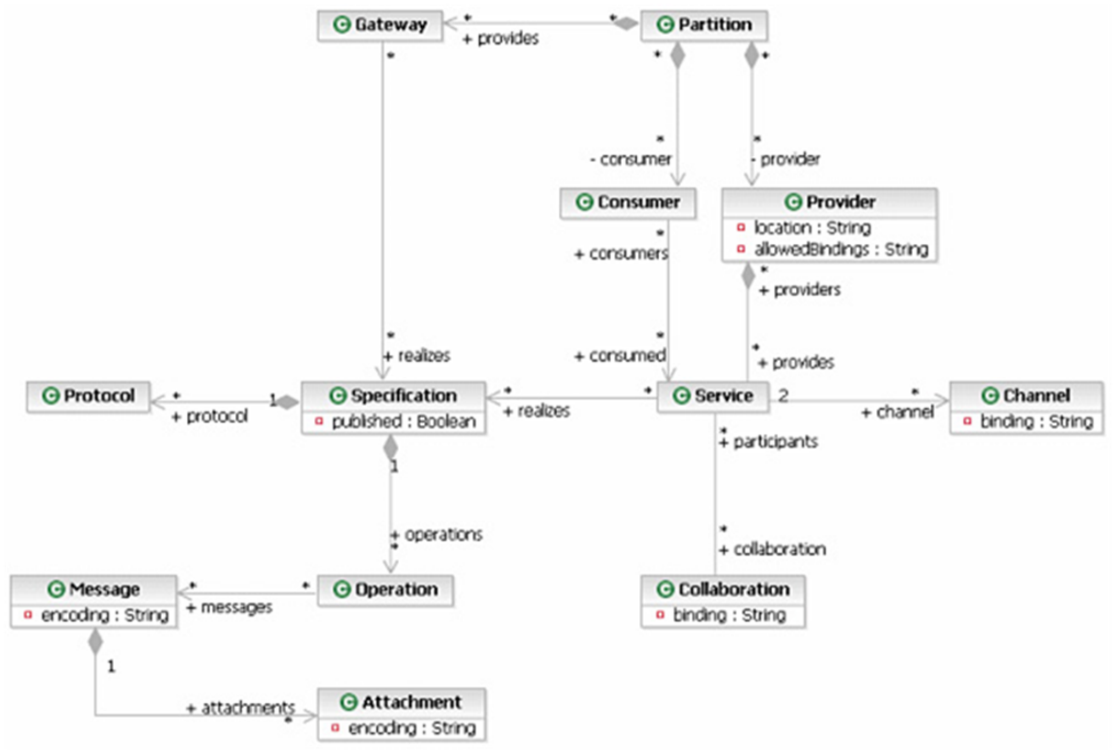


Figure 4.7: Modeling services [Joh05].

oriented model is usually restricted to provider and consumer types. For this reason, this concept is not available or implicit in most of these models. From our point of view, this concept must be explicit according to the need to precise the service roles in different interactions which are surely not the same. In [OMG09], we can find different roles of participants in a service contract.

The interaction between services is achieved usually by basic types of communication like method calls, binding and service channels between service providers and consumers. In some cases, the basic interaction is not sufficient; then, using a protocol specified in the services or participants is possible. The interaction can be specified through activity or collaboration diagrams in the cases of SOA models based on UML like in [EKLM07] and [Joh05].

Model	Concepts									
	Service		Service provider	Service consumer	Role	Service Spec.	Operation with messages	Interaction		
	Composite	Primitive / General						Basic	Contract	Protocol
WSDL [CBZ+04]	by service binding	ok	-	-	implicit in the message header	Port (interface WSDL 2.0)	ok	binding	-	SOAP
SoaML [OMG09]	ok by service choreography	ok	participant/ agent	participant/ agent	role of participants in service contract	port, service point, request points	ok	RPC & ServiceChannel	ServiceContracts	ok
SOARM	implicit by contract	ok	ok	ok	-	Service Interface	-	exchanging messages	ok	ok in the behaviour model
[EKLM07]	composition Component	atomic (primitive)	Service Providing Component	-	-	interface	ok	binding &UML activity diagram	-	Service Interaction Protocol
[RRSA06]	-	ok	component	component	-	-	-	-	-	-
[BHTV03]	-	ok	component	component	-	ports to publish and interfaces	ok	exchanging messages	-	collaboration diagram
[Joh05]	-	ok	ok	ok	-	interface	ok	service channel	-	protocol & service collaboration

Table 4.2: A comparison to extract shared concepts between service models

The participant is a first-class concept in most of the previously viewed models except in the WSDL meta-model. We consider any application as a set of interacting services, then we prefer to precise participants lately in more concrete level. These participants will be components or agents implementing services (as we explained previously in the introduction). For these reasons (the implicit existence of roles and explicit one of participants), we choose to define our own abstract service model where the concept of role is explicit and the one of participant is implicit.

After the presentation of the several models in component, agent and service approaches, we need to put the light on the works that already have been interested in the integration of these approaches or at least two of them. These works are the subject of the next chapter.

This is a tricky domain because, unlike simple arithmetic, to solve a calculus problem - and in particular to perform integration - you have to be smart about which integration technique should be used: integration by partial fractions, integration by parts, and so on.

Marvin Minsky

Chapter 5

Works integrating components, agents and services simultaneously

Contents

5.1 Introduction

In this chapter, we browse the most representative works that already consider simultaneously the three approaches of component, agent and service or at least two of them.

We start by browsing works that integrate component and service approaches followed by the ones that consider agent and service together. Then, we present works that consider both of component and agent approaches through two main categories: The *Agentification* (resp. *Componentification*) is the added values of the agent (resp. component) approach to the existing components (resp. agents) approach.

5.2 Component-based approach and service oriented architecture

A service can be viewed as a logical extension for a component and a component can implement a service, since they both have the main interest of reusability. The web service, which is the most used technology for implementing services, is considered as a special type of components in [LW07,BL07]. A component itself can implement abstract business services.

The authors in [BL07] consider services as a special type of component with a major difference that the services focus on run-time retrieval and run-time deployment, while components focus on design time.

The authors provide a detailed comparison between Component Based Software Engineering (CBSE) and Service Oriented Software Engineering (SOSE) according to principles, process concerns, technology concerns and composition. A component is a tightly coupled entity because it exists always in the frame of a component model and it has to be conformed to its rules.

The service description and implementation are well separated (black boxes) in service-based approaches, while there may be different degree of the component specification (gray, white and black boxes) in component-based approaches. Regarding the composition level, it is always between homogeneous static component in CBSE, which is not the case for SOSE where it composes dynamic heterogeneous services.

Figure 5.1 presents main differences between CBSE and SOSE during their development process. We can see that the SOSE focuses on both specification and run times rather than design and implementation ones, while CBSE consider them all through its development process.

Another interesting comparison between components and services exists in [BCM+07]. The authors view services in a higher level of aggregation. It can be implemented by one or more components. The authors view components as compiled blocks of code that can be reused and assembled in homogenous application. The services overcome the homogeneity limits, they access to heterogeneous environments and they cover the components limitation when they aggregate different components that belong to heterogeneous environments.

The work of [HkO11] provides also a detailed comparison between CBSE and SOSE. On one side, the authors compare quantitative criteria like product and process. They also map the concepts of the two approaches along their lifecycles (i.e., *component and connector types* correspond to an *abstract service* in the design time). On the other side, they compare the two approaches according to quality criteria. They found that

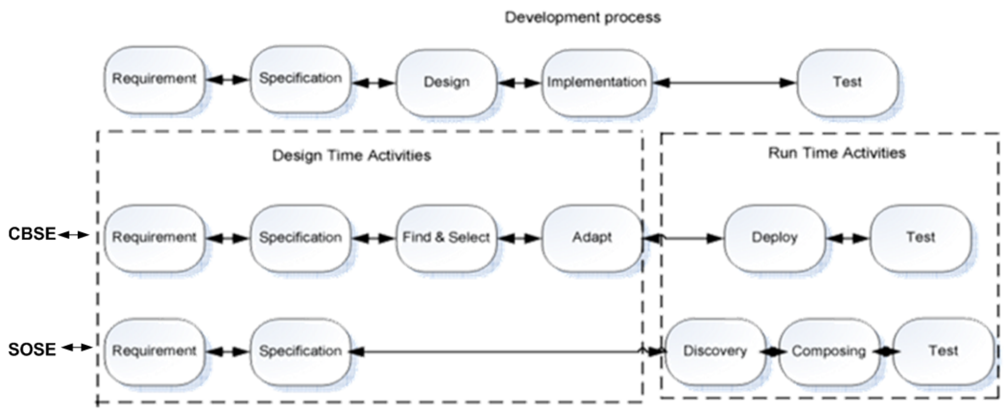


Figure 5.1: Comparison of typical activities during development process in CBSE and SOSE [BL07].

components are more reusable and composable than services while services are better at the dynamicity level.

In the following sections, we present works that already merge services and components in a new entity [ZYCZ08], define a new architecture [CA07] based on the two approaches, or the works that use one approach to enhance the reusability and composition ability of the other approach [Zhu05] and [YP04].

5.2.1 ServiceComponent

Zhang et al. propose a service oriented architecture model based on a traditional architecture description technique (an ADL) named *SO-ADL* [ZYCZ08]. Like any ADL model, there are the three main concepts of component, connector and configuration. This model proposes a new concept named *ServiceComponent* (SC) instead of the component one. A SC differs from traditional ADL's component as it does not present business requirement and it is not the same as the service, since it is a functional unit rather than a business service. The SC entity is defined via four main aspects of the identification, interfaces (provision and request), specification and (pre and post) condition. The second important concept in this model is the *connector*. *Connectors* are defined in explicit manner between different services, their main aspects are the identification, role (provider or requester), specification that defines the protocol glue between roles and coordinator that ensures compatibility between connectors and ServiceComponents (see figure 5.2).

The concept of *configuration* is composed of the declaration of *ServiceComponent* and *connector* instances. At the same time, the composition of both of SC and connector is achieved by declaring their constituent instances. This approach supports dynamic changes from dynamic configuration, composition and recompilation. This work is quite interesting but it was not fairly detailed. A similar work was provided by a group of vendors named Service Component Architecture is presented in the following section.

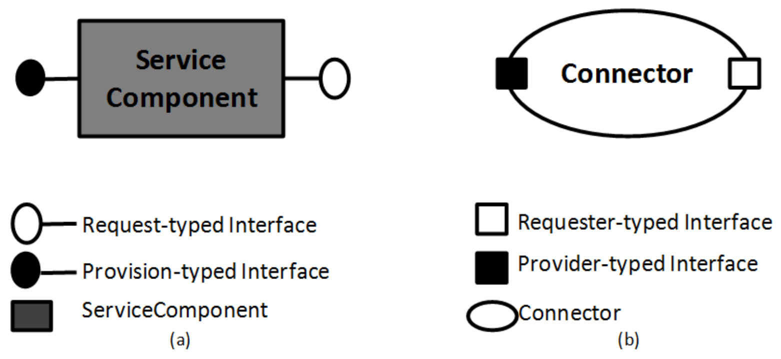
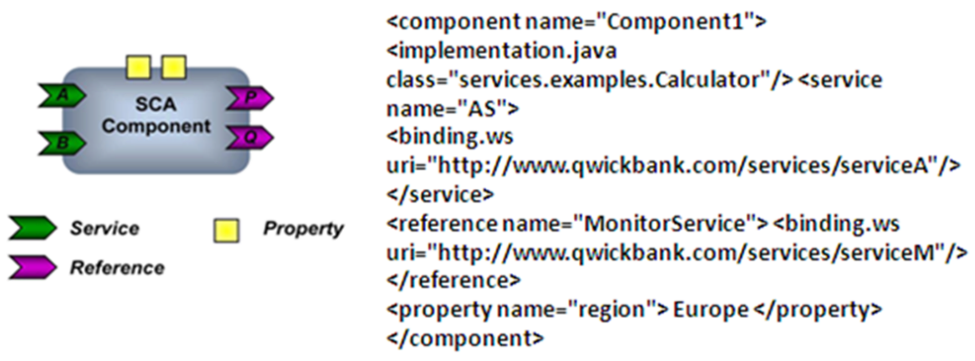


Figure 5.2: Service component (a) and connector types (b) [ZYCZ08].

5.2.2 Service Component Architecture (SCA)

BEA, IBM, Oracle, SAP, and others proposed Service Component Architecture[31]. Here, we have the new term of **"SCA component"** that implements the service. Figure 5.3 represents a SCA component, we can see the three main elements in a *SCA component*: Service, reference and property.



```
<component name="Component1">
<implementation.java
class="services.examples.Calculator"/> <service
name="AS">
<binding.ws
uri="http://www.qwickbank.com/services/serviceA"/>
</service>
<reference name="MonitorService"> <binding.ws
uri="http://www.qwickbank.com/services/serviceM"/>
</reference>
<property name="region"> Europe </property>
</component>
```

Figure 5.3: SCA component and specification for a calculator [CA07].

SCA component exposes services while it requires the references, which are the services that a component needs to accomplish its tasks. The property is a value that helps to access this component like the component location. This is important in the case of composition. The provided service interface is on the outside of the service component, it is visible to service consumers. The SCA component and references (required service interfaces) are part of the internal view. Thus, they are not visible for service consumers. However, this information conflicts the transparency characteristics of services. The main originality of SCA is in providing an assembly mechanism for components implemented over heterogeneous environments. Then, it overcomes one of the component drawbacks (being aggregated in homogeneous environment) [MR09].

[31]http://www.osoa.org/display/Main/Service+Component+Architecture+Home

The work of [HLD04] belongs to the same category, where the service is always in a higher level of abstraction than a component. It focuses on special types of service: *non-functional services* (security, management, transition, etc.) as a composition of components to allow the modularity and the adaptation to all application specificities. These *non-functional services* can be decomposed to sub tasks. Each one of these tasks is implemented by a component. They use the Fractal component model because it allows to change easily the *non-functional service* version for a specified environment. The component model allows the easy substitution of a component by another one to adapt the application in a defined context. Here, the *non-functional services* allow the reconfiguration of the application in deployment time; they present an example for researching services which need a high level of adaptability.

5.2.3 Service for more reusable components

Another study is done by Zhu in [Zhu05], where he finds that SOA provides more chances for the development of reusable components. A service is more proper for reusing software since we are not interested in its implementation. The author shows that "register, find, bind and execute" actions help to reach more reusable components.

The author defines the service in the registry by a tuple **S::=<N, P, I, R, M>**.

- **N** is the service provider; it can be a URL,
- **P** is a pattern of service and it defines the name of service,
- **I** is the format of inputs of the service,
- **R** is the format of returned results,
- **M** is the services implementation.

This definition can be extended to consider the semantic description of the service. The author finds that it is easier to the service requesters to understand the contract of the provider than to implement complicated logic of the service by themselves. Then, he defines the contract for using a service including performance requirement, cost of a service; and the penalty to the service provider's failing to meet the performance requirement.

This work presents different ways to reach service binding in different service oriented architectures (centralized, distributed service centres and distributed providers with a centralized and distributed registry).

5.2.4 Service components for managing the life-cycle of service compositions

A methodological framework for services composition and managing their life-cycle is proposed in [YP04]. The authors use the concept of *service component* to package the composition of services (complex services) and expose their interfaces and operations in a consistent and uniform manner. This *service component* can be considered as an abstraction of web services although it is a web service itself and it can be published, reused and extended. Since *service components* are published, they can be invoked by any service-based application.

After browsing the works that consider component and service together, we present the works considering service and agent approaches together in the next section.

5.3 MultiAgent systems and service oriented architecture

We need to distinguish two main directions of researches considering theses two approaches. Either we make use of the agent approach in the context of SOA or we make use of the interests of SOA in the context of MAS.

5.3.1 Service-oriented computing and software agents

Figure 5.4 presents an Extended Service Oriented Architecture (ESOA) provided by Papazoglou et al. in [PAG04]. This architecture consists of three main layers: basic, composite and managed services.

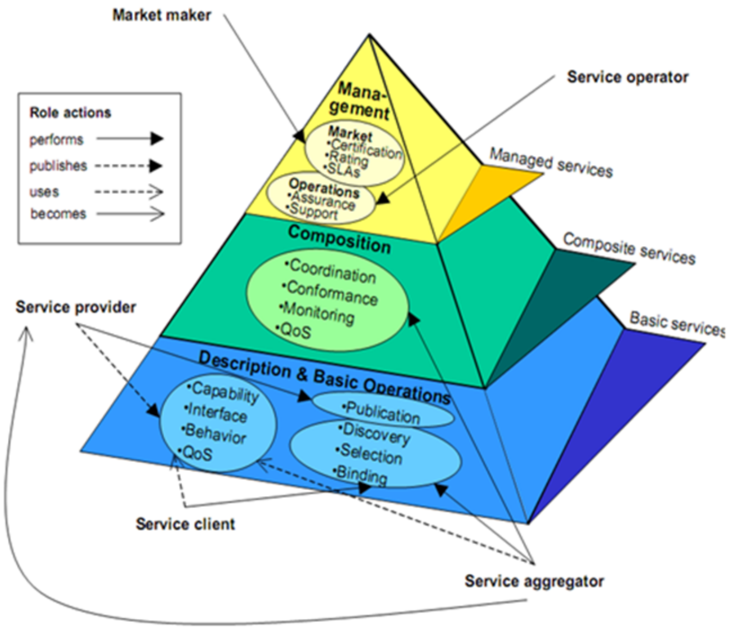


Figure 5.4: The Extended Service Oriented Architecture [PAG04].

This layered architecture uses the SOA in its basic layer. The service composition layer aims to consolidate several services into a single composite service. The aim of ESOA's service management layer is to provide support for open service marketplaces. The authors use Grid Service Bus (GSB), which is the infrastructure to allow services interaction, coordination and aggregation. Then GSB helps to reach an end to end quality of service and other goals. Papazoglou et al. suggest using agent technology with ESOA, which could be used to provide two types of coarse agents: agents as service providers and clients. These two types of agents could be combined when a coarse agent behaves as a service aggregator (or market maker) in order to provide services added value to other agents. They identify four types of agents: coordination, monitoring, QoS composition and policy enforcement agents in addition to the deployment and service selection agents in the service grid.

Another work that uses agent interests in the context of service approach is presented in the following section.

5.3.2 Agents for service composition by negotiation

The authors in [PBBP01] explore the future of bundling service on the fly to respond to the requirements of a customer by negotiation. This approach claims that the agent technology is the best practice to bundle services and to reach to composite ones because of its ability to automate sophisticated negotiation. Then, the resulted composite services respond to the customer needs and do not contain useless services.

Framing the intelligent agent with an overall of SOA and expected benefits of such approach is presented in [ITS$^+$08]. Since the SOA cannot help to reach rational (semi) automated selections in the creation of Virtual Organizations (VO) [DM92]; agents automate such processes of finding collaborative partners for the creation of these VOs based on services and business rules through negotiation.

In the two previous sections, the authors use agents to manage their extended SOA or to facilitate the composition of services, while we find a reverse example in the following work.

5.3.3 Service-oriented approach for MultiAgent system designs

The work in [ODS09] proposes an approach that combines service-oriented principles with organizational based MultiAgent concepts. The aim of this work is to facilitate the design of complex MultiAgent systems by composing predefined reusable MultiAgent *organizations*. Figure 5.5 presents the proposed organizational service meta-model, where we can find that a service is used or provided by an organization (a service consumer or provider is an organization). A *service* is coarse-grained MultiAgent functionality while the *operation* is a fine-grained one. The binding is initialized at design time between the consumer organization and the provider one in order to create a single composite organization. The *connection point* is a pair of goal-role of an organization and it *provides* or *uses* operations (then services). In some way, the services help to reach reusable organizations (service providers).

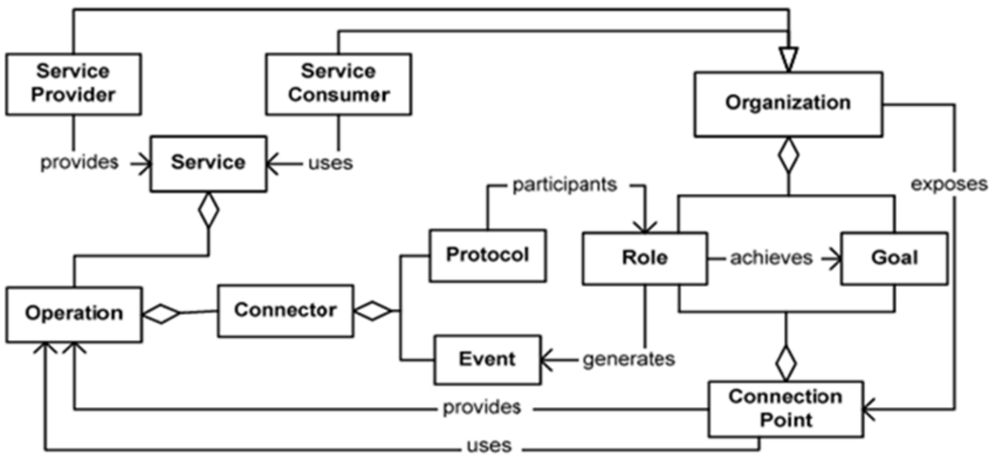


Figure 5.5: Organizational service meta-model [ODS09].

5.3.4 SoaML and an agent meta-model

The work in [HJR10] provides a model driven approach for the integration of agents and services. It shows the possibility of the transformation between SoaML and a defined platform independent (meta-)model of agents (PIM4AGENT). This work has been previously started in [HMMF+06] where the authors studied the Believe Desire Intention (BDI) models of agents [RG95]. The work in [HJR10] integrates the organizational dimension. The SoaML (figure 4.3) is chosen to represent the SOA model where it contains the notion of agent for concretely representing a participant in an application specification.

The authors in [HMMF+06] provide mapping rules between the concepts of the agent meta-model (PIM4AGENT) and the SoaML (PIM4SOA) model (i.e., direct mappings between the agent of the SoaML model of service and the agent concept of their agent meta-model). They use ATL[32] for implementing the mapping rules. This work ensures the possibility of interoperability between MAS and other application technologies. This PIM4AGENT must be able to incorporate all relevant high-level concepts of the target agent platforms then they can ensure to generate a PSM4AGENT according to the agent technology. The mappings between the models of SOA (PIM4SOA) and the agent meta-model ensure the feasibility of this approach.

After browsing works considering service and component together then service and agent together, works that consider both of agent and component approaches together remain to browse.

[32]http://www.eclipse.org/m2m/atl/

5.4 MultiAgent systems and component-based approach

Lind compares component and agent approaches through three categories: entities, interaction, and problem solving [Lin01]. This comparison aims at showing that agents may extend components to facilitate the building of complex distributed systems.

The work in [BGZ06] cites benefits of each approach, such as reusability, compositions, Component Of The Shelf (COTS) properties of component approach. The key properties of MultiAgent approach are such as autonomy, social ability, auto learning and reasoning capability. Then, an interesting comparison is also provided in [BGZ06]. It compares the two approaches according to their state, their communication, delegations of responsibilities and communication with other parties or with their environment in table 5.1.

Feature	**Agent**	**Component**
State	Mental attitudes	Attributes & relations
Communication	ACL	Metaobject protocol
Delegation of responsibility	Task and goal delegation	Task delegation
Interaction between Parties	Capability descriptors	Interfaces
Interaction with the environment	New beliefs	Events

Table 5.1: Adapted features of the agent meta-model and their counter parts of the component meta-model [BGZ06]

Component **state** is a set of attributes and relations that can be changed by relations with other components, while agent mental state can be manipulated by other agents. The **communication** in components is realized by method calling or protocols, which force the receiver components to execute methods. However, agents use declarative language like ACL that helps to convey the mental state of a sender agent to the receiver one to convince it to do certain messages. Considering **tasks delegation**; a sender component does not explain anything to the receiver one. Then, all outcomes are from the sender responsibilities. Agents may delegate part of their goals to other agents; through the tasks delegating, then both of them share the responsibility of the resulted outcomes. The component uses its interfaces to **interact with other parties** while agent does not have interfaces but it describes its capabilities to show what it can do through its roles. This work finds also that agents are more efficient in semantic interoperability than components while the components are more powerful in the syntactic interoperability. The **environment** communicates with components through events while it is an essential concept in the agent approach as any change in it may affect the mental state of an agent.

The authors in [KMW03] present two different ways of combining agent and component approaches. The first one, is called ***agentifying***, which starts from component technology with Added value by agent properties into existing components. The reverse definition is for ***componentifying*** which considers agent technology as starting point and add component features to it. We borrow these two approaches with adapting their names into ***Agentification*** and ***Componentification*** to represent the transformation from component to agent model in the ***Agentification*** and the reverse in the ***Componentification***.

We use these terms in defining relations between component and agent models in chapter 6. They also help us to classify works that consider the two approaches of components added value to agents or the reverse in the following section.

5.4.1 Agentification

Many works try to add agent properties to component model, to reach auto adaptable, well managed and intelligent components like the works of CompAA [AL08], SoSAA [DLCO09] and ActiveComponent [BP11].

5.4.1.1 CompAA

The main objective of this work is to adapt the components automatically by attaching them to agents [AL08]. This is not literally what the ***Agentification*** is about; but it can be viewed as the gain of the dynamicity feature of agent approach to adapt components in a model named CompAA (Component Automatic Adaptation). The component conforms to a specific model named **Ugatze** [SA04b], it differentiates the input and output of data and control (data input (DI), data output (DO), control input (CI), control output (CO)) and it has adjustment points to reach its services (the required input service SI0.. SIn or provided output service SO0.. SOn) (see figure 5.6 (a)). For each component there is an assigned agent responsible for its adaptation (see figure 5.6 (b)). Then, the component makes use of the agent properties (dynamicity) to be auto adapted [LGA06].

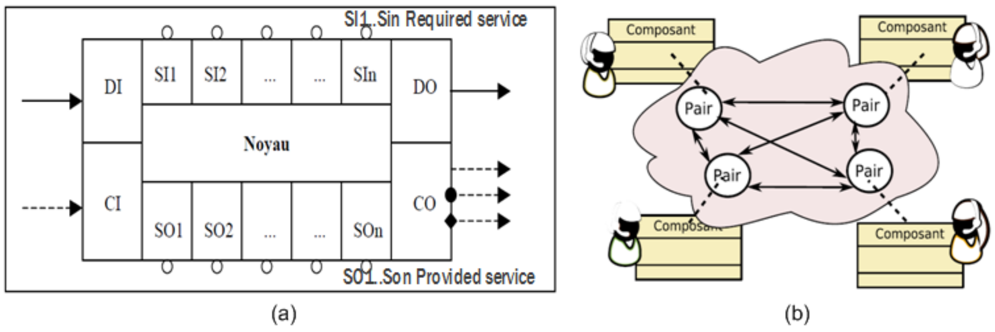


Figure 5.6: An Ugatze component (a) and assigning an agent for each component (b).

5.4.1.2 SoSAA

[DLCO09] provides a framework named Socially Situated Agent Architecture (SoSAA) that aims at integrating the advantages of CBSE and the Agent Oriented Software Engineering (AOSE). This framework requires wrapping functional skills within low-level components before the deliberative component agents manage it in a higher-level (the SoSAA intentional layer). This is achieved through the intermediate layer containing SoSAA adapters. These adapters are agents that interact with the underlying components through defined ways like loading components (load), binding components (wire), changing component parameters (configure), etc. Agents that access to the components are named intentional agents, they can communicate via ACL. Figure 5.7 illustrates the combination between component-based infrastructure framework and a MAS-based high-level infrastructure one.

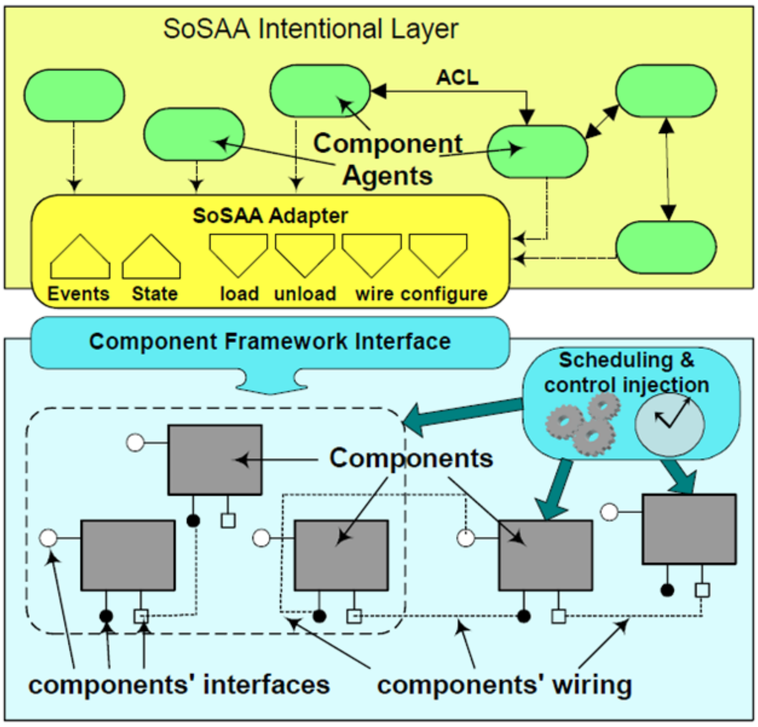


Figure 5.7: SoSAA's hybrid framework strategy [DLCO09].

This work is also listed under the ***Agentification***, since we add agent features to components belonging to a lower level to reach deliberative component agents.

5.4.1.3 ActiveComponent

Braubach and Pokahr propose an intelligent entity named *ActiveComponent* in [BP11]. It facilitates the design of distributed applications. It is based on SCA approach previously viewed in section 5.3 and it integrates the intelligence of BDI agents with it.

The aim of this work is to reach autonomous acting service providers or consumers from passive SCA components. This helps to simulate real world scenarios where there are different active stakeholders.

We choose to list this work under the ***Agentification*** approaches, since it starts by special kind of components communicating by services according to the SCA architecture named SCA components. These SCA components make use of the agents properties like autonomy in a new entity named textitActiveComponent. Figure 5.8 presents an overview of the *ActiveComponent*. The *ActiveComponent* merges autonomous agents with passive components. The agent perceives its environment with its sensors and influences it by its effectors; it owns its goals and internal capabilities. The passive component describes its dependencies with the environment through the services that it requires or the ones that it exposes. The *ActiveComponent* owns agent sensors and effectors and also component required and provided interfaces.

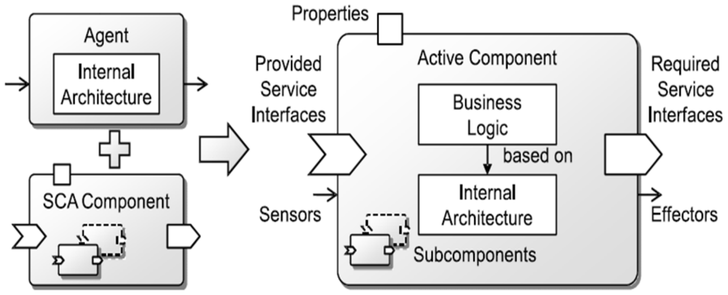


Figure 5.8: An ActiveComponent structure [BP11].

5.4.2 Componentification

Here, we list some works that are based on the agent approach and gain interests of the component one to reach reusable agents, like *AgentComponent* by [KMW03] and another work which does not fully correspond to the definition of the *Componentification* in MaDcAr [GBV06] but it uses components to build agents.

5.4.2.1 AgentComponent

The work of [KMW03] classifies itself under the ***componentifying*** approach. The authors introduce an entity named *AgentComponent* (AC). This entity combines the features of agents and components (see figure 5.9), like reusability and parameterisation of components and communication/interaction and processing complex tasks of agents. An AC has all agent properties like autonomy, reactivity, proactively and interaction because it is an agent in basic but it gains component features.

The authors started from agents with their communication ability and make use of the reusability and parameterization/customization aspects of the component approach. To build an agent, we initialize or reuse **ACs**. The main parts in an AC are: the *Knowledge base* which defines the services to add, remove or use information from this base to other agents, *Slots* which are communication partners, *Ontology's*, *ProcessComponents* **PC** which are the processes that define the behaviour of **AC** instances, they can be added or removed to and from **ACs**.

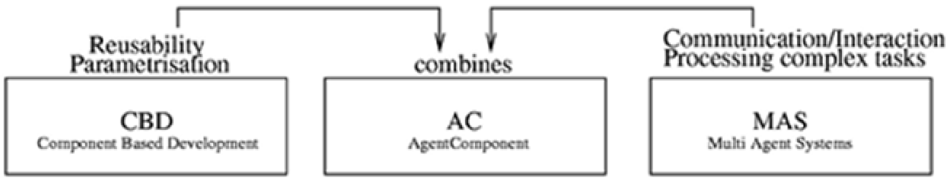


Figure 5.9: The features combination of agent and component approaches [KMW03].

All the services that define the appearance of an **AC** like add / remove (PC, AC instances, slot, ontology) are defined in an interface named *StructuralInterface*. This interface enables to reach reusable and customizable agents by changing its ontology, slots or behaviours through the process components (PCs).

5.4.2.2 MaDcAr

Another interesting work is named **MaDcAr** which stands for Model for automatic and Dynamic component Assembly reconfiguration [GBV06]. It is an abstract model automates the construction of agent (engine) by the component-based approach thanks for its (re-)assembly mechanism. This work cannot be classified under ***Agentification*** or ***Componentification*** because it actually does not add agent or component features but it uses components to build an agent. We have chosen to list it here because MaDcAr profits from the component properties to create an agent. MaDcAr configuration describes a family of similar assemblies as having the same structural constraints. In fact, each configuration consists of a graph of component roles and a set of characterization functions, as shown in figure 5.10. According to the functions definitions, each configuration is more or less appropriate to different contextual situations that may arise during the execution of an application. Then, here we have reusable agents in different contexts.

After presenting works integrating pairs or triples of service, component and agent approaches, we summarize headlines of the presented ways of their integrations in the following section.

5.5 Summary

In this chapter, we browsed most representative works that consider the integration of pair of component, agent and service approaches. Some works consider service and component approaches together. Some works compare the characteristics of the two approaches (reusability composition, dynamicity, loosely and tightly coupled, location transparency) in [LW07], [BL07], [BCM$^+$07] and [HkO11].

