UNIVERSITAT AUTÒNOMA DE BARCELONA
FACULTAT DE CIÈNCIES

TESI DOCTORAL

Open, Reusable, and Configurable Multi Agent Systems: A Knowledge Modelling Approach

Mario Gómez Martínez

Director: Enric Plaza i Cervera

Bellaterra, Maig 2004

CONSELL SUPERIOR D'INVESTIGACIONS CIENTÍFIQUES
INSTITUT D'INVESTIGACIÓ EN INTEL·LIGÈNCIA ARTIFICIAL

UNIVERSITAT AUTÒNOMA DE BARCELONA
FACULTAT DE CIÈNCIES

Tesi Doctoral

Open, Reusable, and Configurable Multi Agent Systems: A Knowledge Modelling Approach

Bellaterra, Maig 2004

Memoria presentada per Mario Gómez Martínez per optar al títol de Doctor en Informàtica per la Universitat Autònoma de Barcelona sota la direcció del Dr. Enric Plaza i Cervera. El treball presentat en aquesta memòria ha estat realitzat a l'Institut d'Investigació en Intel·ligència Artificial (IIIA), del Consell Superior d'Investigacions Científiques (CSIC).

A mis padres

Qué ligereza encierra una gota de lluvia
Qué delicado el roce del mundo
Cualquier cosa acontecida en cualquier lugar y tiempo
escrita está en el agua de Babel.

Wislawa Szymborska
(*El Agua*)[1]

En el jeroglífico había un ave, pero no se podía saber si volaba o estaba clavada por un eje de luz en el cielo vacío. Durante centenares de años leí inútilmente la escritura. Hacia el fin de mis días, cuando ya nadie podía creer que nada hubiese sido descifrado, comprendí que el ave a su vez me leía sin saber si en el roto jeroglífico la figura volaba o estaba clavada por un eje de luz en el cielo vacío.

Jose Angel Valente
(*De la luminosa opacidad de los signos*)

Lloras?... Entre los álamos de oro,
lejos, la sombra del amor te aguarda.

Antonio Machado.
(*LXXX*)

[1] "Traducció d'Ana María Moix i Jerzy Wojciech Slawomirski"

Contents

List of Figures

Agradecimientos

La realización de esta tesis ha sido posible gracias al soporte brindado por el Consejo Superior de Investigaciones Científicas. La mayor parte del trabajo se ha desarrollado bajo la cobertura del proyecto europeo IBROW (IST-1999-190005), y parcialmente bajo el proyecto español SMASH (TIC96-10ajo 38-C04).

Sin el apoyo y el estímulo constante de Enric, mi director de tesis, ésta no se hubiera hecho realidad. Sus recomendaciones y sugerencias transcendieron la labor académica formal y contribuyeron a mi formación integral como investigador en aspectos tales como el respeto frente al trabajo de los otros y la asertividad a la hora de presentar las contribuciones propias.

Esta tesis se ha nutrido de todo eso y mucho más; agradezco a toda la gente del IIIA su compañerismo, su buena disposición a ayudar, y su impagable buen humor, tanto al personal investigador como al personal administrativo y de servicios. Estoy convencido de que la personalidad de las personas es inseparable del ambiente, y viceversa; en ese sentido puedo decir con total sinceridad que el IIIA es un lugar que puede sacar lo mejor de cada uno.

Especial mención merecen los que han colaborado más de cerca en este parto elefantino: Josep Lluís, por su inestimable ayuda con los aspectos más técnicos, sin su implementación de NOOS y su estupenda plataforma para desarrollo de agentes basada en instituciones electrónicas, esta tesis no sería lo que es; a Marc y a todo el equipo de instituciones electrónicas, sin duda otra de las columnas vertebrales de este trabajo; a Santi, autor de la herramienta de visualización Agent World; y a Chema, compañero de casi todo durante estos últimos años, y autor de buena parte del código utilizado en WIM.

Este es el momento de agradecer las historias de pegasos de mi padre, y el regazo de mi madre, más de 11680 días madre, a ellos va dedicada esta tesis. Llegada también la hora de agradecer a mis hermanos su incondicional amistad, es una gran suerte tenerlos como hermanos.

Y por último, infinitas gracias a la persona que me acompañó en este tramo de mi vida, convirtiendo el temido calvario de la tesis en un luminoso paseo.

Resumen

Aunque los Sistemas Multiagente se suponen abiertos, la mayor parte de la investigación realizada se ha centrado en sistemas cerrados, diseñados por un solo equipo de desarrollo, sobre un entorno homogéneo, y un único dominio.

Esta tesis pretende avanzar hacia la consecución de Sistemas Multiagente abiertos. Nuestros esfuerzos se han centrado en desarrollar un marco de trabajo para Sistemas Multiagente que permita maximizar la reutilización de agentes en diferentes dominios, y soporte la formación de equipos bajo demanda, satisfaciendo los requerimientos de cada problema particular.

Por un lado, este trabajo investiga el uso de Métodos de Solución de Problemas para describir las capacidades de los agentes con el objetivo de mejorar su reutilización. Hemos tenido que adaptar el modelo para trabajar con aspectos específicos de los agentes, como el lenguaje de comunicación y los protocolos de interacción.

Por otro lado, esta tesis propone un nuevo modelo para el Proceso de Solución de Problemas Cooperativo, el cual introduce una fase de configuración previa a la formación de un equipo. El proceso de configuración se encarga de diseñar el equipo en términos de las tareas a resolver, las capacidades a utilizar, y el conocimiento del dominio disponible.

El marco de trabajo desarrollado ha sido puesto a prueba mediante la implementación de una infraestructura para agentes. Esta infraestructura proporciona un nivel de mediación social para los proveedores y clientes del sistema de resolución de problemas, sin imponer una arquitectura particular para los agentes participantes, ni un modelo mental o lógico para explicar la cooperación.

Las contribuciones de este trabajo adoptan la forma de un marco de trabajo multinivel, y son presentadas desde los conceptos más abstractos a los más concretos, para terminar con la implementación de una aplicación particular basada en agentes de información cooperativos.

Abstract

Although Multi Agent Systems are supposed to be *open* systems, most of the initial research has focused on *closed* systems, which are designed by one developer team for one homogeneous environment, and one single domain.

This thesis aims to advance some steps towards the realization of the open Multi Agent Systems vision. Our work has been materialized into a framework for developing Multi Agent Systems that maximize the reuse of agent capabilities across multiple application domains, and support the automatic, on-demand configuration of agent teams according to stated problem requirements.

On the one hand, this work explores the feasibility of the Problem Solving Methods approach to describe agent capabilities in a way that maximizes their reuse. However, since Problem Solving Methods are not designed for agents, we have had to adapt them to deal with agent specific concepts concerning the agent communication languages and interaction protocols.

One the other hand, this thesis proposes a new model of the Cooperative Problem Solving process that introduces a Knowledge Configuration stage previous to the Team Formation stage. The Knowledge Configuration process performs a bottom-up design of a team in term of the tasks to be solved, the capabilities required, and the domain knowledge available.

The statements made herein are endorsed by the implementation of an agent infrastructure that has been tested in practice. This infrastructure has been developed according to the electronic institutions formalism to specifying open agent societies. This infrastructure provides a social mediation layer for both requesters and providers of capabilities, without imposing neither an agent architecture, nor an attitudinal theory of cooperation.

The contributions of our work are presented as a multilayered framework, going from the more abstract aspects, to the more concrete, implementation dependent aspects, concluding with the implementation of the agent infrastructure and a particular application example for cooperative information agents.

Chapter 1

Introduction

The main goal of this thesis is to provide a framework for open Multi-Agent Systems that maximizes the reuse of agent capabilities through multiple application domains, and supports the automatic, on-demand configuration of agent teams according to stated problem requirements.

We have devoted considerable effort to the applicability of our proposals, which resulted in the implementation of an infrastructure to develop Multi Agent Systems according to the principles and requirements stated by our framework.

During the rest of this Chapter the main goal of this thesis is analyzed and boiled down to the several issues and problems it encompasses. First, we situate our work in the field of Multi-Agent Systems, focusing on the open problems and challenges that motivated us; second, the main contributions of this thesis are summarized; and third, the structure of the thesis is presented as a guide for readers.

1.1 Motivation and context

Distributed Artificial Intelligence has historically been divided in two main areas: Distributed Problem Solving (DPS) and Multi-Agent Systems (MAS) [Bond and Gasser, 1988a]. In the DPS approach problems are divided and distributed among a number of nodes that cooperate in solving the different parts of the problem; but the overall problem solving strategy is an integral part of the system. In contrast, MAS research is concerned with the behavior of a collection of possibly pre-existing autonomous agents aiming at solving a given problem [Jennings et al., 1998]. MAS have been defined as loosely coupled networks of problem-solving entities working together to find answers to problems that are beyond the individual capabilities or knowledge of the isolated entities [Durfee and Lesser, 1989]. The MAS approach advocates decomposing problems in terms of autonomous agents that can engage in flexible, high level interactions, and this way of decomposing a problem aids the process of engineering complex systems [Jennings, 2000]. Some characteristics of MAS are the following:

- each agent has incomplete information or capabilities for solving the problem, thus each agent has a limited viewpoint;
- there is no global system control;
- data is decentralized; and
- computation is asynchronous.

Some reasons for the increasing interest in MAS research include: the ability to provide robustness and efficiency, the ability to allow inter-operation of existing legacy systems, and the ability to solve problems in which data, expertise, or control is distributed. Agents are defined as sophisticated computer programs that act autonomously on behalf of their users, across open and distributed environments, to solve a growing number of complex problems.

Considering the former definitions, MAS are supposed to be open systems in that agents can enter / leave at any time. Nonetheless, most of the initial work devoted to MAS research has focused on *closed* systems [Klein, 2000], typically designed by one team for one homogeneous environment, with participating agents sharing common high-level goals in a single domain. The communications languages and interaction protocols are typically in-house protocols, and are defined by the design team prior to any agent interactions. Systems are scalable under controlled conditions and design approaches tend to be ad hoc, inspired by the agent paradigm rather than using any specific methodologies.

It is often suggested the need for real open systems that were capable of dynamically adapting themselves to changing environments. Some examples are electronic markets, communities and distributed search engines. All in all, in open MAS the participants (both human and software agents) are unknown beforehand, can change over time and can be developed by different parties. Open systems are opposite to closed or proprietary systems, i.e. open systems can be supplied by hardware components from multiple vendors, and whose software can be operated from different platforms.

According to the predictions of the European Network of Excellence for Agent Based Computing, fully open MAS spanning multiple application domains and involving heterogeneous participants will not be achieved in a foreseeable future, and not before year 2009 [Luck et al., 2003]. This cautious prediction obeys to some challenges yet to be undertaken, including the following:

- provide effective agreed standards to allow open agent systems;
- provide semantic infrastructure for open agent communities;
- develop reasoning capabilities for agents in open environments;
- develop agent ability to understand user requirements;
- develop agent ability to adapt to changes in the environment;
- ensure agent confidence an trust in agents

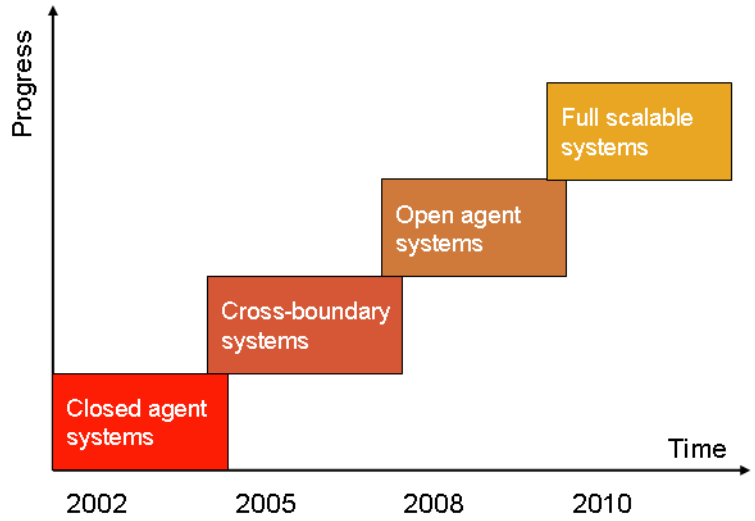

Figure 1.1: Roadmap timeline for agent technologies

Figure 1.1 (adapted from [Luck et al., 2003]) shows a roadmap timeline suggesting how agent technology will progress over time if R & D is aimed at the main challenges identified.

Although we are not going to solve all these problems entirely, we hope to provide tentative solutions to some of them and to bring about some insights that could drive future work on these issues. We are not going to exhaustively describe these problems here, since they are described in Chapter 2; we are rather going to sketch them so as to let the reader become acquainted with the motivations for this work.

Nowadays, MAS are increasingly being designed to cross corporate boundaries, so that the participating agents have fewer goals in common, although their interactions are still concerning a common domain. The languages and protocols used in these systems are being agreed and standardized; however, despite this raising diversity, all participating agents are designed by the same team designing the system and share common domain knowledge.

In order to overcome the limitations of current agent infrastructures for open MAS, researchers must tackle with several problems:

- the connection problem, or how to put providers and requesters (services and customers) in contact;
- the interoperability problem, or how to achieve a meaningful interaction among heterogeneous agents at the syntactic, semantic and pragmatic levels;

- the coalition problem, or how to form and coordinate agent teams to solve problems in a cooperative way;
- the reuse problem, or how to use the same agent capabilities across several application domains;
- the accountability problem, or how to predict or explain the behavior of MAS according to the requirements of the problem.

The MAS community builds intelligent agents capable of reasoning about how to cooperate to solve complex problems. This area uses knowledge intensively: for adding meaning (using ontologies), enabling service discovery and composition (using annotations and reasoning for matchmaking), and coordinating processes (using negotiation strategies). In closed environments knowledge is usually homogeneous and static. In open environments such as the Internet, knowledge is pervasive, distributed, heterogeneous, and dynamic in nature.

Nowadays the Web is shifting the nature of software development to a distributed *plug-and-play* process. This change requires a new way of managing and integrating software based on a software integration architectural pattern called *middleware*. Middleware is connectivity software; it consists of enabling services that allow multiple processes running on one or more machines to interact across a network. It follows that a middleware layer is required to provide a common set of programming interfaces that developers can use to create distributed systems.

Intelligent middleware aims to achieve the highest degree of interoperability, where systems can identify and react to the semantics of data. For this reason, many research communities are focusing their attention to semantic interoperability, for example: MAS, Semantic Web Services, Cooperative Information Systems and Component Based Software Development.

In open MAS the middleware layer is usually provided by *middle agents* [Decker et al., 1997b] that mediate between requesters and providers of capabilities, e.g. matchmakers [Decker et al., 1996], facilitators [Erickson, 1996a, Genesereth and Ketchpel, 1997] and brokers[Nodine et al., 1999]). Typically, the function of a middle agent is to pair requesters with providers that are suitable for them, and this process is called *matchmaking*. To enable matchmaking, both providers and requesters share a common language to describe the requests (tasks or goals) and the advertisements (capabilities or services) in order to compare them. This language is called an Agent Capability Description Language (ACDL).

Matchmaking is the process of verifying whether a capability specification matches the specification of a request (e.g. a task to be solved): two specifications match if their specifications verify some *matching* relation, where the matching relation is defined according to some criteria (e.g. a capability being able to solve a task). Semantic matchtmaking, which is based on the use of shared ontologies to annotate agent capabilities [Guarino, 1997a], improves the matchmaking process and facilitates interoperation.

Semantic matchmaking allows to verify whether a capability can solve a new type of problem (a task), but the reuse of existing capabilities over new

application domains is difficult because capabilities are usually associated to a specific application domain.

The notion of an *Agent Capability Description Language* (ACDL) has been introduced recently [Sycara et al., 1999a] as a key element to enable MAS interoperation in open environments. An ACDL is a shared language that allows heterogeneous agents to coordinate effectively across distributed networks. Sometimes, capabilities are referred as "services" and, consequently, an ACDL can alternatively be called an Agent Service Description Language (ASDL).

In the literature, an ACDL is defined as a language to describe both agent advertisements and requests, and is primarily used by middle agents (e.g. brokers and matchmakers) to pair service-requests with service-providing agents that meet the requirements of the request [Sycara et al., 1999b, Sycara et al., 1999a].

Some desirable features for such a language are *expressiveness*, *efficiency* and *ease of use*:

- *Expressiveness*: the language should be expressive enough to represent not only data and knowledge, but also the meaning of a capability. Agent capabilities should be described at an abstract rather than implementation dependent level.
- *Efficiency*: inferences on descriptions written in this language should be supported. Automatic reasoning and comparison on the descriptions should be both feasible and efficient.
- *Ease of use*: descriptions should not only be easy to read and understand, but also easy to write. The language should support the use of ontologies for annotate agent capabilities with shared semantic information.

However, in addition to capability discovery, an ACDL should bring support to other activities involved in MAS interoperation. On the one hand, once a capability is discovered, it should be enacted automatically; agents should be able to interpret the description of a capability to understand what input is necessary to execute a capability, what information will be returned, and which are the effects or postconditions that will hold after applying the capability. In addition, an agent must know the communication protocol, the communication language and the data format required by the provider of the capability in order to successfully communicate with it.

On the other hand, in order to achieve more complex tasks, capabilities may be combined or aggregated to achieve complex goals that existing capabilities cannot achieve in isolation. This process may require a combination of matchmaking, capability selection among alternative candidates, and verification of whether the aggregated functionality satisfies the specification of a high-level goal.

Our approach to these activities is tightly related with the idea of reuse: how to reuse a capability for different tasks, across several application domains, and in cooperation with other capabilities provided by different, probably heterogeneous agents. The idea of reuse is being addressed by the Software Engineering and the Knowledge Engineering communities.

The reuse of complete software developments and the process used to create them has the potential to significantly ease the process of software engineering by providing a source of verified software artifacts [Wegner, 1984]. It is suggested than reuse of software artifacts can be achieved through the utilization of software libraries [Atkinson, 1997]. Essentially a software library is a repository of information which can be used to construct software systems. The main goal of software libraries reuse is to enable previous development experiences to guide subsequent software development. To this end, MAS designers must be provided with libraries of:

- generic organisation models (e.g., hierarchical organisations, flat organisations);
- generic agent models (e.g., purely reactive agent models, deliberative BDI models);
- generic task models (e.g., diagnostic tasks, information filtering tasks, transactions);
- communication languages and patterns for agent societies;
- ontology patterns for agent requirements, agent models and organisation models;
- interaction protocol patterns between agents with special roles;
- reusable organisation structures; and
- reusable knowledge bases.

From the compositional approach, building a software system is essentially a design problem [Biggerstaff and Perlis, 1989]. The Component-Based Software development (CBSD) approach focuses on building large software systems by integrating previously-existing software components. By enhancing the flexibility and maintainability of systems, the ultimate goal is to reduce software development costs, assemble systems rapidly, and reduce the maintenance burden associated with the support and upgrade of large systems [Brown and Wallnau, 1996].

Constructing an application involves the use of prefabricated pieces, perhaps developed at different times, by different people and possibly with different purposes, therefore integrability of heterogeneous components is a key when considering whether to acquire, reuse, or build new components. Reusable software components can be deployed independently and are subject to composition by third parties [Szyperski, 1996]. There is, however, a major problem with software composition, the so called *Bottom Up Design Problem* [Mili et al., 1995], defined as:

> given a set of requirements, find a set of components within a software library whose combined behavior satisfies the requirements.

The fundamental difficulty when considering this problem is how to decompose the requirements in such a way as to yield component specifications. A reverse approach is to search the space of all possible component compositions until one satisfying the requirements is found [Hall, 1993, Zhang, 2000]. Thus, composition of components can be regarded as composition of their specifications [Butler and Duke, 1998].

Concerning Knowledge Engineering, we are interested in Knowledge Modelling Frameworks that has proposed several methodologies, architectures and languages for analyzing, describing and developing knowledge systems [Steels, 1990, McDermott, 1988, Schreiber et al., 1994a, Fensel et al., 1999]. The goal of a Knowledge Modelling Framework (KMF) is to provide a conceptual model of a system which describes the required knowledge and inferences at an implementation independent way. This approach is intended to support the engineer in the knowledge acquisition phase [Van de Velde, 1993] and to facilitate reuse [Fensel, 1997a].

However, KMFs and reusable software libraries have rarely been applied in the field of MAS to deal with the reuse and interoperation problems arising in open environments. This thesis explores the utility of a KMF to support the automated design and coordination of agent teams according to stated problem requirements; in other words, we translate the Bottom Up Design Problem problem to the MAS field: given a set of requirements, find a set of agent capabilities whose combined competence and knowledge satisfy the requirements.

1.2 Contributions

The main outcome of our efforts to overcome the problems concerning interoperability and reuse in open MAS is a multi-layered framework for MAS development and deployment that integrates Knowledge Modelling and Cooperative Multi-Agent Systems together. This framework is called ORCAS, which stands for Open, Reusable and Configurable multi-Agent Systems.

The ORCAS framework explores the use of a KMF for describing and composing agent capabilities with the aim of maximizing capability reuse and supporting the automatic, on-demand configuration of agent teams according to stated problem requirements. The ORCAS KMF is being used as an ACDL supporting semantic matchmaking and allowing capability descriptions in a domain independent manner, in order to maximize capability reuse.

The Knowledge Modelling Framework of ORCAS has been complemented with an Operational Framework, which describes a mapping from concepts in the Knowledge-Modelling Framework to concepts from Multi-Agent Systems and Cooperative Problem Solving. Specifically, the Operational Framework describes how a composition of capabilities represented at the knowledge-level can be operationalized by a customized team of problem solving agents. In order to do that, the Operational Framework extends the KMF to describe also the communication and the coordination mechanisms required by agents to cooperate. Our approach to describe such aspects of a capability is based on the macro-level

(societal) aspects of agent societies, which is focused on the communication and the observable behavior of agents, rather than adopting a micro-level (internal) view on individual agents. The reason to focus on the macro-level is to avoid imposing a specific agent architecture, thus facilitating the design and development of agents to third parties, a basic requirement of open MAS.

The ORCAS Operational Framework proposes a new model of the Cooperative Problem Solving process that is based on a knowledge-level [Newell, 1982] description of agent capabilities, using the ORCAS KMF. This model includes a Knowledge Configuration process that takes a specification of problem requirements as input and searches a composition of capabilities and knowledge satisfying those requirements. The result of the Knowledge Configuration process is a *task-configuration*, a knowledge-level design of an abstract agent team, in terms of the tasks to be solved, the capabilities to be applied, and the knowledge to be used by those capabilities.

An agent willing to start a cooperative activity requires an initial plan to know which are the capabilities required in order to select suitable agents for that plan. In larger systems, team selection may involve an exponential number of possible team combinations, and a blow-out in the number of interactions required to select the members of a team. There are two approaches to overcome these problems: one approach, that still relies on some kind of global plan is that of guiding the team formation with problem requirements [Tidhar et al., 1996]; another approach is to use distributed tasks allocation methods to make the team selection computationally tractable [Shehory and Kraus, 1998, Sandholm, 1993]; furthermore, a mixture of both approaches is also feasible [Clement and Durfee, 1999].

In this thesis we adopt the approach based on guiding the team formation process with the problem requirements, but the notion of a initial plan is here replaced by the notion of a task-configuration. A task-configuration reduces the complexity of the team formation process by constraining the composition of the team to a certain design that satisfies the requirements of the problem. In spite of its combinatorial nature, the complexity of the team selection process is mitigated, though partially transferred from the team formation process to the Knowledge Configuration process. Therefore, in order to further reduce the complexity of the Knowledge Configuration and the team formation activities, we propose Case-Based Reasoning to heuristically guide the search process over the space of possible configurations.

Finally, we have implemented and agent infrastructure according to the ORCAS model of the CPS process. This agent infrastructure has being implemented using the *electronic institutions* formalism [Esteva et al., 2001, Esteva et al., 2002b], which is based on a computational metaphor of human institutions from a macro-level point of view.

Human institutions are places where people meet to achieve some goals following specific procedures, e.g. auction houses, parliaments, stock exchange markets, etc. Intuitively, the notion of electronic institutions refers to a sort of virtual place where agents interact according to explicit conventions. The

institution is the responsible for defining the rules of the game, to enforce them and impose the penalties in case of violation.

An electronic institution, or e-Institution, is a "virtual place" designed to support and facilitate certain goals to the human and software agents concurring to that place. Since these goals are achieved by means of the interaction of agents, an e-institution provides the social mediation layer required to achieve a successful interaction: interaction protocols, shared ontologies, communication languages and social behavior rules. The interaction is not only regulated by the institution, furthermore it is mediated by institutional agents that offer an added value to participating agents.

The ORCAS e-Institution brings an added value to both requesters and providers: on the one hand, requesters are freed of finding adequate providers and provides a single interface to the multiple and heterogenous providers; on the other hand, the institution provides an advertisement service to capability providers, provides a mediation service for the team formation process, and facilitates coordination during the teamwork activity, allowing agents to solve complex problems that cannot be achieved by an agent alone.

However, in addition to implement an agent infrastructure using the electronic institutions formalism, we are interested on using the concepts proposed by the e-Institutions approach to describe the communication and the operational description of agent capabilities without imposing neither a specific agent architecture, nor an attitudinal theory of cooperation.

The goal of partitioning the ORCAS framework in layers is to bring developers an extra flexibility in adapting this framework to their own requirements, preferences and needs. We claim that a clear separation of layers will support a flexible utilization and extension of the framework to fit different needs, and to build different infrastructures. Therefore, we divide the ORCAS framework in three complementary frameworks:

1. The *Knowledge Modelling Framework* (KMF) proposes a conceptual and architectural description of problem-solving systems from a knowledge-level view, abstracting the specification of components from implementation details. In addition, a Knowledge Configuration model is presented as the process of finding configurations of components that fulfill stated problem requirements.

2. The *Operational Framework* deals with the link between the characterization of components and its implementation, that in our framework is realized by Multi-Agent Systems. This framework comprehends an extension of the KMF to become a full-fledged Agent Capability Description Language, together with a new model of the Cooperative Problem Solving process based on the KMF.

3. The *Institutional Framework* describes an implemented infrastructure for developing and deploying Multi-Agent Systems configurable on-demand, according to the the two layered —knowledge and operational— configuration framework. This infrastructure is designed and implemented ac-

cording to an institutional model of open agent societies. The result is multi-agent platform that supports flexible, extensible and configurable Multi-Agent Systems.

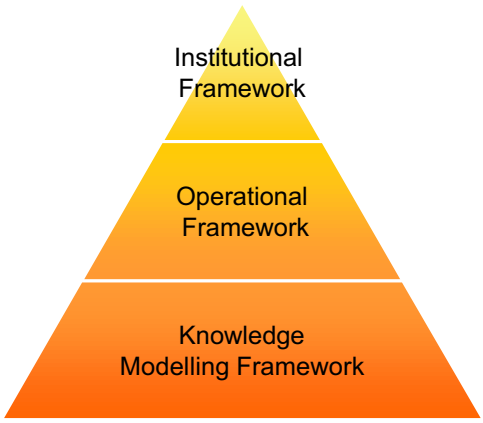

Figure 1.2: The three layers of the ORCAS framework

Figure 1.2 shows the three layers as a pyramid made of three blocks. The block at the bottom corresponds to the more abstract layer, while upper blocks corresponds to increasingly implementation dependent layers. Therefore, developers and system engineers can decide to use only a portion of the framework, starting from the bottom, and modifying or changing the other frameworks according to its preferences and needs.

1.3 Structure

This thesis consist of 7 chapters, including this one, and several appendixes providing technical information. The thesis is organized as follows (Figure 1.3):

Chapter 2 reviews some research relevant to our thesis and discusses some of their contributions that put the basis for our work, together with its limitations and the open issues we are dealing with. Since our work integrates two fields together -knowledge modelling and multi-agent systems-, this chapter have to address very different issues.

Chapter 3 draws the structure of ORCAS framework to give the reader an overall view of it, and remarks the outstanding elements of each layer so as to disclose the logic underpinning that structure.

Chapter 4 proposes a knowledge modelling framework for Multi-Agent Systems. This framework describes a conceptual and architectural characterization of problem-solving systems from a knowledge-level perspective,

abstracting the specification from any implementation details. Moreover, this chapter describes a Knowledge Configuration process that is able to find a configuration of components (tasks, capabilities and domain-models) fulfilling stated problem requirements.

Chapter 5 describes a framework to operationalize a knowledge-level configuration by forming and instructing a team of agents with the required capabilities and domain knowledge. This chapter also describes a model of teamwork for team mates in order to coordinate during the problem solving process in order to fulfill the stated problems requirements.

Chapter 6 introduces the institutional framework, an implemented infrastructure for system development that is based on the two layered approach to multi-agent configuration together with an institutional approach to open agent societies, in support of flexible, customizable and extensible Cooperative Multi-Agent Systems.

Chapter 7 shows an implemented application as a case study of the ORCAS framework, the Web Information Mediator (WIM). WIM is an application to look for medical bibliography in Internet that relies on a library of tasks and agent capabilities for information search and aggregation, linked to a medical application domain.

Chapter 8 presents some conclusions and draws up those open issues that are believed to deceive future work.

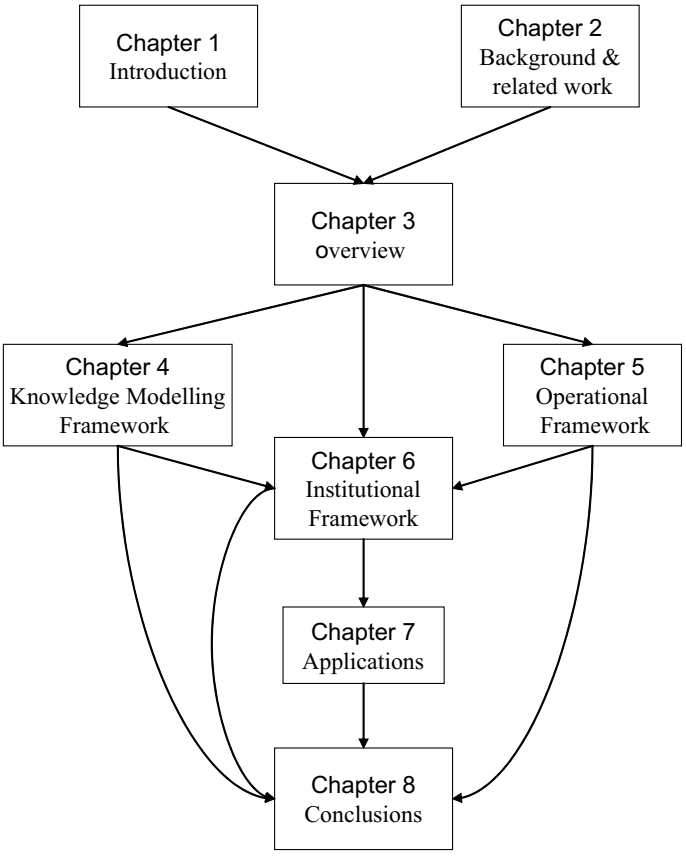

Figure 1.3: Thesis structure

Chapter 2

Background and related work

2.1 Introduction

Our work explores the potential of a Knowledge Modelling Framework for describing Multi-Agent Systems at the knowledge level, with the goal enabling the Cooperative Problem Solving process to be r adapted on-demand to the requirements of the problem at hand. Therefore, our review of related has to deal with two main research areas: Knowledge Modelling (§2.2) and Multi-Agent Systems §2.4.