Other works interest in mixing components and services. Authors in [MR09] reach to communicate components by services. They combine service oriented architecture and the component approach in a Service Component Architecture. In [Zhu05], authors use

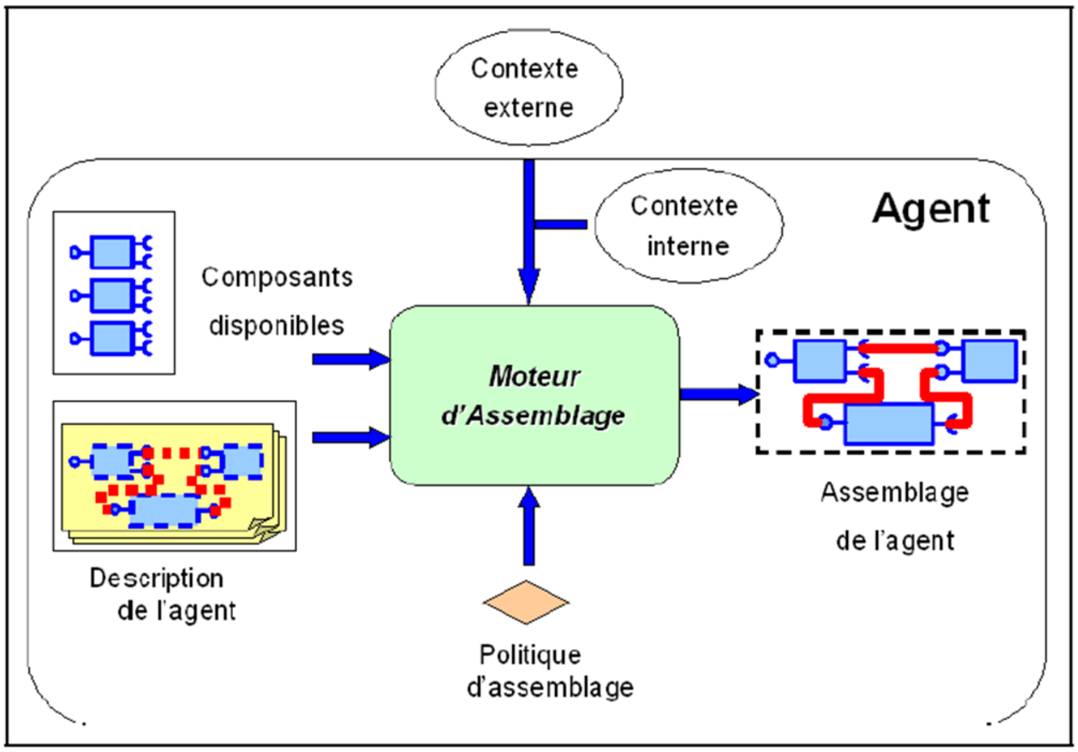


Figure 5.10: Agent Structure in MaDcAr [GBV06].

services to reach more reusable components. Other researches use components to add value for services approaches like in [YP04], the authors provide a *service component* entity for managing the services lifecycles. A similar entity with the same name of *ServiceComponent* is also proposed in [ZYCZ08]; while this intermediate entity is based on ADL models and it owns properties of both component and service.

Services and agents are also key concepts for many researches. Agents are used to add values for service approaches like in ESOA [PAG04] where the authors extend the SOA by adding specific types of agents. The work in [PBBP01] profits from agents ability of negotiation in service composition. The reverse view exists in [ODS09] where the authors make use of service to create reusable organizations of agents which help to build complex systems. [HJR10] browses feasible mappings between service and agent models.

Few works consider agent and component simultaneously. The works of [Lin01, BGZ06] compare the characteristics of these approaches from reusability, autonomy, levels of interaction, etc. We choose to represent the works mixing component and agent through two main categories of *Agentification* and *Componentification*.

Under the *Agentification* category, we find CompAA [AL08] model that attaches an agent for each component to reach auto adaptable components based on the interest of high-level communication of the agents. The work of SoSAA [DLCO09] belongs also to this category where it defines deliberative component agents via controlling functional low-level components by deliberative agents.

Under the *Componentification* category, we find the work of [KMW03] that defines *AgentComponent*, which encapsulates agents in components.

In [GBV06], the authors automate the construction of agents by a component-based approach thanks for its (re-)assembly mechanisms through the *MaDcAr* model. This work does not fully correspond to this category but it makes use of the assembly property of the component approach to build agents. A similar model named MALEVA [BMP06] allows building complex agents behaviours via composite components.

We can already see that many works are already interested in integrating benefits (adding values) of two or three of these approaches in different ways. Their main way to realize this integration is to consider one approach as the specification base and to add features from the other domain on the elements of this base. In other words, they consider that one approach (agent, component or service) is at a higher level or is more preponderant than the other one for the specification of applications. Contrary to these works, we consider component and agent approaches at the same level and with equity.

Lastly, the work in [BP11] integrates the three approaches of component, agent and service in an entity named ActiveComponent. This entity is based on SCA architecture that defines passive components communicating via service and adds the autonomy feature of agents to reach active components. This work owns the same objective as ours (integrating the three approaches of component, agent and service), but it merges their interests in a single entity named ActiveComponent. However, we aim to provide a model offering wide choices to the designer to use components and agents with their classical interests or with new ones.

Most of presented works belong to low levels of conception (mostly related to a specific technology or model). This makes the reusability and the adaptation of these works restricted in special contexts. Our objective is to provide a unified integral model, which can be reused easily in different contexts. This model provides a palette with the widest possibility of conception that allows the designer to choose the suitable elements according to its requirements in the applications specification.

Conclusion

This part started by browsing models under each approach of component, agent and service. We concentrate on the existence of two main concepts of interaction and service (how they exist), in chapters 2, 3 and 4. We ended these three chapters with tables to extract the shared concepts between the studied models in each domain. Then, we browsed in chapter 5, works that already mix two or three of these domains in different ways with clarifying the added value of each domain to the other.

Figure 1 summarizes this state of the art and related work part by representing three circles for each domain (i.e., component, agent and service approaches circles). Each circle contains the main studied models in a domain. We studied more than thirty models, about 10 models for each domain, we cited along this chapter the works that we reuse in some how to achieve our objective. The intersections between these three circles represent main works that belong to two or three approaches jointly. We can find at least four models between pairs of circles (i.e., the intersection between component and agent circles contains five models, the intersection between component and service circles contains five models and the intersection between service and agent circles contains six models). However, the intersection between the three circles of component, agent and service approaches contains only two models ActiveComponent [BP11] and CompAA [AL08]. ActiveComponent is a new single entity that merges the interests of the three approaches. However, in CompAA, we have a model enabling auto adaptable components that makes use of the agent and service approach interests.

In this research, we consider the approaches of component and agent with equity. We aim to overcome the shortages of one approach component (resp. agent) by profiting of the interests of the other approach agent (resp. component) with keeping at the same time the possibility of using the original approaches in their current state.

The next part of this thesis presents our contribution with the help of a running illustrating example. The tables, with the shared concepts of each domain, are the base in designing the general models of each domain respecting our objective.

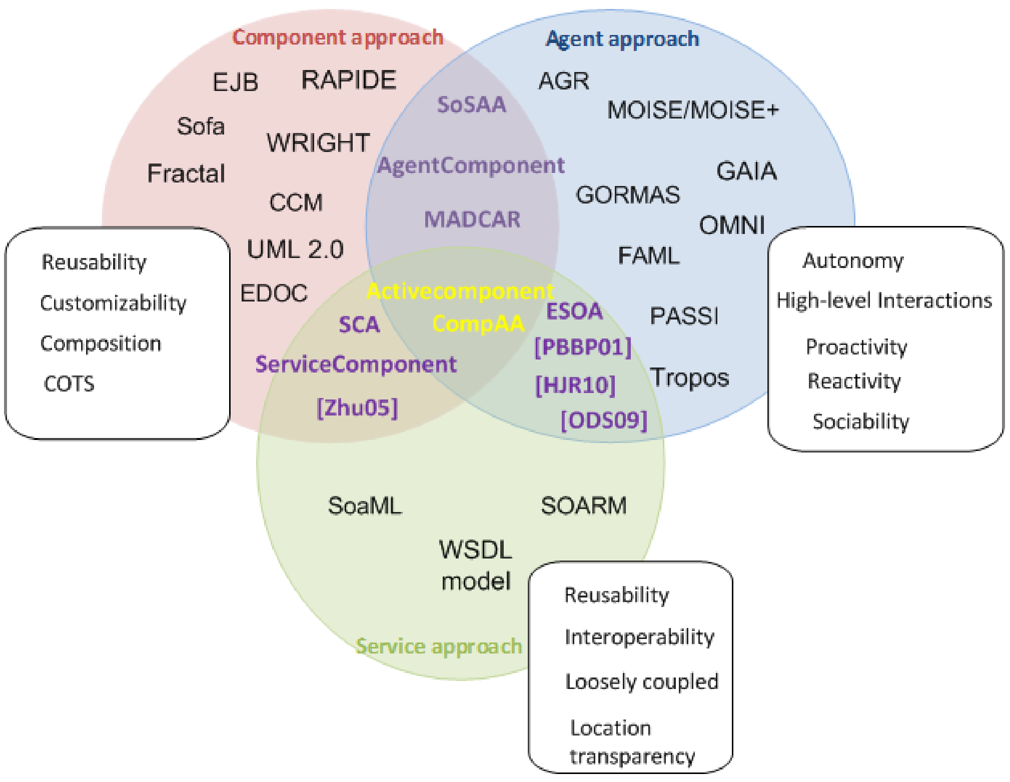


Figure 1: A general view of the related works.

Part III

Contribution

Overview

As we saw previously in the introduction, our contribution consists of several parts, the first part is already presented in the state of the art. We already studied existing models of component, agent and service with the focus on the existence of the two key concepts of service and interaction.

The second part of our contribution is related to the integration of these three domains. This implies to define our own models (Domain Specific Languages (DSLs) [Spi01]) for each domain, since none of the studied models of component, agent and service completely fulfils our requirements. Indeed, we aim to highlight the interaction and service specifications in an application through components, agents or both entities. The need of an abstract service model does not contain the concept of participant explicitly is also from our requirements. These DSLs will be extracted from the already presented comparison tables of each approach (tables 2.3, 3.4, 4.2). We unify the concepts of the studied models in each domain and we consider the two concepts of interaction and service as first-class ones. An additional model named **C**omponent **A**gent **S**ervice **O**riented **M**odel will be defined to reach our objective of integration, where this model enables the application specifications through interacting agents and components.

We start this part by defining a kind of hierarchy between these models depending on MDE principles; we call it our **framework**. This framework defines all kinds of relations (projections and transformations) between component, agent, service and CASOM models. We use a holiday reservation system as an example of application to illustrate our models. Then, we propose our three unified models of component, agent and service with their main concepts before the presentation of **CASOM** one, which allows the application specification with both components and agents. Lately, we provide mappings between the concepts of these models to ensure the feasibility of the transformations from one model to another. Possible variants of mappings and application specification in CASOM are presented in a dedicated chapter named design guide. This design guide chapter groups all the cases needing the designer interventions. Then, we provide the implementation of our contribution in a MDE environment in a dedicated chapter.

Chapter 6

Starting point: Framework and Case study

Contents

6.1 Introduction

This chapter is the gateway of our contribution. It provides a global view of the considered domains (i.e., general models represent each domain) along this research in a single framework. This framework defines also the relations between these domains. It can be interpreted by different entries according to our view (top-down, bottom-up and middle-out). The Model-Driven Engineering approach allows implementing this framework. For this reason, we recall the basic principles of the MDE. A general representation of the used case study to differentiate the application examples conforming to each model is provided at the end of this chapter.

6.2 Framework

We start our contribution by presenting our framework [ACG11, ACGA11] that provides a general view of our models and the relations between them. This framework is a hierarchy that contains four models as shown in figure 6.1. The abstract model is the service

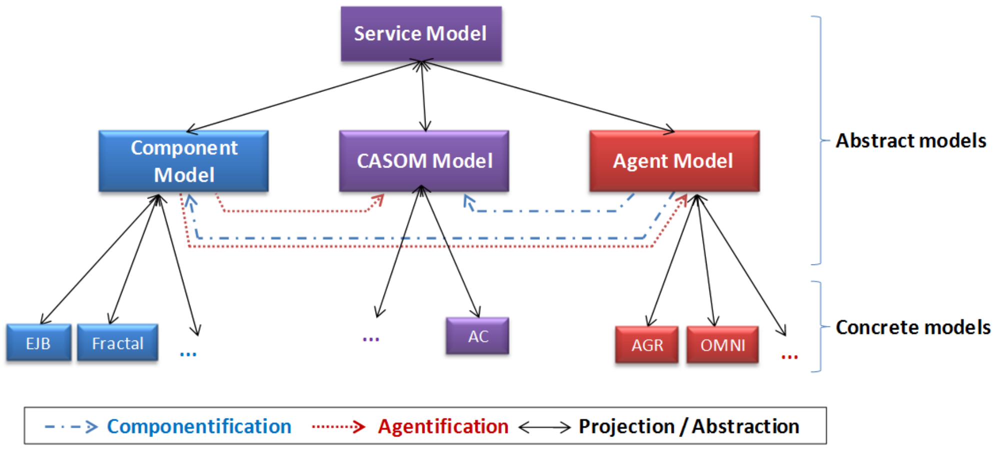


Figure 6.1: Integration framework of service, component and agent approaches.

one; this model defines only the services of an application without requiring to detail which elements (agent and/or components) are implementing them. The models of component, agent and CASOM are at the same level of design. They are the projection of the abstract service model towards agent, component or CASOM models. These four models specify an application in several ways: with only services, with only components, with only agents or with a mix of agents and components. In addition to the four models, we can see the relations between them. The arrows of *Projection/ Abstraction*, *Componentification* and *Agentification* represent these relations in figure 6.1. The *Projection* arrow enables to project any application conforming to our service model towards component, agent and mixed of component agent based applications. The reverse arrow is the *Abstraction* one. The direct *Agentification* (resp. *Componentification*) arrow enables the transformation of any application specification into agents only (resp. components only) conforming our agent (resp. component) model. Arrows between agent and component models towards CASOM one are also named *Agentification* (resp. *Componentification*) according to the model that we start from component (resp. agent) respectively (i.e., the arrow from the component model to CASOM is named *Agentification* since we may add agents to existing components in the application specification via CASOM). This enables to enhance easily any existing component or agent application specification.

Our framework can be interpreted by the following different entry points:

- A **top-down** approach: we can project any application viewed as sets of collaborative services corresponding to our abstract service model into another specification conforming to one of the three other models (component, agent or CASOM). More details on the implementation can further be added by projecting a specification conforming to one of these three models to a specification conforming to concrete implementation models such as EJB and Fractal for components, AGR and OMNI for agent models and AgentComponent (AC) for a mixed agent/component approach (see figure 6.1). The *Projection* arrow corresponds to this view of framework.

- A **bottom-up** approach: any application implemented by any existing component or agent model or mix of them can be abstracted and viewed as a service-oriented application. For example, we can start from an application implemented by Fractal component model, we adapt it to make it conforming to our component model, and then it can be abstracted as only formed of collaborated services conforming to our abstract service model. It is a kind of reverse engineering process. The *Abstraction* arrow corresponds to this view of framework.

- A **middle-out** approach: the emphasis here is on the interaction aspects between components and agents of a system to reach their integration. Services play key roles in their interoperability, as services clarify the specification of what agents or components do. The *Agentification* and *Componentification* arrows correspond to this view of framework.

Our framework respects the principles of the Model Driven Engineering (MDE) [Ken02], then we can call it a MDE framework. Next section puts the light on the MDE approach with its main key concepts that we use along this research.

6.3 Model-Driven Engineering

We present here the basic principles of the MDE that we use to reach our objective along this research. One of the main concerns of MDE is to separate the specification of a system from its implementation. This specification independently of any technology helps to define many platforms for a system. MDE enables also the transformation of system specification in one particular platform to another one. Portability, interoperability and reusability through architectural ways are from the main properties of systems based on MDE approaches [OMG03]. Platform Independent Model (PIM) and Platform Specific Model (PSM) are from the main models in the Model-Driven Architecture (MDA). The MDA is the vision of the MDE using the OMG standards (UML, OCL, MOF . . .). In the PIM, we find the specification of the fixed functionalities of a system that do not change according to the platform. However, the PSM presents the system functionalities in a particular platform.

Figure 6.2 shows a MDE view of the three approaches of component, agent and service. We can see that component and agent models are PIMs, the service model belongs to a higher level of abstraction, it describes the system behaviour and requirement. Therefore, it represents the requirement model, which is a Computation Independent Model (CIM) but this service model can be viewed also as a PIM. These models can be projected to PSMs related to specific models or technologies like EJB for components and AGR on Madkit platform for agents.

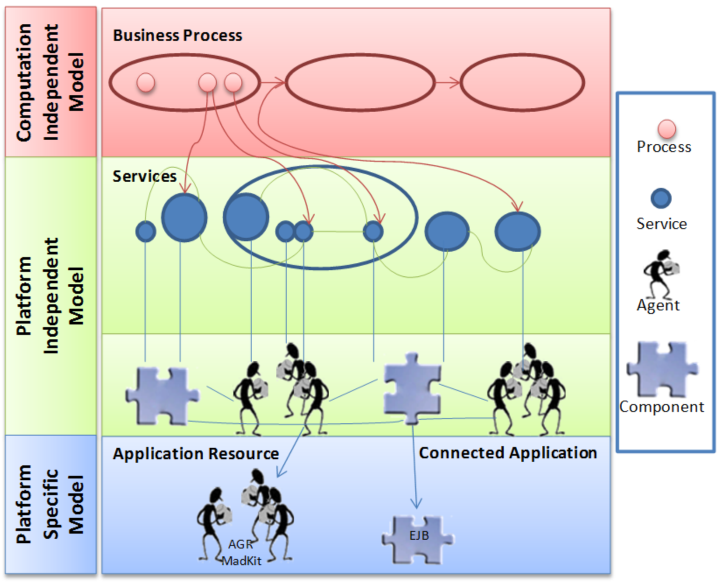


Figure 6.2: A MDE view of the three approaches of component, agent and service.

6.3.1 Main concepts

Figure 6.1 shows our framework with its four models which are actually four PIMs of service, component, agent and mixed of all these domains according to the MDE approach. We present some definitions of MDE concepts that are essential in our framework like models, meta-models and transformations.

6.3.1.1 Model and meta-model

The concept of model is central in MDE, where a model can be projected or transformed to other models. The model itself should conform to a more abstract model named meta-model. Many definitions exist for a model [MFBC10]. For example,

A **model** *is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system* according to [BG01].

We must also cite the famous phrase of [Lud03] that says *nobody can just define what a model is, and expect that other people will accept this definition; endless discussions have proven that there is no consistent common understanding of models.*

A **model modeling** *is relations of representation between modeling concepts, without actually trying to understand the nature of these concepts* according to [MFBC10].

A model itself conforms to a meta-model, which is also a model that defines the concepts used in its conforming models and their relations. The meta-model describes models and the meta-model itself is an instance of a meta-meta-model. We can also say that a meta-model is a model for modeling language.

The OMG [OMG00] provides a hierarchy of four level of meta-modeling (figure 6.3). It contains the real world system in M0 level, the model according to a general modeling language in M1 like UML. It also includes the UML meta-model that defines the used language to describe the model in M2 and the meta-meta model that describes the UML meta-model in M3, which is Meta Object Facility (MOF) [OMG00]. Most of the models in different levels need to be supported with Object Constraint Language (OCL) [OMG10] to remove possible ambiguity.

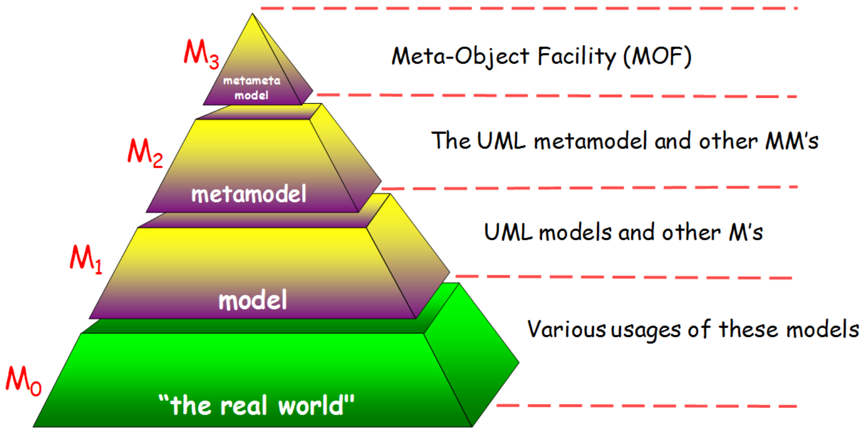

M_3
metameta model
Meta-Object Facility (MOF)
M_2
metamodel
The UML metamodel and other MM's
M_1
model
UML models and other M's
M_0
"the real world"
Various usages of these models

Figure 6.3: Hierarchy levels of meta-modeling.

A Domain Specific Languages (DSL) is a modeling language; it belongs to the M2 level of the previous hierarchy.

A **DSL** *is a language that closely models a certain domain of knowledge or expertise, where the concepts are tied to the constructs of the language*[33].

If we know the domain, a DSL allows us to have more expressive language than a general modeling language like UML; it guides to define common metaphor in specific domains. DSLs help also to separate business logic from application code.

We profit of this short presentation of MDE main concepts to avoid possible ambiguities and to be clear along this document. Our component, agent, service or CASOM models are actually meta-models defining DSLs. They are not model in the sense that are at the M1 level but model in the sense that they define the concepts of a domain.

6.3.1.2 Transformation

Kleppe et al. [KWB03] define as follows:

A **transformation** *is the automatic generation of a target model from a source one, according to a transformation definition.* While a **transformation definition** *is a set of transformation rules that describes how a model in the source language can be transformed into a model in the target language.*

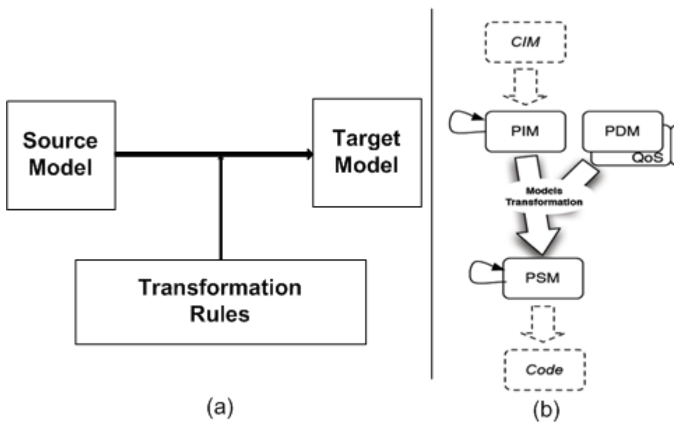


Figure 6.4: MDA transformation process (a). General transformation in MDA (b).

The transformation definition proposed in [KWB03] corresponds to our defined mappings rules between chosen source and target models of component, agent, service and CASOM ones. Transformations[34] can be achieved between different types of models in a model driven architecture like PIMs, PSMs. Figure 6.4 presents a general view of transformations in MDA. Transformations can be vertical according to the level of abstraction and it can be horizontal between models of the same level. The source and target model may have the same meta-model in an endogenous transformation, or different meta-models in exogenous transformation. The arrows of correspondence between the four models according to our framework are actually the transformations between them (see figure 6.1). The *Projection* transformation between service model and the other three models of component, agent and CASOM is vertical one, since our service model belongs to a higher level of abstraction than the other models. The *Agentification* and *Componentification* transformations are exogenous one, since our models represent different languages (DSLs).

In order to realize our objectives, we started by designing general well-defined model for each domain using UML class diagrams. However, UML is not sufficient to realize the parts (models and relations) of our framework. Therefore, we need a framework that supports MDE principles like Eclipse Modeling Framework (EMF) to facilitate the implementation of our framework.

6.3.2 Eclipse Modeling Framework (EMF)

EMF [SBPM09] is a complete open source environment. It supports the MDE principles of models design, transformations, interoperability and code generation. EMF contains a principle meta-meta-model named Ecore. All the information about defined classes and their relations are available in Ecore meta-model. We can implement our DSLs using Ecore meta-meta-model, and the transformations can be defined using Atlas Transforma-

[33]http://www.grails-exchange.com/files/Guilliaume%20LaForge%20-%20DomainSpecificLanguages.pdf.
[34]http://www.theenterprisearchitect.eu/archive/2008/02/18/mda-and-model-transformation

tion Language (ATL) [Gro08]. ATL is a hybrid language mixing declarative (relational) and imperative (operational) constructs that defines the rules of transformations from a source model to a target one. What is an Ecore model?

Ecore model

Ecore meta-meta-model is a general model from which any meta-model can be defined. Figure 6.5 presents a simplified view of Ecore model. We define any meta-model in Ecore by describing its class using EClass element that owns attributes and relations with other classes. The attributes are defined by EAttribute classes, their types are defined by EDataType and the relations between classes are defined by EReference classes.

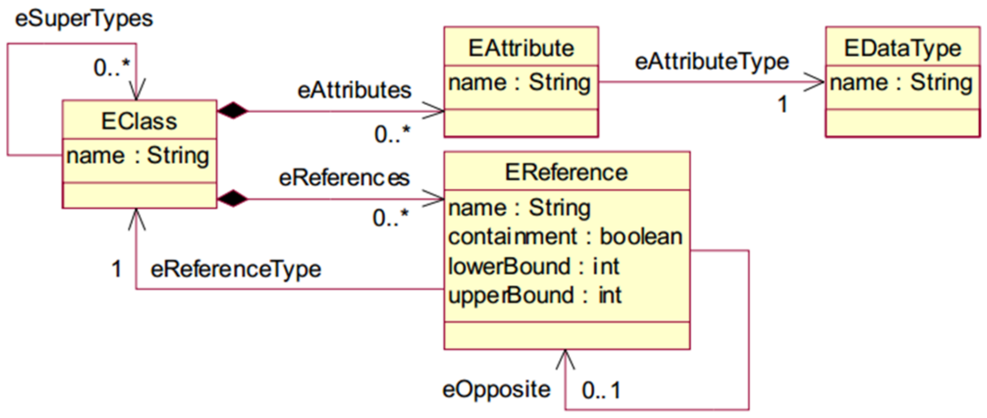


Figure 6.5: Simplified view of Ecore meta-model [MS05].

After presenting our framework in its parts and implementation tool, we browse in the next section the running illustration example that we use along the contribution part to define the applications (models) conforming to our DSLs.

6.4 Case study

Our case study is a typical holiday reservation system. A client addresses a travel agency to find appropriate vacation according to some criteria like number of persons, dates, prices, places and themes (see figures 6.6, 6.7). This travel agency has networks of hotels and airline companies according to geographical zones. In this example, we choose to ignore the payment and bank related issues. A special case of this system will be considered along our contribution part to facilitate the representation of any application specification. It considers only one instance of each actor (client, travel agency, airline company and hotel chain) but for the internal hotels in the hotel chain, they may be multiple.

Figure 6.6 presents the collaboration diagram (the sequence of communications) between instances of main actors in this example. Our proposed scenario is as follows: a client needs to know the possibilities to spend his vacation with defined price **VP** for

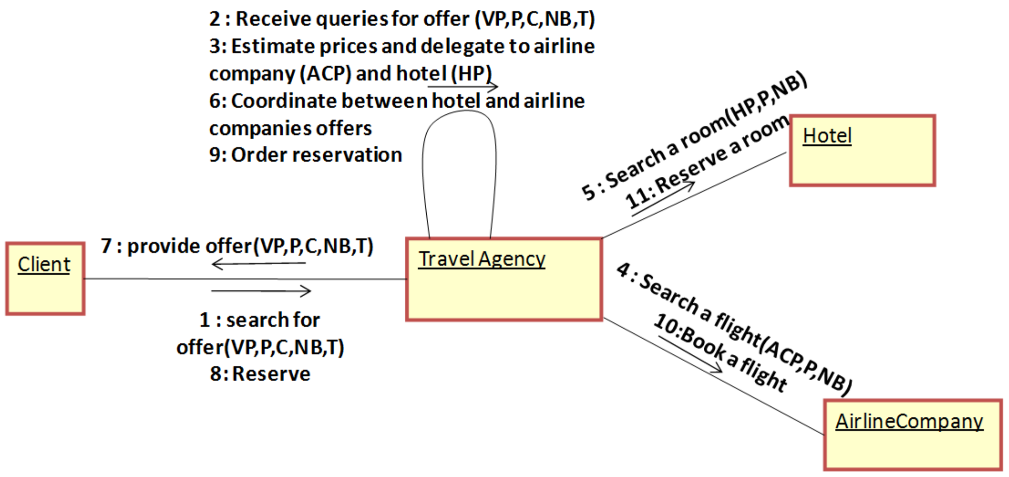

2 : Receive queries for offer (VP,P,C,NB,T)
3: Estimate prices and delegate to airline company (ACP) and hotel (HP)
6: Coordinate between hotel and airline companies offers
9: Order reservation
Client
7 : provide offer(VP,P,C,NB,T)
Travel Agency
1 : search for offer(VP,P,C,NB,T)
8: Reserve
5 : Search a room(HP,P,NB)
11: Reserve a room
Hotel
4 : Search a flight(ACP,P,NB)
10:Book a flight
AirlineCompany

Figure 6.6: The collaboration diagram of our holiday reservation system.

certain period **P** (e.g., Christmas vacation), in a chosen country **C**, number of person **NB** with a desired theme **T** (e.g., to the mountain for skiing). The main steps of the proposed scenario in the collaboration diagram figure 6.6 are the following:

- Step 1: a client addresses the travel agency to find vacation offers. For example, he is looking to spend his vacation in France with budget of 2000 euro, for 10 days, for two persons to learn skiing.

- Steps 2 and 3: the travel agency receives the client query. It estimates new prices for airline company **ACP** and hotel **HP** according to previous deals. The travel agency delegates the query to its partner's airline company and hotel chain according to the country and theme criteria (i.e., airline company and hotel chain in the Alps or Pyrenees mountains).

- Steps 4 and 5: the airline company and the hotel chain provide corresponding offers for a flight and a room reservation respecting the other criteria.

- Step 6 and 7: the agency coordinates offers of the airline company and the hotel chain to provide set of possible vacation offers ranked by price (or any criteria chosen by the client). This agency may need to negotiate some criteria (i.e., price) with the hotel and (or) airline company as there are lots of deals between them (business as loyal customer). It manages to change criteria according to the client ones (if the hotel offered a price higher than the estimated price according to the customer budget, then the agency makes a compromise with this hotel).

- Step 8, 9, 10 and 11: The client chooses the most suitable and interesting vacation offer from the proposed ones and he reserves it. This implies the reservation of a flight and a hotel room.

6.5 The problematic illustrated by an example

The holiday reservation case study (with its actors) is usually implemented according to one approach (i.e., only agent approach or only component one). Figure 6.7 represents a possible architecture of our holiday reservation system. Client and travel agency actors are based on agent approach, while hotel chain and airline company ones are realized by components (they can be primitive component (small airline company) or composite ones (hotel chain X). Travel agency and client actors are based on agent approach, since they need spaces of autonomy in taking decisions and high-level interactions with other parties to negotiate and coordinate with them. Hotel chain and airline company actors are based on component approach, since these actors do not need any mental action in their activities. They reuse their services and make queries on their databases (or on their sub branches one). Many interactions between components and agents exist to exchange their services, like the interaction between the service of *VacationReservation* provided by the travel agency agent to pay to the hotel chain service *ReserveRoom*, that is required by a hotel component (these services will be detailed lately in chapter 7).

A simple type of interaction can be achieved by basic communication, such as a single and a basic service call, but this is not sufficient when the parties need to negotiate for a price or a date to make certain compromise to gain the clients trust.

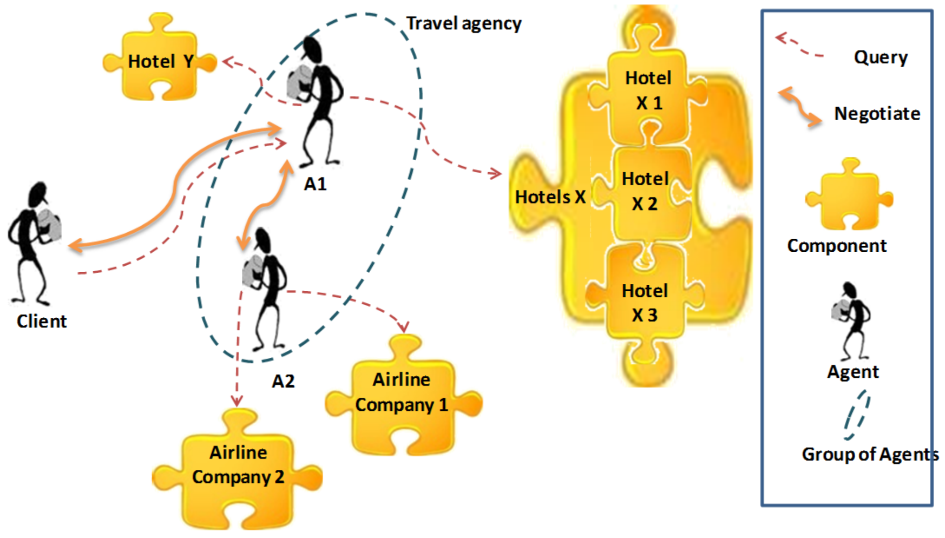


Figure 6.7: A possible architecture of a holiday reservation system.

Unfortunately, there is no such flexibility in components or services communication (see tables 2.3 and 4.2). Then, we need to have more complex communication types such as the existing interaction kinds in the agent approach. At the same time, the services provided by agents could be useful in other contexts (i.e., the travel agency service, that provides the vacation offers for clients, may provide special offers for local product in the target country in addition to the provided vacation offers). However, we cannot reuse agents' services in different contexts because these services are not specified explicitly. The agent services are defined implicitly in its behaviour (more precisely in the functional role of the agent); then, we cannot modify its service due to the absence of the service specification concept (see table 3.4).

These two limitations reflect the need to raise interaction levels between agents and components and to have reusable agents through their well-defined services. In other words, we need to offer component features to agents and conversely.

In the following chapter, we present this case study conforming to DSLs of each domain of service, component and agent separately. The proposed architecture in figure 6.7 can be realized conforming to CASOM model, which allows applications specification with both agents and components and which will be presented lately.

Chapter 7

Our proposed general DSLs of service, component and agent domains

Contents

7.1 Introduction

In this chapter, we propose general models for each domain of service, component and agent. These models are derived from the comparison between the studied models of each domain (already presented in chapters 2, 3 and 4). We use UML class diagram notation along this chapter to provide the graphical representation of our proposed models [ACGA11]. We designed these models from the common concepts in the studied models of each domain. The explicit existence of the two key concepts of interaction and service with considering them as first-class elements are central in these models. They are the main reasons that stand behind our choice to define new representing models for each domain instead of using existing ones. We start this chapter by the representation of our abstract service model.

7.2 The general service model

We start this chapter by the presentation of our unified abstract service model; because it represents a view towards the upcoming models of component, agent and CASOM. This model belongs to a higher level of conception than the last three models and it can be projected towards any of them (see figure 6.1 on page 106). We defined our abstract service model standing on the presented comparison in table 4.2 on page 80. As we stated previously, the SOA views any application as sets of interacted services independently of their locations to satisfy heterogeneous software systems (implemented by different programming languages). From this point of view, we can interpret our proposed model and define its concepts (see figure 7.1).