This background is completed with a section on Software Reuse (§2.3), and another section on Semantic Web Services (§2.5). On the one hand, the reason to include a section about software reuse is that we aim to maximize the reuse of agent capabilities across multiple domains. On the other hand, the reason to include a section devoted to Semantic Web Services (§2.5) is the shared interest of that field and Multi-Agent Systems on interoperation in open environments like the Internet. A more detailed organization of the chapter is sketched below:

- The chapter begins with a review of some Knowledge Modelling Frameworks in §2.2, focusing on those frameworks related with the Task-Method-Domain modelling paradigm: Generic Tasks (§2.2.1), Role-limiting Methods (§2.2.2), Components of Expertise (§2.2.3), KADS/CommonKADS (§2.2.4), UPML(§2.2.5) and current issues in knowledge modelling(§2.2.6). This section concludes with a subsection about reuse of problem solving methods (§2.2.6), and more specifically, it is mainly about the role of ontologies in component reuse.

- Section §2.3 is about those aspects of reuse that are more relevant to our work, and more specifically, those research lines assuming a compositional approach to software development, either explicitly or implicitly. Section §2.3.1 review the field of software libraries, which is similar to the idea

of libraries of problem solving methods in knowledge-engineering; Section §2.3.2 is about Component-Based Software Development (CBSD), which share some concepts with abstract architectures for knowledge-engineering; and finally, Section §2.3.3 deals with the requirements for a semantic-based reuse and the solutions.

- Section §2.4 addresses the wide field of Multi-Agent Systems. Although some dimensions and alternative views of Multi-Agent Systems will be presented, we favor an external, societal view of Multi-Agent Systems, thus avoiding topics such as agent architectures and agent theories. First, we review Cooperative Multi-Agent Systems (§2.4.1), including subsections on collaboration (§2.4.1), coordination (§2.4.1) and Cooperative Problem Solving (§2.4.1). The section continues with Team Formation (§6.5.3), a key activity of the Cooperative Problem-Solving process. Team Formation can be achieved by either a centralized task allocation (§2.4.2) and distributed task allocation (§2.4.2) approaches. Another section that is especially relevant for this thesis deals with agent interoperation in open environments (§2.4.3), including agent communication (§2.4.3), middle agents (§2.4.3), and matchmaking (§2.4.3). Section §2.4.3 ends with a review of infrastructures for developing and deploying Multi-Agent Systems in open environments (§2.4.3). Next section introduces social modelling approaches to Multi-Agent Systems (§2.4.4), and finally, the section on MAS concludes concludes with a review of agent-oriented methodologies (§2.4.5).

- The last section of this chapter is about Semantic Web Services (§2.5), and is mainly concerned with proposed frameworks (§2.5.1) and relevant work on composition and interoperation of Web services (§2.5.2).

At the end of the chapter we review the open issues and describe the type of problems that we expect to contribute to. In addition, the main differences between our approach and related work are discussed.

2.2 Knowledge Modelling Frameworks

There is much consensus that the process of building a knowledge system can be seen as a modelling activity. Building a knowledge system means building a computer interpretable model with the aim of making problem-solving capabilities comparable to a domain expert, but it is not intended to simulate the cognitive processes involved in human problem-solving.

In the early eighties the development of an expert system has been seen basically as a transfer process of human knowledge into an implemented knowledge base. This approach relies on the assumption than the knowledge is available there (in the expert) ready to be collected. Typically, this knowledge is acquired through interviewing experts, and the knowledge is implemented in some kind of production rules executed by some interpreter. But due to the limitation of one single knowledge representation formalism, and to the fact that

expert knowledge is often difficult to acquire (i.e. tacit knowledge) it was realized that the transfer approach adopted by first generation expert systems is only feasible for small prototypical systems, but it failed to produce large, reliable and maintainable knowledge bases. Therefore, with the introduction of the knowledge level [Newell, 1982] in the development of knowledge system, the knowledge acquisition phase is no longer seen as a transfer of knowledge, but as a model construction process [Clancey, 1989] with the following characteristics [Studer et al., 1998]:

- a model is an approximation of the reality, thus it is never ended;
- modelling is a cyclic process, so new observations can lead to a refinement, modification or completion of the already-made model; in the other side, the model may guide the further acquisition of knowledge; and
- modelling depends on subjective interpretations of the knowledge engineer, therefore this process is typically faulty.

Knowledge Modelling Frameworks propose methodologies, architectures and languages for analyzing, describing and developing knowledge systems. While different frameworks may differ on specific details, all of them are based on the idea of building a conceptual model of a system, which describes knowledge and inferences at a domain independent level. These frameworks have been influenced by the notion of the *knowledge level* [Newell, 1982], which proposes to describe a system by focusing on the knowledge they contain rather than the implementation structures of the knowledge (the *symbol level*). In addition, the knowledge level proposes to view a system as an agent with three classes of components: goals, actions and bodies of knowledge; and introduces a *principle of rationality* in the agent behavior: actions are selected to attain goals. Next, we will review the most prominent frameworks for knowledge engineering as a modelling activity at the knowledge level, namely: *Generic Tasks*, Role-Limiting Methods, Components of Expertise, KADS and CommonKADS.

2.2.1 Generic Tasks

In the early eighties, the study of existing knowledge system for design and diagnosis evolved into the notion of a Generic Task (GT) [Chandrasekaran, 1986]. This approach proposes a task-oriented methodology for analyzing and building knowledge-systems. The main intuition underlying this proposal is that there are some recurring patterns in problem-solving activity (e.g. *hierarchical classification*, *abduction assembly* and *hypothesis matching*) that can be reused. Generic tasks are thus viewed as building-blocks that can be combined to build more complex problem-solving tasks (e.g. diagnosis) [Chandrasekaran, 1987]. This approach suggests that the representation of knowledge should closely follow its use *strong interaction problem hypothesis* [Bylander and Chandrasekaran, 1988], and that there are different organizations of knowledge suitable for different types of problem-solving: “Representing knowledge for the purpose of solving some

problem is strongly affected by the nature of the problem and by the inference strategy to be applied to the knowledge". A GT is characterized by:

- a generic description of its input and output;
- a fixed schema of knowledge types specifying the structure of the knowledge; and
- a fixed problem-solving method/strategy specifying the inference steps and the sequence in which these steps have to be carried out.

Since a GT fixes the type of knowledge required to solve a task, it provides a vocabulary that can be used to guide the knowledge acquisition process. A *Task Specific Architecture* (TSA) is an executable shell for a GT that provides a task-specific inference engine and a knowledge-base representation language. Therefore, a particular problem-solver is developed by instantiating a TSA with domain specific terms. However, this approach has the disadvantage of conflating the notion of task and problem-solving method. To overcome this limitation the notion of a *Task-Structure* was proposed [Chandrasekaran et al., 1992]: it makes a clear distinction between a task, which is used to refer to a type of problem, and a method, which is a particular way to accomplish a task. A task structure includes a set of alternative methods suitable for solving a task. A task can be decomposed into subtasks, thus a task structure is a hierarchical decomposition of tasks into subtasks, and the methods suitable for each task. The decomposition structure is refined to a level in which subtasks can be solved "directly" using available knowledge. From this point of view tasks refer to types of problems, while methods are specific ways of solving tasks [Chandrasekaran and Johnson, 1993].

2.2.2 Role-Limiting Methods

This approach [McDermott, 1988] focuses on the characterization of reusable Problem-Solving Methods. A Role-Limiting Method (RLM) is a method that declares the roles knowledge can play in that method. From this approach, whereas domain knowledge should not be acquired and represented with independence of the method, it is still explicitly and separately represented. An advantage of this approach is that method roles prescribe what knowledge should be acquired, therefore the expert only have to instantiating the generic roles with available knowledge. Furthermore, the problem-solving method facilitate explanations beyond a simple recall of inference steps as was usual in first generation expert systems. However, some limitations have been presented to this approach: first, knowledge-acquisition is completely driven by the RLM [Steels, 1990] and thus it is difficult to reuse domain-models for new RLMs; and second, RLMs have a fixed structure that is not well suited to deal with tasks that should be solved by a combination of several methods [Studer et al., 1998]. In order to overcome the inflexibility of RLMs, the concept of configurable RLMs has been proposed.

Configurable *Role-Limiting Methods* exploits the idea of a complex PSM being decomposed into subtasks, where each subtask may be solved by different methods [Poek and Gappa, 1993].

Role-limiting methods lead to a streamlined methodology for doing knowledge acquisition which have resulted in several tools that have been successfully applied in a variety of real-world applications.

2.2.3 Components of Expertise

Components of Expertise [Steels, 1990] is an attempt to synthesize the idea of Role-Limiting Methods and the task-structure analysis of the GT approach. In addition, this approach presents a *componential framework* that is expected to overcome some of the problems detected in previous work by imposing more modularity on the different components of expertise and emphasizing pragmatic constrains. The componential framework considers three classes of components to describe and build a problem solver: *tasks*, *models* and *Problem-Solving Methods*.

From a conceptual point of view, a task is characterized in terms of the type of problem to be solved (e.g. diagnosis, interpretation, design, planning, and so on). This characterization is based on properties of the input, output, and nature of the operations that map the input to the output. Usually, there is a main task that describes the application problem, but tasks can be decomposed into subtasks with input/output relations between them, resulting in a task structure [Chandrasekaran et al., 1992]. However, the pragmatic view focuses on task constrains that result from the environment or from the epistemological limitations of humans (models are limited in their accuracy and scope of prediction).

The componential framework addresses the question of knowledge modelling having in mind the separation of deep and surface knowledge [Steels, 1988]. From the perspective of deep expert systems, problem solving is viewed as a modelling activity in which some models of the world are constructed in order to solve a problem using that models. *Case models* are about the particular problem solving situation, which is determined by the task and methods at hand. However, *domain-models* are valid for a variety of cases. Domain-models describe domain specific knowledge that is used by Problem-Solving Methods to construct case models. There are two further subtypes of domain-models: *Expansion models* can be used to expand a case model by inference or data gathering; and *mapping models* are used to construct or modify a case model based on a mapping from elements of other models. Models can be constructed from different perspectives (for example, there are *functional models*, *causal models*, *behavioral models* and *structural models*) and represented in heterogeneous forms, like rules, hierarchies or networks.

Problem-Solving Methods (PSM) are responsible for applying domain knowledge to solve a task. A problem-solving method might decompose a task into subtasks or directly solve a subtask. In either case they can consult domain-models, create or change intermediary knowledge structures, perform actions to gather more data or expand a case model by adding or changing facts.

2.2.4 KADS and CommonKADS

KADS [Schreiber et al., 1993, Wielinga et al., 1993] is methodology for the analysis and design of knowledge system, which was further developed to *CommonKADS* [Schreiber et al., 1994a]. A basic characteristic of KADS is the construction of a collection of models, where each model captures specific aspects of the knowledge system to be developed as well as of its environment. In CommonKADS the *Organization Model*, the *Task Model*, the *Agent Model*, the *Communication Model*, the *Expertise Model* and the *Design Model* are distinguished:

- The *Organizational Model* describes the organizational structure in which the knowledge system will be introduced.
- The *Task Model* provides a hierarchical description of the tasks which are performed in the organizational unit, and the agents assigned to the different tasks.
- The *Agent Model* specifies the capabilities of each agent involved in the execution of tasks. In general an agent can be a human or some kind of software system.
- The *Communication Model* specifies the interactions between the different agents, including the type of information exchanged.
- The *Design Model* describes the system architecture, the representation and the computational mechanisms to realize a problem-solver according to the requirement of the target system captured at the Expertise Model and the Communication Model.
- The *Expertise Model* describes the knowledge required by agents to solve tasks, approaching knowledge modelling from three different perspectives: static, functional and dynamic. Accordingly, three layers for the expertise model are distinguished: *domain layer*, *inference layer* and *task layer*.
  - At the *domain layer* the domain specific knowledge required to solve tasks is modelled. Domain knowledge is conceptualized through domain-models. A domain-model provides a partial view on a part of the domain knowledge, which statements are conceptualized by a particular *ontology*. The main goal of structuring the domain layer is facilitate its reuse for solving different tasks.
  - At the *inference layer* the reasoning process of a knowledge system is specified following the Role-Limiting Methods approach [McDermott, 1988]. Problem-Solving Methods are described by primitive reasoning steps called *inference actions* as well as the *roles* played by domain knowledge. Dependencies between inference actions and roles are specified by an *inference structure*, which specifies how the

knowledge roles are used and produced by inference actions. Furthermore, the notion of roles provides a domain independent view of the domain

- The *task layer* provides a description of tasks in terms of input and output roles, and specifies the goals characterizing it. In addition, the task layer describes a decomposition of the task into subtasks, as well as the control flow over subtasks.

Both semi-formal and formal languages have been proposed to describe the Expertise Model: CML (Conceptual Modelling Language) [Schreiber et al., 1994b], which is a semi-formal language with a graphical notation, oriented towards the human understanding; and $(ML)^2$, which is a formal specification language based on first order predicate logic, meta-logic and dynamic logic.

The clear separation between the domain knowledge and the reasoning process at the inference and task layers enables two kind of reuse: on the one hand, a domain-model may be reused by different methods; on the other hand, a method may be reused in a different domain [Studer et al., 1998] by defining a new view on the domain. This approach weakens the *strong interaction problem hypothesis* [Bylander and Chandrasekaran, 1988], which is thus redefined as the *relative interaction hypothesis* [Schreiber et al., 1994a]: whereas some kind of dependency exists between the structure of the knowledge and the type of the task, it can be minimized by explicitly stating the dependencies between reasoning methods and domain knowledge [van Heijst, 1995]. Task and PSM ontologies may be defined as two viewpoints on an underlying domain ontology.

Since one of the goals of the CommonKADS approach is to facilitate reuse, a library of reusable and configurable components has been defined that can be used to build up an Expertise Model [Valente et al., 1994], including components to solve the following type of (generic) tasks: modelling, design, planning, assignment, prediction, monitoring, assessment and diagnosis [Breuker, 1994].

2.2.5 UPML

The *Unified Problem-solving Method Development Language* (UPML) [Fensel et al., 1999] is a framework for developing knowledge-intensive reasoning systems based on libraries of generic problem-solving components. UPML is a description language that integrates work on knowledge modelling, interoperability standards and ontologies. Rather than providing the object level description language, UPML provides an architectural framework that specifies the components, the connectors and a configuration of how the components should be connected (architectural constraints) to build a system. Moreover, design guidelines define a process model for building complex knowledge systems out of elementary components [Fensel and Motta, 2001].