7.2.1 Main concepts

The main shared concepts between the studied models in chapter 4 are composite and general services, participants, service specifications, operations, messages and interaction via basic types or protocols. Our general service model keeps most of these concepts with slightly changes in their names (to unify the shared concepts name between the four domains) as the service specification concept is named service point, messages are named parameters. We also added the concept of role to our proposed model although it is not of the most shared concept, because we believe in the necessity of defining service responsibilities in the interactions with other services via their service points. Since our service model is abstract and it can be projected to lower-level designed models; we choose to hide the concept of participant, as it will be added in these lower-level models. Table 7.1 provides the definitions of the main concepts in our service model:

A *service* is composed of *operations* through *service points*. A *service point* regroups *operations* in a *required* or a *provided* mode. Then, a *service* is the logical assembly of sets of *operations*. Some sets are provided and others are required (this is done according to the service definition in WSDL for instance). A *service* can be *composite* that is structurally composed of other services, or it can be a leaf *primitive*. Different services interact between each other via *interactions* associated with their service points. A service plays a given *role* in an interaction. There are three main types of *basic interaction*: *function calls* (RPC / RMI) between a required service point and a provided one or a *delegation* between a service point of a composite service and another service point of the same type of one of its internal services or sending *messages* between a sender service point to a receiver one. If an interaction is complex, we use a *protocol* to define it. A protocol allows for instance the definition of orchestrations and choreographies

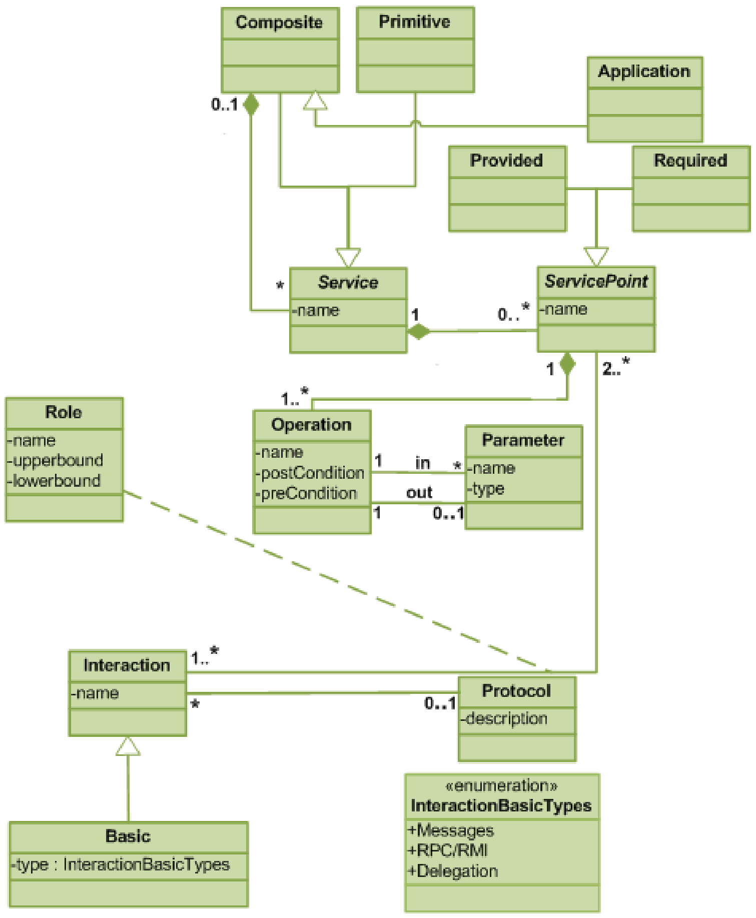


Figure 7.1: Our abstract service oriented model.

between services[35].

Finally, the concept of *application* corresponds to a global application or system. Since an application is formed of interacting services, it is viewed as a special kind of composite service that does not have service points.

In order to remove ambiguity in the proposed service model, we need to define some OCL constraints in the following section.

7.2.2 OCL constraints

In order to reach a well-defined model, we need to define some OCL constraints [WK98] to control this proposed model.

An application is a special kind of composite service because, as it is the root of the composition hierarchy, it is not embedded in a composite. Moreover, it does not own service points:

[35] http://www.soa-in-practice.com/soa-glossary.htm.

Concept	**Definition**
Service	A logical representation of a repeatable business activity that has a specified outcome.
Service Point	An access point of a service. The service specification or interface that determines the role of service and ways to use it (as provided or required).
Role	The responsibilities of a service within an interaction with other services.
Interaction	A kind of action or influence in the dynamic relation between services in order to respond to their needs. It enables services to achieve their expected results.
Protocol	A process flow specification between services. It ensures aggregation (choreography) or recursive composition (orchestration) of services.
Basic	A low-level communication, such as RPC/RMI, message passing or delegation between internal and external service points of a composite service.
Operation	A basic functionality in a service including required pre and post conditions for a service application.
Parameter	Inputs parameters of an operation or output ones to expose its results.

Table 7.1: Definitions of the concepts of the abstract service model

```
context Application inv application:
self.composite.oclIsUndefined() and
self.servicePoint -> isEmpty()
```

All services, except in the particular case of an application, must own at least one service point:

```
context Service inv getServicePoints:
not self.oclIsTypeOf(Application) implies self.servicePoint -> notEmpty()
```

Basic interactions only connect two service points:

```
context Basic inv twoBasicServicePoints:
self.servicePoint -> size() = 2
```

For simplifying the definitions of the constraints dealing with the basic interactions, we consider that the collection of service points for an interaction is ordered. Here are the OCL helpers allowing to get each service point in the context of basic interactions:

```
context Basic def: firstServicePoint : ServicePoint =
self.servicePoint -> first()

context Basic def: secondServicePoint : ServicePoint =
self.servicePoint -> last()
```

RPC/RMI interactions connect a provided service point with a required one. The first service point is the required one and the second is the provided one:

```
context Basic inv rpcrmiSP:
self.type=#RPC/RMI implies
   self.firstServicePoint.oclIsTypeOf(Required) and
   self.secondServicePoint.oclIsTypeOf(Provided)
```

Delegation interactions connect the same type of service points:

```
context Basic inv delegationSP:
self.type=#delegation implies
   self.servicePoint -> forAll ( sp | sp.isTypeOf(Required)) or
   self.servicePoint -> forAll ( sp | sp.isTypeOf(Provided))
```

A RPC/RMI or a message interaction deals with an horizontal assembly, then connects two services embedded directly in the same composite (or application):

```
context Basic inv rpcrmiConnection:
self.type=#RPC/RMI or self.type=#message implies
self.firstServicePoint.service.composite = self.secondServicePoint.service.composite
```

A delegation deals with vertical assembly. The first service point is one of a composite service and the second service point is one of the internal services of this composite:

```
context Basic inv delegationConnection:
self.type=#delegation implies
   self.firstServicePoint.service.oclIsTypeOf(Composite) and
   self.secondServicePoint.service.composite = self.firstServicePoint.service
```

A basic interaction does not have an associated protocol:

```
context Basic inv noProtocolForBasic:
self.protocol.oclIsUndefined()
```

A non basic interaction has a protocol and must at least connect one required service point and one provided service point:

```
context Interaction nonBasic:
self.oclIsTypeOf(Interaction) implies
   not self.protocol.oclIsUndefined() and
   self.servicePoint -> exists ( sp | sp.oclIsTypeOf(Required)) and
   self.servicePoint -> exists ( sp | sp.oclIsTypeOf(Provided))
```

Within a composite, the internal components have different names:

```
context Composite inv uniqueNames:
self.service -> forAll ( s1, s2 | s1 <> s2 implies s1.name <> s2.name)
```

The holiday reservation system conforming to our service model is presented in next section.

7.2.3 Service application example

Let us take the previous case study of the holiday reservation system and represent it conforming to our service model. This representation is provided in figure 7.2.

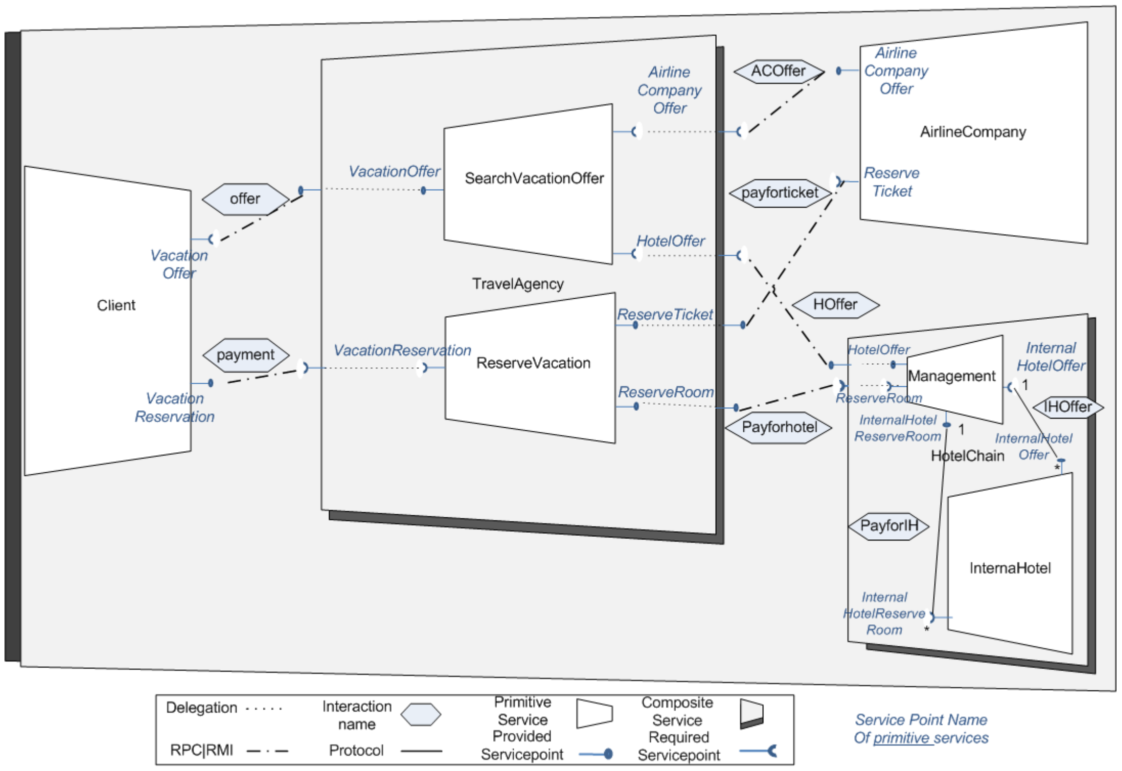


Figure 7.2: Another possible specification of the holiday reservation application specified through our service model.

We choose to present the services (composite or primitive) of each actor with the name of this actor, then, we have four principle services: the ones of *Client, TravelAgency, AirlineCompany and HotelChain*. The composite service of *TravelAgency* and *HotelChain* contains internal primitive services. Any service contains at least one service point, the operations of the internal service point are presented in table 7.2[36], however the external service points of a composite contains the same list of operations than the corresponding internal service point. Here we present the main internal services and service points for each composite, which are the following:

- The client service requires vacation offers after providing his preferences through the *VacationOffer* service point. A client orders a reservation offer through the *Va-*

[36]In this table, we do not detail the types of parameters, and we need to precise that the names of the service and service points do not contains spaces, however regarding the size of the table, we add some spaces where it should not be, e.g., *InternalHotelReserve Room* is actually *InternalHotelReserveRoom*.

cationReservation service point which provides payment information to the travel agency.

- The travel agency provides a list of vacations offers after requiring offers from the airline company and hotel chain in its service *SearchVacationOffer* through *AirlineCompanyOffer* and the*HotelOffer* service points. The travel agency reserves the chosen vacation offer by a client and it delegates the reservation information to the considered airline company and hotel chain in its service *ReserveVacation* through the *ReserveTicket* and *ReserveRoom* service points.

- The airline company provides a list of offers through its service point *AirlineCompanyOffer* and it requires the payment information to accomplish tickets reservation through the *ReserveTicket* one.

- The hotel chain provides offers to reserve rooms in the hotel *Management* service. The last service groups all the internal hotels offers provided by *InternalHotelOffer* service points to build a list of offers. The *Management* service also delegates the payment information to the considered *InternalHotel* service that requires these information to accomplish the room reservation.

These services interact with each other's in an application. We have six main interactions between the services in figure 7.2, the *offer* interaction communicates travel agency offers to the client. The client pays to reserve his chosen offer by the *payment* interaction. *ACOffer* and *HOffer* interactions communicate airline company and hotel chain offers respectively to the travel agency. *payforticket* and *payforhotel* interactions communicate the travel agency payment information to the airline company and hotel chain.

Many interactions of delegation type exist between the service points of composite services and the ones of their internal services. We do not present the name of the external service points in a composite with their delegation interactions and roles in figure 7.2 to keep it legible. Most interactions are from the basic type, else when a service dispatches information over an interaction for other service(s) and receive their responses. In such case, we use protocol type of interaction, like the case of the interaction between *Management* service and the *InternalHotel* services.

Figure 7.3 presents another possible specification of our holiday reservation system according to our service model. In this figure, we choose to represent each actor by a composite service. This choice of representation comes from our choice of abstraction in ignoring participants that provide or use services in the service model. For example, the client services exist in a composite service named *Client*. It contains two primitive service *VacationOffer* and *VacationReservation*, each service has one service points with the same of name of this service. The service points' operations are just the same for the two figures 7.3 and 7.2 as presented in table 7.2. We can find an additional level of composition in figure 7.3, where the *HotelChain* composite service contains composite services of *InternalHotel*s. We precise here the respected mechanism of naming certain

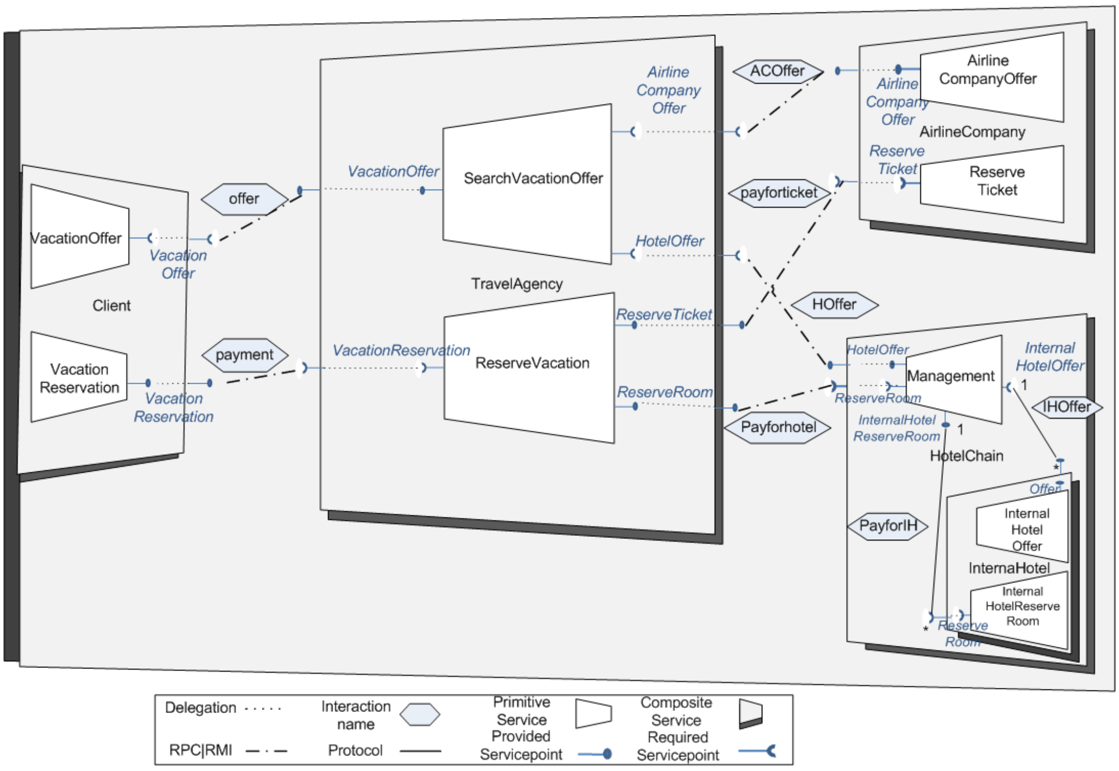


Figure 7.3: Holiday reservation application specified through our service model.

elements, like service points and services in an application specified using our service model.

- The primitive service with only one service point has the same name as its service point name (e.g., the primitive service *VacationOffer* has one required service point with the same name *VacationOffer*).

- The service points of a composite service (external) have the same names as the delegated internal service points, but we put the name of the composite service at the beginning of their names (as prefix before the internal service points' names). For example, the names of the external service points of the composite service *Client* are *ClientVacationOffer* and *ClientVacationReservation*. The *ClientVacationOffer* name comes from the concatenation of the name of this composite service *Client* and the name of internal service point *VacationOffer*.

This service model presents a set of core-shared artefacts between the next three models of component, agent and CASOM; since it can be projected towards any of them.

Composite Service	**Service**	**ServicePoint**		
		Name	Type	Operations
Client	VacationOffer	VacationOffer	Required	VacationOffer(Price, Period, Destination, NumberOfPersons, Theme): listOfVacationOffers
	VacationReservation	VacationReservation	Provided	reserveVacationOffer(NumberOfOffer, PaymentInfo): ConfirmationLetter
Travel Agency	SearchVacationOffer	VacationOffer	Provided	VacationOffer(Price, Period, Destination, NumberOfPersons, Theme): listOfVacationOffers estimateACprice(Price): newPriceForAirlineCompany estimateHotelprice(Price): newPriceForHotel genrateVacationsOffers(listofHotelOffers, listofAirlineCompanyOffers): listOfVacationOffers
		AirlineCompanyOffer	Required	ACOffer(newPriceForAirlineCompany, Period, Destination, NumberOfPersons, Theme): listofAirlineCompanyOffers
		HotelOffer	Required	HotelOffer(newPriceForHotel, Period, Destination, NumberOfPersons, Theme): listofHotelOffers
	ReserveVacation	VacationReservation	Required	reserveVacationOffer(NumberOfOffer, PaymentInfo): ConfirmationLetter creatConfirmationLetter(ACConfirmationLetter, HotelConfirmationLetter): ConfirmationLetter
		ReserveTicket	Provided	reserveACOffer(numberOfACOffer, PaymentInfo): ACConfirmationLetter

Composite Service	**Service**	**ServicePoint**		
		Name	Type	Operations
		ReserveRoom	Provided	reserveHotelOffer(numberOfHotelOffer, PaymentInfo): HotelConfirmationLetter
Airline Company	AirlineCompanyOffer	AirlineCompanyOffer	Provided	ACOffer(newPriceForAirlineCompany, Period, Destination, NumberOfPersons, Theme, Deadline): listofAirlineCompany-Offers
	ReserveTicket	ReserveTicket	Required	reserveACOffer(numberOfACOffer, Pay-mentInfo): ACConfirmationLetter
Hotel Chain	Management	HotelOffer	Provided	HotelOffer(newPriceForHotel, Period, Destination, NumberOfPersons, Theme, Deadline): listofHotelOffers
		ReserveRoom	Required	reserveHotelOffer(numberOfHotelOffer, PaymentInfo): HotelConfirmationLetter
		ManagementInternal HotelOffer	Required	broadcastCriteria(newPriceForHotel, Period, Destination, NumberOfPersons, Deadline): listofInternalHotelOffers
		ManagementInternal HotelReserveRoom	Provided	reserveInternalHotelOffer (numberOfIn-ternalHotelOffer, PaymentInfo): Room-Number
InternalHotel	InternalHotelOffer	InternalHotelOffer	Provided	InternalHotelOffer(newPriceForHotel, Period, Destination, NumberOfPersons, Theme, Deadline): listofInternalHotelOf-fers
	InternalHotelReserve Room	InternalHotelReserve Room	Required	reserveInternalHotelOffer(PaymentInfo, numberOfInternalHotelOf-fer):RoomNumber

Table 7.2: Operations of the service points

7.3 The general component model

Here we present our general component model that represents the domain of components. Similarly to the previously proposed service model, we base on the shared concepts in table 2.3, between studied models in chapter 2 to define our general component model. The concepts of this model cover almost the ones of studied models with the originality of the explicit existence of service and interaction concepts.

7.3.1 Main concepts

The main shared concepts among studied models in chapter 2 are general component, ports, interfaces, composition, connectors and low-level interactions (RMI, events call and delegation). Our proposed component model keeps most of these concepts with slightly changes in their names as the general component becomes primitive one, the composition becomes the composite component, ports become service points and interfaces become explicit service (see figure 7.4). These changes in some concept names are achieved in order to unify the names of concepts in three models (e.g., the concepts of port and service). The concept of role exists in our general component model, although it is not from the most shared concepts of the studied component models. But because it defines the component responsibilities through interactions with other components via their service points. We added also the concept of component connector to reify possible high-levels of interaction and to make them reusable. The gap for adding this concept is tight, since it extends existing connector and component entities. Our model contains also the concept of protocol to specify possible complex interactions. Table 7.3 provides the signification of the main concepts in our component model:

A *component* with well-specified interaction points named *service points* can be primitive (simplest type of component) or composite when it is structurally composed of other components. The primitive component is the basic entity in an assembly of components (horizontal assembly) or their hierarchical composition (vertical assembly). Each service point is associated to a *service* in a *required* or a *provided* mode. A service is composed of *operations*. Different components interact with each other via *interactions* associated to their service points. A component plays a given *role* in an interaction. There are three main types of basic interaction: *functions calls* (RMI / RPC) between required and provided service points of different components, sending messages between sender and receiver components service points or *delegation* between a service point of a composite service and another service point of the same type of one of its internal components. If a more complex interaction is required between the service points of different components, it is realized by a *connector* which implements a *protocol*. A *component connector* is a special kind of component dedicated to the communication between components [CBJ02]. Then, it is also a connector and it implements a protocol.

Finally, the concept of *application* corresponds to a global application or system. Since an application is formed of interacting components, it is viewed as a special kind of composite component.

We can see that the concepts of role, interaction, protocol, operation, parameter

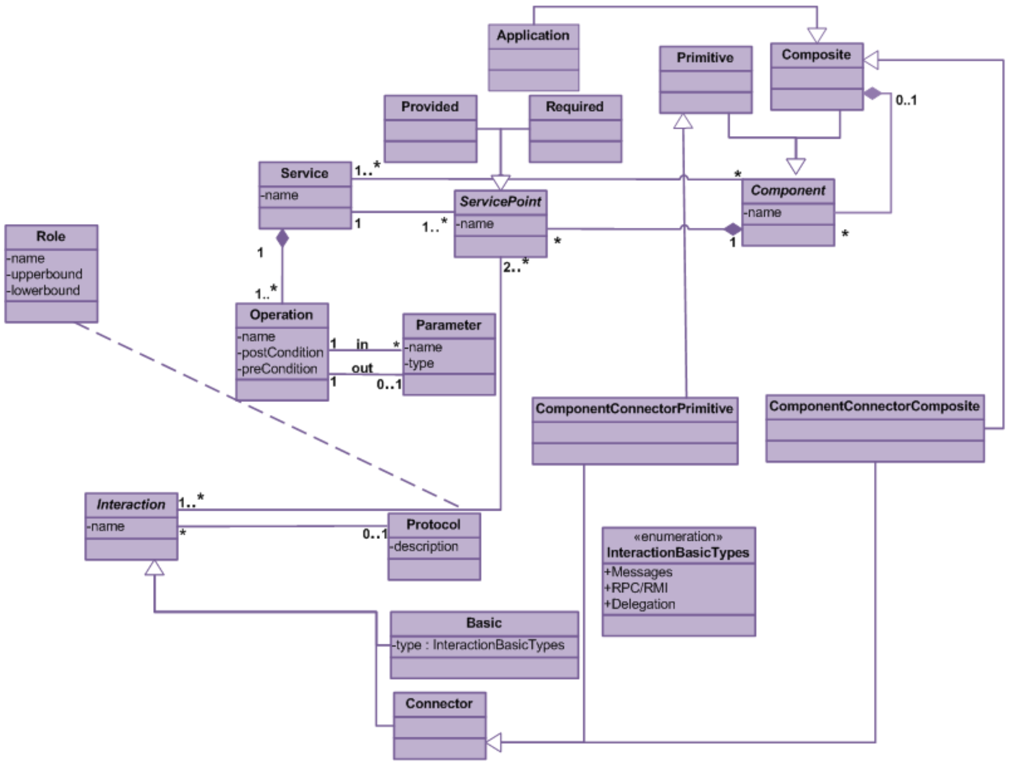


Figure 7.4: Our unified component model.

and application have almost the same significance as the ones in the already proposed service model. The concepts of service and service point are slightly modified, where the operation are related to the service concept in the component model instead of being related to the service point concept in the service model. In order to remove ambiguity in our proposed component model, we need to define some OCL constraints in the following section.

7.3.2 OCL constraints

The OCL constraints of the component model are very similar to the ones of the service model as these models are very close. Most of the differences deal with modifying the context or the navigation of the OCL expressions but their goals are the same.

An application is a special kind of composite component because, as it is the root of the composition hierarchy, it is not embedded in a composite. Moreover, it does not own service points:

```
context Application inv application:
self.composite.oclIsUndefined() and
self.servicePoint -> isEmpty()
```

Concept	**Definition**
Component	A reusable abstract entity with well-specified access points (service points) to expose or use services
Service point	A port of a component either exposing some of its services (provided specialization) or specifying services it has to use (required specialization).
Service	It is a static unit of functions (a classical component interface).
Role	The responsibilities of a component through its service points within an interaction with other components.
Interaction	Communication between components through their service points to exchange their services.
Protocol	A complex interaction specification between components.
Connector	The explicit representation of a complex interaction, its behaviour is specified by a protocol.
Component Connector	An interaction at the same level of a component [CBJ02]. It can be primitive or composite.
Basic	A low-level communication, such as RPC/RMI, messages passing or delegation between component service points for a component composition.
Operation	It is a basic functionality of a service including the required pre and post conditions for a component application.
Parameter	Inputs parameters of an operation or output ones to expose its results.

Table 7.3: Definitions of the concepts of the general component model

All components, except in the particular case of an application, must own at least one service point:

```
context Component inv getServicePoints:
not self.oclIsTypeOf(Application) implies self.servicePoint -> notEmpty()
```

Basic interactions only connect two service points:

```
context Basic inv twoBasicServicePoints:
self.servicePoint -> size() = 2
```

For simplifying the definitions of the constraints dealing with the basic interactions, we consider that the collection of service points for an interaction is ordered. Here are the OCL helpers allowing to get each service point in the context of basic interactions:

```
context Basic def: firstServicePoint : ServicePoint =
self.servicePoint -> first()

context Basic def: secondServicePoint : ServicePoint =
self.servicePoint -> last()
```

RPC/RMI interactions connect a provided service point with a required one. The first service point is the required one and the second is the provided one:

```
context Basic inv rpcrmiSP:
self.type=#RPC/RMI implies
   self.firstServicePoint.oclIsTypeOf(Required) and
   self.secondServicePoint.oclIsTypeOf(Provided)
```

Delegation interactions connect the same type of service points:

```
context Basic inv delegationSP:
self.type=#delegation implies
   self.servicePoint -> forAll ( sp | sp.isTypeOf(Required)) or
   self.servicePoint -> forAll ( sp | sp.isTypeOf(Provided))
```

A RPC/RMI or a message interaction deals with an horizontal assembly, then connects two components embedded directly in the same composite (or application):

```
context Basic inv rpcrmiConnection:
self.type=#RPC/RMI or self.type=#message implies
self.firstServicePoint.component.composite = self.secondServicePoint.component.composite
```

A delegation deals with vertical assembly. The first service point is one of a composite component and the second service point is one of the internal components of this composite:

```
context Basic inv delegationConnection:
self.type=#delegation implies
   self.firstServicePoint.component.oclIsTypeOf(Composite) and
   self.secondServicePoint.component.composite = self.firstServicePoint.component
```

A basic interaction does not have an associated protocol:

```
context Basic inv noProtocolForBasic:
self.protocol.oclIsUndefined()
```

A connector has a protocol and must at least connect one required service point and one provided service point:

```
context Connector protocolConnector:
not self.protocol.oclIsUndefined() and
self.servicePoint -> exists ( sp | sp.oclIsTypeOf(Required)) and
self.servicePoint -> exists ( sp | sp.oclIsTypeOf(Provided))
```

Within a composite, the internal components have different names:

```
context Composite inv uniqueNames:
self.component -> forAll ( c1, c2 | c1 <> c2 implies c1.name <> c2.name)
```

Our holiday reservation system conforming to our component model is presented in next section.

7.3.3 Component application example

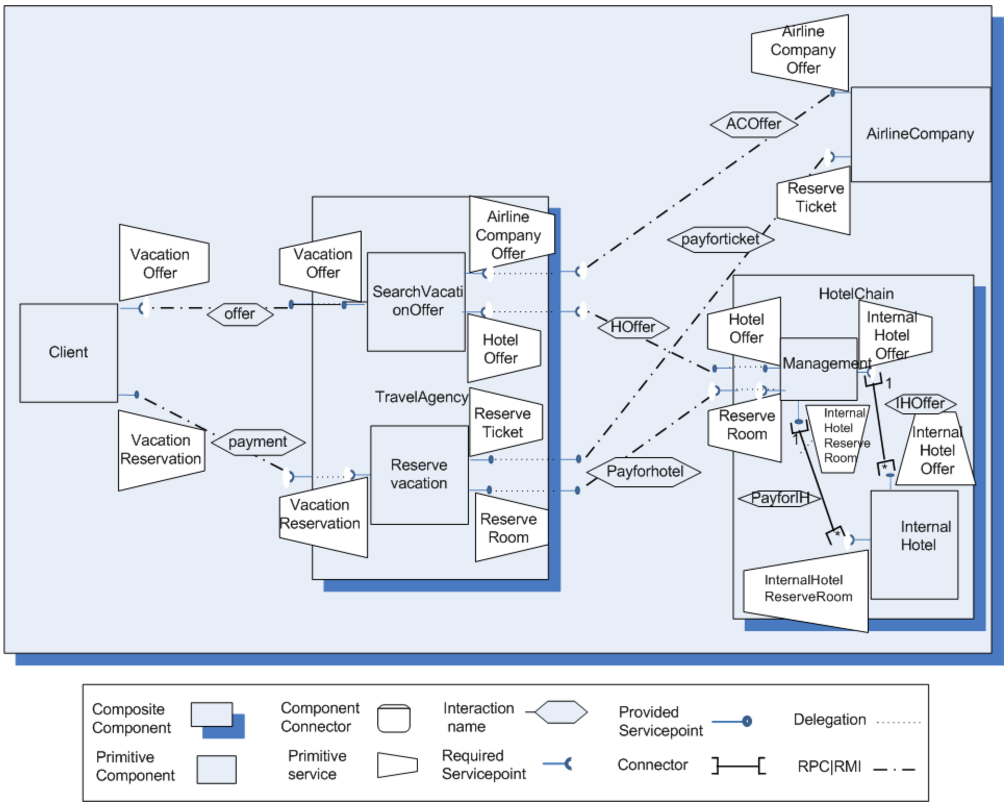


Figure 7.5: Holiday reservation application specified through our component model.

We present here our running application example of the holiday reservation system according to our component model in figure 7.5. Some details are not presented in this figure to avoid overlapping view, like role names of service points over interactions and sets of operations associated with service points (these operations are just the same as the one already presented in 7.2). We can always distinguish the four main actors of *Client*, *TravelAgency*, *AirlineCompany* and *HotelChain* in four components. The *Client* primitive component requires *VacationOffer* service and provides *VacationReservation* one. This component interacts with the *TravelAgency* one via two interactions (offer and payment). The *TravelAgency* composite components receives the offers from *AirlineCompany* and *HotelChain* over the *ACOffer* and *HOffer* interactions respectively. The connector appears in the *HotelChain* under the form of a relation that dispatches the information from *Management* component to other *InternalHotel* ones.

This model is much related to the service one, they share many concepts. The component model implements the service one, where the concept of component implements the one of service in the service model and the concept of connector implements the one of interaction (it is possible to use a component connector concept instead of connector one to raise the interaction level to be at the same one as a component).

7.4 The general agent model

In this section, we propose our unified organizational MultiAgent model after studying different models and comparing their essential concepts in table 3.4. As we stated previously, we limit our research in organizational MultiAgent models, since an organization is needed whenever agents work together in a shared environment [SH05]. This organization structures the interactions of its participating agents.

7.4.1 Main concepts

Figure 7.6 presents our proposed agent model following the same way as previously presented models (by keeping common concepts between the studied models in table 3.4 and adding other essential not shared concepts that serve us to reach our objective).

The common concepts according to table 3.4 are agents, organizations, roles, tasks, interactions by protocols, ACL and coordination, goals, service (implicit under role) and implicit environment. All these concepts are considered in our proposed model except environment one (to avoid considering some issues related to agents and resources interaction). Slightly changes are applied on some of these concepts in our model, we choose to represent the concept of role by two separated kinds as presented in [Lin01]. The interactional role that defines agent roles through interactions and the functional one that defines agent behaviours (it describes agent services) within a group. The concept of service becomes explicit and abstract, it is specialized by functional role concept. Since we aim to detail all kinds of interaction in our proposed model, we added the negotiation kind, although it is not from the most shared concepts between studied models. The concept of capability is also considered in our model to represent specific concept in some agent model like agent capabilities of perception, taking decision and communication. Our model contains also the group concept since it provides additional level of agents' assembly.

Table 7.4 provides the definitions of the main concepts of our agent model.

Concept	Definition
Organization	The overall architecture of a system organized as collaborative services, each modeled as an agent, into a coherent whole [SH05]. It defines also the authority between sets of agents in a group or between groups.
Group	A structural entity composed of roles and agents. An agent can be member of one group, if and only if he plays a role associated to this group.
Agent	An autonomous rational entity that plays a defined role in a group. It owns capabilities, which enable it to play its roles (functional) within a group.
Goal	It represent functional requirements of an organization. It could be divided in to sub goals related to agents or groups.

Concept	**Definition**
Role	The responsibilities and tasks that an agent assumes within its interactions with other agents. It can be of two types: **FunctionalRole** or **InteractionalRole** [Lin01].
Service	It is what an agent may provide for other agents within a group. It is an abstract concept extended by the **functionalRole** one, which defines itself the agent behaviour.
Interaction	The dynamic relation between agents through their played roles. It has different types: *communication*, *coordination*, *negotiation*.
Protocol	A specification of types of interaction between agents from basic types like Message Transferring Protocol (MTP) to the negotiation and coordination ones.
FunctionalRole	It defines agent behaviours and services within a group, in addition to the definition of the tasks that must be carried out in the system.
InteractionalRole	A classical role used in the definition of interaction protocols. This role enables an agent to expose or to request the needed services.
Capability	The knowledge or capacities that an agent owns to play his roles in a group or to participate within an interaction. We just consider the already defined capabilities like perception, communication, reasoning and taking decision.
Task	It is a unit of action that an agent performs. It may be similar to operation concept, but here we do not need to specify input and output parameters. Then an operation belongs to a lower level than a task.
Communication	It is used through agent languages like FIPA ACL, KQML and basic protocols like MTP.
Coordination	It is used among agents that share some resources, to avoid conflicts, such as coordination by planning, coordination through the organizational structure or by signing contracts.
Negotiation	It is needed when a compromise has to be reached between some agents to solve occurred conflicts, such as auction and fish market protocols.
Operation	Basic functionality of a service including the required pre and post condition for the component application.
Parameter	Input parameters of an operation or output ones to expose its results.