The UPML architecture for describing a knowledge system consists of six different elements: tasks, domain-models, PSMs, ontologies, bridges, and refiners: tasks define the type of problems, PSMs specify the reasoning process, and

domain-models characterize the domain knowledge used by PSMs. Each of these elements is described independently to enable the reuse of task descriptions in different domains, the reuse of PSMs for different tasks and domain, and the reuse of domain knowledge for different tasks and PSMs. Ontologies provide the terminology used in tasks, PSMs and domain definitions. Again, this separation enables knowledge sharing and reuse. For example, different tasks or PSMs can share parts of the same vocabulary and definitions. A fifth element of a specification of a knowledge system are adapters, which are necessary to adjust the other (reusable) parts to each other and to the specific application problem. UPML provides two types of adapters: bridges and refiners. Bridges explicitly model the relationships between two distinguished parts of an architecture, e.g. between domain and task or between task and PSM. Refiners can be used to express the stepwise adaptation of elements of a specification, e.g. a task is refined into another more specialized task, or a PSM is refined into a more specialized PSM [Fensel, 1997b]. Again, separating generic and specific parts of the reasoning process maximizes reusability.

A very important role within the UPML framework is borne by ontologies. An ontology provides an explicit specification of a conceptualization, which can be shared by multiple reasoning components communicating during a Teamwork process. In UPML, ontologies are used to define the terminology and its properties used to define tasks, PSMs, and domain-models. UPML does not commit to a specific language style for defining a signature and its corresponding axioms. However, two styles of specifying signature and axioms are studied and has been proposed as suited formalisms: logic with sorts and a frame-based representation using concepts and attributes.

UPML systems are made by adapting and integrating components in a way constrained by the abstract architecture. The overall configuration process is guided by tasks that provide generic descriptions of problem classes. After selecting, combining and refining tasks they are connected with PSMs suitable for those tasks, and both tasks and PSMS are filled in with domain knowledge for the given domain.

2.2.6 Recent issues in knowledge modelling and reuse

The work on Problem-Solving Methods started in the eighties, when a number of researchers recognized common patterns in the reasoning processes of various knowledge systems and described them at a higher level of abstraction, decoupling them from the application domain. These conceptual models of the reasoning process make it easier to understand and maintain knowledge-based systems, and can be used to improve explanation facilities and reuse. The knowledge modelling community has carried out a large body of work on formalizing problem solving methods, on building libraries and on characterizing methods in terms of their assumptions and competencies. Research in this area is concerned with developing new PSMs, methodologies for PSM reuse, libraries of reusable PSMs, tools to support PSM development, and languages for representing PSMs. Recent developments of the knowledge mod-

elling community are undertaking the possibilities of Internet as a medium [Benjamins, 1997, Benjamins et al., 1999, Monica Crubezy and Musen, 2001].

A recent approach that is being addressed from both the software engineering and the knowledge engineering communities is that of software architectures [Shaw and Garlan, 1996, Garland and Perry, 1995]. The goal of *software architectures* is learning from system developing experience in order to provide the abstract recurring patterns for improving further system development. As such, software architectures contribution is mainly methodological in providing a way to specify systems. A software architecture has the following elements: (i) components, (ii) connectors, and (iii) a configuration of how the components should be connected [Garland and Perry, 1995]. Software architectures are designed to build applications by matching the specification of abstract components with the specification of components in a library or repository. Work on software architectures establishes an abstract level to describe the functionality and the structure of software artifacts, thus they are suited to describe the essence of large and complex software systems. Such architectures specify classes of application problems instead of focusing on the small and generic components from which a system is built up. The work on formalizing software architectures in terms of assumptions over the functionality of its components [Penix and Alexander, 1997, Penix, 1998, Penix and Alexander, 1999] shows strong similarities to recent work on PSMS, which define the competence in terms of assumptions over the domain knowledge [Benjamins et al., 1996c, Fensel and Straatman, 1996, Fensel and Benjamins, 1998a, Musen, 1998]. PSMs require specific types of domain knowledge and introduce specific restrictions on the tasks that can be solved by them. These requirements and restrictions are assumptions that play a key role in reusing Problem-Solving Methods, in acquiring domain knowledge, and in defining the problems that can be tackled by a knowledge-based system.

A complementary line of research is the work on ontologies to characterize consensual, formal and declarative knowledge models [Gruber, 1993a]. While Problem-Solving Methods describe the reasoning process of a knowledge-based system, ontologies provide the means to describe the domain knowledge that is used by these methods [Musen, 1998]. The availability of reusable methods and reusable domain knowledge reduces the development process of knowledge-based systems to a "plug-and-play" process [Walther et al., 1992].

Libraries of Problem-Solving Methods

Today, there exist several repositories or *libraries* of PSMs at different locations, with different scope, including the following: diagnosis [Benjamins, 1993], planning [Benjamins et al., 1996b], assessment [Valente and Lockenhoff, 1993], and design [Chandrasekaran, 1990].

All these libraries aim at facilitating the knowledge engineering process, yet they differ in various dimensions such as generality, formality, granularity and size. The type of a library is determined by its characterization in terms of these dimensions. Each type has a specific role in For example, the more general PSMs (i.e. task neutral) in a library are, the more reusable they are, because they do

not make any commitment to particular tasks. However, at the same time, very general PSMs will require a considerable refinement and adaptation effort. This phenomenon is known as the *reusability-usability trade-off* [Klinker et al., 1991].

Another issue concerning component libraries studies the way component specifications are organized and retrieved. There are several alternatives for organizing a library, and each of them has consequences for indexing PSMs and for their selection. Several researchers propose to organize libraries following a task-method decomposition structure [Chandrasekaran et al., 1992, Puerta et al., 1992, Steels, 1993, Shadbolt et al., 1993, Terpstra et al., 1993]. According to this organization structure, a task can be realized by several PSMs, each consisting of primitive or composite subtask that can again be realized by alternative methods. Guidelines for library design according to this principle have been discussed [Orsvarn, 1996], pointing out that PSMs are indexed according to two factors: the competence of the PSMs, and the assumptions under which they can be correctly applied. Selection of PSMs from such libraries should consider first the competence of the PSMs (selecting those whose competencies match the task at hand), and then the assumptions of PSMs (selecting those whose assumptions are satisfied).

Brokering of Problem Solving Methods

Problem-Solving Methods for knowledge systems establish the behavior of such systems by defining the roles in which domain knowledge is used and the ordering of inferences. Developers can compose PSMs that accomplish complex application tasks from primitive, reusable methods. The key steps in this development approach are task analysis, method selection from a library, and method configuration [Eriksson et al., 1995]. From the knowledge modelling community, this approach is described as a configuration process with the following activities: PSM selection according to their competence to solve a given task, verification of domain requirements, combining PSMs together, and mapping them to domain knowledge. This approach to software development is intended to support the engineer in the knowledge acquisition phase [Van de Velde, 1993] and to facilitate reuse [Fensel, 1997a]. The question of reuse has received a lot of attention last years from the knowledge modelling community [Benjamins et al., 1996a, Motta, 1999, Fensel and Motta, 2001]. Nowadays, the World-Wide Web is changing the nature of software development to a distributive *plug-and-play* process, which requires a new kind of managing software: intelligent software brokers. For selecting PSMs from a library, a broker needs to reason about characteristics of PSMs like their competence and their assumptions. A recent project, IBROW, [Benjamins et al., 1998, Benjamins et al., 1999], aims to provide an intelligent brokering service on the Web. One problem that appears while brokering components is the need to annotate components with semantic information (meta-data). From the knowledge-modelling approach, meta-data is provided by the three classes of components: PSMs, tasks and domain-models.

The IBROW approach to brokering libraries of problem-solving components is that of configuring a knowledge-based system by selecting, adapt-

ing and integrating components retrieved from distributed libraries available on the Web [Monica Crubezy and Musen, 2001]. Essentially, the broker is a mediator between customers and providers of Problem-Solving Methods [Benjamins, 1997, Fensel and Benjamins, 1998b]. A customer is someone that has a complex problem but can provide domain knowledge that describes it and that supports problem-solving. The providers are developers of Problem-Solving Methods to be stored in libraries accessible through the Internet. PSMs are annotated with meta-data to support their selection process and invocation. The core of an intelligent broker for Problem-Solving Methods consists of an *ontologist* that supports the selection process of Problem-Solving Methods for a given application. Basically, such a broker has to provide support in building or reusing a domain ontology and in relating this ontology to an ontology that describes generic classes of application problems. This problem-type ontology has to be linked with PSM-specific ontologies that allow the selection of a method.

Ontology-based reuse

The knowledge modelling community has focused on ontologies [John H. Gennari and Musen, 1998, Studer et al., 1996] as a way to share consensual conceptualizations that are required to facilitate reuse. Ontologies are defined as "shared agreements about shared conceptualizations". Shared conceptualizations include conceptual frameworks for modelling domain knowledge; content-specific protocols for communicating among interoperating agents; and agreements about the representation of particular domain theories. In the knowledge sharing context, ontologies are specified in the form of definitions of representational vocabulary [Guarino, 1997b].

Although the definition of what ontologies are is still a debated issue [Guarino, 1997b], this term has achieve a considerably attention by the AI community, and in particular, it has been declared as a key issue in maximizing reuse [Fensel et al., 1997, Fensel, 1997a, Fensel and Benjamins, 1998b]. From that viewpoint, the main goal of an ontology is to facilitate knowledge sharing [Chandrasekaran et al., 1998]. In addition to enable reasoning about components in order to compare, retrieve, reuse or adapt them, ontologies play a central role in connecting software components by allowing the comparison of components using different vocabularies. The comparison of components described with different ontologies or different concepts of the same ontology is allowed by the annexion of ontology *mappings*.

Ontology mappings are declarative specifications of *matching relations* between ontologies, which consist of explicit specifications of the transformations required to match elements of one ontology to elements in another ontology. An example of a mapping is a *renaming*, but mapping can include any kind of syntactic or semantic transformation: *numerical mapping*, *lexical mapping*, *regular expression mapping* and others. In our framework the core components (tasks, capabilities and domain-models) are described with explicit, independent ontologies [Fensel et al., 1997]. In the context of modern componential frameworks for knowledge systems development, ontology mappings are required to

match problem requirements to tasks, Problem-Solving Methods to tasks, and domain-models to Problem-Solving Methods.

The importance of ontologies has even originated what has been called *ontology engineering* [Guarino, 1997b]. A remarkable outcome of this approach is that of reusing also the ontology-based connectors or *mappings*; not surprisingly, reuse of mappings is also improved by specifying a mapping ontology [Park et al., 1998].

2.2.7 Conclusions

Although there are some differences between the different frameworks presented above, a first conclusion of the review on Knowledge Modelling is that there exist much consensus about the use of three classes of components to model knowledge-based systems from a knowledge level approach [Fensel et al., 1999, McDermott, 1988, Chandrasekaran, 1986, Steels, 1990, Schreiber et al., 1994a]. This paradigm proposes three types of components: there are *tasks* describing problem types, *Problem-Solving Methods* (PSM) describing the reasoning steps required to solve a class of problems in a domain-independent way, and *domain-models* describing the properties of domain knowledge. We use the term *Task-Method-Domain* (TMD) frameworks to refer to this architectural pattern found in modern Knowledge Modelling Frameworks.

We are interested in TMD frameworks as a key to maximize reuse [Benjamins et al., 1996a, Motta, 1999, Fensel and Motta, 2001] and specifically, we are mainly interested in its application to the dynamic, on-demand configuration of Cooperative Multi-Agent Systems. Modern approaches from the knowledge engineering community are explicitly addressing reuse as an activity of selecting, configuring and assembling knowledge components from distributed libraries [Gennari and Tu, 1994, Eriksson et al., 1995, Fensel, 1997a, Fensel and Benjamins, 1998b, Benjamins et al., 1999]. TMD approaches envisage an scenario in which developers can compose Problem-Solving Methods that accomplish complex application tasks from primitive, reusable methods. This way of developing a knowledge-system is described as a configuration process (§2.2.6) and is intended to support the engineer in the knowledge acquisition phase [Van de Velde, 1993] and to facilitate reuse [Fensel, 1997a]. However, configuring such a system to build a completely new application is far away of becoming an automated process and is rather described as a semi-automated process [Gaspari et al., 1998, Gaspari et al., 1999, Benjamins et al., 1999, Penix, 1998]. Semi-automatic configuration can be used to support and assist knowledge engineers in the configuration and adaptation of knowledge-based systems [Eriksson et al., 1995, Tu et al., 1995, Studer et al., 1996, Fensel, 1997a, Fink, 1998, Penix and Alexander, 1997, Monica Crubezy and Musen, 2001], Such an approach to configuring a KBS has not been used to configure teams of problem solving-agents. The reason is probably the consideration of Team Formation as a runtime activity, whilst configuration in Knowledge Modelling Framework is considered as a knowledge engineering activity taking place on design time.

This thesis shows how a TMD modelling framework can be used to guide the Team Formation process on runtime by fully automating the configuration of the team in terms of the competence required by a team to solve a problem. An automated configuration process is required to allow agent teams to be designed on-demand, according to the requirements of the specific problem at hand. Moreover, in addition to automating the configuration process, our thesis imposes that the problem specification should be done by the end-user, and not by a knowledge or software engineer.

Another keystone of TMD frameworks is the use of ontologies as explicit, declarative specifications of the conceptualizations used to characterize components. The use of ontologies to annotate components maximizes its reuse because enables the semantic comparison of components [Fink, 1998, Gaspari et al., 1999, Fensel and Benjamins, 1998b, Gaspari et al., 1998]. Consequently, we are including explicit ontologies to describe agent capabilities in a way that facilitates their reuse and configuration in the context of cooperative Multi Agent Systems.

2.3 Software reuse

The reuse of complete software developments and the processes used to create them have the potential to significantly ease the process of software engineering by providing a source of verified software artifacts [Wegner, 1984]. It is suggested than reuse of software artifacts can be achieved through the utilization of software libraries [Atkinson, 1997].

2.3.1 Software libraries

Essentially, a software library is a repository of information which can be used to construct software systems. The main goal of software libraries reuse is to enable previous development experiences to guide subsequent software development. Reuse have been partitioned in *compositional* and *generative* reuse. In particular, we are interested in compositional-approaches to software development [Biggerstaff and Perlis, 1989], which are characterized by the idea of selecting and composing existing components in order to achieve a desired system behavior.

An important aspect of compositional reuse is about the relationship between components in a software library. The are two major relationships: one relationship is the one between the client and supplier [Atkinson, 1997], where the definition of a component by a client refers to the existence of a component in the library as provided by its supplier; and the second relationship is that of inheritance, which allows the definition of one component to include the definitions of another. In general, there may be many relationships between two software components that can be used for retrieval from a software library: one component may be a subtype [Liskov and Wing, 1993] of another, be behaviorally

compatible [Smith, 1994] with another, or be substitutable [Duke et al., 1991] with another.

According to [Mili et al., 1995], there is an open problem of software composition called the *The Bottom Up Design Problem*, defined as:

> given a set of requirements, a set of components within a software library whose combined behavior satisfies the requirements.

The fundamental difficulty when considering this problem is how to decompose the requirements in such a way as to yield component specifications. A reverse approach is to search the space of all possible component compositions until one satisfying the requirements is found [Hall, 1993, Zhang, 2000]. Thus, composition of components can be regarded as composition of their specifications [Butler and Duke, 1998].