Table 7.4: Definitions of the concepts of the general agent model

An *organization* is associated with a functional *goal* which can be decomposed to several sub-goals. These sub-goals are associated with *groups* that present the structural entities of an organization and define the actors and their roles within it. In this context, an *agent* associated with a sub-goal or individual one plays *roles* in a group. The agent

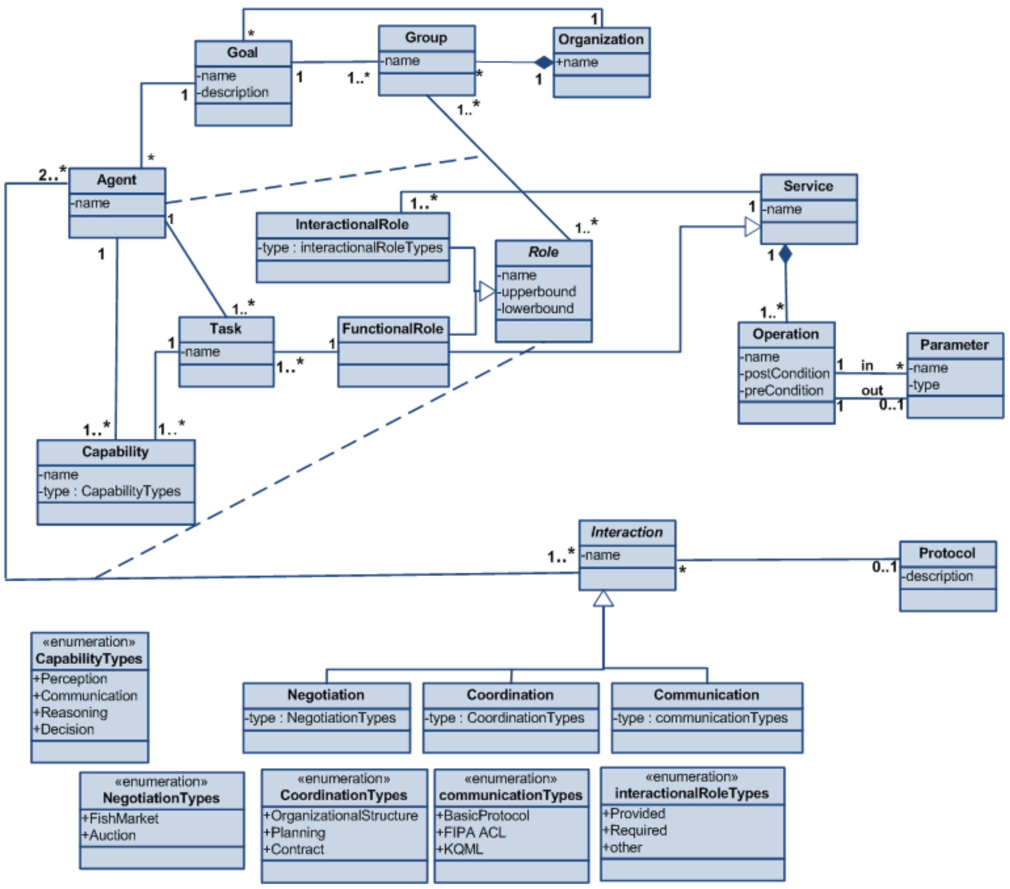


Figure 7.6: Our unified agent model.

individual goals may lead to possible conflicts with the collaborated ones. These conflicts may be resolved through the interaction specification (i.e., coordination). The *functional role* allows the specification of the agent's tasks or behaviours through *services*. A service is a set of *operations*. An *interactional role* defines the responsibilities assumed by an agent in an interaction with other agents. Particularly, it allows to provide or to require services. Capturing this service will be relatively complex because of the autonomy characteristic of an agent. An agent may change its behaviour through its interaction with other agents, this leads to changes in its services. For these reasons, agents must have an agreement with each other to keep certain services through their interactions (i.e., contract). To play a given role, an agent must own certain *capabilities* like perception, communication and reasoning.

Generally, the agents interact with each other through their interactional roles by using high-levels interactions (*communication languages* (such as FIPA-ACL), *negotiation*, *coordination*) based on *protocols*.

It is obvious that the agent model is richer in its concepts than the previously presented models of service and component, since it addresses the design of more open systems. However, we can find core shared concepts between the three proposed models like operations, parameters, roles, interactions, protocols and services (abstract).

Next section provides needed OCL constraints to remove possible ambiguity in this model.

7.4.2 OCL constraints

An interaction is associated with interactional roles:

```
context Interaction inv interactionRoles:
self.role -> forAll ( r | r.oclIsTypeOf(InteractionalRole) )
```

A group is associated with functional roles:

```
context Group inv groupRoles:
self.role -> forAll ( r | r.oclIsTypeOf(FunctionalRole) )
```

An interaction connects at least one required and one provided interactionnal roles:

```
context Interaction inv requiredAndProvided:
self.role -> exists ( r | r.oclAsTypeOf(InteractionalRole).type=#Provided) and
self.role -> exists ( r | r.oclAsTypeOf(InteractionalRole).type=#Required)
```

Within an organization, the groups have different names:

```
context Organization inv groupUniqueNames:
self.group -> forAll ( g1, g2 | g1 <> g2 implies g1.name <> g2.name)
```

Within a group, the agents have different names:

```
context Agent inv agentUniqueNames:
self.agent -> forAll ( a1, a2 | a1 <> a2 implies a1.name <> a2.name)
```

For an agent, each of its interactional roles is associated with a service. This service is actually a functional role and this functional role must belong to the same agent:

```
context InteractionalRole inv roles:
self.agent -> forAll ( a | (self.service.oclAsTypeOf(FunctionalRole)).task.agent = a)
```

Next section presents our holiday reservation system conforming to our agent model.

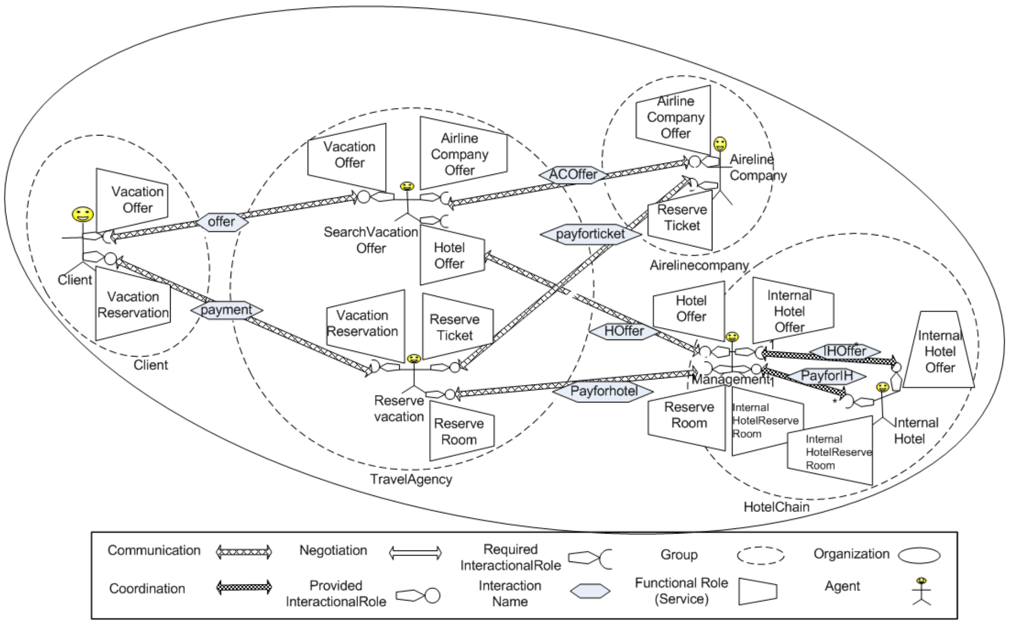


Figure 7.7: Holiday reservation application specified through our agent model.

7.4.3 Agent application example

Figure 7.7 presents the same holiday reservation system conforming to our agent model. There are four principle groups in the organization representing our holiday reservation system, respecting the main actors in this system: *Client*, *TravelAgency*, *AirlineCompany* and *HotelChain*. Since agents cannot be directly contained in an organization without belonging to a group, then we may have groups of one agent like *Client* and *AirlineCompany*. Most of defined agents interact with each other via communication type of interaction like the interaction between the *client* agent in the *client* group that provides the *VacationReservation* service to the *ReserveVacation* agent of the *TravelAgency* group. However, we can find an interaction by coordination in the *HotelChain* group where the *Management* agent coordinates all internal hotel offers provided by *InternalHotel* agents through provided *InternalHotelOffer* services which are required by the *Management* agent. Interactional role, task and capability instances are not presented in this figure to keep it legible.

7.5 Summary

After presenting our proposed general models of service, component and agent domains, we find that there are core common concepts between these models. We think that it would be interesting to present these three models in a single figure that helps in visualizing common concepts between them to avoid their repetition. Figure 7.8 presents a unified view of the three previously proposed Models. Role, service, interaction, protocol, operation and parameter concepts are obviously the common ones between the three models. Some concepts are shared only between service and the component models like required and provided service point, basic type of interaction and application. Table 7.5 groups the definitions of the concepts in figure 7.8 which are already presented earlier in this chapter separately according to their models.

We believe that after optimizing the concepts with the same semantic in figure 7.8, we can achieve to define an additional model. This resulted model will authorize the application specification using the two concepts of component and agent simultaneously through interoperable service. It integrates also the interests of these approaches as it contains their key concept representing key properties of each domain. Next chapter presents this outcome model, which is named Component Agent Service Oriented Model.

Concept	**Model**	**Definition**
Service	Service	A logical representation of a repeatable business activity that has a specified outcome.
	Component	A static unit of functions (a classical component interface).
	Agent	It is an abstract concept specified by the **functionalRole** concept that defines itself the agent behaviour.
Service Point	Service	A port of a component either exposing or using services.
	Component	A static unit of functions (a classical component interface).
Role	Service	The responsibilities of a service within an interaction with other services.
	Component	The responsibilities a component takes through its service points within an interaction with other components.
	Agent	The responsibilities and tasks that an agent assumes within an interaction with other agents. It can be of two types: **FunctionalRole** or **InteractionalRole** [Lin01].
Interaction	Service	A kind of action or influence in the dynamic relation between services.
	Component	Communication between components through their service points to exchange their services.
	Agent	The dynamic relation between agents through their played roles. It has different types: *communication*, *coordination*, *negotiation*.
Protocol	Service	A process flow specification between services that ensures services choreography or orchestration.
	Component	A complex interaction specification between components.
	Agent	A specification of the types of interactions between agents from basic types like Message Transferring Protocol to the negotiation and coordination ones.

Concept	**Model**	**Definition**
Operation	Service Component Agent	Basic functionality of a service including the required pre and post condition for the component application.
Parameter	Service Component Agent	Inputs parameters of an operation or output ones to expose its results.
Basic	Service Component	A low-level communication, such as RPC/RMI, message passing or delegation between services/components for a service/component composition.
Component	Component	A reusable abstract entity with well-specified access points (service points) to expose or use services
Connector	Component	The explicit representation of a complex interaction, its behaviour is specified by a protocol.
Component Connector	Component	An interaction at the same level of a component [CBJ02]. It can be primitive or composite.
Organization	Agent	The overall architecture of the system organized as collaborative services that defines the authority between sets of agents in a group or between groups.
Group	Agent	A structural entity composed of roles and agents. An agent can be member of one group if and only if he plays a role associated with this group.
Agent	Agent	An autonomous rational entity that plays a defined role in a group.
InteractionalRole	Agent	A classical role used in the definition of interaction protocols. This role enables an agent to expose or to request the needed services.
Capability	Agent	The knowledge or capacities that an agent owns to play his role in a group or to participate within an interaction.
Task	Agent	A unit of action that the agent performs, it may be similar to the definition of operation concept, but without input or output parameters.
Communication	Agent	It is used through an agent language like FIPA ACL, KQML and basic protocols like MTP.
Coordination	Agent	It is used among agents that share some resources, to avoid conflicts.
Negotiation	Agent	It is needed when a compromise has to be reached between some agents to solve occurred conflicts.

Table 7.5: Definitions of the concepts of service, component and agent models

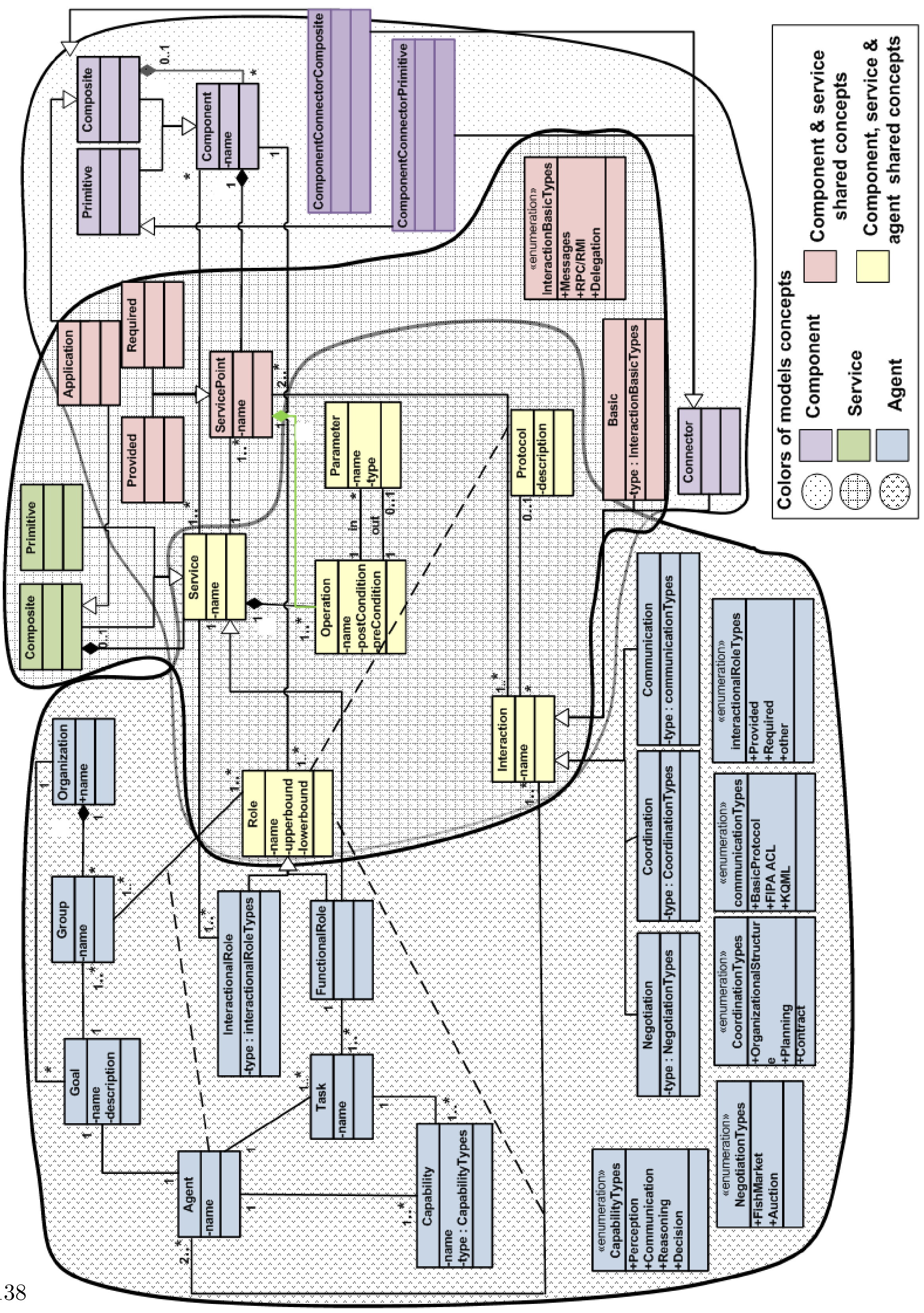


Figure 7.8: Unified view of the three general models (component, agent and service) [ACGA11].

Chapter 8

CASOM: a DSL for application specification by components, agents and services simultaneously

Contents

8.1 Introduction

Figure 8.1 provides a unified view of the already proposed component and agent models. This figure excludes the service model from figure 7.8. Indeed, primitive services with their related concepts exist in both component and agent models, however the composition is on the component (i.e., composite) and agent (i.e., group)(service providers or customers) levels. In the global view of figure 8.1, we can see the feasibility of designing a single model that allows the application specification by interoperable components and agents through services. We may ask several questions to clarify and analyze how to start from the component and agent models to design the new Component Agent Service Oriented Model (CASOM). We can find the concepts related to the interests of one approach available to the other one in CASOM, it provides wide ways to combine and use these interests (see figure 8.3).

8.2 Main concepts

CASOM is surely based on the contents of component and agent models concurrently. We ask the following questions in order to design our target model. The answers for these questions clarify the concepts that we need to add or remove from figure 8.1 to reach CASOM concepts. The questions are:

- What are the same elements in the three models or which are sufficiently close to be considered as equivalent?
- What are the elements coming from different models that can be abstracted under a same general concept in order to express that they are interchangeable?
- What are the elements of a model that are not available in the other one but can nevertheless be used by its elements?
- What are the elements that are fully specific to a domain? What are the secondary elements of a model that are not required to be kept?

In the following four sections, we answer to the previously proposed questions to detail the main concepts in CASOM.

8.2.1 Common or equivalent elements

We can see on the unified view of the component and agent models (figure 8.1) that we have several common elements, which are *Service*, *Operation Parameter*, *Role*, *Interaction* and *Protocol*. This is logical as our key idea of the integration of component and agent approaches is to consider a service as a business abstraction of an entity (an agent or a component) and to make interacting entities through their services. Then, we will retrieve these concepts without any changes in CASOM.

However, there are some differences in the way to manage the services. On the component side, a component owns service points and a service point, in a provided or required mode, is associated to a service. A component plays roles in interactions through its service points. On the agent side, a service is reified through a functional role and an agent offers or requires such functional roles (service) thanks to the interactional roles associated to an interaction. We decide to unify these elements in CASOM using the concepts of the component side: functional and interactional roles of an agent become services, service points and roles. We choose to use the component view for the service definition as the service of a component (that is, a component interface) since it is an essential and mandatory part of its definition whereas the service of an agent is usually more implicit.

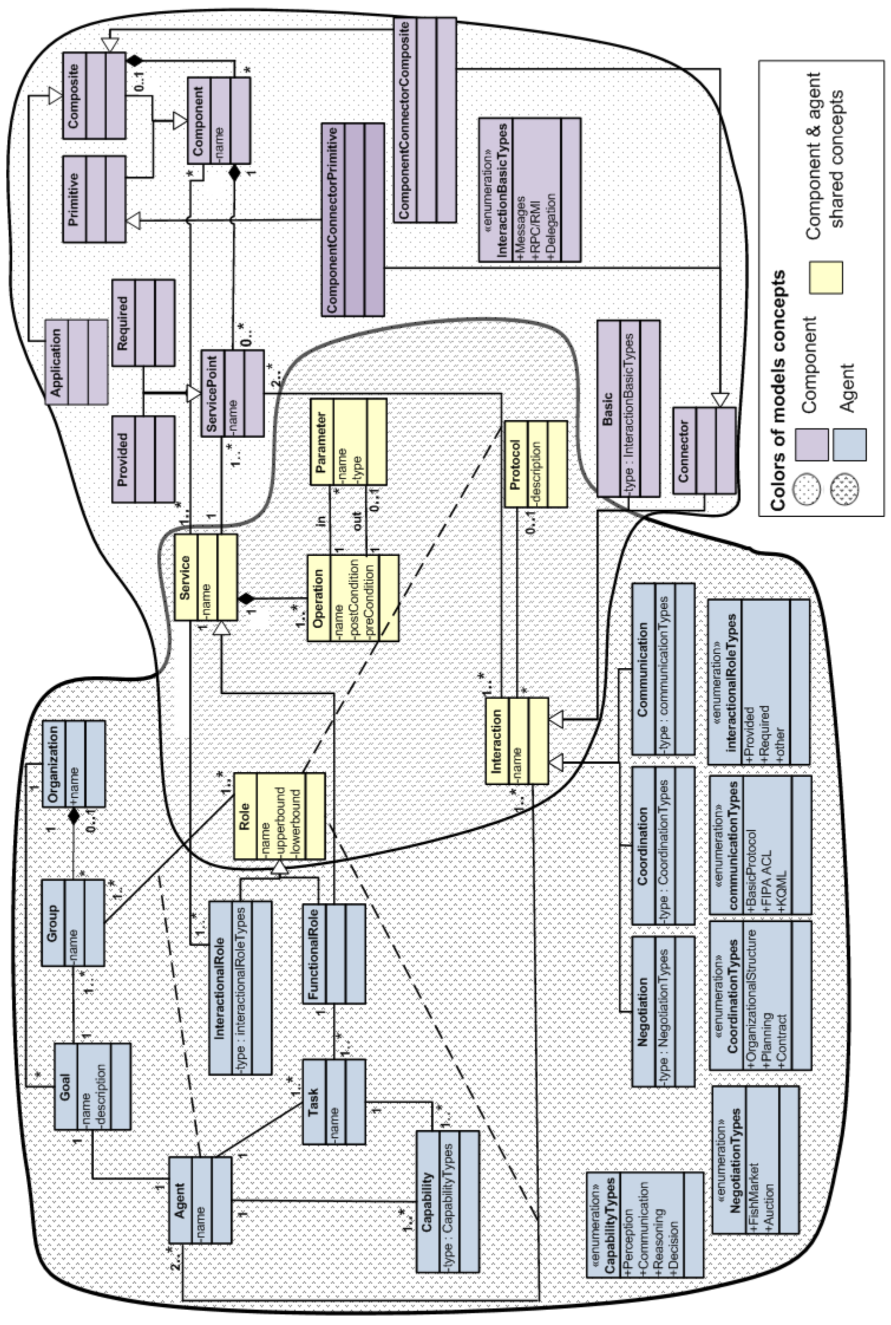


Figure 8.1: Unified view of the component and agent models.

On the component side, the element representing the application as a whole is the *Application* concept and on the agent side, it is the *Organization* one. These two concepts are equivalent and we decide in CASOM to use the *Organization* concept coming from the agent approach. This choice results from the fact that an organization is a first-class entity of OMAS whereas in components, the application as itself is defined usually only implicitly.

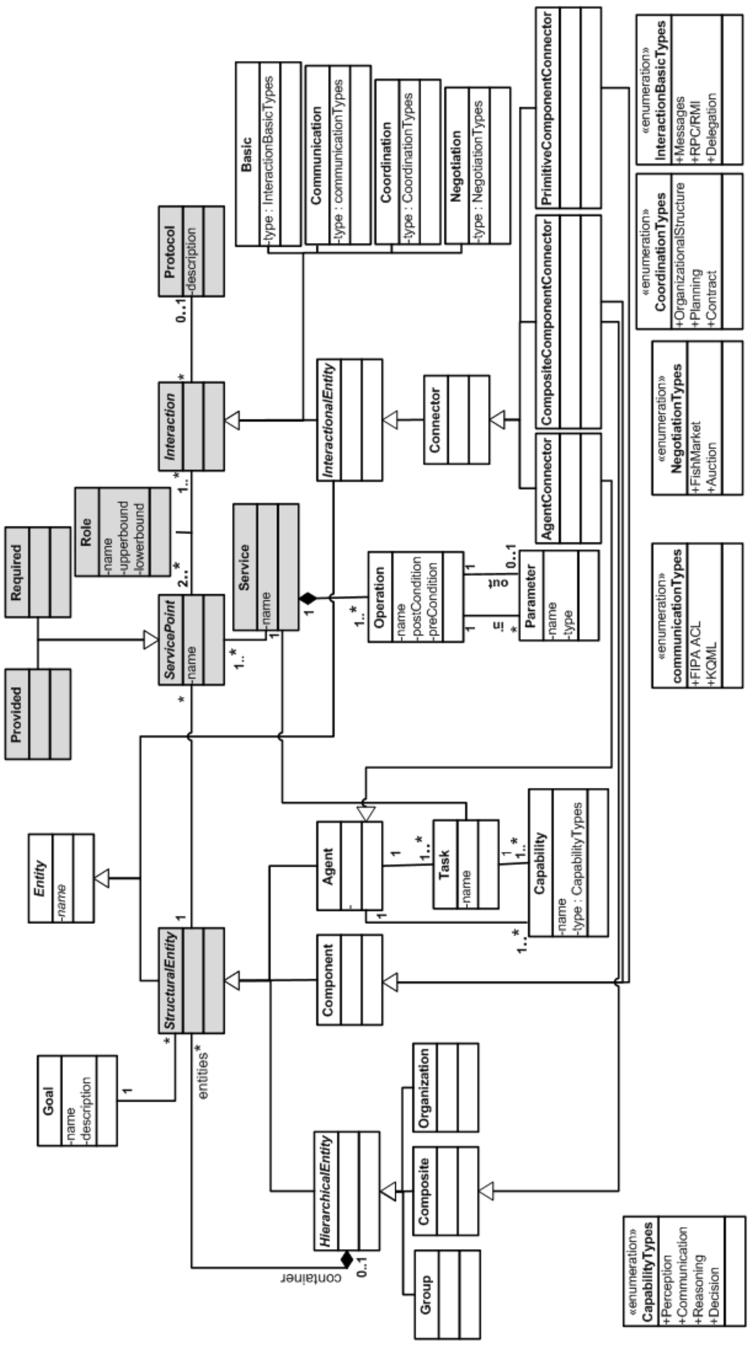


Figure 8.2: Component Agent Service Oriented Model.

8.2.2 Abstraction and generalization of concepts

8.2.2.1 Structural and Hierarchical Entity

The goal of defining abstract concepts that have different realizations is to be able to use indifferently any of these concrete realizations in an application specification. This is what we want to do with agents and components. Then, we define the abstract concept of *StructuralEntity* that is specialized in *Component* and *Agent*. To limit the number of elements in CASOM, the *Component* concept represents now a primitive component.

With the same logic, there exist two hierarchical structures: the composite on the component side and the group on the agent side. Even if they are not exactly similar and at the same level because a group is simply a logical combination of agents whereas the composite is a component, that is a first-class structural entity requiring and offering services, we decide to unify them under the concept of *HierarchicalEntity*. It is a specialization of *StructuralEntity* and contains a collection of *StructuralEntity* elements. These general specializations and associations offer the widest combinations of agents and components: a component or an agent can belong to a group and a composite can contain components (primitive or composite), groups and agents. The *Organization* concept is made specializing the *HierarchicalEntity* one as an organization/application contains components, agents or groups. However, not all combinations are allowed. A group or an organization remains a logical structure and does not offer or require services, that is, does not own service points. An organization is the only hierarchical entity that does not have a container as it is the root of the hierarchy. An agent cannot be directly contained in an organization (it must be contained in a group or in a composite). These constraints are expressed as OCL invariants as shown in section 8.3.

8.2.2.2 Interaction

Concerning the interactions, we also want to make the interaction elements of a domain available to the other one. The interaction kinds of agents (*Communication*, *Coordination* and *Negotiation*) are specializations of the general *Interaction* concept. These kinds of interaction can then be used indifferently between components or agents. In the same way, the *Basic* and *Connector* (*ComponentConnector* and *AgentConnector* specialization) elements inherit from the *Interaction* and can also be used indifferently between components or agents. Then, any element (agent or component) can interact with other elements through any of the agent or component based interactions. The agent interaction classification is directly available to components since it is defined as specialization of the *Interaction* element. Then, the agent interaction classification can be used directly between services points of components. In the same way, an agent can use a structured and reusable interaction through a connector or a component connector since they are also specializations of the *Interaction* element.

8.2.2.3 Interactional Entity

Concerning high-level interactions, component and agent approaches mainly take two different ways to manage them. On the component side, a high-level interaction is defined by a protocol implemented by a connector. A connector is a concept coming from the ADLs (*Architecture Description Languages*) [MT00]. It is an architectural first-class entity structuring and embedding the interactions between components. Some works go even further, such as the communication components of [CBJ02], in reifying a protocol in a component, that is a more structured element with well-specified interfaces (service points and services in our approach) allowing to easily reuse an interaction. Such components dedicated to an interaction are represented by our *ComponentConnector* element. On the agent side, there exists a classification of interactions: negotiation, communication and coordination with several variants. An agent can then reuse easily an existing interaction of this classification or build a new one based on them and defined with a protocol.

However, with the same idea as for the communication components [CBJ02], some works on protocol engineering emphasis the idea of making a clear separation between the functional part of an agent and the management of interaction protocols. In [HSB04], a specific agent, called moderator, is dedicated to enforce the protocol rules during a conversation between functional agents. According to the authors, this provides solution to problems in MASs engineering such as the reuse and adaptive maintenance of protocols and the separation of the aspects in the design and implementation of agents. Actually, they have the same advantages advocate by [CBJ02] for the communication component in the component field. In order to support such design approach for agents, and also to generalize some common features, we decide to define the concept of “agent connector” which is an agent dedicated to manage an interaction between others elements.

One can now ask a question: the component connector and agent connector concepts appear rarely in their respective fields; few works are interested in them. So, why having a component connector directly in the component model and not an agent connector in the agent one? As we saw previously in section 7.3.1, there already exists a first-class entity reifying an interaction in many component models: the connector. This is why the connector is presented in our component model. Then, the gap for adding the component connector concept is tight: we just need to extend this existing connector entity, even if it is not a widely shared concept between all existing component models. However, on the agent side, in almost all agent models, there does not exist such an entity reifying an interaction. It was then not suitable to add an agent connector as it will introduce a new first-class entity that is not widely defined in agent models. But since such an entity reifying an interaction exists in a connector, then, it can be added in CASOM, for the same reason that we have introduced the component connector in the component model.

All these entities integrating and realizing a protocol are specified through a hierarchy of elements in CASOM. At the top, there is the abstract *InteractionalEntity* specializing the *Interaction* element. It is specialized by the *Connector* element, itself being specialized by the *ComponentConnector* element (inheriting also from *Component*) and the *AgentConnector* element (inheriting also from *Agent*).

CASOM contains also a class named *Entity*, which represents the abstraction of both *StructuralEntity* and *InteractionalEntity*.

8.2.3 Extension of concepts to the other domain

8.2.3.1 The goal concept

For an agent, the designer must specify its goal, that is, he must precise what an agent is doing or which purposes need to be achieved from it. A group of agents has also a goal. This *Goal* concept is a first-class feature of agent approaches. Concerning a component, it has also a goal: it has been designed for a given purpose and is realizing actions dedicated to this purpose. However, in most of component approaches, this goal remains implicit within its specification. But as a component has a goal also in principle, like an agent or a group, we decide in CASOM to attach this *Goal* concept to the *Entity* element. Now, a component has an explicit goal.

8.2.4 Secondary or specific elements

The *Task* and *Capability* concepts of agents have no equivalent concepts on the component side but they are full part of the specification of an agent. They are then presented in CASOM and associated with the *Agent* element. These concepts were the last ones that have not been analyzed in the above sub-sections.

Therefore, all the concepts and elements of the component and agent models are presented in CASOM, none has not been taken into account. Table 8.1 groups the definitions of CASOM concepts.

Concept	**Definition**
Hierarchical Entity	It is an abstract concept that contains components, agents or groups with many possible combinations.
Organization	The overall architecture of a system, it contains directly primitive components, composite components and groups of interacted agents and components.
Group	A logical combination of agents or components.
Composite	A hierarchical entity contains components, composites, groups and agents.
Structural Entity	It is an abstract concepts that represents a container of components, agents and hierarchical entities other than organizations.
Goal	A functional requirement of a structural entity that could be divided in to sub goals related to agents, components, composites or groups.
Component &Agent	A first-class structural entity that requires and offers services.

Concept	**Definition**
Entity	An abstract concept specified by structural and interactional entities.
Interaction	It is the dynamic relation between structural entities through their service points else organizations and groups. It has different specification: *Basic*, *Communication*, *Coordination*, *Negotiation* and *InteractionalEntity*.
Interactional Entity	It is an entity that specializes an interaction. It is specialized by the Connector element.
Connector	The explicit representation of a complex interaction, its behaviour is specified by a protocol
Component Connector	A component dedicated for communications. It defines structured and reusable interaction.
Agent Connector	An agent dedicated to manage interactions between interacted elements, like components and agents.
Service point	It is a port of structural entities except organizations and groups. It exposes some of its services (provided specialization) or uses other services (required specialization).
Service	It is a static unit of functions (a classical interface definition).
Role	The responsibilities of structural entities (except organizations and groups) through their service points within an interaction.

Table 8.1: Definitions of the concepts of CASOM

The concepts of *Protocol*, *Operation*, *Parameter*, *Negotiation*, *Coordination*, Communication, Task and Capability have the same significations as presented in tables 7.5. Since they have been already presented in the main concepts tables of service, component and agent models, then we did not rewrite them in table 8.1 to avoid disagreeable repetitions.

The base of CASOM is to define a set of general elements that will be specialized by component and agent concepts (these general elements are represented with a grey background colour on figure 8.2). A structural entity is providing or requiring services through service points. Structural entities interact between each other by playing roles in interactions associated with their service points. Then, the two general concepts of structural entity and interaction are specialized by all the specific elements of components and agents, making them available and mixable in most opened-ways in an application specification. A structural entity can be a component, a composite, an agent or a group. An interaction can be a basic component interaction, a high-level agent interaction or an interactional entity, that is, a connector in its agent and component specializations.

8.3 OCL constraints

Similarly to the proposed OCL constraints for the previous models in chapter 7, we remove the possible ambiguity in CASOM defining the following constraints: An organization is the only hierarchical entity that does not have a container as it is the root of the hierarchy:

```
context StructuralEntity inv noContainerInOrganization:
if self.oclIsTypeOf(Organization)
then self.container.oclIsUndefined()
else not self.container.oclIsUndefined()
endif
```

A group or an organization remains a logical structure and does not offer or require services, that is, does not own service points. Other structural entities own at least one service point:

```
context StructuralEntity inv noServicesWithGroupsAndOrganizations:
if self.oclIsTypeOf(Organization) or self.oclIsTypeOf(Group)
then self.servicePoint -> isEmpty()
else self.servicePoint -> notEmpty()
endif
```

An agent cannot be directly contained in an organization (it must be contained in a group or in a composite):

```
context Agent inv notInOrganization:
not self.container.oclIsTypeOf(Organization)
```

Basic interactions only connect two service points:

```
context Basic inv twoBasicServicePoints:
self.servicePoint -> size() = 2
```

For simplifying the definitions of the constraints dealing with the basic interactions, we consider that the collection of service points for an interaction is ordered. Here are the OCL helpers allowing to get each service point in the context of basic interactions:

```
context Basic def: firstServicePoint : ServicePoint =
self.servicePoint -> first()

context Basic def: secondServicePoint : ServicePoint =
self.servicePoint -> last()
```

RPC/RMI interactions connect a provided service point with a required one. The first service point is the required one and the second is the provided one:

```
context Basic inv rpcrmiSP:
self.type=#RPC/RMI implies
   self.firstServicePoint.oclIsTypeOf(Required) and
   self.secondServicePoint.oclIsTypeOf(Provided)
```

Delegation interactions connect the same type of service points:

```
context Basic inv delegationSP:
self.type=#delegation implies
   self.servicePoint -> forAll ( sp | sp.isTypeOf(Required)) or
   self.servicePoint -> forAll ( sp | sp.isTypeOf(Provided))
```

A RPC/RMI or a message interaction deals with an horizontal assembly, then connects two entitiers embedded directly in the same composite (or group or application):

```
context Basic inv rpcrmiConnection:
self.type=#RPC/RMI or self.type=#message implies
self.firstServicePoint.structuralEntity.container = self.secondServicePoint.structuralEntity.container
```

A delegation deals with vertical assembly. The first service point is one of a hierarchical entity that can only be a composite and the second service point is one of the internal entities:

```
context Basic inv delegationConnection:
self.type=#delegation implies
   self.firstServicePoint.structuralEntity.oclIsTypeOf(Composite) and
   self.secondServicePoint.structuralEntity.container                    =
self.firstServicePoint.structuralEntity
```

The basic interaction is the only kind of interaction that is not associated with a protocol:

```
context Interaction inv noProtocolForBasic:
if self.oclIsTypeOf(Basic)
then self.protocol.oclIsUndefined()
else not self.protocol.oclIsUndefined()
endif
```

Each kind of interaction must at least connect one required service point and one provided service point:

```
context Interaction requiredAndProvidedSP:
self.servicePoint -> exists ( sp | sp.oclIsTypeOf(Required)) and
self.servicePoint -> exists ( sp | sp.oclIsTypeOf(Provided))
```

Within a hierarchical entity, the internal entities have different names:

```
context HierarchicalEntity inv uniqueNames:
self.entities -> forAll ( e1, e2 | e1 <> e2 implies e1.name <> e2.name)
```

8.4 Mixed of agents and components application example

As we saw previously, our goal is to combine agents and components within the same application specification. We also want to make characteristics and advantages of one approach available to the elements of the other one. As an example, figure 8.3 shows such a specification for our holiday reservation system. Concretely, this specification is based on our CASOM model. This application example corresponds to the already proposed possible structure in figure 6.7 on page 113. The elements that are mainly based on agent concepts are the *Client*, which is an agent and the *TravelAgency* element defined by a group. However, this group does not only contain agents as the *ReserveVacation* element is a primitive component. For the component side, the *AirlineCompany* element is a primitive component and the *HotelChain* is a composite one. With the same logic as for the *TravelAgency* group, this composite (*HotelChain*) does not only contain components, its *Management* element is an agent. We could have of course combined the agent and component elements in other ways.

Our choice is based on the idea that agents are more relevant to define elements with autonomous behaviours such as taking decision or negotiating. This is why the client and the travel agency are mainly based on agents as the client must take a decision concerning its reservation and that the travel agency manages all the offers from the hotels and airline companies.

Independently of the reason of choosing between a component or an agent for a given element of the application, one can notice that we can indifferently use an agent or a component. This is made possible because an interaction between elements is only based on their services (via their services points) and that the same service can be concretely implemented or required by an agent or by a component. For instance, for the three versions of our application, the client element is offering the *VacationReservation* service. This service is the same in the three specifications (it contains the same set of operations). In the same way, the agent and component versions of the client element require the same *VacationOffer* service. The concrete implementation of the *VacationReservation* service can be made with a component (as in figure 7.5) or with an agent (as in figures 7.7 and 8.3).

Concerning our mixed specification, if we decide to transform this agent onto its component variant, this will have absolutely no impact on the rest of the specification because this component will still implement the same service and require the same service. In other words, we can easily substitute an agent with a component and conversely.

Regarding interactions, those that manage, through the travel agency, the offers from the airline company and from the hotels for the client are agent-based negotiations. Indeed, these elements interact to make a choice, which may require negotiation. For the payment of a reservation, we use basic interactions (component-based basic function calls or agent-based communication) because no complex interaction is required. Finally, there are connectors used for dispatching the requests between the *Management* element and the *InternalHotel* elements.

Some concepts like task, role and capability do not appear in the figure to keep it

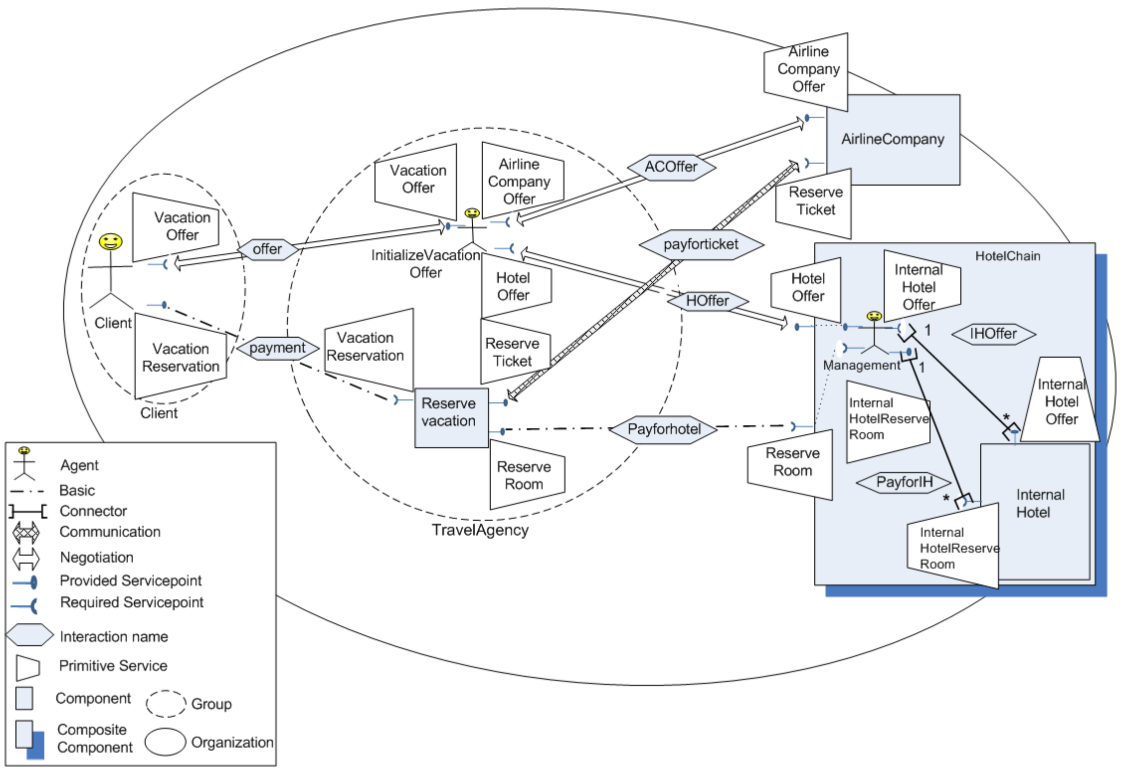


Figure 8.3: Holiday reservation application specified through the CASOM model, mixing agents and components.

legible. Many refinement can be added to this example, for instance we can replace the *Management* agent by a primitive component which interacts with the internal hotel components.

8.5 Summary

CASOM specifies an application (organization) by interacted structural entities with defined goals. These interactions are done through the required or provided service points of structural entities to exchange services within an organization. The structural entities can be specified by components, agents or hierarchical entities of composite and group that combine agents and components. The interaction can be a basic component interaction, a high-level agent interaction or an interactional entity, that is, a connector by its agent and component specializations.

To complete the class diagram defining CASOM (figure 8.2), we added OCL invariants specifying the constraints between the elements that cannot be expressed directly in the class diagram in section 8.3.

To end this chapter, we need to know if this model does really meet our objectives, then we ask the following questions:

- Does component approach makes use of agent characteristic?

  Yes it does, as the *Entity* concept has one or more goals (through the derived relation). Then a component owns an explicit goal to be achieved. This represents the gain of goal-directed behaviours feature of agent approaches. A component can also use the advanced types of interaction related to the agent approach.

- Does the agent approach gain some characteristics of the component approach? Yes it does, agents become more reusable, since an agent owns provided or required service points to reuse services, which is a key property of component and service approaches. An agent can be integrated in a composite structure, where in CASOM a composite component can contain agents directly. This represents the gain of the composition feature of component-based approaches.

Many concepts related to the interests of agent approach are available to enhance component-based approach via CASOM, and the reverse is true for the component.

We already defined in this chapter the CASOM model, which allows mixing the concepts of component, agent domains through interoperable services in one application specification. We need to define its relations with the previously presented models and how to move on smoothly from one model to another in the next chapter.

Chapter 9

Semantic Mappings

Contents

9.1 Introduction

In this chapter, we define the main relations between the previously defined models of service, component, agent and CASOM. These relations are actually the arrows in our framework in figures 6 on page 105, 6.1 on page 106. They are the transformations in MDE words to move on from one domain to another. The *Projection* from service model towards component, agent and CASOM models are vertical transformations with the reverse transformation named the *Abstraction*. The horizontal transformations are the *Agentification* and *Componentification*. The *Agentification* is the transformation from the component model to agent one and the reverse transformation is the *Componentification*. They can be direct from the component to the agent model or indirect by adding components to existing agents to move from the agent model to CASOM one via the *Componentification* transformation (reversely by adding agents to existing components to move from component to CASOM model via the *Agentification* transformation).

We present here the mappings between the concepts of the four previously proposed models of service, component, agent and CASOM. Specifically, these mappings allow translating a specification of one model to another. Most of the proposed mapping rules in this section are already presented in [ACGA12, ACG12]

The defined rules of mappings can be applied to the three examples of holiday reservation application specifications via the service / components / agents models (figures (7.2, 7.5, 7.7 and 8.3)) as each of them can be transformed into another.

Many common concepts exist between the three models of service, component, agent and CASOM. These concepts are presented with yellow backgrounds in figure 7.8. Operation, parameter and protocol are from these shared concepts, which are easy to translate from one domain to another since they are just the same. Then, we do not detail the mappings related to these concepts, as they are just simple and direct ones. Although, the concepts of service and interaction belong also to the shared concept but their relations with other concepts are not the same, then we detail the mappings rules related to these concepts and to the other unshared ones in this chapter.

9.2 Projection: service and component models

As we saw previously in chapter 7, the service and component models are very close. We can consider the component model as a direct projection of service one by adding the elements those implement the services, which are components. As a consequence, there are direct and bidirectional mapping rules between the two models: a primitive service is mapped to a primitive component and a composite service is mapped to a composite component and an interaction is mapped to a similar interaction (RPC|RMI, delegation and Protocol), etc.

Table 9.1 details the bidirectional mappings between the concepts of the service model and those of the component. As presented in the table, mappings are written in the direction from service to component concepts. However, these mappings can directly be applied in the other direction, from component to service concepts. Then, they are in effect bidirectional mappings[37]. There is one major point to take into account concerning service and service point concepts between the two models. The particularity of our service model is in not defining explicitly the elements supporting the implementation of the services (components or agents in other models). In the service model, a service contains a set of operations, each set is associated with a provided or required service point. Here, the service acts as a logical link between these sets of operations. On the component side, this notion of global service disappears since the component becomes its concrete implementation. Then, each set of operations of a service associated with a service point in the service model is mapped onto the same set of operations associated with the service point of the equivalent component. On the component side, we name "service" this set of operations associated with a service point. A component is then associated with multiple services – each service is a set of operations – which correspond to the set of operations of the service points on the service model side. Therefore, the concept of service, even if consistently associated with operations, is not exactly the same

[37]Along this chapter all the mappings between service and agent concepts or between component and agent concepts are also to be understood as bidirectional mappings except if explicitly stated.

Service side	**Component side**
Primitive service	**Primitive component** with the same name
Composite service (resp. **application**) with internal services	**Composite component** (resp. **application**) with the same name and internal components equivalent to internal services
Provided (resp. required) **service point** of a service and its set of operations with a playing **role**	**Service** of a component with the same name and set of operations
	Provided (resp. required) **service point** of a component with the same name and playing the same **role**
Basic interaction of certain type (resp. related to a **protocol**) and its **roles** of *service point* through this interaction	**Basic interaction** of the same type (resp. related also to a **protocol** and a **connector**) with the same **roles** of the corresponding service point of the equivalents components

Table 9.1: Service / component mappings

in both models. Figure 9.1 illustrates an example of the mapping between a primitive service associated with two service points (and then the set of operations) in part (a) and a primitive component with the two equivalent services (containing the same set of operations) in part (b).

The connector is a particular kind of element as it exists only on the component side. Actually, the connector plays the same kind of role as the component as being the concrete support for realizing an abstract concept: the component realizes a service and the connector realizes a protocol.

Regarding the connector component, it has particular mapping rules. In the direction from the service model to component one, there is no mapping between elements of the service model and the component connector concept (except the mapping variant that we propose lately in the design guide). In the other direction, from components towards services model, the component connector concept is viewed as a regular component by applying the rules of table 9.1 for primitive or composite component according to the structure of the component connector. The only difference is that its associated protocol is then ignored.

9.3 Projection: service and agent models

Similarly to a component, an agent is an element realizing the implementation of a service. One can see that the service and agent models are relatively distant with many

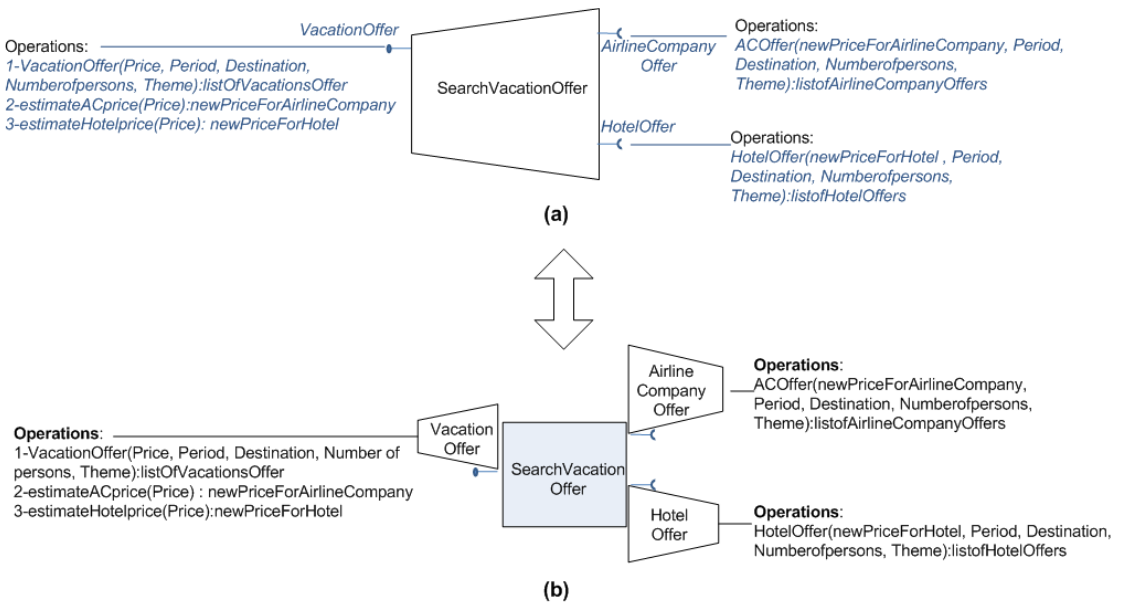

VacationOffer
Operations:
1-VacationOffer(Price, Period, Destination, Numberofpersons, Theme):listOfVacationsOffer
2-estimateACprice(Price):newPriceForAirlineCompany
3-estimateHotelprice(Price): newPriceForHotel
SearchVacationOffer
AirlineCompany Offer
Operations:
ACOffer(newPriceForAirlineCompany, Period, Destination, Numberofpersons, Theme):listofAirlineCompanyOffers
HotelOffer
Operations:
HotelOffer(newPriceForHotel , Period, Destination, Numberofpersons, Theme):listofHotelOffers
(a)
Operations:
1-VacationOffer(Price, Period, Destination, Number of persons, Theme):listOfVacationsOffer
2-estimateACprice(Price) : newPriceForAirlineCompany
3-estimateHotelprice(Price):newPriceForHotel
Vacation Offer
SearchVacation Offer
Airline Company Offer
Operations:
ACOffer(newPriceForAirlineCompany, Period, Destination, Numberofpersons, Theme):listofAirlineCompanyOffers
Hotel Offer
Operations:
HotelOffer(newPriceForHotel, Period, Destination, Numberofpersons, Theme):listofHotelOffers
(b)

Figure 9.1: A primitive service with several service points (a) to a primitive component with several service points mapping (b).

specific concepts for each domain. However, there are mappings between the main concepts of each domain.

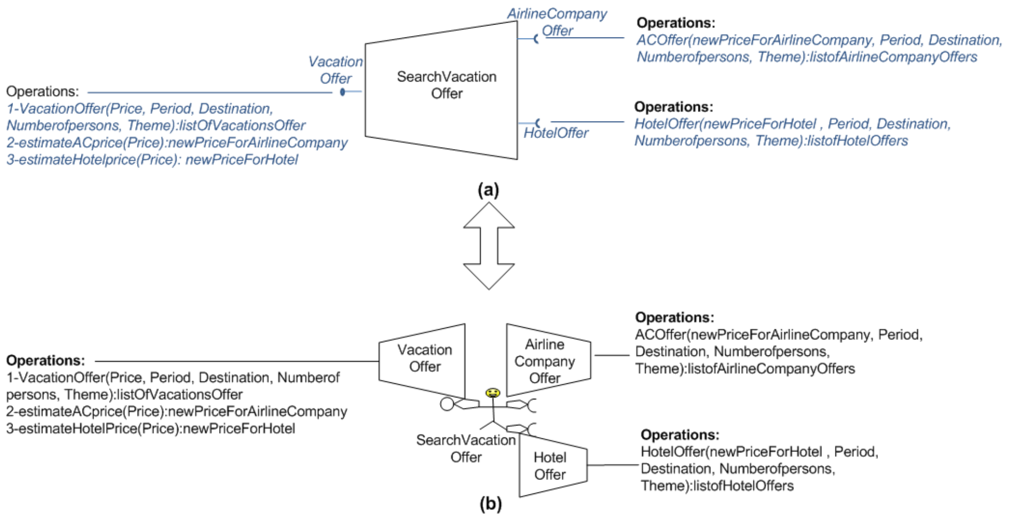

AirlineCompany Offer
Operations:
ACOffer(newPriceForAirlineCompany, Period, Destination, Numberofpersons, Theme):listofAirlineCompanyOffers
Vacation Offer
SearchVacation Offer
Operations:
1-VacationOffer(Price, Period, Destination, Numberofpersons, Theme):listOfVacationsOffer
2-estimateACprice(Price):newPriceForAirlineCompany
3-estimateHotelprice(Price): newPriceForHotel
HotelOffer
Operations:
HotelOffer(newPriceForHotel , Period, Destination, Numberofpersons, Theme):listofHotelOffers
(a)
Operations:
1-VacationOffer(Price, Period, Destination, Numberof persons, Theme):listOfVacationsOffer
2-estimateACprice(Price):newPriceForAirlineCompany
3-estimateHotelPrice(Price):newPriceForHotel
Vacation Offer
Airline Company Offer
Operations:
ACOffer(newPriceForAirlineCompany, Period, Destination, Numberofpersons, Theme):listofAirlineCompanyOffers
SearchVacation Offer
Hotel Offer
Operations:
HotelOffer(newPriceForHotel , Period, Destination, Numberofpersons, Theme):listofHotelOffers
(b)

Figure 9.2: A primitive service with several service points (a) to an agent with associated functional and interactional roles mapping(b).

The service concept is associated with operations in both models. But this association cannot be viewed in a similar way, because of the absence of the service point concept in the agent model. Actually, in an organizational multi-agent approach, all interactions between agents are done through roles: the interactional roles allow an agent to provide or to require services and these services are defined through functional roles. Then a functional role on the agent side is mapped onto a service point with its associated operations on the service side.

Service side	**Agent side**
Primitive service	**Agent** of the same name and associated with a **goal** by default
Composite service with internal primitive services	**Group** of the same name containing corresponding agents to the internal services and with a **goal** by default (figure 9.3)
Application	**Organization** of the same name and a **goal** by default
Provided (resp. required) **service point** of a service and its set of operations with a playing **role**	**Functional role** with the same name and set of operations
	Interactional role with the name of the role associated with the service point (on the service side) and the type of the service point (provided or required), see figure 9.2
Interaction with **protocol** between service points	Equivalent high-level **interaction** like **Communication**, **Coordination** or **Negotiation** associated with an equivalent **protocol** between the equivalent interactional roles associated with functional ones that correspond to the service points (figure 9.4)
Basic interaction of certain type between service points	**Communication** of basic protocol type between the **interactional roles** associated with functional ones corresponding to the service points

Table 9.2: Service / agent mappings

Table 9.2 presents the bidirectional mappings for the major concepts of the service and agent models. Regarding “goal by default” for each agent or group when moving on from service to agent side, it means that the added goal has a description field that is not specified. The designer has to set it afterwards. In the other direction, from agents to services, the goal associated with an agent or a group is ignored.

Since service model belong to a different domain with different purposes than the ones of agent model (service oriented approaches belong to software engineering domain and MultiAgent systems belong to artificial intelligence domain); there is a large gap between their concepts in their numbers and semantics. Here we present some specific issues when applying transformation from the service model to agent one and via versa.

- We do not find an equivalent concept for composite on the agent side. This concept must be taken into account at the level of the organizational structure. It raises then different problems at the structure level itself but also for delegation between service points. The organizational structure of the agent model offers only two levels: the group and the agent. This causes problems for hierarchical composition of more than two levels (for example, a composite service containing composite services[38]). One way to overcome this problem is to use shared agents between different groups to represent the structural levels of an organization. This approach is used in AGR model for instance [FGM04]. If an agent member of a group of level n is also member of the group of level $n+1$, then it can be considered as the representing element of the group of the lower level and all the interactions between the two levels must be managed by this agent.

- The second problem deals with the delegation of external service points to internal ones for a composite service: such a delegation principle does not exist on agent side. When we transform a composite service to a group of agents, we consider only the service points of the internal services to create the functional roles of the agents (the same set of operations is associated with an internal service point and with its delegated external service point on the composite service). For the other direction, from agents to services, we must create external service points on the composite and the required delegations between internal and external service points with associated roles.

- The absence of the composite concept on the agent model side makes the transformation of an application (on the service model) more complex. Intuitively, one can consider a mapping between the concept of application of the service model and the concept of organization of the agent one. As an organization is always composed of groups, there is no problem to move on from the agent model to the service one (an organization composed of groups becomes an application consisting of composite services). However, an application on the service side is not always composed of only composite services; it can also directly contain primitive services, which cause a problem. In this case, when moving on from services to agents, each primitive service of the application is mapped onto a single agent that is contained in a dedicated group. This group has the same name of the agent it contains.

- When a primitive service interacts with other services via a protocol (e.g., broadcast, facilitator, mediator, etc.), the mapped agent must own the required capabilities in terms of communication (FIPA ACL language and basic protocols), possibly reasoning and decision making (coordination, negotiation) in order to play the associated interactional roles through this protocol. For example, the service

[38]For this reason, the table 9.2 presents the mapping of a composite service containing only primitive ones. In this case, the mapping is direct and simple: it is a group composed of agents.

Management in a *HotelChain* offers a list of available hotels after broadcasting the request for different hotels and collecting their offers. On the agent side, such broadcasting and collecting actions require communication and perception capabilities of *Management* agent (an example of these capabilities are represented by ovals in figure 9.3 but they are not available in all figures to avoid their overloading). Unfortunately, on service side, there are no equivalent concepts for the capability concept, neither for task or goal ones. Then, when moving on from the agent model to the service one, all these elements will be "removed", while in the other direction, the designer must add or modify these elements manually on the default specification obtained automatically

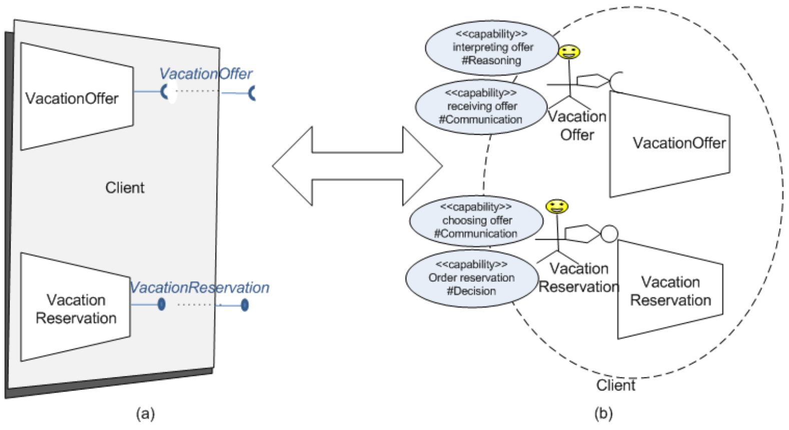

VacationOffer
VacationOffer
Client
Vacation Reservation
VacationReservation
(a)
<<capability>> interpreting offer #Reasoning
<<capability>> receiving offer #Communication
Vacation Offer
VacationOffer
<<capability>> choosing offer #Communication
<<capability>> Order reservation #Decision
Vacation Reservation
Vacation Reservation
Client
(b)

Figure 9.3: A composite service (a), a group of agents (b).

Regarding the non-basic interactions, there is a single concept of protocol on the service side while there is a richer classification on the agent side based on three main types of interactions (communication, coordination and negotiation). Then, we do not have the same level of details on each side. When we move on from agent to service side, a non-basic interaction is systematically associated with a protocol, but in the other direction, an automatic default choice must be made or the designer must choose between one of the three types of interactions (see figure 9.4).

9.4 Projection: service and CASOM models

We need to clarify here that most of the mappings related to CASOM model do not propose systemically rules of transformations. However, we propose at least two possibilities of transformations and the designer must take a decision to precise his choices in the application specification. In CASOM, we find ourselves in front of two principle choices to implement system services by components or agents with their originally related concepts and other acquired ones. Components own high-levels of interactions with well-defined goals. Agents can be a part of a composite structure with reusable services via their service points but an agent cannot be composed itself of other agents.

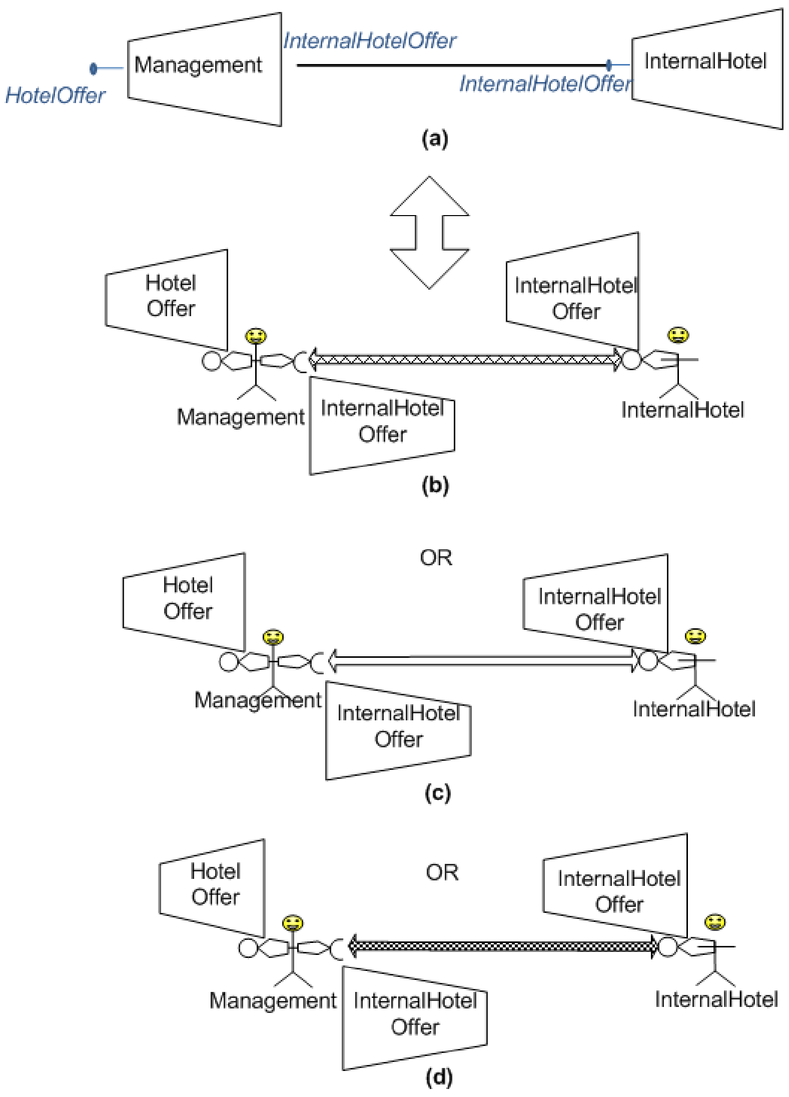


Figure 9.4: Services and interaction protocols (a), agents interacting by communication (b), coordination (c) and negotiation (d).

We try with CASOM to enhance the applications based on only one of these two domains (components or agents) by adding complementary concepts of the other added-value domain.

This projection groups the two already presented rules of mappings (from the service model to component and agent ones). It contains some changes related to the names of the concepts in CASOM (e.g., the functional and interactional roles of agents become services, service points and roles in CASOM) as presented in chapter 8. The service concept in the service model can be implemented by an agent or a component according to the choice of the designer using the coming proposed hints in the design guide chapter.

For example, when a service interacts with several services by a protocol, it will be implemented by an agent, however when a service responds to a single interaction like RMI, implementing it by a component is sufficient. Table 9.3 provides the possible bidirectional mappings between the concepts of service and CASOM models.

Service side	*CASOM side*
Primitive service	**Primitive component** or **Agent** with a **goal** by default
Composite service with internal services	**Composite component** or **Group** containing a component or an agent for each internal service[⋆] with a **goal** by default
Application	**Organization** with a **goal** by default
Provided (resp. required) **service point** of a service and its set of operations and playing a **role**	**Service** of a component or an agent with the same set of operations
	Provided (resp. required) **service point** of a component or an agent with the same name and playing the same **role**
Interaction associated with a **protocol** between service points	**Connector** in its two specialization **ComponentConnector** and **AgentConnector** or one of **Communication**, **Coordination** or **Negotiation** specialization of **Interaction** associated with an equivalent **protocol** between the equivalent service points
Basic interaction of a given type between service points	**Basic interaction** or **Communication** of basic protocol type[†] between the equivalent service points

[⋆] Only if these internal services are primitive. See section 9.3 for the explanations.
[†] Except for the delegation of the service. See section 9.3 for the explanations.

Table 9.3: Service / CASOM mappings

Similarly to the transformation between agent and service models in section 9.3, the designer also should interfere to define the needed capabilities and tasks when an agent is chosen to implement a service. However, in the reverse direction these concepts will be ignored. After presenting the vertical transformations between the service model and the three other models, we browse in the following sections the horizontal ones.

9.5 Agentification / Componentification: direct mappings between component and agent models

We saw previously, that the component model is very close to the service one. We find here some similarities in the mapping rules between component and agent models except the ones between service and component ones; the agents like the components are elements that implement services. Whereas, there will be the same limitation regarding the domain specific concepts, especially the notion of composition on the component side and the notion of goal on the agent one. In the agent model, the service concept is perceived in the same way as in the component model except that it is considered as an abstraction of a functional role. An agent (resp. primitive component) can be associated with several functional roles (resp. services); each functional role (resp. service) being composed of operations. An interactional role (resp. service point) is associated with a functional role (resp. service). Agents (resp. primitive components) interact with each other through interactions associated with their interactional roles (resp. service points). In the previous example of figure 9.1, the component part (a) corresponds to the agent one in figure 9.2 (b), which leads to figure 9.5. We must note that the notion of interactional role on the agent side groups the two notions of service point and role on the component side. This requires making the choice of a convention to name the interactional role to save information. We can use a compound name as *<name of role>.< name of service point>*.

Table 9.4 presents the bidirectional mappings between main concepts of component and agent models.

We find same problems related to the transformation of a composite component as those encountered for the transformation of a composite service (section 9.3), these problems are further detailed in section 10.4 which considers special cases of mappings. As a solution, a similar approach as for the treatment of hierarchies of composition, of applications and of the delegation mechanism can be taken.

With the same logic as for the mapping between services and components, a component connector is managed as follow. From agent to component model, except the proposed variant in the design guide (next chapter), there is no mapping leading to the definition of a component connector on the component side. In the direction from component to agent, a component connector is managed as an ordinary component ignoring its protocol. However, a designer must add manually to the mapped agent(s) the required capabilities to implement this component connector.

Finally, there are no equivalent concepts for the capability, task or goal on the component model side. Then, when moving on from the agent model to the component one, all these elements will be “removed”, while in the other direction, the designer must add or modify these elements manually on the default specification obtained automatically.