Although the use of a software library by a software engineer requires to know the processes of how to retrieve, insert and adapt components of the library, the issue of how to retrieve components is probably the central one, since the purpose of software libraries is to provide access to reusable verified components [Atkinson, 1997]. Component retrieval is defined as the process of locating the components that can be used in the construction of a particular application. From that view, retrieval is a process of obtaining a component in a library, $S \in \mathcal{L}$ satisfying a given query Q. A query Q imposes some requirements that should be compared with the specification of existing components $\mathcal{L}$ to check wether a particular retrieval criteria holds. There have been three classes of proposed solutions to this problem: *faceted* (classification), *signature-matching* (structural) and behavioral (functional) retrieval. These retrieval techniques can use many different indices as representations of components:

- *External Indices* seek to find relevant components based upon controlled vocabularies external to the component; including facets [Prieto-Daz, 1987], frames [Rosario and Ibrahim, 1994], lexical affinity [Maarek et al., 1991] and feature-based techniques [Börstler, 1995]

- *Static Indices* include type signature matching [Zaremski and Wing, 1995] and specification matching techniques [Rollins and Wing, 1991, Fischer et al., 1995, Zaremski and Wing, 1997], which seek to find relevant components based upon elements of the structure of components.

- *Dynamic Indices* seek to find relevant components by comparing input and output spaces of components, which are used by behavioral techniques [Hall, 1993, Mili et al., 1997]

2.3.2 Component-Based Software Development

Following the idea of reuse for component-based systems, Component-based software development (CBSD) focuses on building large software systems by integrating previously-existing software components. By enhancing the flexibility

and maintainability of systems, the ultimate goal is to reduce software development costs, assemble systems rapidly, and reduce the maintenance burden associated with the support and upgrade of large systems [Brown and Wallnau, 1996].

CBSD shifts the development emphasis from programming software to composing software systems [Clements, 1996], thus the notion of building a system by writing code is replaced by the notion of assembling and integrating existing software components. From the CBSD approach, constructing an application involves the use of prefabricated pieces, perhaps developed at different times, by different people and possibly with different purposes; therefore integrability of heterogeneous components is a key consideration when deciding whether to acquire, reuse, or build new components. In other words, software components can be deployed independently and are subject to composition by third parties [Szyperski, 1996].

Component capabilities and usages are specified by *interfaces*. From a compositional approach, an interface can be defined as a service abstraction, which defines the operations that the service supports independently from any particular implementation [Iribarne et al., 2002]. Interfaces can be defined using many different notations and representation languages. Usually component interfaces are described in three levels: signature level, semantic level and protocol level. Current approaches at the signature level use Interface Description Languages such as the ones defined by CORBA, COM and CCM. At the protocol level there are many interaction protocol description languages like those based in finite-state-machines [Yellin and Strom, 1997], Petri-Nets [Bastide et al., 1999], temporal logic [Han, 1999] or π-calculus [Canal et al., 2001]. At the semantic level the operational semantics of components are described using formal notations ranging from the Larch [Garland et al., 1993] family of languages based on pre-conditions and post-conditions to algebraic equations [Goguen et al., 1996] or refinement calculus [Mikhajlova, 1999].

2.3.3 Semantic-based reuse: ontologies

The informality of feature-based classification schemes for reuse is an impediment to formally verify the reusability of a software component. However, the use of formal specifications to verify reusability has associated a high reasoning cost, which jeopardizes the scalability of reusable software libraries. A way to increase the efficiency of formal specifications is to shift the overhead of formal reasoning from the retrieval to the classification phase of reuse [Penix et al., 1995]. This is done by using a classification scheme to reduce the number of specification matching proofs (usually some kind of implication) that are required to verify reusability. Components can be classified using semantic features that are derived from their formal specification. Retrieval can then be accomplished based on the stored feature sets, which allow an efficient verification of reusability relations [Penix and Alexander, 1999].

Following the philosophy underlying the *external indices* approach to software reuse and the semantic enrichment of component descriptions, it seems

natural to introduce ontologies as shared vocabularies to describe reusable components.

We agree with [Guarino, 1997b] about the potential role of explicit ontologies to support reuse, since ontologies can be used to enable semantic matching between components [Guarino, 1997a, Paolucci et al., 2002]. Therefore, semantic matching between an application specification and the components in a library can be used to verify a reusability relation. A key concept of ontology based reuse is that of mapping. Ontology mappings are declarative specifications of *matching relations*, which consist of explicit specifications of the transformations required to match elements of one ontology to elements in the other ontology. An example of a mapping is a *renaming*, but a mapping can include any kind of syntactic or semantic transformation, including *numerical mappings*, *lexical mappings*, *regular expression mappings* and others classes of mappings. An interesting property of ontology mappings are that mapping patterns themselves can become reusable components [Park et al., 1998].

2.3.4 Conclusions

An open issue of software reuse that is addressed within this thesis is the kind of language to be used for specifying components. Recent approaches remark the need for semantic information to describe software components, and there is an increasing consensus about the class of properties to describe a component; however, there are many differences among the object languages proposed to specify these properties, from simple keywords, to First-Order Logic. Since time is an important factor to take into account when considering the on-the-fly configuration of MAS, our goal is to achieve a trade-off between the expressive power of the object language and the computational efficiency of the inference mechanism (see §4.3.1).

2.4 Multi Agent Systems

Distributed Artificial Intelligence has historically been divided in two main areas [Bond and Gasser, 1988a]: Distributed Problem Solving (DPS) and Multi-Agent Systems (MAS). In the DPS approach, a problem is divided and distributed among a number of nodes which cooperate in solving the different parts of the problem. In the DPS model, the overall problem solving strategy is an integral part of the system. In contrast, MAS research is concerned with the behavior of a collection of possibly pre-existing autonomous agents aiming at solving a given problem [Jennings et al., 1998]. From that view, a MAS is a loosely coupled network of problem-solving entities that work together to find answers to problems that are beyond the individual capabilities or knowledge of the isolated entities [Durfee and Lesser, 1989]. More recently the term "Multi-Agent System" has come to a more general meaning, and it is now used to refer to all types of systems composed of multiple (semi-) autonomous components [Jennings et al., 1998]. The MAS approach advocates decomposing problems in

terms of autonomous agents that can engage in flexible, high level interactions, and this way of decomposing a problem aids the process of engineering complex systems [Jennings, 2000]. The characteristics of MAS are:

- each agent has incomplete information or capabilities for solving the problem, thus each agent has a limited viewpoint;
- there is no global system control;
- data is decentralized, and
- computation is asynchronous.

Some reasons for the increasing interest in MAS research include: the ability to provide robustness and efficiency; the ability to allow inter-operation of existing legacy systems; and the ability to solve problems in which data, expertise, or control is distributed.

There are two main perspectives when approaching Multi Agent Systems: a macro or social level, focused on external, observable behavior, and a micro, or agent-oriented level, focused on the internal architecture of individual agents. We are more interested on the *macro* phenomena, rather than the *micro* phenomena; therefore, we are going to focus on the coordination and cooperation mechanisms of open agent societies, without paying much attention to neither agent theories nor agent architectures.

2.4.1 Cooperative Multi-Agent Systems

Cooperation is often presented as one of the key concepts which differentiates Multi-Agent Systems from other related disciplines such as distributed computing, object oriented systems, and expert systems [Doran et al., 1997]. However, the idea of cooperation in agent-based systems is yet unclear and sometimes inconsistent. There are many open questions like for example:

- What is cooperation? How does it relate to concepts like communication, coordination and negotiation?
- What sorts of cooperation are likely to be found in multi-agent systems? Which factors will affect cooperation strategies, and how?
- Is it meaningful to talk about reactive cooperation? Is cooperation a mentalistic, a behavioral notion or a mixture of the two?
- What are the key mechanisms and structures giving rise to cooperation?

The range of answers to these questions are many and varied, probably due to the different approaches adopted when addressing the cooperation issue; thus a typology of cooperation approaches seems necessary to help understand cooperation without offering a single definition of cooperation (Figure 2.1).

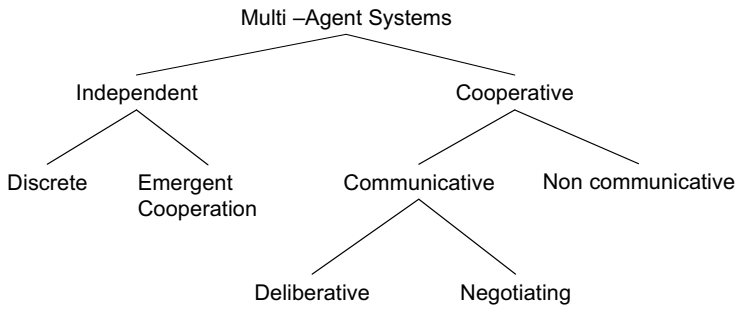

Figure 2.1: Cooperation typology

A Multi-Agent System is *independent* [Franklin and Graesser, 1996] if each agent pursues its own agenda independently of the others. A Multi-Agent System is *discrete* if it is independent and the agendas of the agents bear no relation to one another, thus there is no cooperation involved. However, cooperation is possible though agents have no intention of doing so, what can occur when cooperation is an *emergent* behavior resulting from the interaction of individuals (stigmergy is a good example [Beckers et al., 1994] of such a class of cooperation). On the other hand, there are systems in which the agendas (plans) of the agents include some way of cooperating with other agents. In *non-communicative* cooperation agents can coordinate by observing the behavior of the others [Franklin,]. In *communicative* systems agents can achieve coordination through the intentional sending and receiving of signals, which usually follow a *speech acts* style of communication. *Deliberative* agents jointly plan their actions so as to cooperate with each other. *Negotiating* agents are also deliberative, but the agents are basically competing, thus there is basically a different degree of self-interestingness.

There is an alternative viewpoint on cooperation that regards it as a property of the actions of the agents [Doran and Palmer, 1995]: cooperation occurs when agents have a (possibly implicit) goal in common (which no agent could achieve in isolation) and their actions tend to achieve that; or agents perform actions which enable or achieve not only their own goals but also the goals of other agents. This approach focuses on actions and goals, irrespective of how they arise. Therefore, from this view agents do not require to deliberate and goals may be implicit.

In contrast to the former approaches to cooperation that do not require intention, there is a more restrictive view on cooperation as a motivated activity. From that viewpoint, cooperation is defined as acting with others for a common purpose and a common benefit where the purpose should be motivated by an intention to act together [Norman, 1994]. That intention is usually referred as a commitment to joint activity [Bratman, 1992, Jennings, 1993]. This class of

cooperation relying on motivational attitudes is sometimes called collaboration [Wilsker, 1996, Grosz and Kraus, 1996] to differentiate it from other classes of cooperation.

Collaborative agents

Most early work in DAI dealt with a group of agents pursuing common goals [Lesser et al., 1989, Lesser, 1991, Durfee, 1988, Cammarata et al., 1983]. Agent interactions are guided by cooperation strategies meant to improve their collective performance, therefore early work on distributed planning took the approach of complete planning before action. These systems have to recognize, avoid or resolve dependencies or interactions between subproblems. For instance, [Georgeff, 1983] proposes a synchronizer agent to recognize and resolve such interactions.

Agents embedded in dynamic environments that are continuously sensing their environment and performing actions to change it are called *situated agents* [Rao et al., 1992]. These agents are resource-bounded [Bratman, 1988], they must reason and act under possibly stringent constrains on time and information. According to [Bratman, 1990], the intentions of the agent play a crucial role in such cases. Intentions can be seen as constrains on the deliberating and planning processes, hence reducing the reasoning effort. Systems based on mental attitudes like intentions are commonly called Belief-Desire-Intention (BDI) architectures [Bratman, 1988, Rao and Georgeff, 1991, Rao and Georgeff, 1995].

We review below three of the most influential contributions to the field, namely: Joint Intentions, SharedPlans and Planned Team Activity.

The *Joint Intentions* model [Cohen and Levesque, 1990, Levesque, 1990, Cohen and Levesque, 1991] represents one of the first attempts to establish a formal theory of multi-agent collaboration. This theory is a formal model of what motivates agent communication about teamwork. The basic premise rests in the idea of intention as the commitments to act in a certain mental state [Levesque, 1990]. A commitment represents a goal that persists over time. From this view, a team is composed of agents that jointly commit to the achievement of a team goal, called a *joint persistent goal* (JPG) [Cohen and Levesque, 1990]. An important conclusion of this theory is that by virtue of its joint commitments, an agent in a team has the responsibility to communicate private beliefs if it believes that the JPG is either achieved, unachievable or irrelevant [Cohen and Levesque, 1991]. Hence, the need of some form of communication is implicit in this model. Team goals are formed by an individual agent nominating a task as a proposed team goal, and communicating that intention until consensus is formed.

The *SharedPlans* [Grosz and Kraus, 1996, Grosz et al., 1999] model of collaboration emphasizes the need for a common high-level team model that allows agents to understand all requirements for plans to achieve a team goal [Grosz and Sidner, 1990], even if the individuals do not know the specific details of the collaborative plan or how to met the requirements The SharedPlans theory of collaboration is based on a rich view of plans. Rather than associating a

plan for some goal with a group of actions that can achieve it, a plan is instead a structure describing relationships between intentions (commitments) and information needs. Having a SharedPlan implies a joint mental state to do a group action: (1) mutual beliefs of a (partial) recipe; (2) individual intentions that the joint action will be carried over; (3) individual intentions that collaborators succeed in performing the constituent subactions; and (4) individual or collaborative plans for subplans. As a way of describing motivational attitudes, four different intention operators are introduced [Grosz and Kraus, 1993]: *intention-to*, *intention-that*, *potential-intention to* and *potential intention-that*. The first two are intentions that can be adopted by an agent, while potential intentions represent and agent's mental state when it is considering adopting and intention, but it is yet considering another possible courses of action. An *intention-to* perform some action represents an individual commitment on the part of an agent to perform that action, while an *intention-that* instead represents a commitment to certain states or conditions holding. Intentions-to serve a number of functions: (a) they constrain deliberations (an agent will seek ways to accomplish an intended action); (b) they represent commitments to action (an agent will not normally adopt new intentions that conflict with existing ones); and (c) agents monitor the success or failure of attempts to achieve an intention (failures can engender replanning). In contrast to an intention-to, an intention-that does not directly connote an action; rather, it implies that an agent will behave in a manner consistent with a collaborative effort, and can engender helpful behavior and also spawn monitoring actions. Communication requirements may arise from intentions-that, as opposed from being mandatory in the Joint Intentions model. If an agent has an intention-that about some group action to succeed then it will adopt a potential intention to perform any action it believes will help the group action to succeed. Therefore if an agent believes communicating some event or belief will aid in successfully prosecuting the group action, then it will be communicated.