Component side	**Agent side**
Primitive component	**Agent** of the same name with a **goal** by default
Composite component with internal primitive components	**Group** of the same name containing an agent for each internal component with a **goal** by default
Application	**Organization** of the same name and with a **goal** by default
Service of a component with set of operations	**Functional role** of the same name and with the same set of operations assigned to the corresponding agent of the component associated with the service
Required (resp. provided) **service point** of a component with an associated **role**	**Interactional role** having the name composed of the service point name and its associated role one. It takes the type of the service point (provided or required). This role is assigned to the corresponding agent of the component
Interaction associated with a **protocol** between service points	Equivalent high-level **interaction** like **Communication**, **Coordination** or **Negotiation** associated with an equivalent **protocol** between the equivalent interactional roles associated with functional ones that correspond to the services
Basic interaction of a certain type	Basic protocol type of **communication**
Connector associated with a **protocol**	**Interaction** associated with an equivalent **protocol**

Table 9.4: Component / agent mappings

9.6 Agentification / Componentification: mappings between component, agent models and CASOM

As we presented previously in section 9.4, all CASOM related mappings need the intervention of a designer; we propose at least two possibilities of mapping and a designer need to precise his choices. In this section, no new mapping rules will be defined because we can use the already presented mappings between component and agent models in the previous section except two obligations. These two obligations differ according to the source model. From component model to CASOM, an application concept becomes an organization. From agent model to CASOM a functional role becomes a service and

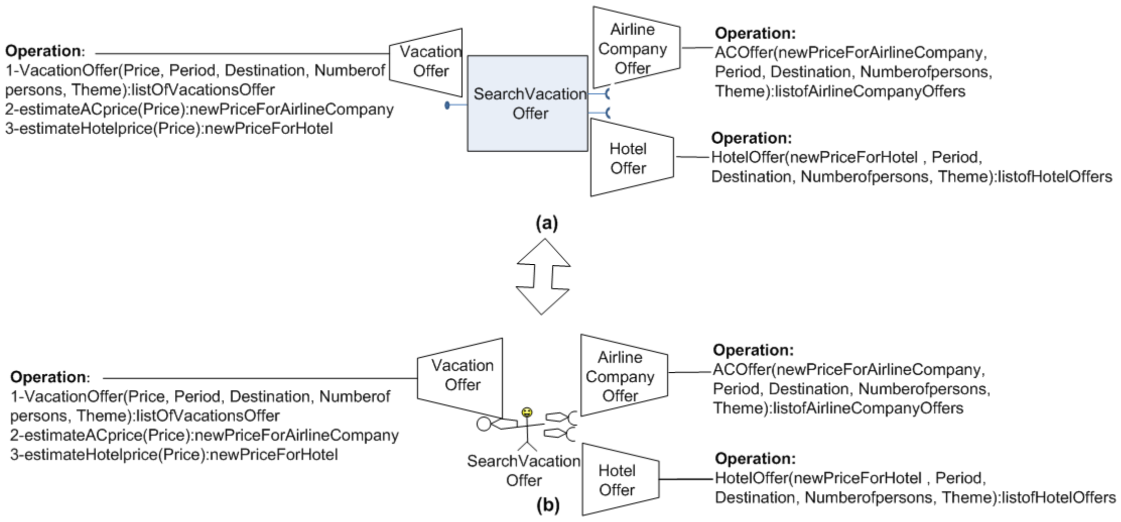

Operation:
1-VacationOffer(Price, Period, Destination, Numberof persons, Theme):listOfVacationsOffer
2-estimateACprice(Price):newPriceForAirlineCompany
3-estimateHotelprice(Price):newPriceForHotel
Vacation Offer
SearchVacation Offer
Airline Company Offer
Operation:
ACOffer(newPriceForAirlineCompany, Period, Destination, Numberofpersons, Theme):listofAirlineCompanyOffers
Hotel Offer
Operation:
HotelOffer(newPriceForHotel , Period, Destination, Numberofpersons, Theme):listofHotelOffers
(a)
(b)

Figure 9.5: A primitive component with several service points (a), an agent with several interactional roles (b).

an interactional one becomes a service point with a role. In the transformation towards CASOM, a designer can keep original concepts whenever he finds that they are sufficient. He can transform other concepts to the possible mapped concepts when he finds that they are more efficient in a defined context. For example, conforming to our component model, we can find primitive and composite components that interact with each other to provide or use services via basic or advanced interactions. In the transformation of this application towards CASOM, the designer may decide to keep components without any changes but he might change other components to agents with the same service points. Similar decisions may be taken regarding interactions. For example, when a designer finds that many components using the same service by RMI/RPC methods, he may use coordination protocols or create a dedicated agent connector to avoid bottleneck problems. This is done through the *Agentification* via CASOM which adds the concepts originally from the agent approach to the component-based one. This is also true for the *Componentification* via CASOM, where the designer may transform reactive agents into components (e.g., *AirlineCompany* presented in figure 9.6), respecting the already presented rules of mappings in the previous section where the functional roles and interactional roles become services, service points and roles respectively and an agent becomes a component. Connectors and component connectors can also be used instead of the agent connectors or advanced kinds of interaction.

9.7 Summary

In this chapter, we defined the relations between the four defined models of service, component, agent and CASOM according to our framework in figure 6.1 on page 106.

The component model can be seen as an implementation of the abstract service

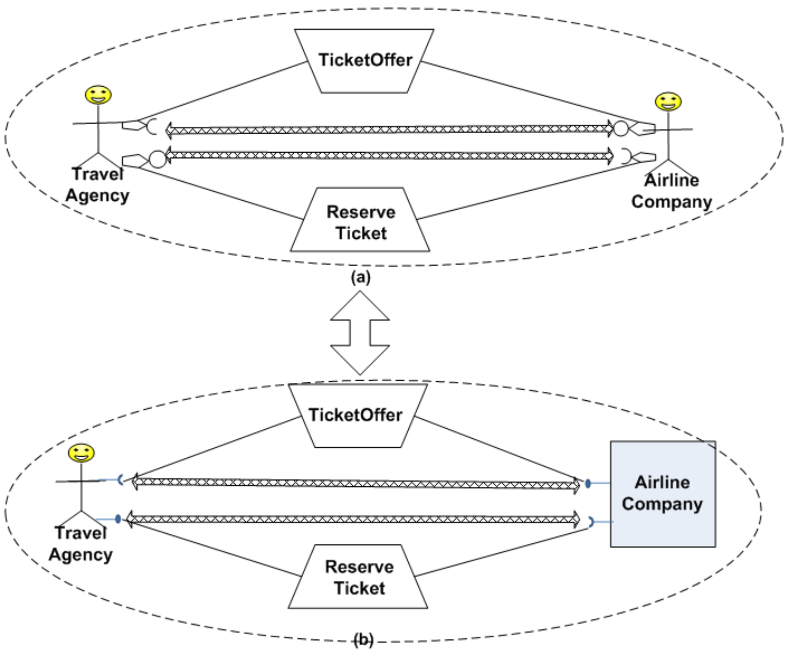


Figure 9.6: Indirect Componentification: reactive agent in agent model (a) to a component (b) in CASOM.

model, where a component implements a service and a connector a complex interaction. Thus, there are bidirectional mappings allowing to move on systematically and totally from a service specification to a component one and vice versa. Regarding the agent side, defining the mappings to the service and the component models is a bit more complicated. The main concepts of interaction and service have systematic bidirectional mappings but some secondary concepts exist only on the agent side without equivalent in the other two models. From the point of view of interactions, they are richer on the agent side compared to the service and component sides. In return, the definition of the composition is less natural on the agent side. These differences in interaction and composition are actually quite logical: we retrieve here the strengths and weaknesses of agent approaches regarding the other two approaches, as stated in the introduction. CASOM contains the concepts of component and agent models, then the vertical or horizontal transformations towards it group the already presented ones between the other three models of service, component and agent. A service can be implemented by an agent or a component and the interaction can be from agent or component kinds of interaction. However, these transformations are not achieved automatically, where they require the intervention of a designer because he has to choose between the concepts originally of component or agent models according to the system requirements. Table 9.5 groups the main mappings between the concepts of the four models.

Service side	*Component side*	*Agent side*	*CASOM side*
Primitive service	**Primitive component**	**Agent** with a **goal** by default	**Primitive component** or **Agent** with a **goal** by default
Composite service with internal services	**Composite component** with internal components	**Group** containing an agent for each internal service/component* with a **goal** by default	a **Composite** or a **Group** containing a component or an agent for each internal service
Application	**Application**	**Organization** with a **goal** by default	**Organization** with a **goal** by default
Provided (resp. required) **service point** of a service and its set of operations and playing a **role**	**Service** of a component with the same set of operations	**Functional role** with the same set of operations	**Service** of a component or an agent with the same set of operations
	Provided (resp. required) **service point** of a component with the same name and playing the same **role**	**Interactional role** with the name of the role associated with the service point (on the service side) and the type of the service point (provided or required)	Provided (resp. required) **service point** of a component or an agent with the same name and playing the same **role**
Interaction associated with a **protocol** between service points	**Connector** associated with a **protocol** between the equivalent service points	Agents high-level **Interactions** like **Communication**, **Coordination** or **Negotiation** associated with an equivalent **protocol** between the equivalent interactional roles	**Connector** in its two specialization **ComponentConnector** and **AgentConnector** or one of **Communication**, **Coordination** or **Negotiation** specialization of **Interaction** associated with an equivalent **protocol** between the equivalent service points
Basic interaction of a given type between service points	**Basic interaction** of the same type between the equivalent service points	**Communication** of basic protocol type† between the equivalent interactional roles	**Basic interaction** or **Communication** of basic protocol type† between the equivalent service points

Table 9.5: Service / component / agent /CASOM mappings

⋆ Only if these internal services are primitive. See section 9.3 for the explanations.

† Except for the delegation of the service. See section 9.3 for the explanations.

Next chapter presents the core of our design guide that presents possible variants of some of the already presented mappings in this chapter. It helps the designer to determine his choices in application specifications through CASOM.

Chapter 10

Design Guide

Contents

10.1 Introduction

This chapter groups all the cases that need the intervention of a designer to take a decision or to precise a choice. This choice must ensure the robustness and the flexibility of the system. Our design guide contains some hints that help the designer to choose the appropriate element when specifying an application via CASOM. It also presents possible variants of the mappings presented in the previous chapter, in addition to the presentation of some special cases that need the intervention of a designer through the transformations. We need to mention that this design guide is a basic one, however it is an open guide. It can become richer in its hints and mapping variants since a designer may add new hints for the application specifications through CASOM or new possible variants of transformation between the four models according to the system requirements analysis. A designer may also face other possible special cases through the transformation and he can add the specification of such cases to the guide that may help other designers in the future.

10.2 Hints for an application specification via CASOM

As we saw previously, the main idea of CASOM is to specify an application with both components and agents, where they interact with each other by using or providing services. The design guide helps the designer to choose the suitable structural entities (groups, agents, composites and components) to represent system elements. The first step to do is to identify the used or provided services of each actor in the system. In order to facilitate the choices between structural entities to represent a system element; you need to ask the following questions:

1. Does this element simulate human behaviour (it deals with humans or with other applications)?

2. Does this entity need to take decisions through its interactions with other system elements?

3. Do the interacted system elements consider a general objective although they may own their own goals and different capabilities?

4. Do the system elements have well-defined, and clearly separated income and outcome services?

5. Can these elements be decomposed to several parts that interact between each other in parallel or sequential ways?

If the answers of the first two questions are positive then the agent concept is the ideal candidate to represent such elements because we need an autonomous entity that is able to take decisions.

A positive answer to the third question leads to the group of agents that interact between each other to realize a defined goal.

Positive answers to the last two questions lead us to the components and the composites that can contain also primitive and composite components.

Many possible compositions of positive answers are possible for these questions. The designer needs to study and know the system requirements, he needs also to know which entity will decrease the cost of the system and increase its efficiency in the same time (e.g., using agents with decision-making and reasoning capabilities may affect the speediness of the system. Then this choice may decrease the system efficiency if it was supposed to be a speedy system).

In CASOM, all kinds of advanced interactions are available to components, and they own their goals. Moreover, agents become reusable through their service points. Then, we have reusable entities (components and agents) with high-level interactions and defined goals. However, the main difference is that a component can be composite itself of other primitive or composite components, but an agent cannot although it can be a part of a composite. Then, the component is always the ideal candidate for any composite structure. On the other side, the autonomy is still restricted in agents, where they own their capabilities of perception, reasoning and making decisions. Then an agent is still the best candidate to represent autonomous entities.

One may ask why not adding relations between capability and component concepts, so components become also autonomous. The response to this question is related to other issues like, how to define components capabilities, and how these capabilities can affect the component services since these services must be well-defined and not changeable. Specification hints by CASOM are open, where a designer may add new hints according to the specified system. The content of our design guide is built incrementally by designers' experience, where using these variants requires designers intervention to avoid the systemically proposed mappings.

Next section browses possible variants of some already presented mappings in the previous chapter.

10.3 Mapping variants

In this section, we present possible variants of mappings instead of some direct mappings presented previously. These variants can be translated to transformation rules but the designer needs to precise whether he wants to apply the direct mappings or the variants.

10.3.1 Service and component models

The first variant consists in not transforming systematically a composite (resp. primitive) element into a composite (resp. primitive) element of the other model. The rule is that if the internal elements in a composite (component or service) do not interact with each other, we can transform this composite element into a primitive element on the other side[39]. This variant is notably interesting in the case where we do not want to specify a service as implicitly formed of provided or required sets of operations (via associated service points). We dedicate explicitly each set of operations to a particular service (that is, a primitive service that contains only one service point), defining in this way a composite service containing primitive ones. However, regarding the component side, it is more relevant to have a primitive component as each set of operations is directly associated with a service point, then with a single service. Figure 10.1 presents a composite service *Client* that contains two internal primitive services having each one a single service point. This composite service is mapped to a primitive component *Client* directly defining the two service points equivalent to those of the internal services of the composite service. The external service points of the composite service with its roles and delegation interactions are ignored in the new mapped component.

A second variant consists, on the component side, to prefer using a primitive ComponentConnector instead of a connector associated to a protocol to map a complex interaction on the service side. To add a component connector concept, we need to add

[39]If the internal elements are connected, this means that a designer has explicitly specified internal interactions between these elements and by "merging" them in just a single one, we would cause to lose these interactions and the associated explicit internal structure. Then, there would be information lost.

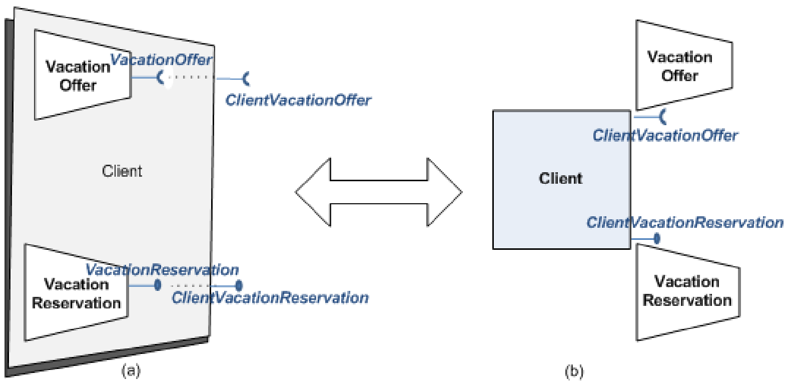


Figure 10.1: A composite service (a) to a primitive component mapping(b) mapping.

the symmetric service points (associated to the same service but by providing it instead of using it or the reverse) with interactions of basic types like RPC/RMI between the ComponentConnector and the connected components. Figure 10.2 shows the two possible choices of mapping a complex interaction on the service side to a *Connector* or *ComponentConnector* on the component side. In service side (part (a)), two services are related via protocol through their service points. In component side, the two service points of the corresponding components are related using *Connector* (part (b)) or primitive *ComponentConnector* owning the symmetric service points (part (c)).

10.3.2 Service and agent models

Similarly to what we have presented for the mappings of service and component models, here we have a mapping variant that does not associate systematically a primitive service of the service model with an agent of the agent model. If we take the example of a composite service where its two internal services are not internally interacted, it is not necessary to have two different agents. We can make the choice to group these services within a single agent by assigning both corresponding functional and interactional roles to the two service points. As an example, the composite *Client* service containing two primitive services (figure 10.3 (a) is mapped onto a group containing a single agent instead of two agents (b)).

Another possible variant of mapping of a composite service to a group of agents (instead of the one already presented in figure 9.3 on page 159) is to add systemically an agent that represents the composite component and its external service points (the agent client in figure 10.4 (b)). This variant is very interesting to be used in the case where we have many composite services and the reverse transformation (from the resulted agent model to the service one) may be needed. This variant keeps most information during the transformation (i.e., external service points (names, roles) and the communication of delegation between the internal service points).

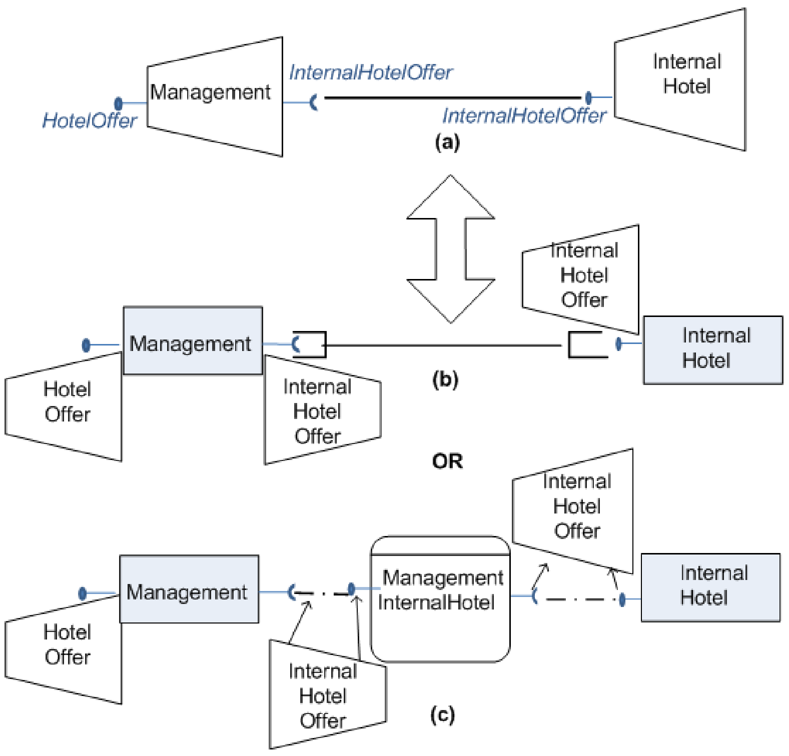


Figure 10.2: A protocol (a) to a connector (b) or a component connector(c) mapping.

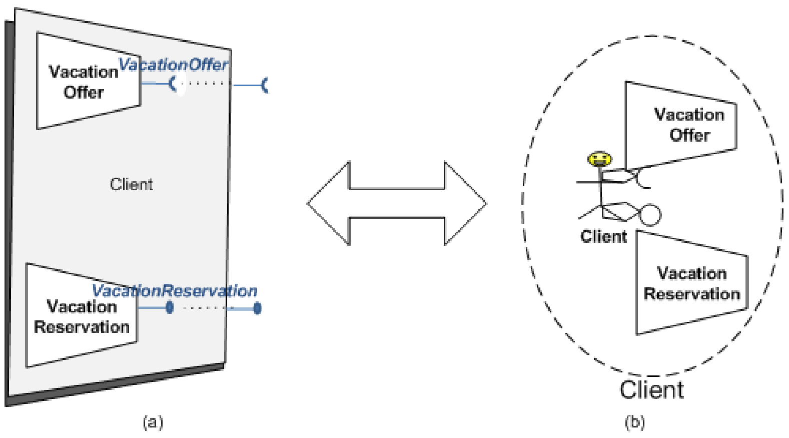


Figure 10.3: Internal services of a composite service (a) to a group of one agent (b) mapping.

10.3.3 Service and CASOM models

Similarly to the two already presented variants in the transformations from the service model to the component or agent one, we have two main variants. The first is related to the structural element. Following the already presented rules, we do not transform

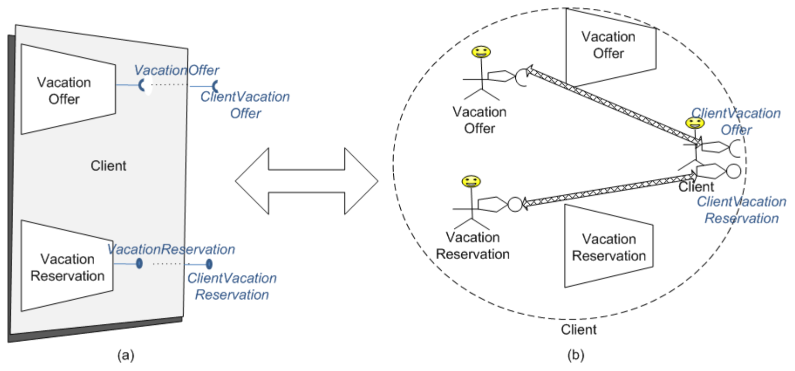


Figure 10.4: A composite service (a) to a group of agents (b) mapping.

systemically a composite service into a composite component, a group of agents or components whenever the internal elements do not interact between each other. A primitive component or a group of one agent may be sufficient to implement such kind of composite service.

The second variant is related to the implementation of complex interaction in the service model. Instead of the choice of using the classification of interaction specialising this interaction, we implement it by a connector. Then, we have two possibilities to specify and manage the behaviour of such connector either by the use of a dedicated component connector or agent one. It depends after all on the desired implementation of the connector behaviour management. It can be transparent through the agent connector or it can be encapsulated in a black box and composed of other elements by the component connector.

10.3.4 Component and agent models

The first variant is not to transform systematically a primitive component within a composite onto an agent. We may choose to group non-interacting internal components of a composite within the same agent and vice versa (figure 10.5). We keep the same idea in the proposed mapping variant between a composite service and a group of agent in figure 10.4 here. Figure 10.6 presents a possible variant of a composite component mapping to a group of agents. In this proposed mappings we keep all the information in the composite component (external and internal service points name, roles and delegation interactions). Whenever a reverse transformation is needed (from the resulted agent model to the source component one), this variant of mapping is ideal to be used.

Similarly, based on the mapping variant between service and component models, we can map a complex interaction like negotiation between agents (figure 10.7 (a)) onto a primitive component connector “with its symmetrical service points” instead of a connector (figure 10.7 (b)).

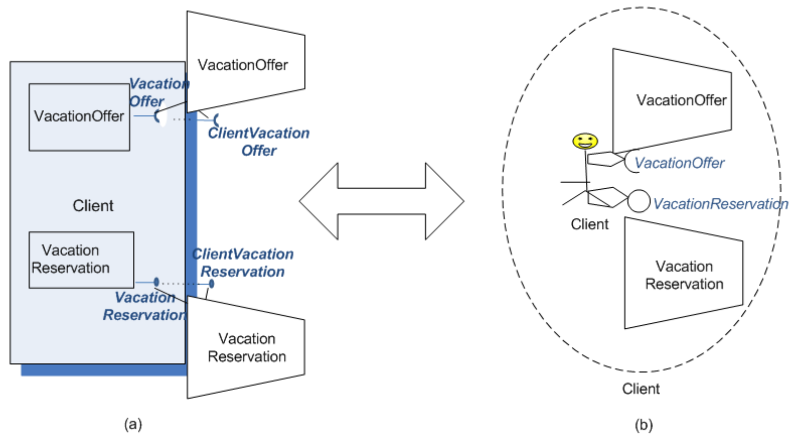


Figure 10.5: A composite component with several internal components to a group of one agent mapping.

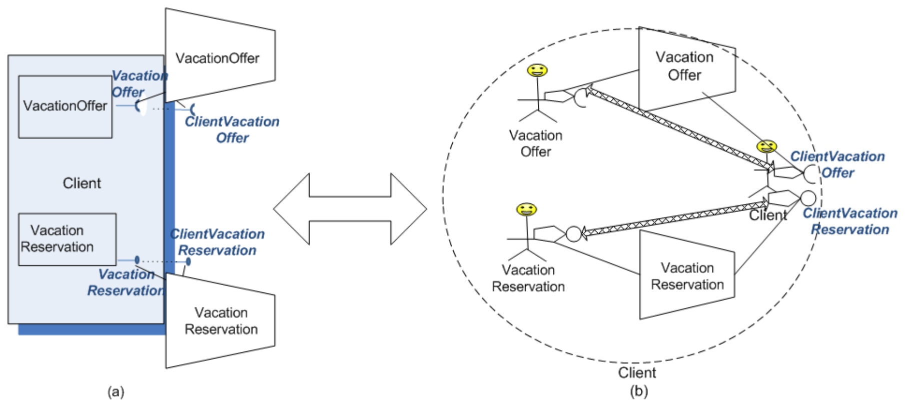


Figure 10.6: A composite component (a) to a group of agents (b) mapping.

10.4 Special cases of mappings

We need to discuss some special cases, which may not occur frequently in an application specification and may need the intervention of a designer, like:

- Composite component of two or more levels: this problem is already addressed in section 9.3 on page 158 between the composite service and agent transformations. The absence of the composition on agent side causes different problems in the transformation related to the composition and delegation concerns. Using representing agent for the composite of composites component is already proposed as a solution in the mappings of a composite of composites service from service model to agent one. We use the same principles in the illustrative example (figure 10.8) to clarify the case of the transformation of a composite of composite component to groups of agents with an additional representing agent. The rule here is to map each (internal or external) composite component (resp. service) to a

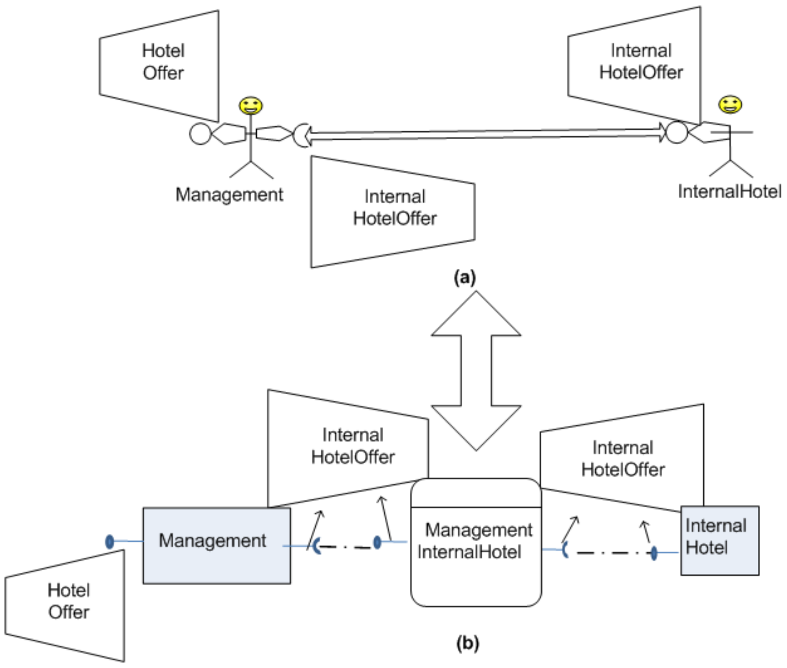


Figure 10.7: Agents negotiation interaction(a) to a component connector (b) mapping.

group of agents corresponding to the internal primitive components with adding by default a representing agent for this composite. This agent has a derived name from the name of the interacting components and the actual composite component (e.g., the agent is named AB in figure 10.8) and it provides or uses the services provided or used by the internal delegated service points of a component. We need to precise here that this special case occurs from the component to agent model transformation. However, whenever we apply the reverse transformation (from the resulted agent model to component one), we will not have the exact model as the original component one if we apply the direct mapping rules because we have this additional agent that can be mapped to a new component (which does not exist in original model). This shared agent helps only to define the composite of composite components, then the designer need to interfere to avoid applying the direct mappings rules for this case.

- Agents that belong to two groups: this case may occur from the agent model to component one. Our agent model does not prevent an agent to be member of two groups at the same time. The logical mapping of this case is to have a shared component. This kind of component is not allowed in most of components models that causes a problem here. The proposed solution is to apply the previously defined mappings rules (table 9.4 on page 163) for all the group members but to duplicate the corresponding component for this shared agent twice with its services in the two corresponding composite components. In figure 10.9, we find the *A*

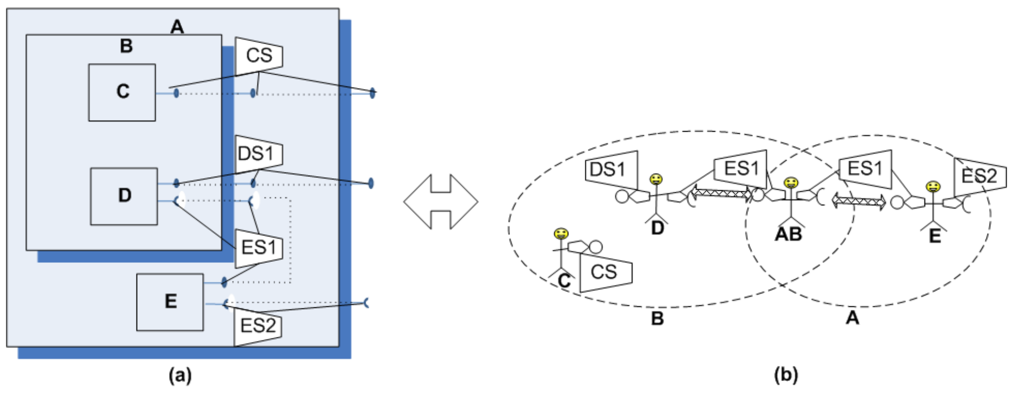


Figure 10.8: A composite of composite component (a) to groups of agents (b) mapping.

composite component mapped to the A group, we have a primitive component C corresponding to the shared agent C and another primitive component with the same name of this agent C also exists in the other composite component B corresponding to the second group B). The names of the exchanged services

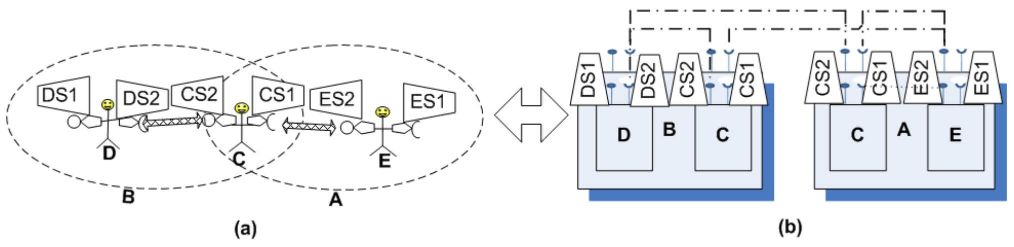


Figure 10.9: A shared agent between two groups (a) to two composite components (b) mapping.

over the interactions and the agent name help to distinguish between the reverse transformations of the representing agents to composite components used in the previous case and the shared agent here since they look similar.

- Application specifications via CASOM: as we saw previously in chapter 8, CASOM allows widest combinations of agent and component model concepts. A designer is able then to define composite components containing groups of agents, these groups in their turns can contain composite components, agents and groups in the application specified by CASOM. Then, the transformations from CASOM to component or agent models require the intervention of the designer after the application of the direct mapping rules (already presented in section 9.4 on page 159 and 9.6 on page 163) to consider the already presented special cases or new ones.

10.5 Summary

We presented in this chapter the base of our design guide. It helps the designer to precise his choices when he specifies an application by CASOM or to use certain mapping variants during the transformations instead of the systematically proposed ones. The system requirements play a key role in the application specification. From this chapter, we find that a designer must get in the way through the application specifications via CASOM and the transformation from / to this model. A designer may interfere to use the proposed mapping variants in this section instead of the direct already proposed one in the previous chapter. The table 10.1 presents the transformations with their main variants. The transformations are completely reversible between component and service models. However, they need a designer intervention in any transformation to the target of agent model to define the needed capabilities and associated tasks. The transformations towards CASOM require also a designer intervention to choose the target mapped elements, since there are at least two choices that are originally from component or agent model. A choice by default may be proposed for the transformation to the target of CASOM, which may not respond to the designer or system requirements, and then the designer has to change the default proposed mapped elements manually. The transformations from CASOM in the direction of the other models are already included in the transformation from component to agent models, service to component and service to agent transformations. It depends on the used elements in CASOM and the needed ones in the target model (e.g. from CASOM to component, we can translate an agent to a component and a component to a component).The development of an environment that manages the choices for designer interventions is one of our main concerns in near future.

The process of building this guide is an incremental one, where any designer facing special cases and personalizes new solutions, can add them to the guide since it is open.

Next chapter ends the contribution part by presenting the implementation of our framework in its four models and the relations between them in a modeling driven engineering environment.

Transform.	**Automatic**	**Manual (designer)**	**Lost Information**	**Variants**
Service To Component	Yes	N/A	None	Complex Interaction can be mapped to a connector or a component connector
Component To Service	Yes	N/A	Associated protocol to a component connector⋆	N/A

Transform.	**Automatic**	**Manual (designer)**	**Lost Information**	**Variants**
Service To Agent	Yes	Tasks, capabilities and goals of agents	None	A complex interaction can be mapped to communication, coordination and negotiation kinds of interaction‡
Agent To Service	Yes	N/A	Tasks, capabilities and goals of agents†	N/A
Component To Agent	Yes	Tasks, capabilities and goals of agents	None	A connector or a component connector can be mapped to communication, coordination, and negotiation kinds of interaction‡
Agent To Component	Yes	N/A	Tasks, capabilities and goals of agents	An interaction by communication or coordination or negotiation can be mapped to a connector or component connector.
Service/ Component/ Agent To CASOM	No±	Yes, elements implementing services and complex interactions	None	N/A

Transform.	**Automatic**	**Manual (designer)**	**Lost Information**	**Variants**
CASOM To Service/ Component /Agent	Yes	N/A$^{\mp}$	Tasks, capabilities and goals of agents if its used or associated protocols to a component connector if it was used in CASOM	N/A$^{\mp}$

Table 10.1: Summary of the accomplished transformations

* A component connector is viewed as an ordinary component.
† If we apply a transformation from the service to agent model and in the service model, there were compositions of many levels. The resulted agent model will have representing agents to ensure the representation of these compositions. However, in the reverse transformation, from the resulted agent model to the service one, a designer should check the services resulted from the translation of the representing agents to be sure to have the exact source service model.
‡There is no hierarchical composition in the agent model, therefore we use a dedicated agent to represent the N nested levels of composition.
$^{\pm}$ It cannot be automatic because there are at least two choices, to map the source model elements, that are originally from component or agent model.
$^{\mp}$ The lost information and variants are already included in the transformation from component to agent models, service to component and service to agent transformations.

Next chapter ends the contribution part by presenting the implementation of our framework in its four models and the relations between them in a modelling driven engineering environment.

Chapter 11

Implementation

Contents

11.1 Introduction

In this chapter, we implement our framework (models and transformations) via Eclipse Modeling Framework (EMF) (already presented in chapter 6). We presented the service, component, agent and CASOM models in figures in chapters 7 and 8 using UML class diagram specification to present these models conceptually. However, this representation needs some reformatting in order to be implemented in Ecore meta-models. This chapter presents then the implementation of the four models in Ecore, in addition to the implementation of the relations between these models (the mappings already presented in chapter 9) via ATL transformation rules at the end of this chapter. The general steps of our implementation in EMF are the following.

- We implement the four models of service, component, agent and CASOM using Ecore model.
- We implement the mappings between pairs of service, component, agent and CASOM models via Atlas Transformation Language.

Next section presents how to move on from our conceptual models to DSLs / Ecore (meta-)models.

11.2 From conceptual models to DSLs / Ecore models

Many changes need to be applied on our conceptual models to represent them via Ecore model. These changes are related to the absence of some elements in Ecore model that exist in UML like association classes or they are related to the respected ways of defining a model. These ways exist in Ecore and do not exist in UML, which imply the addition of some concepts like a root element, composition relations, factoring elements for common concepts. Here, we present the applied changes to move on from the conceptual models to the DSLs (the implemented model in Ecore).

- **The association class:** we want to implement our conceptual models using Ecore meta-model since EMF contains plug-ins to implement the transformations between them. However, the main difference between Ecore meta-model and UML class diagram is the fact that Ecore does not have association classes that we use frequently in our proposed models. Then, when implementing these models in EMF by Ecore meta-model, some changes must be applied. Figure 11.1 represents an example of the translation of an association class from UML class diagram to Ecore model. We can see that an *interaction* is related to two *ServicePoints* at least by a *Role* (see figure 11.1 (a)). In Ecore, we break the association class by having two relations instead of one. The first relation is between an *interaction* and at least two *Roles*. The second relation is between the *Role* which is related to one *ServicePoint* (see figure 11.1 (b)).

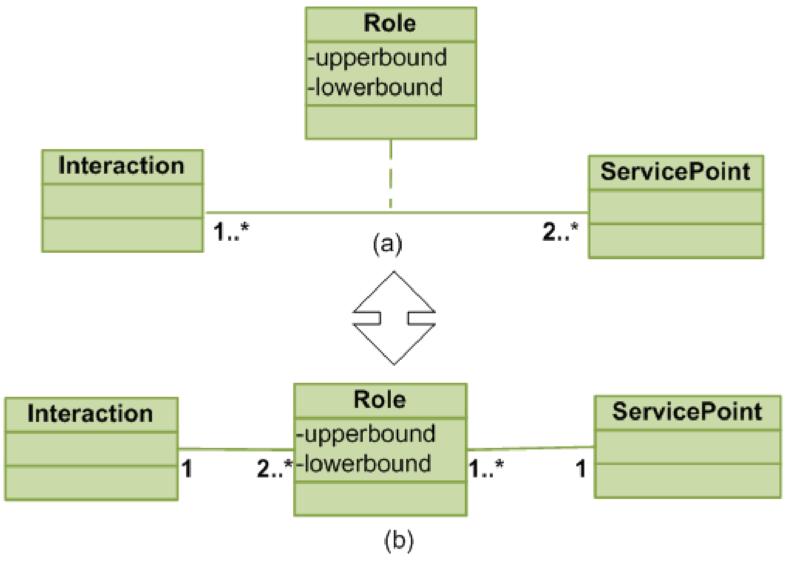


Figure 11.1: An association class from UML to Ecore specification.

- **The root element:** In Ecore, for helping in editing a model within a single XML Metadata Interchange (XMI)[40] file, a particular design rule has to be applied when

[40]http://www.omg.org/spec/XMI/

creating a meta-model. It is base on the fact that if a meta-element A has a composition reference on a meta-element B, then, when editing the model, it is possible to directly create instances of B whithin an instance of A. The particular design rule consists in having a kind of root or container element and to have direct or transitive composition links from it for all the elements of the meta-model. For our DSLs, we decide to make the *Application* element in the service and component meta-models and the *Organization* element in the agent and CASOM meta-models, being this root element for their respective meta-models. Indeed, such elements represent the application as a whole, they are then natural candidates for embedding directly or transitively all the elements of a model. This design rule requires also to add some composition links starting from the root element that have no equivalent or are not relevant at the conceptual level but they will help in editing the models. For instance, in the service meta-model, a composition link has been added from *Application* to *Interaction*.

- **The factoring element:** in order to avoid the repetition of the attribute *Name* that exists in almost all the concepts of the four models; We added a new concept *NamedElement* as an abstract class and all concepts inherit its attribute *Name*.

11.3 The four DSLs in Ecore

We present here service, component, agent and CASOM models in Ecore. This representation respects the already presented points to move from the conceptual model to the one in Ecore.

- Figure 11.2 shows our already presented service model (figure 7.1 on page 117) in Ecore. There is no association class in this model, where we break it as we presented previously in two relations (a relation between an *interaction* and at least two *Role*s and a relation between the *Role* and *ServicePoint*). The *Application* class is a root concept with relations of compositions with the other concepts. We can see the class of *NamedElement* as an abstract father concept for all the other concepts.

- Figure 11.3 represents our component model (already presented in figure 7.4 on page 126) in Ecore. Similarly to the service model in Ecore we can find the same changes (i.e., no association class, an *Application* root class with composition relations to reach the other concepts and the abstract *NamedElement* concept also exists.

- Figure 11.4 demonstrates our already presented agent model (figure 7.6 on page 133) in Ecore. The implementation of the association class between the *Group*, *FunctionalRole* concepts by the *Agent* one (see figure 11.5 (a)) leads to a relation that says an agent belongs to one group and it is associated with one *FunctionalRole* (i.e., service) in the Ecore model of agent (see figure 11.5 (b)). The interpretation of these relations is not logic then we change manually two cardinalities (see (c) figure 11.5). The modified relations cardinalities allow an agent to have many

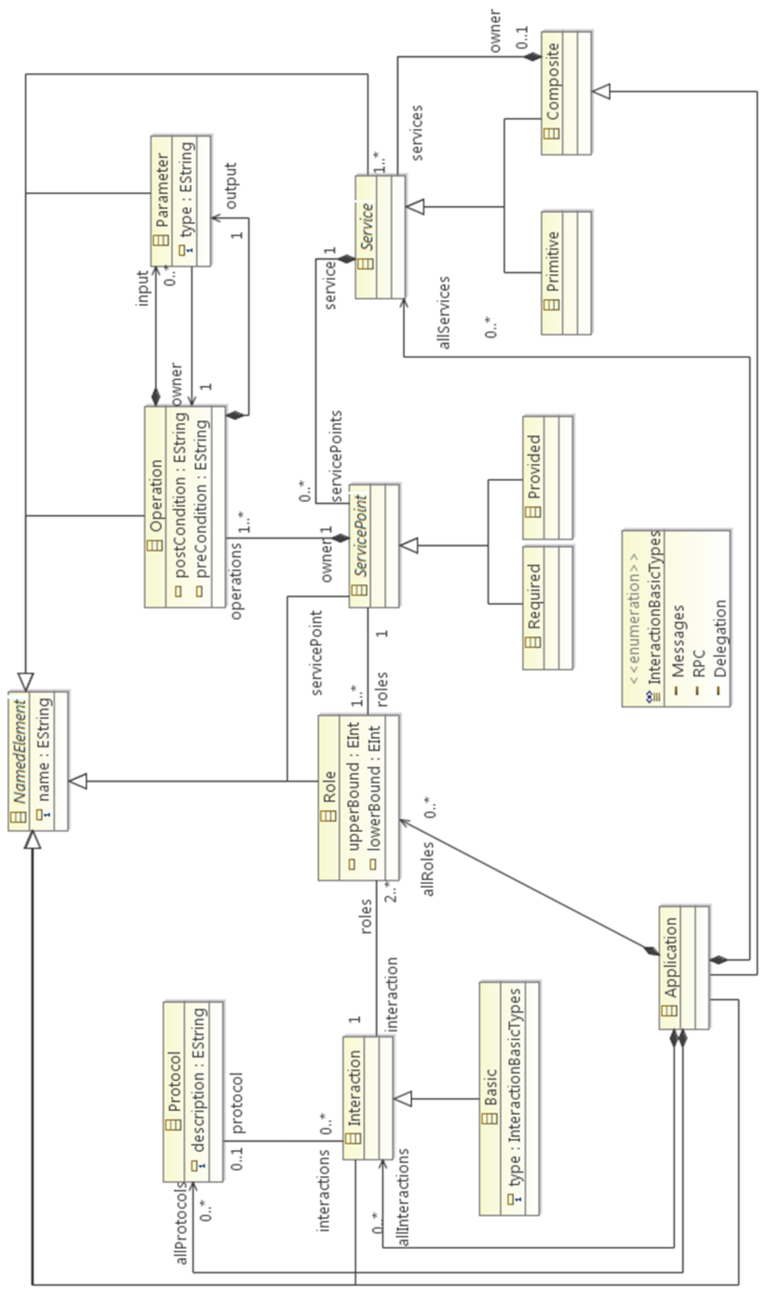


Figure 11.2: The service model in Ecore.

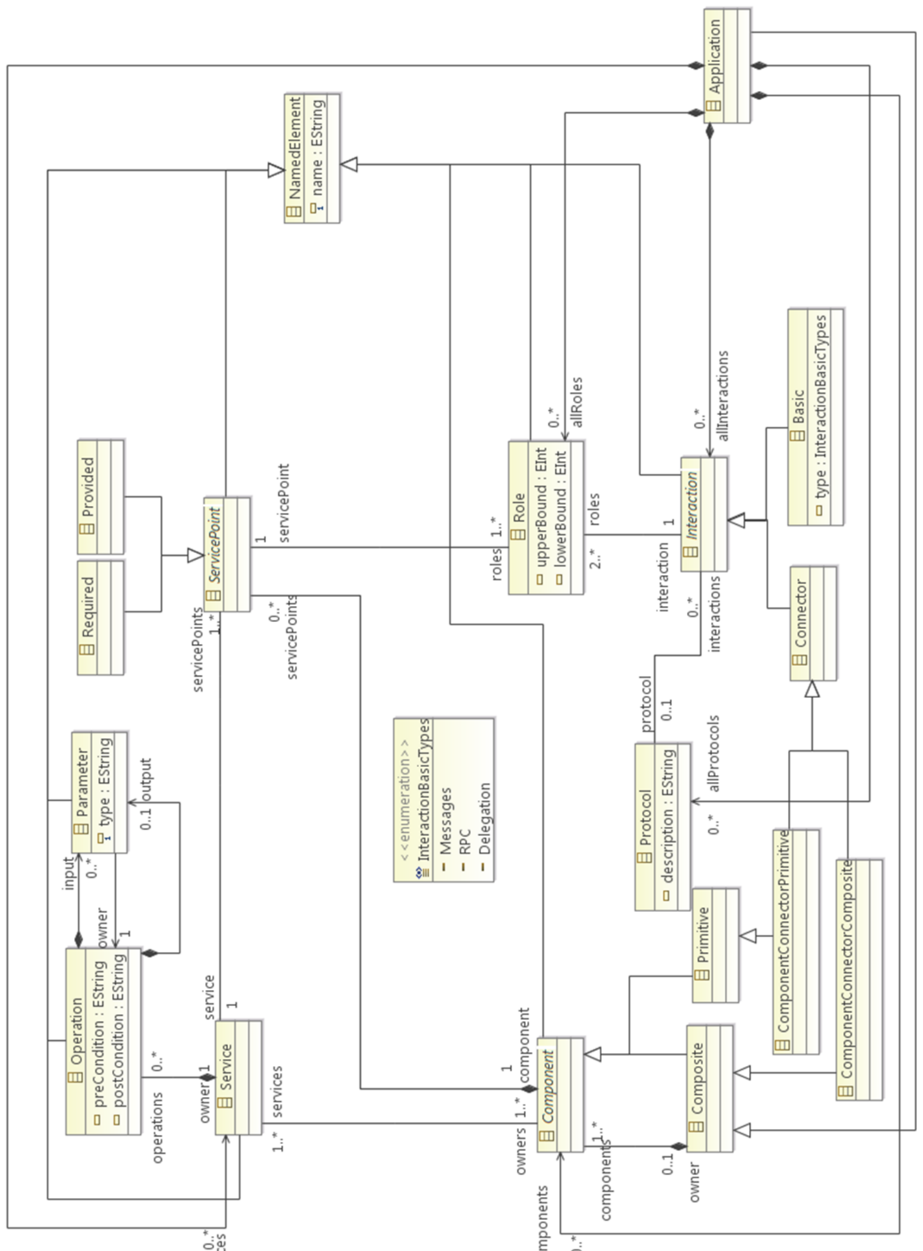


Figure 11.3: The component model in Ecore.

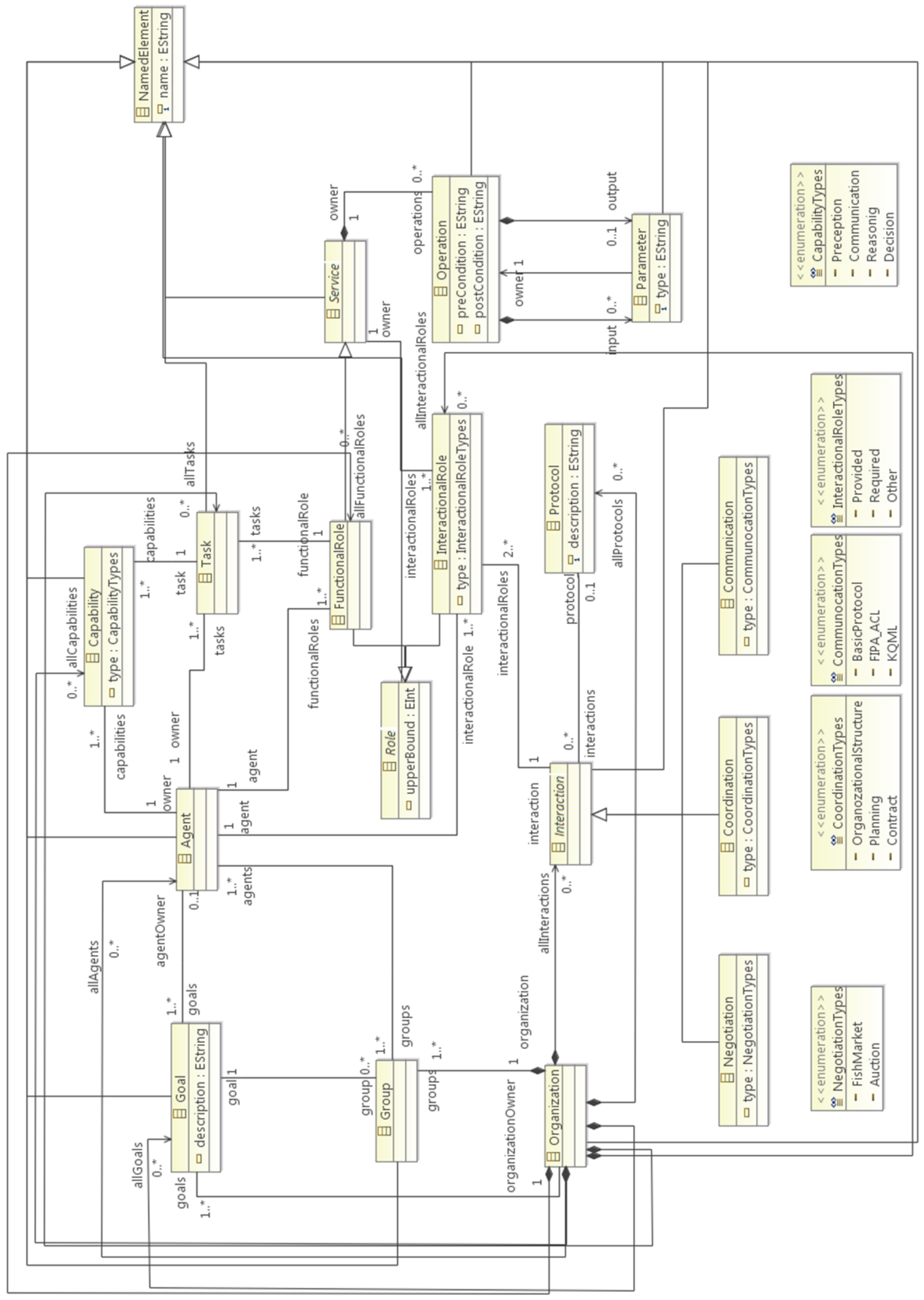


Figure 11.4: The agent model in Ecore.

FunctionalRoles. They allow also having a shared agent between two groups. We already used this shared agent in the mappings from service / component model to agent one where we use a shared agent to represent a composite of composite service / component. Then, we change manually this relation in the agent model (see figure 11.5 (c)). The root concept is the *Organization* one with its composition

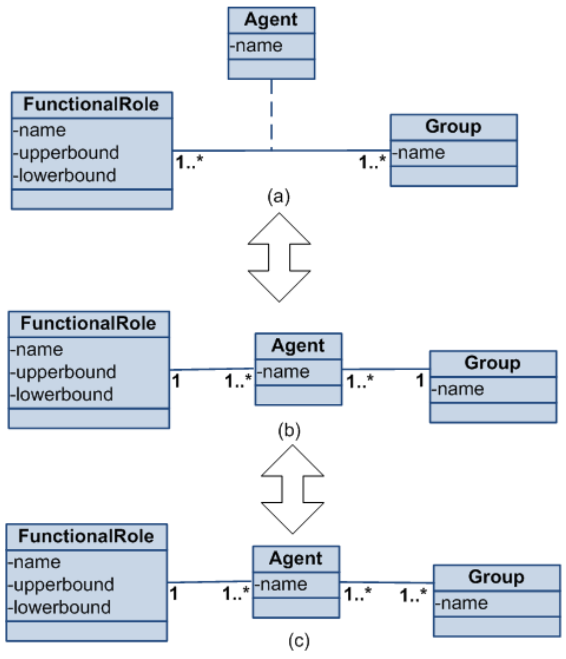


Figure 11.5: The actual association class between group and functional role class by agent one (a) the corresponding relation in Ecore (b) the actual relation in Ecore (c).

relations with other concepts and the abstract *NamedElement*) concept.

- Figure 11.6 represents our CASOM model (already presented in figure 8 on page 139) in Ecore, with the *Organization* concept as the root one in addition to the existence of the abstract *NamedElement*) concept.

11.4 The ATL transformations implementing the mappings

ATL is a model transformation language; it contains functions that help to produce a target model from a source one. Any ATL program is composed of transformation rules and some functions named helpers. In this section, we implement our already proposed mappings between the four models in chapter 9. We present some main rules to generate a component model from a service one. We start by structural entities related rules, like the transformation from composite and primitive service to a composite or primitive component (see figure 11.7). As well as the required (resp. provided) service points to

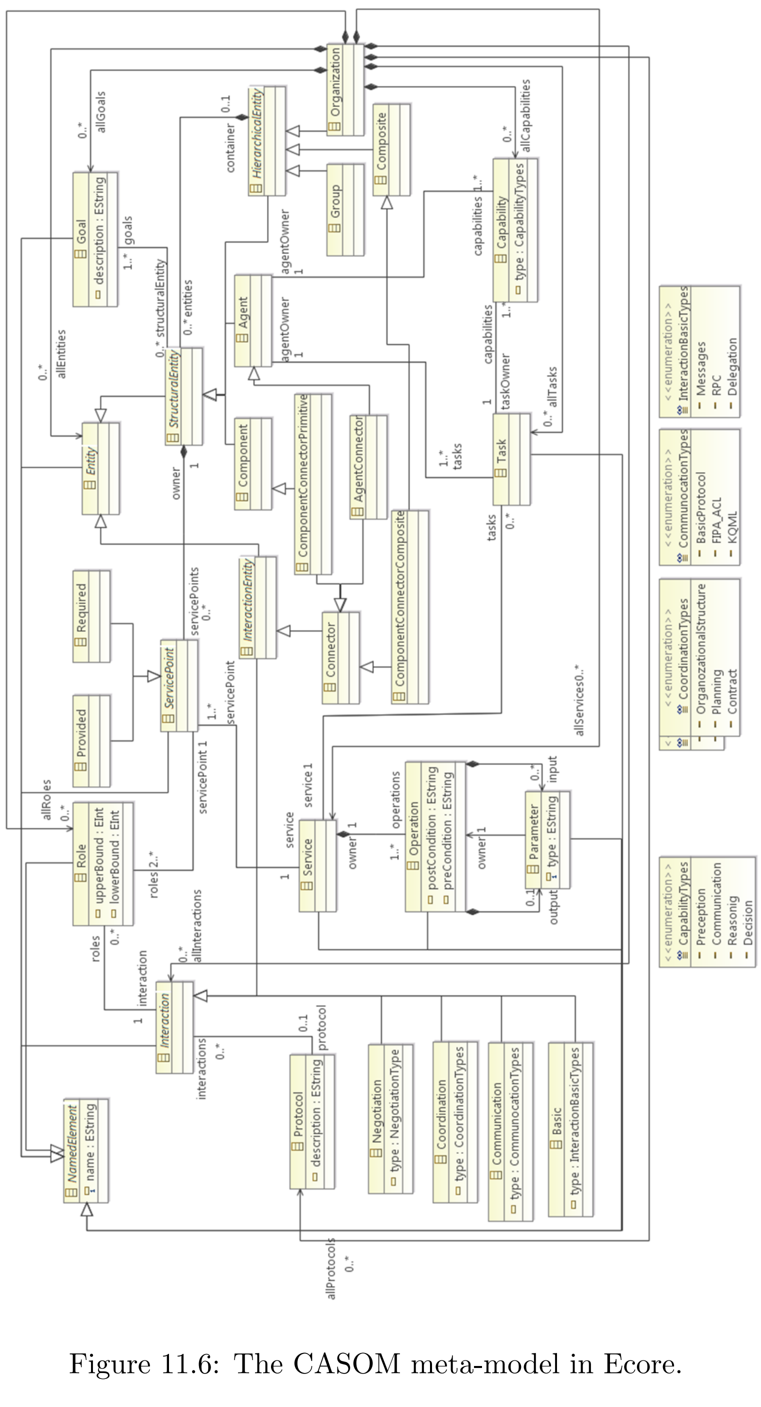


Figure 11.6: The CASOM meta-model in Ecore.

required (resp. provided) service point and a service with the same list of operations as already presented in table 9.1 on page 155.

```
rule ServiceCompositeToComponentComposite {
From
sourceService : ServiceModel!Composite
to cibleComponent : ComponentModel!Composite (
name<- sourceService.name,
owner<- sourceService.owner,
components<- sourceService.services
) }
rule ServicePrimitiveToComponentPrimitive {
from
sourceService : ServiceModel!Primitive
to
cibleComponent : ComponentModel!Primitive (
name<- sourceService.name,
owner<- sourceService.owner
) }
rule ServicePointRequiredToServicePointRequiredAndService {
from
sourceServicePoint : ServiceModel!Required
to
cibleService : ComponentModel!Service (
name<- sourceServicePoint.name,
owners<-cibleService.owners->including(sourceServicePoint.service),
operations<-sourceServicePoint.operations
)
,
cibleServicePoint : ComponentModel!Required (
name<- sourceServicePoint.name,
component<-sourceServicePoint.service,
service<-cibleService,
roles<-sourceServicePoint.roles
)
do{
thisModule.getComponent(sourceServicePoint.service.name).servicePoints<-
thisModule.getComponent(sourceServicePoint.service.name).servicePoints-
>including(cibleServicePoint);
}
}
```

Figure 11.7: ATL rules for structural entities from the service model to component one.

Another transformation rule is related to the interaction concept. Whenever, we have an interaction related to a protocol in a service model (complex interaction), it becomes a connector specified by the same protocol in the component model (see figure 11.8).

```
rule complexInteractionToConnector {
from
sourceInteraction : ServiceModel!Interaction(not sourceInterac-
tion.protocol.oclIsUndefined())
to
cibleConnector : ComponentModel!Connector (
name<- sourceInteraction.name,
protocol<- sourceInteraction.protocol,
roles<-sourceInteraction.roles
) }
```

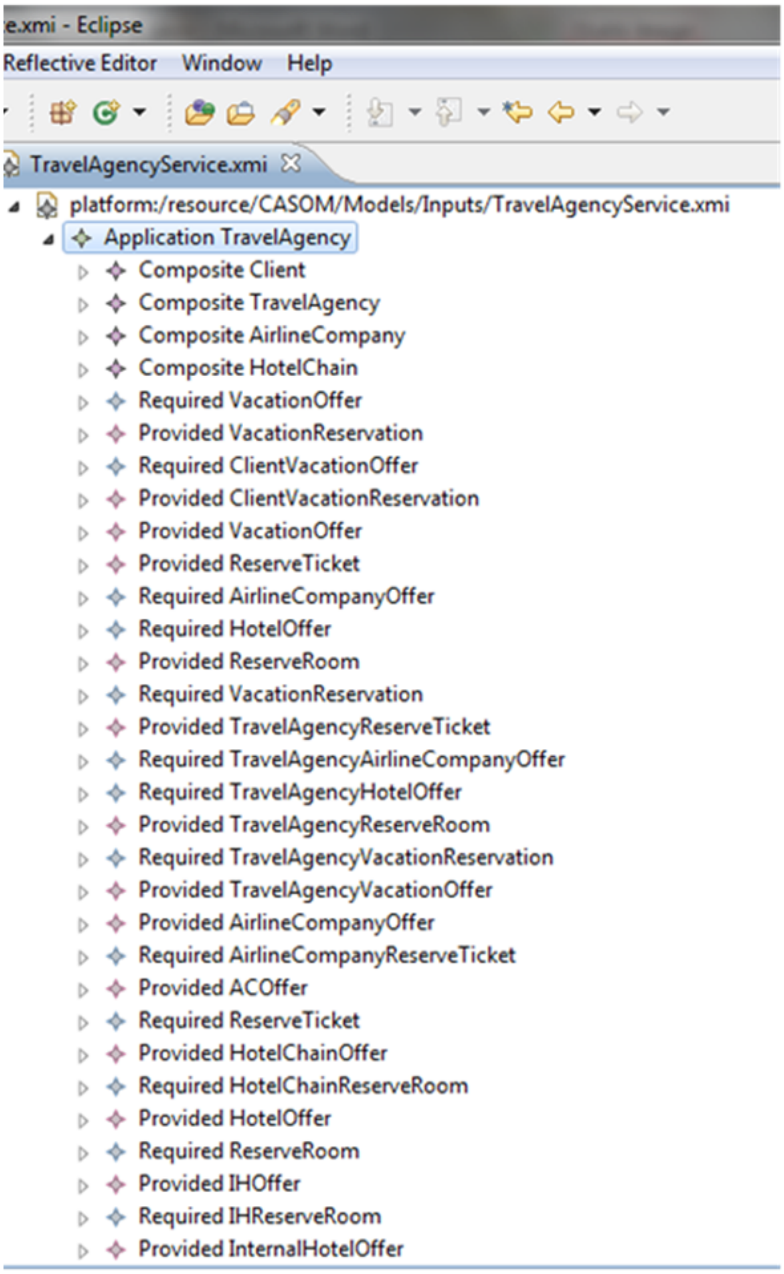
Figure 11.8: ATL rule to transform complex interaction from the service model to the component one.

In the same way, we provide the rules of transformations from service to agent model and from agent to component one, in addition to the reverse transformations for all the previous ones.

11.5 How to use our MDE framework?

After the implementation of our four models and the relations between them, we need to know how to use this framework. Firstly, we need to create an application example for one of our four models of service, component, agent and CASOM (e.g., service application example). Secondly, we apply the suitable ATL transformations corresponding to the already presented mappings between the four models in chapter 9. We get as a result the application example corresponding to the target model according to the applied transformation. The application example conforming to a defined model is created by the creation of the class instances and relations between them conforming to this model. The creation of the instances can be generated from the root class (i.e., the *Application* class in the service / component model and the *Organization* class in the agent/CASOM model). We get in EMF a file that groups all instances of each concept and their relations with other concepts, this file is from the type XMI. We can create any of our already presented travel agency examples in figures 7.2 on page 120, 7.5 on page 129, 7.7 on page 135 and 8.3 on page 150. Figure 11.9 shows a part of the vacation reservation example conforming to our service model where it is not possible to presents all the instances in one figure. One of our main perspectives is to provide a friendly view of these models using the Graphical Modeling Framework (GMF)[41]. GMF allows us to realize graphic editors that edit and assign graphical shapes for the classes in our models. These editors allow us to visualize our application examples in the same way presented in figures 7.2, 7.5, 7.7 and 8.3 by using the same shapes. This will facilitate the reading of the XMI files.

[41]http://www.eclipse.org/modeling/gmp/



platform:/resource/CASOM/Models/Inputs/TravelAgencyService.xmi
Application TravelAgency
Composite Client
Composite TravelAgency
Composite AirlineCompany
Composite HotelChain
Required VacationOffer
Provided VacationReservation
Required ClientVacationOffer
Provided ClientVacationReservation
Provided VacationOffer
Provided ReserveTicket
Required AirlineCompanyOffer
Required HotelOffer
Provided ReserveRoom
Required VacationReservation
Provided TravelAgencyReserveTicket
Required TravelAgencyAirlineCompanyOffer
Required TravelAgencyHotelOffer
Provided TravelAgencyReserveRoom
Required TravelAgencyVacationReservation
Provided TravelAgencyVacationOffer
Provided AirlineCompanyOffer
Required AirlineCompanyReserveTicket
Provided ACOffer
Required ReserveTicket
Provided HotelChainOffer
Required HotelChainReserveRoom
Provided HotelOffer
Required ReserveRoom
Provided IHOffer
Required IHReserveRoom
Provided InternalHotelOffer

Figure 11.9: Part of the service model instances conforming to our service model.

11.6 Summary

In this chapter, we implemented our framework by implementing its models and the relations between them in Ecore Modeling Framework. The service, component, agent and CASOM meta-models are defined in Ecore, that requires some changes on the con-

	Service To Component	Component To Service	Service To Agent	Agent To Service	Component To Agent	Agent To Component	CASOM To Service
lines of code	171	165	376	298	429	300	350

Table 11.1: Transformations implemented in ATL

ceptual models which are already represented in UML class diagram. These changes are related to the absence of some elements in Ecore model that exist in UML like association classes or to the respected ways of defining a model in Ecore. These respected ways imply the addition of some concepts like a root element for each model with composition relations to reach the other concepts in the model.

The relations between service, component, agent and CASOM models are implemented in ATL transformation language. Table 10.1 on page 180 already presented the main transformations. The transformations between component and service models are completely reversible. The transformations from service / component model towards agent/ CASOM one are automatic but the designer needs to define the agent tasks and capabilities. However, the reverse transformation from the resulted agent/ CASOM model towards service / component one are direct and automatic where some concepts will be removed (i.e. task and capability).

Table 11.1 shows the implemented transformations of table 10.1 and the number of code lines for each one. We can see from this table that the code size for ATL rules between service and component models is relatively small, since the component model implements directly the service one. However, the code size of ATL rules between the component/ service models and agent/CASOM one are relatively large regarding their riches in the number of concepts.

In order to use these transformations on application examples, a designer must follow the next steps:

- design the application model conforming to one of the service, component, agent and CASOM models,
- select a target model,
- launch the ATL transformation engine,
- complete, if needed, the resulted model.

Conclusion

In this part, we presented our contributions starting by the presentation of our framework. This framework relates the three domains of service, component and agent with the domain allowing mixing the three previous ones in one model (chapter 6). This chapter also presents our running example of holiday reservation system that we use along this part to illustrate the application examples (to create instances of the model concepts corresponding to our holiday reservation system elements). We propose a general model for each domain in chapter 7, with their main concepts and OCL constrains to remove possible ambiguity. We propose in chapter 8, our general model that allows specifying applications by interacting agents and components via services. The relations between the four models are defined in chapter 9. A design guide that helps the designer to precise his choices (in specifying an application by CASOM or choosing variants of mappings through transformations) is proposed in chapter 10. We present the implementation of the contribution through a MDE environnement (EMF/Ecore for implementing the meta-models and ATL to realize the transformation between them) in chapter 11.

Part IV

Conclusion and Perspectives

Chapter 12

Conclusion

12.1 Summary of achievements

We started this dissertation with a main goal in front of our eyes: the integration of the two approaches of component-based systems and MultiAgent ones by service-oriented approaches, since there is a nice complement between the advantages of one approach and the shortages of the other. The service-oriented approach represents a pivot of interoperability between the component and agent approaches. Service and the interaction concepts are key ones to reach our objective.

The response to our goal is proposed in CASOM model in chapter 8. CASOM allows us to use interacting agents and components via services in one application specification although one can think these two approaches are completely separated. These interacted components or agents gain some interests through CASOM, where an agent can be part of a composite structure and its services can be reusable and a component can interact with other parties using advanced kinds of interaction. We can say then that the figure 1.3 is implemented.

As we have seen, this objective imposes several other issues, which we summarize in the following achievements in this research:

- We studied works that consider each domain separately with a focus on service and interaction concepts in chapters 2, 3, 4. After studding these models, we make a comparison between their main concepts (component models in table 2.3 on page 35, agent models in table 3.4 on page 66 and service models in table 4.2 on page 80). From this tables, we can identify clearly the lack of high-level interactions in the service and component models, else few models that contain connector and component connector concepts [CBJ02]. We also identified the shortages of agent models considering reusability and composition purposes. It is also clear that the interests of one approach (components vs. agents) overcome the shortages of the other.

- We looked for the works that are already interested in mixing these domains in chapter 5. We found many works that concern considering pairs of approaches jointly, like service and component approaches in the Service Component Architecture [CA07], service and agent approaches like [HJR10] and component and agent ones together [KMW03]. However, few works consider the three of them together like [BP11]. Most of the studied approaches that mix two domains use one domain to enhance the other or merge them in a new entity like ActiveComponent in [BP11]. However, we consider the two domains at the same level and with equity. Then we aim to provide wide choices to a designer to use components and agents with their classical interests or with the new gained ones.

- We defined a kind of framework that groups all the domains that we consider in a single view. This framework represents the gateway of our contributions. It consists of two levels (see figure 6.1 on page 106), the abstract level that contains the models that represent the four domains (i.e., the component model represents the studied component models. The agent model represents the studied agent models. The service model represents the studied service models and Service Component Agent Oriented Model represents the models that mix the three previous ones). The second level is the concrete one and it contains the application examples in specific models. This framework is presented in chapter 6 and it defines a kind of hierarchy between the four representing models, where the service model belongs to a higher level of abstraction. It also defines the relations between these models.

- We defined general models for component and agent domains where the two key concepts of service and interaction are explicit and central. We defined also an abstract service model that does not consider service provides and consumers but only interacting service; these providers (resp. consumers) will be specified in a lower level of conceptions. These general service, component and agent models are defined in chapter 7. This chapter provides also a unified view of the three models in order to present their shared concepts with their relations. the concepts in the four models were considered only from a structural view.

- The analysis of the unified view was the basis to design our CASOM model in chapter 8 after choosing representing concepts for the ones with almost the same significations in the three models.

- The relations between the four models of service, component, agent and CASOM are presented in chapter 9. These relations are the transformations between service model and the other three models of component, agent and CASOM are named *Projection / Abstraction*. Direct transformations between component and agent models or indirect towards CASOM are named *Agentification / Componentification*. They are applied to our own service, component and agent models. However, these models are relatively general because we defined main conceptual principles in each field. Our study of mappings between these three fields can also be considered as a “theoretical” and general one, and not just dedicated to our models. It is possible, through an adequate adaptation, to easily reuse the principles of mappings we established for specific models of service, component or agent.

- A core of a design guide is proposed in chapter 10. It presents the cases that need the intervention of a designer through the design of applications in CASOM model or in using variants of mappings through the transformations between models instead of the direct provided mappings. This design guide is open, where a designer may change or define new variants.
- The implementation of all the parts of our contribution is presented in chapter 11. We used the Ecore Modeling Framework to implement the conceptual models by Ecore meta-models and the transformations between models by ATL rules.

12.2 Perspectives

We have several general perspectives and implementation related ones for this research.