The two former models have implicitly assumed that when agents establish either a Joint Commitment or a SharedPlan, they do so immediately and completely [Wilsker, 1996]; without any allowance for an intermediate mental attitude, like an expression of interest. Unlike the two previous theories, in the Planned Team Activity approach [Kinny et al., 1992, Sonenberg et al., 1994] is that plans to achieve some goal are supplied in advance, not generated by the agents, and that agents have complete knowledge of the full plans prior to joining a team. Therefore, agent behavior is bound and predictable, making this model advantageous in dynamic and real-time environments. On the other hand, teamwork is more brittle, since plans may fail within unpredictable environments. In addition, there is a greater responsibility on the agent designed, since the success of teamwork is tightly dependent on how well the plans are specified. The semantics of team's beliefs, goals and intentions are different from those in Joint Intentions. Specifically, the joint intentions of a team are expressed in terms of the joint intentions of its members, which reduce to single agent attitudes rather than by modal operators expressing shared attitudes. A team has a joint

intention towards a plan if: (1) every member has the joint intention towards the plan; (2) every member believes that the joint intention is held by the team; and (3) every member believes that all the members executing their respective individual plans results in the team executing the plan. This definition is similar to that of SharedPlans but without the intention that collaborators succeed. The process of Team Formation begins with an agent wishing to achieve some goal, but realizing that it is unable to do so by himself. The agent communicates with other potential participants by announcing the joint goal, joint plan and the individual roles to be assumed by each participant. An agent is capable of adopting a joint goal, a joint plan and a role within a plan if and only if: (a) has the necessary skills; (b) does not already believe the formula that needs to be adopted as a joint goal; (c) the preconditions of the plan are already believed by the team; (d) the joint goal is compatible with the current goals of the agent; and (e) the joint plan and role plan are compatible with the current intentions of the team member. Two strategies for Team Formation are considered by this proposal [Kinny et al., 1992]: (1) *commit-and-cancel*, and (2) *agree-and-execute*. In the commit-and-cancel strategy the team leader sends a request to each participant to "commit" to the joint goal, joint plan and role. If all the participants reply with a "committed" message within the permitted time the team has been formed; else the team leader sends a "cancel" message to them each agent committed and any team activity is abandoned. In the agree-and-execute strategy the team leader sends an "agree" to all participants, and if all reply affirmatively, sends an explicit request to all of them to execute the plan. Only at that point are the joint goal, joint plan and roles adopted by participants. Unlike the former strategy, no explicit message needs to be sent when an agent does not agree to participate, as the other agents have made no commitment yet. If a member is unable to achieve its goals within the team it has the responsibility to make other team members aware of its failure.

Another direction in multi-agent planning research is oriented towards modelling teamwork explicitly. This is particularly helpful in dynamic environments where team members may fail or where they may encounter new opportunities. For instance [Singh, 1994] proposes a family of logics for representing intentions, beliefs, knowledge, know-how, and communication in a branching-time framework. Whereas other theories are based exclusively on mental concepts, this approach combines mental and social concepts and proposes a formal theory of intentions for teams that considers the structure of teams explicitly, in terms of their members' commitments and coordination requirements [Singh, 1998]. Furthermore, this approach distinguishes between *exodeictic* (outward) and *endodeictic* (inward) intentions, which considers team structure. A team structure is defined by the constraints on the interactions —at the commitment and coordination levels— of its members.

Other works on collaboration based in multi-agent planning can be found for example in the *Social Plans* proposal [Rao et al., 1992] and in the Shared Planning and Activity Representation (SPAR) effort [Tate, 1998].

Coordination frameworks

Partial Global Planning (PGP) is a flexible approach to coordination that does not assume any particular distribution of sub-problems, expertise or other resources, but instead allows nodes to coordinate themselves dynamically [Durfee, 1988]. Cooperating agents adjust its own local planning so that the common planning goals are met, and communicate its plan to others to improve predictability and network coherence. The PGP approach to distributed coordination improved the coordination of agents in a network by scheduling the timely generation of partial results, avoiding redundant activities, shifting tasks to idle nodes, and indicating compatibility between goals. Identifying and generalizing the types of coordination relationships that were used by the basic PGP algorithm has lead to the Generalized PGP (GPGP) [Decker and Lesser, 1992]. *Generalized Partial Global Planning*(GPGP) is a coordination algorithm described in a modular, domain independent way, which can be tuned for particular intra-task environment behaviors(primarily the creation and refinement of local scheduling constraints). GPGP extends (as well as generalizes) the PGP algorithm along two lines: handling real-time deadlines and improving the distributed search among schedulers. GPGP can be seen as an extendable family of coordination mechanisms that form a basic set of mechanisms for teams of cooperative autonomous agents [Decker and Lesser, 1995]. This approach provides a set of modular coordination mechanisms; a general specification of these mechanisms involving the detection and response to certain abstract *coordination relationships* (not tied to a particular domain); and a more clear separation of the coordination mechanisms from an agent's local scheduler that allows each to better do the job for which it was designed.

TAEMS [Decker, 1996] was designed as a modelling language for describing the task structures of agents, supporting the GPGP approach to coordinated agent behavior. The acronym stands for Task Analysis, Environmental Modelling and Simulation. A TAEMS task structure is essentially an annotated task decomposition tree (actually a graph). The highest level nodes in the tree, called task groups, represent goals that an agent may try to achieve. Below a task group there will be a sequence of tasks and methods which describe how that task group may be performed. Tasks represent sub-goals, which can be further decomposed in the same manner. Methods, on the other hand, are terminal, and represent the primitive actions an agent can perform. Annotations on a task describe how its subtasks may be combined to satisfy it. Another form of annotation, called an interrelationship, describes how the execution of a method, or achievement of a goal, will affect other nodes in the structure. The TAEMS framework is designed to handle issues of real-time (e.g. scheduling to deadlines) and meta-control (e.g. to avoid the need of detailed planning at all possible node interactions). Much of what is represented in TAEMS structures is also quantitatively described, including expected execution characteristics, resource usage and specific ways to derive the quality of a task from the qualities of the combined subtasks. These quantitative aspects allow the agent to compare and contrast possible plans, predict their effects, and reason about the need for

coordination with other agents.

TEAMCORE [Tambe, 1997, Pynadath et al., 1999, Tambe et al., 2000] is an agent architecture that integrates many of the basic principles of the joint intentions theory and the Shared Plans approach. This is a perspective based on an explicit, domain independent model of teamwork that has include learning [Tambe et al., 1999] as the most remarkable issue.

The Cooperative Problem-Solving process

A general framework for the Cooperative Problem-Solving process has been described in [Wooldridge and Jennings, 1999], with four stages: recognition (an agent identifies the potential for cooperation), Team Formation, plan formation (collective attempts to construct an agreed plan) and execution. The authors adopts an internal (endodeictic) perspective, the approach is to characterize the mental states of the agents that leads them to solicit and take part in cooperative action. The model is formalized by expressing it as a theory in a quantified *multi-modal logic* . Starts from the following desiderata: agents are autonomous, cooperation can fail, communication is essential, communicative acts are characterized by their effects, agents initiate social processes, are mutually supportive and are reactive.

The view on Multi-Agent Systems as decoupled networks of autonomous entities is usually associated to a distributed model of expertise, regarded as a collection of specialized agents with complementary skills. Thus *team selection* is defined as the process of selecting a group of agents that have complimentary skills to achieve a given goal [Tidhar et al., 1996].

2.4.2 Team Formation

Team Formation is defined as the process of selecting a group of agents that have complimentary skills to achieve a given goal [Tidhar et al., 1996]. Most approaches to team selection view the process of achieving team goals as means-end analysis [Bratman et al., 1991, Rao, 1994] with two steps: first, selecting a group of agents that will attempt to achieve a goal [Levesque, 1990, Cohen and Levesque, 1991, Wooldridge and Jennings, 1994, Rao and Georgeff, 1995]; and second, selecting a combination of actions that agents must perform to achieve the goal [Grosz and Kraus, 1996, Tambe, 1997, Wooldridge and Jennings, 1999]. This combination of actions is typically described as a sequence of actions or a plan, like in in the Social Plans approach [Rao et al., 1992], the Planned Team Activity model [Kinny et al., 1992, Sonenberg et al., 1994], and the SPAR proposal (Shared Planning and Activity Representation) [Tate, 1998].

There are several approaches and variations over this basic schema. One of the more extended approach is to use plans as "recipes" [Georgeff and Lansky, 1987, Bratman et al., 1991, Sonenberg et al., 1994, Tidhar et al., 1996]. Since plans are provided by the user at compile time

the process of planning in the classical sense is unnecessary and can lead to significantly better performance.

Centralized Task allocation

One of the first methods for selecting agents for cooperative action was the *contract-net* protocol [Smith, 1940]. Given a task to perform, an agent determines whether the task can be decomposed into subtasks and announce these tasks to other agents by sending a "*call for proposals*". Bidders can reply with a bid to perform a task, indicating how well (price, quality, time, etc.) can they perform it and, finally, the contractor collects the bids and awards the task to the best bidder. This protocol enables dynamic task allocation, allows agents to bid for multiple tasks at a time, and provides natural load balancing [Jennings et al., 1998]. This protocol has however some limitations, like the absence of conflict detection and resolution, the impossibility for agents to refuse bids, or the absence of pre-emption in task execution [Jennings et al., 1998]. Some extensions of the protocol have been proposed to rectify some of its shortcomings [Sandholm, 1993]. The contract net protocol has been so extensively used, modified and extended by researchers in the field of multi-agent coordination [Sandholm, 1993, Dignum et al., 2001], that it has been included in the standardization effort carried out by the Foundation for Intelligent Physical Agents (FIPA) [FIPA, 2002]. However, the suitability of the Contract Net protocol for open MAS is under analysis, as it seems to be very dependent on the value of the deadline used when waiting for bids, and in the number of agents [Juhasz and Paul, 2002]. Such unguided team selection involves an exponential number of possible team combinations, and a blow-out in the number of interactions required to select the members of a team. Some attempts to overcome these problems that still rely on some kind of global plan employ problem requirements to guide the team selection and reduce the number of possible teams [Tidhar et al., 1996].

Distributed Task Allocation

Anytime algorithms with low ratio bounds have been proposed based on distributed coalition formation algorithms [Shehory et al., 1997, Shehory and Kraus, 1998]. A coalition is defined as "a group of agents who have decided to cooperate in order to achieve a common goal". Given a set of tasks and agents, these algorithms search a combination of coalitions (overlapping or not) to solve each task, taking into account the agents limited resources.

Distributed task allocation methods are appropriate for DPS and cooperative MAS [Sycara et al., 1996], since agents cooperate to increase the overall outcome of the system. Non super-additive environments (in a super-additive environment any combination of two groups of agents into a new group is beneficial) consider also task dependencies (task precedence and competing resources). Other examples of distributed planning propose methods for coordinating plans

at abstract levels [Clement and Durfee, 1999] by using information about how abstract plans can be refined in order to identify and avoid potential conflicts.

In the *Team Formation by dialog* approach [Dignum et al., 2001] autonomous agents are able to discuss the Team Formation, using structured dialogues, with an emphasis on persuasion. Follows the four stages model of the Cooperative Problem-Solving process drawn in [Wooldridge and Jennings, 1999]. This approach is based on the Dialogue theory [Walton and Krabbe, 1995], that proposes to structure dialogues by rules, so dialogues are not completely free neither completely fixed. The initial situation of negotiation is a conflict of interests, together with a need for cooperation, where the main goal is to make a deal. This theory adopts an internal view on agents based on a BDI model and an architecture containing reasoning, planning, communication and social reasoning modules. The initiator agent makes a partial plan for the achievement of a goal and looks for potential teams based on abilities (static), opportunities (situational), and willingness.

2.4.3 Interoperation in open environments

One of the current factors fostering MAS development is the increasing popularity of the Internet, which provides the basis for an open environment where agents interact with each other to reach their individual or shared goals. To successfully communicate in such an environment, agents need to overcome two fundamental problems: first, they must be able to *find* each other (since agents might appear or disappear at any time), and once they have done that, they must be able to *interact* [Jennings et al., 1998].

Interaction is one of the most important features of an agent [Nwana and Woolridge, 1996]. It is in the nature of agents to interact to share information, knowledge and goals to achieve. Three key elements have been outlined for a successful interaction:

- A common agent communication language and protocol
- A common format for the content of communication
- A shared ontology

Furthermore, in open environment there is a need of mechanisms for advertising, finding, using, combining, managing and updating agent services and information [Decker et al., 1997b]. Next follow a very brief section on agent communication topics and a review of middle agents as one way of implementing the aforementioned requirements.

Agent Communication

Agents must share a communication language to be able to interoperate. There are two main approaches to agent communication languages [Genesereth and Ketchpel, 1997]: in the *procedural* approach communication

is based on executable content, which can be accomplished by using programming and scripting languages, e.g. Tcl [Ousterhout, 1990]. Since procedural languages are difficult to control, coordinate and merge, declarative languages are preferred for the design of agent communication languages, specially in open environments. Most declarative communication languages [FIPA, 2003, Finin et al., 1994, Labrou and Finin, 1997] are based on the *speech acts* theory [Searle, 1969]. For this approach, communication is modelled through illocutionary acts called *performatives* (e.g. request, inform, agree), which are conceptualized as actions intending to produce some effect on the receiver, like performing some task (request) or giving some information (query). Although such performatives can characterize message types, efficient languages for expressing message content so as to allow a meaningful communication, have not been effectively demonstrated [Jennings et al., 1998]; thus the problem of representing and sharing meaning through ontologies is still open [Gruber, 1993b].

Middle agents

A common approach to overcome the interoperability problems agents face in open environments is the introduction of a middleware layer between requesters and providers of services and the use of a shared language and ontologies for describing both the tasks to be solved and the capabilities available. Having a mediation service is very useful since problem solving agents can advertise their capabilities, and the requester may look for agents with the capabilities more appropriate for the problem at hand. Usually, the mediation layer is realized by *middle agents* [Decker et al., 1997b] specialized in reasoning about and supporting the activities of other agents.

Many ideas about middle agents have precedents in the work on *mediators*. The notion of mediators was initially proposed in the field of Information Systems. A foundational paper is [Wiederhold, 1992], which introduces *mediators* as a technique to handle large-scale information systems in open and distributed environments. Thus, it is not strange that current ideas on middle agents were initially applied in the field of Intelligent Information Integration [Wiederhold, 1993] and Information Brokering [Jeusfeld and Papazoglou, 1996, Martin et al., 1997] as a way to locate and combine information coming from multiple and heterogeneous sources, e.g. relational and object-oriented databases. The notion of mediators is also studied as a software pattern [Rising, 2000] and is used in multi-layer information architectures [Wiederhold and Genesereth, 1997]. Mediators and brokers have been originally conceived as providing an added-value services for information-based applications. A middle agent can be seen as a mediator between requesters and providers of services, in which the information being mediated is constituted by the description of available services. Some introductions to middle agents can be found in [Decker et al., 1996] and [Decker et al., 1997b], and a taxonomy of middle agents appears in [Wong and Sycara, 2000]. Below follows a brief review of approaches to middle agents, though there is not clear differentiation between

the different types considered.

- *Facilitators* are agents to which other agents surrender their autonomy in exchange of the facilitator's services [Erickson, 1996a, Genesereth and Ketchpel, 1997]. Facilitators can coordinate agents' activities and can satisfy requests on behalf of their subordinated agents.
- *Mediators* are agents that exploit encoded knowledge to create services for a higher level of applications [Wiederhold, 1992]. For a detailed account of the differences between mediators and facilitators see [Wiederhold and Genesereth, 1997].
- *Matchmakers* and *yellow pages* assist service requesters to find service providers based on advertised capabilities. Services found that matches a given request are communicated to the requester, thus it must choose and contact the selected provider directly. [Decker et al., 1996].
- *Brokers* are agents that receive requests and are able to contacting with appropriate providers on behave of the requester. Thus, tasks are delegated to brokers that locate and communicate with suitable by themselves, freeing the requester of knowing the details required to communicate with a specific provider. The difference between brokers and matchmakers is that the matchmaker only introduces matching agents to each other, whereas a broker handles all the communication with the capability providers [Decker et al., 1996].
- *Blackboards* are repository agents that receive and hold requests for other agents to process [Nii, 1989].