12.2.1 General perspectives

We list below the envisaged improvements for some parts of our contribution:

- **General models:** our provided general models of component, agent, service and CASOM are well-defined and consistent since they are extracted from representative models of each domain. But these domains are still evolving and new models appear. Morevover, in our study, we focused on the aspect of services and interactions that could lead us to ignore some concepts that we considered secondary. In this context, we may need other ways to specify certain things; therefore, we plan to have variants of models to handle these cases. For example, the following cases could be adressed:
    1. We will consider behavioural aspects in the models variants since the actual models consider only the structural aspects. For example detailing the behavioural specification related to the concept of protocol.
    2. We may have groups within groups in a new variants of the agent model since it is not possible currently. This should simplify the mappings between agent model and service or components ones.
    3. We may integrate the resource concept in a new variants of the agent model, this will allow us to take into account the environment dimension of MAS and it implies considering coordination between agents to share these resources.
    4. We may add the capability concept to the component in a variant of CASOM, and this implies to study how to define these capabilities.
- **Design guide:** our design guide is a very basic one. We need to make it richer in its transformation variants and special cases of application specification via CASOM. We must provide our framework (implementation) to be used by several designers using different application examples. This variety of examples helps to see critical cases and hidden problems, which are not discovered yet.

- **Validation:**In the short term, we plan to experiment the implementation of CASOM model: starting from a specification that conforms to CASOM and implementing it on concrete platforms according to specific models of agents and components. We already started a work in that direction on the travel agency example. The goal is to make interacting Jade agents and Fractal components via web services.

  In the longer term, the overall validation of this work requires an empirical experimentation on several examples to verify models transformations. The first step is to create a repository specification from the examples. Ideally, each example should be specified using the four models. However, depending on the nature of the example, it can be assumed that some models do not have interest and that the specification can be limited to an origin model and a target one. Then, the second step consists in applying the transformations and chains of transformations on repository items and check if the resulting specification belongs to the repository, or, if some transformations lost elements, that the resulting specification contains the expected conserved elements. These verifications can be processed using model transformation contracts [CBBD09].

  Beyond empirical validation of our approach, these experiments will help in enriching the designer guide.

12.2.2 Implementation perspectives

- **A framework supporting the management of a designer choices:** as we saw previously in the design guide chapter, the manual interventions of a designer are required. The designer choices and interventions are required through the application specification in CASOM or through the transformations to determine which variant to use instead of the direct proposed mappings. Then, the implementation of a transformation environment that includes the management of the designer choices and interventions is the first on our implementation perspectives list. For example, a designer should be able, via CASOM, to transform a component (and only one among all other components of an application) to an agent via the selection of this component by the designer.

- **Towards a transformation engine between the four models:** the same framework (from the previous point) must define a plug-in that groups the four models of service, component, agent and CASOM with their *Projection / Abstraction*, *Agentification* and *Componentification* transformations. This plug-in is a transformation engine with a friendly interface. This interface allows us to choose the source and target models and to load the source model conforming to the chosen source model (a file containing all the instances of the model concepts) though this interface. The result of this transformation is a generated target model conforming to the chosen target model. This engine will be named CAST that stands for Component Agent Service Transformation Engine. For example, we load a file, that is actually an application example conforming to our component model and the desired output is an application that contains interoperable agents and components (conforming to CASOM model). Figure 12.1 presents a general view of CAST transformation engine.

- **A graphical representation of the applications:** another perspective related to the implementation is to use the Graphical Modeling Framework (GMF) of EMF to view and modify the application examples visually. We need then to assign graphical shapes to the concepts of component, agent service and CASOM models (i.e., the same shapes which are presented in the legend of the application examples in figures 7.2 on page 120, 7.5 on page 129, 7.7 on page 135 and 8.3 on page 150 can be assigned to the concepts of their models). This will facilitate the complexity in finding relations between the instances in the XMI file, then to facilitate the reading of the application examples.

In our preliminary study, we do realize the difficulty of considering the three domains jointly, then some choices were taken like the restriction of the number of studied models in each domain. Nevertheless, we have proposed a relevant and consistent contribution to the area considering the integration of the three domains of service, component and agent simultaneously. But we are aware that there is still work to be done on the implementation and validation of the approach that should be adressed in future work.

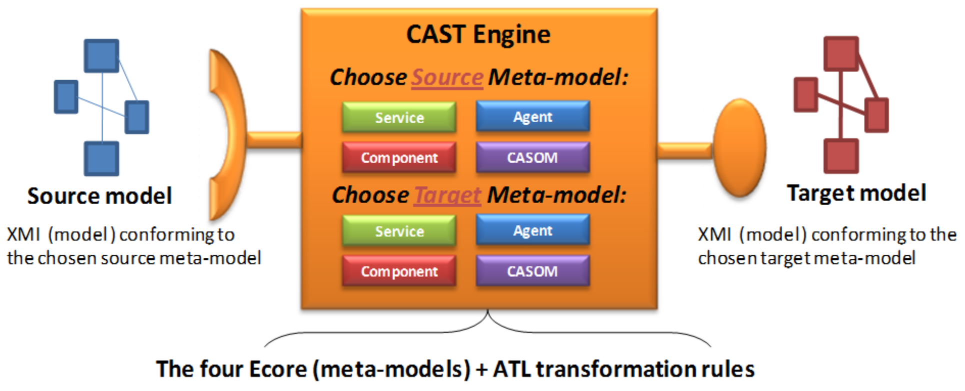

Source model
XMI (model) conforming to the chosen source meta-model
CAST Engine
Choose Source Meta-model:
Service
Agent
Component
CASOM
Choose Target Meta-model:
Service
Agent
Component
CASOM
Target model
XMI (model) conforming to the chosen target meta-model
The four Ecore (meta-models) + ATL transformation rules

Figure 12.1: A general view of CAST engine.

Bibliography

[ABB$^+$02] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jurgen Wust, and Jorg Zettel. *Component-based product line engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[ABJ09] Estefania Argente, Vicente J. Botti, and Vicente Julián. GORMAS: An Organizational-Oriented Methodological Guideline for Open MAS. In Marie Pierre Gleizes and Jorge J. Gómez-Sanz, editors, *AOSE*, volume 6038 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2009.

[ACG11] Nour Alhouda Aboud, Eric Cariou, and Eric Gouardères. Towards a Component Agent Service Oriented Model. In *5ème Conférence Francophone sur les Architectures Logicielles (CAL 2011)*, 2011.

[ACG12] Nour Alhouda Aboud, Eric Cariou, and Eric Gouardères. Correspondances sémantiques entre des modèles de services, de composants et d'agents. In *6ème Conférence Francophone sur les Architectures Logicielles (CAL 2012)*, 2012.

[ACGA11] Nour Alhouda Aboud, Eric Cariou, Eric Gouardères, and Philippe Aniorté. Service-oriented Integration of Component and Agent Models. In María José Escalona Cuaresma, Boris Shishkov, and José Cordeiro, editors, *ICSOFT (1)*, pages 327–336. SciTePress, 2011.

[ACGA12] Nour Alhouda Aboud, Eric Cariou, Eric Gouardères, and Philippe Aniorté. Semantic Mappings between Service, Component and Agent Models. In *The 15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE-2012)*, June 2012.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[AL08] Philippe Aniorté and Jérôme Lacouture. CompAA : A Self-Adaptable Component Model for Open Systems. In *ECBS*, pages 19–25. IEEE Computer Society, 2008.

[Ami03] Matthieu Amiguet. MOCA : Un modèle componentiel dynamique pour les systèmes multi-agents organisationnels phd thesis, 2003. Université Neuchatel Institut d'Informatique et d'Intelligence Arctificielle, Suisse.

[Aus62] John L. Austin. *How to Do Things with Words*. Harvard University Press, William James Lectures, 1962.

[Bar] Douglas K. Barry. Web Services and Service-Oriented Architectures. `http://www.service-architecture.com/web-services/articles/service-oriented_architecture_soa_definition.html`.

[BCM+07] Amit Bahree, Shawn Cicoria, Dennis Mulder, Nishith Pathak, and Chris Peiris. *Pro WCF: Practical Microsoft SOA Implementation*. Apress, 1st ed. 2007. corr. 3rd printing edition, January 2007.

[BCP05] Carole Bernon, Massimo Cossentino, and Juan Pavón. Agent-oriented software engineering. *Knowledge Eng. Review*, 20(2):99–116, 2005.

[BCS04] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The Fractal Component Model, 2004.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *ASE*, pages 273–280. IEEE Computer Society, 2001.

[BG08] Vicente Botti and Adriana Giret. *ANEMONA: A Multi-agent Methodology for Holonic Manufacturing Systems*. Springer Publishing Company, Incorporated, 2008.

[BGZ06] Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli. On the Use of Agents as Components of Software Systems. In *Methodologies and Software Engineering for Agent Systems*. Springer US, 2006.

[BHTV03] Luciano Baresi, Reiko Heckel, Sebastian Thö"ne, and Dániel Varró. a UML profile for Servie Oriented Architecture. `http://lists.oasis-open.org/archives/semantic-ex/200610/pdf00000.pdf`, 2003.

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-Based performance prediction with the palladio component model. In *Proceedings of the 6th international workshop on Software and performance*, WOSP '07, pages 54–65, New York, NY, USA, 2007. ACM.

[BL07] Hongyu Pei Breivold and Magnus Larsson. Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles. In *EUROMICRO'07: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 13–20, Washington, DC, USA, 2007. IEEE Computer Society.

[BLHS+09] Ghassan Beydoun, Graham Low, Brian Henderson-Sellers, Haralambos Mouratidis, Jorge J. Gomez-Sanz, Juan Pavon, and Cesar Gonzalez-Perez. FAML: A Generic Metamodel for MAS Development. *IEEE Transactions on Software Engineering*, 99(RapidPosts):841–863, 2009.

[Blo03] Jason Bloomberg. The role of the service-oriented architect. http://www.ibm.com/developerworks/rational/library/content/RationalEdge/may03/bloomberg.pdf, 2003.

[BM93] S Bussmann and J Müller. *A Negotiation Framework for Cooperating Agents*, pages 1–17. Dake Centre, University of Keele, 1993.

[BMO01] Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.

[BMP06] Jean-Pierre Briot, Thomas Meurisse, and Frédéric Peschanski. Architectural Design of ComponentBased Agents: A Behavior Based Approach. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *PROMAS*, volume 4411 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2006.

[BN84] Andrew Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[BN95] Ron Ben-Natan. *CORBA - a guide to common object request broker architecture.* J. Ranade Workstation series. McGraw-Hill, 1995.

[Boo95] Grady Booch. *Object-oriented analysis and design with applications (2.ed.).* Benjamin/Cummings series in object-oriented software engineering. Addison-Wesley, 1995.

[BP11] Lars Braubach and Alexander Pokahr. Addressing Challenges of Distributed Systems using Active Components. In *In Proceedings of 5th International Symposium on Intelligent Distributed Computing (IDC-2011)*, 2011.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8:203–236, May 2004.

[Bra87] Michael E. Bratman. *Intention, Plans, and Practical Reason.* Harvard University Press, November 1987.

[CA07] David Chappell and Associates. Introducing SCA OASIS SCA Tutorial Part 1. http://www.davidchappell.com/articles/Introducing_SCA.pdf, 2007.

[Car03] Eric Cariou. *Contribution à un Processus de Réification d'Abstractions de Communication.* PhD thesis, Université de Rennes 1, école doctorale Matisse, 2003.

[CBBD09] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. OCL Contracts for the Verification of Model Transformations. In *the Workshop The Pragmatics of OCL and Other Textual Specification Languages at MoDELS 2009*, volume 24. Electronic Communications of the EASST, 2009.

[CBJ02] Eric Cariou, Antoine Beugnard, and Jean-Marc Jézéquel. An Architecture and a Process for Implementing Distributed Collaborations. In *The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2002).* IEEE Computer Society, 2002.

[CBZ$^+$04] Fei Cao, Barrett R. Bryant, Wei Zhao, Carol C. Burt, Rajeev R. Raje, Andrew M. Olson, and Mikhail Auguston. A Meta-Modeling Approach to Web Services. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '04, pages 796–, Washington, DC, USA, 2004. IEEE Computer Society.

[CCSV07] Ivica Crnkovic, Michel Chaudron, Séverine Sentilles, and Aneta Vulgarakis. A Classification Framework for Component Models. In *Proceedings of the 7th Conference on Software Engineering and Practice in Sweden*, October 2007.

[CCZ05] Luca Cernuzzi, Massimo Cossentino, and Franco Zambonelli. Process models for agent-based development. *Journal of Engineering Applications of Artificial Intelligence*, 18:205–222, 2005.

[CD01] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-based Software Systems.* Addison-Wesley, 2001.

[CL00] David J. Carney and Fred Long. What Do You Mean by COTS? Finally, a Useful Answer. *IEEE Software*, 17(2):83–86, 2000.

[CLG$^+$12] J.J. Coyle, C.J. Langley, B. Gibson, R.A. Novack, and E.J. Bardi. *Supply Chain Management: A Logistics Perspective.* Cengage Learning, 2012.

[Cos05] Massimo Cossentino. From Requirements to Code with the PASSI Methodology. In Henderson B. Sellers and P. Giorgini, editors, *Agent-Oriented Methodologies*, volume 3690 of *LNCS*, pages 79–106. Idea Group Pub, June 2005.

[CT01] Bill Councill and George T.Heineman. *Definition of a software component and its elements*, pages 5–19. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Dem95] Yves Demazeau. From interactions to collective behaviour in agent-based systems. In *Proceedings of the First European conference on cognitive science*, pages 117–132, Saint Malo, France, April 1995.

[DLCO09] Mauro Dragone, David Lillis, Rem W. Collier, and Gregory M. P. O'Hare. SoSAA: a framework for integrating components & agents. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 722–728. ACM, 2009.

[DM92] William H. Davidow and Michael S. Malone. *The virtual corporation : structuring and revitalizing the corporation for the 21st century*. HarperCollinsPublishers, 1st ed. edition, 1992.

[DS83] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, 1983.

[DYC$^+$05] Jing Dong, Sheng Yang, Lawrence Chung, Paulo Alencar, and Donald Cowan. A COTS architectural component specification stencil for selection and reasoning. *SIGSOFT Softw. Eng. Notes*, 30(4):1–4, May 2005.

[EDO04] UML Profile for Enterprise Distributed Object Computing (EDOC). http://www.omg.org/spec/EDOC/1.0/PDF/, 2004. Version 1.0.

[EKLM07] Christian Emig, Karsten Krutz, Stefan Link, and Christof Momm. Model-Driven Development of SOA Services. In *University Karlsruhe (TH)*, 2007.

[Fer95] Jacques Ferber. *Les Systèmes Multi-Agents : vers une intelligence collective*. Inter-Editions, 1995.

[FG98] Jacques Ferber and Olivier Gutknecht. A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems. In Yves Demazeau, editor, *ICMAS*, pages 128–135. IEEE Computer Society, 1998.

[FGM04] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From Agents to Organizations: an Organizational View of MultiAgent Systems. In *Agent-Oriented Software Engineering (AOSE) IV*, volume 2935 of *LNCS*. Springer, 2004.

[fip01] FIPA English Auction Interaction Protocol Specification. http://www.fipa.org/specs/fipa00029/SC00029H.html, 2001.

[fip02a] FIPA ACL Message Structure Specification. http://www.fipa.org/specs/fipa00061/SC00061G.html, 2002.

[fip02b] FIPA Communicative Act Library Specification. http://www.fipa.org/specs/fipa00029/SC00037J.html, 2002.

[fip02c] FIPA Contract Net Interaction Protocol Specification. http://www.fipa.org/specs/fipa00029/SC00029H.html, 2002.

[FMBB04] Jacques Ferber, Fabien Michel, and José-Antonio Báez-Barranco. AGRE: Integrating Environments with Organizations. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *E4MAS*, volume 3374 of *Lecture Notes in Computer Science*, pages 48–56. Springer, 2004.

[GBV06] Guillaume Grondin, Noury Bouraqadi, and Laurent Vercouter. Assemblage Automatique de Composants pour la Construction d'Agents avec MaDcAr. In *Journées Multi-Agent et Composant (JMAC 2006)*, pages 39–48, Nîmes, France, 21-21 mars 2006. École des Mines d'Alès.

[GF01] Olivier Gutknecht and Jacques Ferber. The MADKIT Agent Platform Architecture. In *Revised Papers from the International Workshop on Infrastructure for Multi-Agent Systems: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, pages 48–55, London, UK, 2001. Springer-Verlag.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall, 1991.

[GMW97] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1997.

[Gro08] INRIA ATLAS LINA Group. ATLAS transformation language. http://www.eclipse.org/m2m/atl/, April 2008.

[GSP02] Jorge J. Gómez-Sanz and Juan Pavón. Meta-modelling in Agent Oriented Software Engineering. In Francisco J. Garijo, José Cristóbal Riquelme Santos, and Miguel Toro, editors, *IBERAMIA*, volume 2527 of *Lecture Notes in Computer Science*, pages 606–615. Springer, 2002.

[Har07] Chris Harding. An Open Marketplace for Services. *DM Review*, 17(9):20–39, 2007.

[HB02] Hugo Haas and Allen Brown. Web Services Glossary, 2002. http://www.w3.org/TR/ws-gloss/#service.

[HBSS00] Mahdi Hannoun, Olivier Boissier, Jaime Simão Sichman, and Claudette Sayettat. MOISE: An Organizational Model for Multi-agent Systems. In Maria Carolina Monard and Jaime Simão Sichman, editors, *IBERAMIA-SBIA*, volume 1952 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 2000.

[Hew77] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artif. Intell.*, 8(3):323–364, 1977.

[HJR10] Christian Hahn, Sven Jacobi, and David Raber. Enhancing the interoperability between multiagent systems and service-oriented architectures through a model-driven approach. In *Proceedings of the 8th German conference on Multiagent system technologies*, MATES'10, pages 88–99, Berlin, Heidelberg, 2010. Springer-Verlag.

[HkO11] Anthony A. Hock-koon and Mourad Oussalah. The Product-Process-Quality Framework. In *EUROMICRO-SEAA*, pages 20–27. IEEE, 2011.

[HLD04] Colombe Herault, Sylvain Lecomte, and Thierry Delot. New Technical Services Using the Component Model for Applications in Heterogeneous Environment. In Thomas Böhme, Victor Larios-Rosillo, Helena Unger, and Herwig Unger, editors, *IICS*, volume 3473 of *Lecture Notes in Computer Science*, pages 99–110. Springer, 2004.

[HMMF$^+$06] Christian Hahn, Cristián Madrigal-Mora, Klaus Fischer, Brian Elvesæter, Arne-Jørgen Berre, and Ingo Zinnikus. Meta-models, Models, and Model Transformations: Towards Interoperable Agents. In Klaus Fischer, Ingo J. Timm, Elisabeth André, and Ning Zhong, editors, *MATES*, volume 4196 of *Lecture Notes in Computer Science*, pages 123–134. Springer, 2006.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[HR85] Barbara Hayes-Roth. A blackboard architecture for control. *Artif. Intell.*, 26(3):251–321, 1985.

[HRHL01] Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. Intelligent Agents - Summary of an Agent Infrastructure. In *In 5th International conference on autonomous agents*, 2001.

[HS94] M. Huhns and M.p. Singh. Ckbs-94 tutorial: Distributed artificial intelligence for information systems. In *Second International Conference on Cooperating Knowledge Based Systems*, Keele, UK, 1994.

[HSB02a] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. In *SBIA'02*, pages 118–128, 2002.

[HSB02b] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. MOISE+: towards a structural, functional, and deontic model for MAS organization. In *AAMAS*, pages 501–502. ACM, 2002.

[HSB04] Chihab Hanachi and Christophe Sibertin-Blanc. Protocol Moderators as Active Middle-Agents in Multi-Agent Systems. *Autonomous Agents and Multi-Agent Systems*, 8(2):131–164, 2004.

[ITS+08] Ioannis Ignatiadis, Dimitrios Tektonidis, Adomas Svirskas, Jonathan Briggs, Stamatia-Ann Katriou, and Adamantios Koumpis. A service oriented and agent-based architecture for the e-collaboration of smes. In Makoto Oya, Ryuya Uda, and Chizuko Yasunobu, editors, *II3E*, volume 286 of *IFIP Advances in Information and Communication Technology*, pages 125–137. Springer, 2008.

[JC05] Clement Jonquet and Stefano A. Cerri. The STROBE model: Dynamic Service Generation on the Grid. *Applied Artificial Intelligence, Special issue on Learning Grid Services*, 19(9-10):967–1013, 2005.

[JCJÖ92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-oriented software engineering - a use case driven approach.* Addison-Wesley, 1992.

[Joh05] Simon Johnston. UML 2.0 Profile for Software Services. http://www-128.ibm.com/developerworks/rational/library/05/419_soa, 2005.

[Jon05] Steve Jones. Toward an acceptable definition of service. *IEEE Software*, 22(3):87–93, 2005.

[JSW98] Nicholas Jennings, Katia Sycara, and Michael Wooldridge. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7 – 38, July 1998.

[Ken02] Stuart Kent. Model Driven Engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag.

[KG91] David N. Kinny and Michael P. Georgeff. Commitment and effectiveness of situated agents. In *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 1*, pages 82–88, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[KM85] Jeff Kramer and Jeff Magee. Dynamic Configuration for Distributed Systems. *IEEE Trans. Softw. Eng.*, 11:424–436, April 1985.

[KMW03] Richard Krutisch, Philipp Meier, and Martin Wirsing. The Agent Component Approach, Combining Agents, and Components. In Michael Schillo, Matthias Klusch, Jörg P. Müller, and Huaglory Tianfield, editors, *MATES*, volume 2831 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2003.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[LF97] Yannis Labrou and Tim Finin. A Proposal for a new KQML Specification. http://www.csee.umbc.edu/csee/research/kqml/papers/kqml97.pdf, 1997.

[LGA06] Jérôme Lacouture, Guy Gouarderes, and Philippe Aniorté. Vers l'adaptation dynamique de services : des composants monitorés par des agents. In *2ème journée Multi-Agent et Composant*, pages 5–15, 2006.

[Lin01] Juergen Lind. Relating Agent Technology and Component Models, 2001. http://www.agentlab.de/documents/Lind2001e.pdf.

[LKA^{+}95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Trans. Softw. Eng.*, 21(4):336–355, 1995.

[LMSW05] Michael Luck, Peter McBurney, Onn Shehory, and Steve Willmott. *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*. AgentLink, 2005.

[Lud03] Jochen Ludewig. Models in software engineering - an introduction. *Software and Systems Modeling*, 2(1):5–14, March 2003.

[LW07] Kung-Kiu Lau and Zheng Wang. Software Component Models. *IEEE Trans. Software Eng.*, 33(10):709–724, 2007.

[McI68] Douglas McIlroy. Mass-Produced Software Components. In J. M. Buxton, Peter Naur, and Brian Randell, editors, *Software Engineering Concepts and Techniques*, pages 88–98. NATO Science Committee, oct 1968.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[Mey03] Bertrand Meyer. The grand challenge of Trusted Components. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 660–667, Washington, DC, USA, 2003. IEEE Computer Society.

[MF07] Saber Mansour and Jacques Ferber. Un modèle organisationnel pour les systèmes multi-agents ouverts déployés à grande échelle (présentation courte). In Valérie Camps and Philippe Mathieu, editors, *JFSMA*, pages 107–116. Cepadues Editions, 2007.

[MFBC10] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoit Combemale. Modeling Modeling Modeling. *Journal of Software and Systems Modeling (SoSyM)*, 2010.

[MR09] Jim Marino and Michael Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.

[MRT99] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE'99: Proceedings of the 21st international conference on Software engineering*, pages 44–53, New York, NY, USA, 1999. ACM.

[MS05] Ed Merks and Dave Steinberg. Models to Code with Eclipse Modeling Framework. `http://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial11final.pdf`, 2005.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26:70–93, January 2000.

[MT07] Lisa Jean Moya and Andreas Tolk. Towards a taxonomy of agents and multi-agent systems. In *Proceedings of the 2007 spring simulation multi-conference - Volume 2*, SpringSim '07, pages 11–18, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[NLJ96] H. S. Nwana, L. C. Lee, and Nicholas R. Jennings. Co-ordination in Software Agent Systems. *The British Telecom Technical Journal*, 14(4):79–88, 1996.

[Nwa96] Hyacinth S. Nwana. Software Agents: An Overview. *Knowledge Engineering Review*, 11(3):205–244, 1996.

[ODS09] Walamitien H. Oyenan, Scott A. DeLoach, and Gurdip Singh. A service-oriented approach for integrating multiagent system designs. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 1363–1364, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[OLKM00] Robvan Ommering, Frankvander Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33:78–85, March 2000.

[OMG00] OMG. *Meta Object Facility (MOF) Specification*. Object Modeling Group, 2000.

[OMG03] OMG. MDA Guide Version 1.0.1. `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`, June 2003.

[OMG05] OMG Unified Modeling Language (OMG UML), Specification. `http://www.omg.org/cgi-bin/doc?formal/05-04-01/05-04-01.pdf`, 2005. 1.4.2/.

[OMG09] OMG. Service oriented architecture Modeling Language (SoaML). http://www.omg.org/spec/SoaML/1.0/Beta1/PDF/09-04-01.pdf, 2009.

[OMG10] OMG. Object Constraint Language (OCL) Specification, version 2.2, 2010. http://www.omg.org/spec/OCL/2.2/.

[ONL05] James Odell, Marian Nodine, and Renato Levy. A metamodel for agents, roles, and groups. In *Proceedings of the 5th international conference on Agent-Oriented Software Engineering*, AOSE'04, pages 78–92, Berlin, Heidelberg, 2005. Springer-Verlag.

[PAG04] Mike Papazoglou, Marco Aiello, and Paolo Giorgini. Service-Oriented Computing and Software Agents. In Lawrence Cavedon, Zakaria Maamar, David Martin, Boualem Benatallah, and Gerhard Weiss, editors, *Extending Web Services Technologies*, volume 13 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 29–52. Springer US, 2004.

[PBBP01] Chris Preist, Andrew Byde, Claudio Bartolini, and Giacomo Piccinelli. Towards agent-based service composition through negotiation in multiple auctions. *AISB Journal*, 1(1), 2001.

[PBJ98] F. Plásil, D. Bálek, and R. Janecek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, Washington, DC, USA, 1998. IEEE Computer Society.

[PCW98] Martin Purvis, Stephen Cranefield, and Roy Ward. Distributed Software Systems: From Objects to Agents, Software Engineering: Education. In *Practice, Proceedings of the 1998 International Conference, IEEE Computer Society Press, Los Alamitos, CA*, pages 158–165, 1998.

[PH07] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *VLDB J.*, 16(3):389–415, 2007.

[PLR$^+$99] Nikos Prekas, Pericles Loucopoulos, Colette Rolland, Georges Grosz, Farida Semmak, and Danny Brash. Developing Patterns as a Mechanism for Assisting the Management of Knowledge in the Context of Conducting Organisational Change. In Trevor J. M. Bench-Capon, Giovanni Soda, and A. Min Tjoa, editors, *DEXA*, volume 1677 of *Lecture Notes in Computer Science*, pages 110–122. Springer, 1999.

[Pra86] Vaughan Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[PTDL08] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: a Research Roadmap. *Int. J. Cooperative Inf. Syst.*, 17(2):223–255, 2008.

[RBP$^+$91] James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, Frederick Eddy, and William E. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[RG95] Anand S. Rao and Michael P. Georgeff. BDI Agents: From Theory to Practice. In Victor R. Lesser and Les Gasser, editors, *ICMAS*, pages 312–319. The MIT Press, 1995.

[RL90] Philip R.Cohen and Hector J. Levesque. Intention is choice with commitment. *Artif. Intell.*, 42:213–261, March 1990.

[RNC$^+$96] Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[RRSA06] Adel Torkaman Rahmani, Vahid Rafe, Saeed Sedighian, and Amin Abbaspour. An MDA-Based Modeling and Design of Service Oriented Architecture. In Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science (3)*, volume 3993 of *Lecture Notes in Computer Science*, pages 578–585. Springer, 2006.

[SA04a] Silvia N. Schiaffino and Analía Amandi. User - interface agent interaction: personalization issues. *Int. J. Hum.-Comput. Stud.*, 60(1):129–148, 2004.

[SA04b] Frédérick Seyler and Philippe Aniorté. Ugatze: un meta-modèle de composants orienté réutilisation. In *INFORSID*, pages 195–210, 2004.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework.* Addison-Wesley, Boston, MA, 2 edition, 2009.

[SG96] Mary Shaw and David Garland. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, Upper Saddle River, NJ, 1996.

[SH05] Munindar P. Singh and Michael N. Huhns. *Service-oriented computing - semantics, processes, agents.* Wiley, 2005.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60:51–92, March 1993.

[Sol05] Richard Mark Soley. Model driven architecture: Next steps. page 3, 2005.

[Som04] Ian Sommerville. *Software Engineering.* Addison Wesley, 7 edition, 2004.

[Spi01] Diomidis Spinellis. Notable design patterns for domain-specific languages. *J. Syst. Softw.*, 56(1):91–99, February 2001.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[Tow08] Erik Townsend. The 25-Year History of Service Oriented Architecture, 2008. http://www.eriktownsend.com/white-papers-technology.html/The25\discretionary{-}{}{}yearhistoryofSOA.pdf.

[VSDD05] Javier Vázquez-Salceda, Virginia Dignum, and Frank Dignum. Organizing Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, 11:307–360, November 2005.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[WJK00] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3:285–312, September 2000.

[WK98] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, October 1998.

[Woo00] M. Wooldridge. Intelligent Agents. In G. Weiss, editor, *Multiagent Systems A Modern Approach to Distributed Artificial Intelligence*, pages 27–77. MIT Press, 2000.

[Woo09] Michael J. Wooldridge. *An Introduction to MultiAgent Systems (2. ed.)*. Wiley, 2009.

[WSW$^+$02] A. Wigley, M. Sutton, S. Wheelwright, R. Burbidge, and R. Mcloud. *Microsoft .Net Compact Framework: Core Reference*. Microsoft Press, Redmond, WA, USA, 2002.

[WZ02] Michael Winter and Christian Zeidle. The PECOS Software Process. In *ICSR7 Workshop on Component-Based Software Development Processes*, 2002.

[YP04] Jian Yang and Mike P. Papazoglou. Service components for managing the life-cycle of service compositions. *Inf. Syst.*, 29:97–125, April 2004.

[Zhu05] Haibin Zhu. Building reusable components with service-oriented architectures. In Du Zhang, Taghi M. Khoshgoftaar, and Mei-Ling Shyu, editors, *IRI*, pages 96–101. IEEE Systems, Man, and Cybernetics Society, 2005.

[ZJW03] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12:317–370, July 2003.

[ZYCZ08] Tao Zhang, Shi Ying, Sheng Cao, and Jiankeng Zhang. A modelling approach to service-oriented architecture. *Enterp. Inf. Syst.*, 2:239–257, August 2008.

Résumé

Les travaux présentés dans cette thèse concernent des problématiques d'architecture logicielle multi-domaines pour le développement d'applications distribuées. Ces applications sont caractérisées aujourd'hui comme des systèmes ouverts, complexes, hétérogènes et à large échelle. Les approches traditionnelles, telles que l'approche orienté objet, n'offrent plus un paradigme de conception suffisant pour appréhender la complexité de tels systèmes. Ces nouvelles tendances ont conduit à l'émergence d'approches de plus haut niveau telles que les approches orientées services, composants ou agents. Chacune de ces approches offrent des intérêts et des caractéristiques propres dans le développement d'applications distribuées. Les services offrent une abstraction et une interopérabilité à large échelle. Abstraction dans le sens où un service permet de spécifier un élément fonctionnel sans préciser comment cet élément est implémenté. Les composants sont une approche robuste basée sur la composition et la réutilisation d'éléments clairement définis par leurs interfaces. Les agents sont eux des éléments présentant un comportement dynamique dirigé par un but et des interactions de haut niveau avec les autres agents formant l'application, vue comme une organisation de services collaboratifs.

D'un point de vue conceptuel, le service peut donc être perçu comme le modèle « métier » de l'application, alors que les composants et les agents constituent un modèle d'implémentation. L'étude de ces différents domaines et des modèles associés, a montré que les approches composants et agents sont complémentaires, les points forts d'une approche représentant les faiblesses de l'autre. Face à ce constat, il nous est paru intéressant d'intégrer ces deux approches, au sein d'une même démarche de conception. Cela permet, d'une part, qu'une approche puisse bénéficier des intérêts de l'autre et d'autre part, d'utiliser conjointement des agents et des composants dans la conception d'une même application. La démarche que nous avons adoptée consiste à considérer les services comme pivot d'interaction afin de rendre possible l'interopérabilité des agents et des composants.

Pour supporter cette démarche, nous avons défini un processus de conception basé sur l'Ingénierie Des Modèles qui contient quatre modèles conceptuels (Domain Specific language) dont l'intérêt est de mettre l'accent sur les concepts de services et d'interaction. Nous avons ainsi défini un modèle de services, un modèle de composants et un modèle d'agents. Enfin, un modèle mixte appelé CASOM, Component Agent Service Oriented Model, permet de spécifier une application via une combinaison des trois domaines précédents. Ensuite, des règles de correspondances ont été définies entre les quatre modèles pour pouvoir par exemple transformer une spécification agents en une spécification composants ou mixte. L'implémentation de ces transformations a été réalisée en langage ATL (ATLAS Transformation Language).

Mots-clés: architecture logicielle multi-domaines, approche a base de composants, sys-

tèmes multiagents organisationnels, Architecture Orientée Service.

Abstract

The presented work considers problems related to multi-domain software architecture for the development of distributed applications. These applications are large-scaled, heterogeneous, open and complex software systems. Traditional approaches such as object-oriented are no longer sufficient to represent such complex systems. These trends lead to the emergence of higher-level approaches such as service-oriented, components or agents. Each one of these approaches offers interests and characteristics in the development of distributed applications. Services provide an abstraction and interoperability in a large scale. Abstraction is in the sense that a service can specify a functional element without specifying how this element is implemented. The components are a robust approach based on composition and reusability through their clearly defined interfaces. Agents are elements which are characterized by dynamic goal directed behaviours and high-level interactions with other agents forming the application, seen as an organization for collaborative services.

From a conceptual point of view, the service can be seen as the "business" model of an application, while components and agents are the implementation models. The study of these different domains, with their related models, showed that the components and agents approaches are complementary; the strengths of one approach overcome the weaknesses of the other. Therefore, we are interested in the integration of these two approaches in a single design approach. This allows an approach to benefit from the interests of the other, on one hand and the use of agents and components jointly in the design of an application on the other hand. To reach our objective, we consider services as pivot of interaction between agents and components.

The result of our analysis leads us to develop a design process based on Model-Driven Engineering which contains four conceptual models (Domain Specific Languages) with the main interest of focusing on the concepts of services and interaction. We then defined a service, component and agent models. Finally, a hybrid model called CASOM, Component Agent Service Oriented Model, was proposed that allows application specification via a combination of the three domains. Then, mapping rules have been defined between the four models in order to transform agents specification into components specification or mixed. The implementation of these transformations was done in ATL language (ATLAS Transformation Language).

Keywords: multi-domain software architecture, Component-based approach, Organizational MultiAgent systems, Service Oriented Architecture.