Preliminary experiments [Decker et al., 1997b] shows that each type of middle agent have its own performance characteristics and is best suited for a certain type of environment. For example, while brokered architectures ar more vulnerable to failures, they are also able to cope more quickly with a rapidly fluctuating agent work-force. A general problem with the existing systems is that they do not overcome the gap between *push* and *pull* access to information [Haustein and Ludecke, 2000], since they commit to only one access model. There is, however, a considerable interest in combining both ways of accessing information. By caching data from a "push" source, a combination of a mediator and a broker could solve the access mismatch problem, providing both access modes.

Matchmaking and Agent Capability Description Languages

Typically, the function of middle agents is to match service-requests with service providers, where services are provided by agents. To enable matchmaking, both providers and requesters should share a common language to describe both service-requests and agent capabilities, which is called an Agent Capability Description Language (ACDL) [Sycara et al., 2001] (called also an Agent Service

Description Language, due to the quite usual view on agent capabilities as "services" provided to clients).

Matchmaking is the process of finding an appropriate provider of capabilities (or services) for a requester [Sycara et al., 1999a, Sycara et al., 1999b]. Some ACDLs supporting matchmaking are reviewed below, namely: the Logical Deduction Language (LDL++), the Interagent Communication Language (ICL), the Language for Advertisement and Request for Knowledge Sharing (LARKS), and the DARPA Agent Markup Language (DAML-S).

- LDL++ is a logical deduction language similar to Prolog that is used by brokers in the Infosleuth [Nodine et al., 1999] distributed agent architecture. LDL++ supports inferences about whether an expression of requirements matches a set of advertised capabilities.

- ICL is the interface, communication, and task coordination language shared by OAA agents, regardless of what platform they run or on what computer language they are programmed in [Martin et al., 1999, Cheyer and Martin, 2001]. OAA agents employ ICL to perform queries, execute actions, exchange information, and manipulate data in the agent community. ICL includes a layer of conversational protocols (such as KQML or FIPA), and a content layer. The content layer has been designed as an extension of PROLOG, to take advantage of unification and other features of PROLOG. Every agent participating in an OAA-based system defines and publishes its capabilities expressed in ICL. These declarations are used by a facilitator to communicate with the agent and also for delegating service requests to the agent.

- LARKS [Sycara et al., 2002] (Language for Advertisement and Request for Knowledge Sharing) is a language used by matchmaking agents to pair service-requesting agents with service-providing agents that meet the requesting agents [Sycara et al., 1999a], and is used by agents in the RETSINA [Sycara et al., 2001] agent infrastructure. When a service-providing agent registers a description of its capabilities with a middle agent, it is stored as an "advertisement" and added to the middle agent's database. Therefore, when an agent inputs a request for services, the middle agent searches its database of advertisements for a service-providing agent that can fill such a request. Requests are filled when the provider's advertisement is sufficiently similar to the description of the requested service. LARKS is capable of supporting inferences. It also incorporates application domain knowledge in agent advertisements and requests. Domain-specific knowledge is specified as local ontologies in the concept language ITL.

- ATLAS (Agent Transaction language for Advertising Services) is a DAML-based agent advertising language that will enable agents and devices to locate each other and interoperate [Paolucci et al., 2002]. ATLAS is based in DAML-S, an ontology to annotate Web Services with semantic information

[The DAML-S Consortium, 2001]. The DAML-S ontology uses concepts that are similar to the purpose and the requirements of an Agent Capability Description Language. Therefore, we find very similar elements in both ACDLs and Semantic Web Services description languages. Specifically, the view of the DAML-S consortium is that DAML-S descriptions are used by agents in support of the automated discovery, interoperation, composition, execution and monitoring of services. A matchmaker for DAML-S matchmaking has been proposed that utilizes two separate filters: one compares *Functional Attributes* to determine the applicability of advertisements, and the other compares *Services Functionalities*. *Subsumption* is the inference operation used to determine if two specifications match.

There are other proposals for ACDLs, based upon some extension of Petri Nets, like *Possibilistic Petri Nets* [Jonathan Lee and Chiang, 2002], the object-based extension called *G-Net* [Xu and Shatz, 2001] and the constraint-based model of *fitness-for-purpose* [White and Sleeman, 1999], among others.

It is also interesting to review other research on capability descriptions not specifically designed to describe agent capabilities, although they can be adapted for that purpose, like skills modelling in human organizations [Stader and Macintosh, 1999], capability descriptions for PSMs [Aitken et al., 1998], or process/action modelling techniques such as SPAR [Tate, 1998] and ADL [Pednault, 1989]. See [Wickler and Tate, 1999] for a wide survey on capability description for software agents.

Agent infrastructures for Cooperative Problem-Solving

There are several architectures and standards focused on open agent architectures and mechanisms to achieve interoperability. These infrastructures are based on some notion of mediation or middle agents: like *yellow-pages* in the FIPA abstract architecture , *matchmakers* in Retsina, *brokers* in OAA and task-planning agents in UMDL. However, there are some architectures oriented towards industrial applications, e.g. GRATE and ARCHON.

- *FIPA*[1] has produced a collection of specifications which aim is to become an standard for the interoperability of heterogeneous software agents. The standardization effort includes an *Abstract Architecture* dealing with the abstract entities that are required to build agent services and an agent environment. A FIPA platform contains a communication channel, an Agent Name Server (ANS) that is used as a "white pages" service, and a Directory Facilitator(DF), which acts as a "yellow pages" service.

- *UMDL*[2] provides a distributed architecture [Birmingham et al., 1995] for a digital library that can continually reconfigure itself as users, contents, and services come and go. This has been achieved by the development

[1]FIPA stands for the Foundation for Intelligent Physical Agents
[2]University of Michigan Digital Library.

of a multi-agent infrastructure with agents that buy and sell services from each other by using commerce and communication services/protocols [Vidal et al., 1998], that is called the Service Market Society (SMS). The SMS allows for the decentralized configuration of an extensible set of users and services [Durfee et al., 1998]. There are many types of agents in the UMDL agent architecture: there are information agents specialized in complementary knowledge areas; there are user interface agents that support the user in specifying queries; and there are also task planning agents [Vidal and Durfee, 1995] that are able to perform matchmaking between queries and agent services. Services are described in Loom.

- *RETSINA*[3] is an open multi-agent architecture that supports communities of heterogeneous agents [Sycara et al., 2001]. Distributed approach to information and problem-solving tasks (search, gathering, filtering, fusion, etc). The RETSINA system has been implemented on the premise that agents in a system should form a community of peers that engage in peer to peer interactions. Any coordination structure in the community of agents should emerge from the relations between agents, rather than as a result of the imposed constraints of the infrastructure itself. In accordance with this premise, RETSINA does not employ centralized control within the MAS; rather, it implements distributed infrastructure services that facilitate the interactions between agents, as opposed to managing them.

- *OAA*[4] [Cheyer and Martin, 2001] is a framework for building flexible, dynamic communities of distributed software agents. OAA enables a cooperative computing style wherein members of an agent community work together to perform computation, retrieve information, and serve user interaction tasks. Communication and cooperation between agents are brokered by one or more facilitators, which are responsible for matching requests, from users and agents, with descriptions of the capabilities of other agents [Martin et al., 1999].

- *DECAF* (Distributed, Environment-Centered Agent Framework) is a toolkit which allows a principled software engineering approach to building Multi-Agent Systems. The toolkit provides a platform to design, develop, and execute agents. DECAF provides the necessary architectural services of a large-grained intelligent agent: communication, planning, scheduling, execution monitoring, coordination, and eventually learning and self-diagnosis . This is essentially, the internal "operating system" of a software agent, to which application programmers have strictly limited access. The control or programming of DECAF agents is provided via a GUI called the Plan-Editor. In the Plan-Editor, executable actions are treated as basic building blocks which can be chained together to achieve a more complex goals in the style of a *Hierarchical Task Network*. This issue provides

[3]Reusable Environment for Task-Structured Intelligent Networked Agents.

[4]OAA stands for the Open Agent Architecture, developed at the SRI International's Artificial Intelligence Center (AIC)

a software component-style programming interface with desirable properties such as component reuse (eventually, automated via the planner) and some design-time *error-checking*. The chaining of activities can involve traditional *looping* and *if-then-else* constructs. This part of DECAF is an extension of the RETSINA and TAEMS task structure frameworks. Unlike traditional software engineering, each action can also have attached to it a performance profile which is then used and updated internally by DECAF to provide real-time local scheduling services . The reuse of common agent behaviors is thus increased, since the execution of agent behaviors does not depend only on the specific construction of the task network but also on the dynamic environment in which the agent is operating. Furthermore, this model allows for a certain level of non-determinism in the use of the agent action building-blocks.

GRATE [Jennings et al., 1992] is a general framework which enables to construct MAS for the domain of industrial process control. Embodies in-built knowledge related to cooperation, situation assessment and control. Designer can utilize this knowledge (reuse , configuration of preexisting knowledge) and augment it with domain specific information, rather than starting from scratch. More focused on agent architecture than MAS architecture. The in-built knowledge is represented by generic rules encoding sequences of actions. ARCHON [Wittig et al., 1994] is an extension of GRATE [Jennings et al., 1992] with reactive mechanisms. Many of GRATE's generic rules encode sequences of actions resulting in common patterns of rule firing, these patterns can be grouped into units of activity similar in nature to reactive planning systems (precompiled plans that behaves like in an unplanned, reactive manner). The result is a hybrid approach in which both general rules and reactive mechanisms are combined. ARCHON concentrates upon loose coupling of semiautonomous agents. There is no representation of an overall goal, but only goals of the agents that together met the overall goals of the community. ARCHON has been applied to pre-existing computational systems, although its concepts may well be used as enhancements to more conventional (e.g. client/server) integration architectures.

2.4.4 Social approaches

There are some aspects of complex system development that become more difficult by adopting an agent-based approach [Jennings, 2000]. Since agents are autonomous, the patterns and the effects of their interactions are uncertain, and it is extremely difficult to predict the behavior of the overall system based on its constituent components, because of the strong possibility of emergent behavior. These problems can be circumvented by imposing rigid and preset organizational structures, but these restrictions also limit the power of the agent-based approach. As an answer to these difficulties a *social level* view has been proposed [Jennings and Campos, 1997] that takes the *knowledge level* [Newell, 1982] anal-

ysis approach as a starting point. Whereas the knowledge level view stripped away implementation and application specific details from problem solvers, the social level view focuses on the organizational aspects of agent societies with the primary goal of analyzing system behaviors abstracted from implementation details or specific interaction protocols [Jennings and Campos, 1997]. The GAIA methodology [Wooldridge et al., 2000] follows this approach which allows to describe agent-based systems as computational organizations that are defined in terms of roles, interactions and obligations.

The *Civil Agent Societies* (CAS) [Dellarocas, 2000] is a framework for developing agent organizations which follows the metaphor of civil human societies based on social contracts, and is oriented towards marketplaces and B2B e-commerce. The CAS approach uses the Contract Net interaction protocol, social norms, notary services and exception handling services.

Another social approach to Multi-Agent Systems is described in [Panzarasa and Jennings, 2001] that is based on a conception of cognition, both at the individual and the collective level, and examined in relation to contemporary organization theory. Yet another organization-oriented model for agent societies is found [Dignum et al., 2002].

Another view of open agent organizations is that of *electronic institutions* as a metaphor of human institutions. A electronic institution is a virtual place where agents meet and interact according to the communication policies and norms defined by the institution. Formalization of electronic institutions [Esteva et al., 2001] underpins the use of structured design techniques and formal analysis, and facilitates development, composition and reuse. ISLANDER is a formal language with a graphical representation that allows to define [Esteva et al., 2002a] and verify [Huguet et al., 2002] the specification of an electronic institutions. Another advantage of such a formal language is that, given the specification of an electronic institution, it is possible to generate skeletons for the development of agents for that institution [Vasconcelos et al., 2001].

2.4.5 Agent-Oriented Methodologies

Agent technology has received a great deal of attention in the last few years and is also beginning to attract the industry. But in spite of the extensive research and successful application of agent theories, languages and architectures, there is little work for specifying techniques and methodologies to develop agent-based applications. Furthermore, the usual approach to the development of agent-oriented methodologies have been to adapt or extend an existing methodology to deal with the relevant aspects of agent-oriented programming [Iglesias et al., 1998]. These extensions have been carried out mainly in three areas: *object-oriented* methodologies, *software engineering* and *knowledge-engineering*.

Extensions of Object-Oriented Methodologies

There are several reasons to use object-oriented methodologies as the basis for an agent-oriented methodology: (1) there are many similarities between both paradigms [Burmeister, 1996, Kinny and Georgeff, 1996], and specifically, there is a close relationship between DAI and object-based concurrent programming [Bond and Gasser, 1988b, Gasser and Briot, 1992, Yoav Shoham, 1993]; and (2) there is a considerable experience is using object-oriented languages to implement agent-based systems.

Some examples of agent-oriented methodologies based on OOP are the following: Agent-Oriented Analysis and Design [Burmeister, 1996], Agent Modelling Technique for Systems of BDI agents [Kinny and Georgeff, 1996], Multi-Agent Scenario-Based Method (MASB) [Moulin and Brassard, 1996] and Agent Oriented Methodology for Enterprize Modelling [Kendall et al., 1995]

However, there are some aspect of agents not addressed by object oriented methodologies [Burmeister, 1996, Yoav Shoham, 1993, Kendall et al., 1995]: (1) the agent style of communication can be much more complex than the method invocation style in OOP; (2) agents can be characterized by their mental state; and (3) agents can include a social dimension not existing in OOP.

Extensions of Knowledge-Engineering Methodologies

Knowledge engineering methodologies can provide a good basis for MAS modelling by exploiting a human inspired style of problem solving. Since agents have cognitive features, the experience achieved in *knowledge acquisition* and *knowledge modelling* methodologies can be applied to agent development. In addition, existing tools and libraries of Problem-Solving Methods [Breuker and Van de Velde, 1994] can be reused.

CoMoMAS [Glaser, 1996] is an extension of CommonKADS [Schreiber et al., 1994a] for MAS modelling. The following models are defined:

- The *Agent Model* defines the agent architecture and the agent knowledge, that is classified as social, cooperative, control, cognitive or reactive knowledge.
- The *Expertise Model* defines the cognitive and reactive competencies of agents, distinguishing between tasks, problem solving (PSM) and reactive knowledge.
- The *Task Model* describes the task decomposition and details if a task is solved by an user or an agent
- The *Cooperation Model* specifies the communication primitives and interaction protocols required to cooperate and resolve conflicts
- The *System Model* defines the organizational aspects of the agent society together with the architectural aspects of agents

- The *Design Model* collects the previous models and captures the non functional requirements required to operationalize them.

MAS-CommonKADS [Iglesias et al., 1997] extends the models defined in CommonKADS adding techniques from object-oriented methodologies (OOSE, OMT) and from protocol engineering (MSC and SDL). This methodology starts with an informal *conceptualization phase* used to obtain the user requirements and a first description of the system from the user point of view. For this purpose, *use cases* from OOSE are used, and its interactions are formalized with Message Sequence Charts. Then, the different models described below are used for analysis and design of the system, that are developed following a risk-driven life cycle. For each model the methodology defines the constituents (entities to be modelled) and the relationships between the constituents. A textual template and a set of activities for building each model are provided according to a development state (empty, identified, described or validated). MAS-CommonKADS defines the following models:

- *Agent model*: describes the main characteristics of agents, including capabilities, skills (sensors/effectors), services, goals, etc.
- *Task model*: describes the tasks (goals) carried out by agents and tasks decomposition, using textual templates and diagrams.
- *Expertise model*: follows the KADS approach, that distinguishes domain, task, inference and problem solving knowledge. The MAS-CommonKADS methodlogy proposes a distinction between autonomous PSMs, that can be carried out by the agent itself, and cooperative PSM, that decompose a goal into subgoals that are carried out by the agent in cooperation with other agents.
- *Coordination model*: describes the conversations between agents. A first milestone is intended to identify the conversations and the interactions. The second milestone is intended to improve conversation with more flexible protocols such as negotiation and identification of groups and coalitions. The interactions are specified using MSC (Message Sequence Charts) and SDL (Specification and Description Language).
- *Organization model*: describes the organization in which the MAS is going to be introduced and the organization of the agent society. The agent society is described using an extension of the object model of OMT, and describes the agent hierarchy, the relationship between the agents and their environment, and the agent society structure.
- *Communication model*: details the human-software agent interactions, and the human factors for developing these user interfaces.
- *Design model*: collects the previous models and is subdivided into three submodels: *application design*, *architecture design*, and *platform design*. The application design is about the composition or decomposition of the

agents according to pragmatic criteria and selection of the most suitable agent architecture for each agent. The architecture design deals with the relevant aspects of the agent network: required network, knowledge and telematic facilities. The platform design refers to the selection of the agent development platform for each agent architecture.

Other approaches

Several formal approaches have tried to bridge the gap between formal theories and implementations [d'Inverno et al., 1997]. Formal agent theories are specifications that allow the complete specification of agent systems. Though formal methods are not easily scalable [Fisher et al., 1997], they are specially useful for verifying and analyzing critical applications, prototypes and complex cooperating systems. Some examples include the use of Z [Luck et al., 1997], temporal modal logics [Wooldridge, 1998], and DESIRE [Brazier et al., 1997]. DESIRE (DEsign and Specification of Interacting REasoning components) proposes a component-based approach to specify the following aspects: task decomposition, information exchange, sequencing of subtasks, subtask delegation and knowledge structures. It is well suited to specify the task, interaction and coordination of complex tasks and reasoning capabilities in agent systems.

During the development of Multi-Agent Systems some developers have adopted a software engineering approach that can be used as the basis for an agent oriented methodology. Although in some cases they have not explicitly defined an agent oriented methodology, they have given general guidance, and in some other cases, they have proposed a methodology based on their personal experience. Some examples are:

- In *ARCHON* [Wittig et al., 1994], the analysis combines a *top-down* approach, that identifies the system goals, the main tasks and their decomposition, and a *bottom-up* approach, that allows the reuse of preexisting systems, thus constraining the top-down approach. The design is subdivided into *agent community design* (defines the agent granularity and the role of each agent) and *agent design* (encodes the skills for each agent).

- *MADE* [O'Hare and Woolridge, 1992] is a development environment for rapid prototyping of MAS. It proposes a methodology that extend the five stages of knowledge acquisition proposed in [Buchanan et al., 1983]: Identification, Conceptualization, Decomposition (added for agent identification), Formalization, Implementation and Testing (adding the integration of the MAS).

- The *AWIC* [Müller, 1996] method proposes an iterative design. In every cycle five models are developed: Agent model (tasks, sensors and actuators, world knowledge and planning abilities), World model, Interoperability model (between the world and the agents), and Coordination model (protocols and messages, study the suitability of joint plans or social structuring).

- The *Decentralizing Refinement Method* [Singh et al., 1993] proposes to start with a centralized solution to the problem. Then a general PSM is abstracted out. Next step is the identification of the assumptions made on the agent's knowledge and capabilities, and the relaxation of these assumptions in order to obtain a more realistic version of the distributed system. Finally, the system is formally specified. The method takes into account the reuse of the PSMs by identifying connections among parts of the problems and the agents that solve them.

2.4.6 Conclusions

Most research in the field of Cooperative Problem Solving (CPS) falls within the context of the Cooperative Problem Solving process [Wooldridge and Jennings, 1994], with four stages: recognition, Team Formation, planning and execution. In this framework the problem solving process starts with an agent willing to solve a task and realizing the potential for cooperation, but the process of deciding the goals to achieve and the way to achieve them is skipped, assuming that they are provided by the user [Wooldridge and Jennings, 1999]. Moreover, task allocation among cooperating agents is typically based on a preplan that decomposes a task into subtasks [Shehory and Kraus, 1998], without specifying the algorithms to build such a plan, neither the criteria to be taken into account, e.g. the Planned Team Activity [Sonenberg et al., 1994] and the SharedPlans approach [Grosz and Kraus, 1996]. Our work focuses on the feasibility and utility of a componential approach to build such initial plans using the the knowledge-level description of the Multi-Agent system (i.e. building a configuration of tasks, capabilities and domain knowledge).

When addressing the problem of designing the behavior of a Multi-Agent System we agree with other researchers that users matter [Erickson, 1996b]: people may need to understand what happened and why a system alters its responses, have some control over the actions of the system, even though agents are autonomous, or predict the overall system behavior. There is a need for methods to guide the Team Formation process according to stated problem requirements and user needs. Existing frameworks for developing cooperative MAS assume that both team plans and individual plans are known beforehand, and the stage called "planning" in fact is a re-planing stage, because agents refine the initial plans until a agreed plan is decided.

We claim the need of a framework with two integrates the problem specification within the CPS process; and second, provides a fully automated configuration process (equivalent to build a plan) of a MAS on-demand. The idea of the configuration process is to build a hierarchical TMD structure encompassing the capabilities required for a Team to solve a particular problem, and this process is driven by problem requirements in place of beforehand plans. We contribute to this issue by introducing a Knowledge Configuration process as the initial stage of the Cooperative Problem Solving process. Such a TMD configuration is an extension of the idea of matchmaking to deal with the domain, which facilitates

the reuse of existing capabilities not only for new requirements, but also for new application domains.

There is another MAS topic we have to have a closer look, *matchmaking*. While existing frameworks support matchmaking between tasks and capabilities, we are also considering matchmaking among capabilities and domain models in order to enhances the reusability of agent capabilities across different application domain. Moreover, the ORCAS Knowledge Configuration process fills the gap between the matchmaking process, that pairs a specification to a capability, and the global configuration of a team plan, which is required to solve the "bottom-up design problem" (§2.2): *given a set of requirements, find a set of agent capabilities and domain knowledge within a MAS whose aggregated competence satisfies the requirements*. The fundamental difficulty when considering this problem is how to decompose the requirements in such a way as to yield component specifications (i.e. capabilities and domain-models). Our approach to this problem is to search the space of all possible compositions of components —configurations— until one satisfying the requirements is found. Moreover, our work has extended the Knowledge Configuration process to apply Case-Based Reasoning during the selection of components. The idea is that past experience can be used to improve the search process by guiding the exploration of possible configurations according to the similarity of the current problem to past configuration problems. The reader is referred to §4.4 for the general configuration strategy and §4.5 for the case-based configuration approach

Concerning the topic on Agent Capability Description Languages (ACDL), most approaches distinguish among tasks (or goals) and capabilities (or services), but these components are tightly coupled to a particular application domain. Our approach to improve the reuse of agent capabilities is using the TDM approach for describing agent capabilities and tasks in a domain independent manner, abstracted from the application domain. This decoupling of capabilities and domain is enabled by allowing agent capabilities and domain knowledge to be described independently, as proposed by TMD frameworks to specify knowledge-systems. Capability descriptions are compared to domain-models during the Knowledge Configuration process to verify that the domain knowledge satisfies the capability assumptions, and this process is fully automated to enable the on-demand, on-the-fly configuration of the MAS.

In addition to the reuse issue, there are another aspects of MAS design that can benefit from a Knowledge Modelling framework:

- Domain knowledge acquisition is facilitated by an abstract description or model of the domain knowledge required for some application, that can be used as a guide during the knowledge acquisition process.

- Problem specifications are also oriented by the abstract level description of a system, which facilitates the task of posing appropriate problem requirements to the user.

Moreover, there are some limitations of TMD frameworks to be applied in MAS configuration. These methodologies conceive a knowledge system as a cen-

tralized one, without considering neither the social dimension of agents, nor the issue of autonomy. On the other hand, agents are autonomous entities that can decide if accepting or refusing requested actions. Consequently, TMD frameworks must be extended to allow some kind of distributed control and coordination rather than applying a centralized control schema. On the other hand, usually agents interoperate through a conversational model, using speech-acts to communicate the purpose of a message, and following structured interaction protocols. In consequence, and to sum up, our framework should deal with specific agent properties, deserving special attention to the fact agents are autonomous and communication is based on speech-acts and interaction protocols.

Concerning software libraries and MAS, the idea of reusable software libraries is related to concepts from Multi-Agent Systems, like *directory facilitator* or *yellow pages* services. In addition, selection of components in software reuse is defined in terms matching, in a similar way to the *matchmaking* process as performed by middle agents in open MAS. Our approach aims at integrating research on software libraries and reuse together with recent work on open agent architectures, more specifically, we propose the following ideas:

- that agent capabilities can be registered and managed as components in a library (or a middle agent), and can be queried by other agents in order to locate them;
- and knowledge modelling can be used to describe the components with semantic information, abstracting them from implementation details in order to maximize reuse.

2.5 Semantic Web services

Semantic Web Services are defined as "self contained, self describing modular applications that can be published, located and accessed across the Web" [Tidwell, 2000], and also as "loosely coupled, reusable software components that semantically encapsulate discrete functionality and are distributed and programmatically accessible over standard internet protocols" [McIlraith et al., 2001]). Today's Web was designed primarily for human interpretation, to be used as a repository of information. But nowadays the Web is becoming also an open environment for distributed computing, where new applications can be built by assembling information services on-demand from a montage of networked legacy applications and information sources [International Foundation on Cooperative Information Systems, 1994].

Web services technologies are beginning to emerge as a defacto standard for integrating disparate applications and systems [Peltz, 2003]. Nevertheless, most Web service interoperation is realized through APIs that incorporate hard code to locate and extract content from HTML pages. These mechanisms are not suited to deal with an open, changing environment like the Internet. In order to implement reliable, large scale interoperation of Web services it is fundamental to make such services computer interpretable, which could be achieved by

creating a Semantic Web [Berners-Lee et al., 1999] of services whose properties, capabilities, interfaces and effects are encoded in an unambiguous, machine understandable form. The realization of the Semantic Web is underway with the development of new markup languages with well defined semantics and able to manipulate complex relations between entities [Stab et al., 2003]. Some examples of such languages are OIL [Fensel et al., 2000], DAML+OIL [Horrocks, 2002] and DAML-S [The DAML-S Consortium, 2001].

Semantically annotated Web services can support the automated discovery, execution, composition and interoperation of services by computer programs or agents [McIlraith et al., 2001]. The distributed control nature of agents make them suited to become accessible through the Internet likewise services. From that view, agent capabilities can be seen as services provided to users or other agents through mediation services such as yellow pages; therefore, developing agent-based applications by reusing existing agent capabilities in open environments like the Internet will face the same problems encountered in the research on Semantic Web Services; and there is likewise a need for languages and infrastructures supporting the automated discovery, execution, composition and interoperation of agent capabilities.

Agent description languages could be used for describing Web services and viceversa with little adaptation effort. Markup of services exploits ontologies to facilitate sharing, reuse, composition, mapping and markup [Fensel et al., 1997, Fensel and Bussler, 2002]. Service description should include a functional view to enable automatic service discovery, a pragmatic view, or some kind of operational metrics are also very desirable, e.g. QoS [Cardoso and Sheth, 2002], but also a process model of the service designed to facilitate service composition. While there are a considerable level of consensus with respect to the functional aspects of a service (inputs, outputs, preconditions or prerequisites and postconditions or effects), and few standardization proposals (WSDL, DAML-S profile ontology), the panorama is quite different when talking about the process model. There are a lot of languages proposed for this purpose, including WSFL, XLANG, WSCI, BPML, BPEL4WS and YAWL [van der Aalst and ter Hofstede, 2002]. These languages are called Web Service composition languages, workflow languages, process languages, and Web Service orchestration languages.

An introduction to the Web Services approach can be found in [Vaughan-Nichols, 2002] and recent trends and controversies in [Stab et al., 2003].

2.5.1 Semantic Web Services Frameworks

For an overview of existing technologies and research directions on service description, advertising and discovery, the reader is referred to [Lemahieu, 2001]. A brief review of the most influencing proposals is provided below:

- The Web Service Modelling Framework (WSMF) [Fensel and Bussler, 2002, Bussler et al., 2002] proposes a conceptual

model for developing and describing services and their compositions. WSMF is based on maximal decoupling and scalable mediation services.

- DAML-S [The DAML-S Consortium, 2001, Ankolekar et al., 2002] is a DAML+OIL [Horrocks, 2002] ontology for describing the properties and capabilities of Web Services. The purpose of the DAML-S ontology is to allow automatic reasoning about Web services in order to improve the discovery, access, composition and interoperation of Web services. The approach to do that is to enrich service markup with semantics. An example of semantic matching for service discovery is described in [Paolucci et al., 2002].

- DLML [Euzenat, 2001] proposes to describe Web services using a Description Logics Markup Language (DLML) for ensuring interoperability among semantically heterogeneous Web services. DL allows to formally define transformations(mappings), proof of properties, and checking of compound transformations.

- WSTL Web Service Transaction Language [Piresa et al., 2003]. Extends WSDL for enabling the composition of Web services.

- Web Services Semantic Architecture is based upon a mix of standard Web services technology (UDDI, WSFL and DAML-S) and the DAML-S ontology.

2.5.2 Composition and interoperation of Web services

The most extended approach for composing Web services is the use of a *workflow* language. But there is a lack of consensus [Wil M. P. van der Aalst, 2002], which results in a variety of languages to describe service composition from a workflow approach , like WSFL, XLANG, WSCI, BPML and more recently BPEL4WS (see for instance [Peltz, 2003] for a review of emerging technologies, tools and standards). A particular approach is to use Petri Nets as the computational formalism of a workflow language, like WRL [van der Aalst et al., 2001] and YAWL [van der Aalst and ter Hofstede, 2002]. A great challenge is to integrate heterogeneous services, which requires to address the interoperability issue not only at the syntactic level, but also at the semantic level. For example, [Cardoso and Sheth, 2002] describes an ontology-based approach for the discovery and integration of heterogeneous Web services within workflow processes. This approach takes into account both functional and operational requirements (Quality of Services) and provides some algorithms to measure the degree of integration.

Action-based planning [Wil M. P. van der Aalst, 2002] is an approach to services composition that views Web Services as a collection of actions available to build a plan. An example of this approach is demonstrated by ConGolog [McIlraith et al., 2001, McIlraith and Son, 2001], a concurrent logic programming language based on *situation calculus*. ConGolog can be used to program

agents which can perform simulation and execution of composite Web services customized for the user.

Transactional approaches are inspired by a business perspective; for example, in [Piresa et al., 2003] a multi-layered mediated architecture is presented together with a language to compose Web services from a transactional viewpoint.

To conclude, there is yet another approach that thinks of Web services as behavioral extensions of agent capabilities. From this point of view, Web services are located, invoked, composed and integrated by intelligent software agents [Bryson et al., 2002, McIlraith and Son, 2001]. DAML-S is proposed as a language for the semantic markup of Web services. Semantic markup of services could enable to apply simulation, verification and automatic composition of services, for instance using Petri Nets [Narayan and McIlraith, 2002]. An agent-oriented architectural framework for Web services based on the notion of a Flexible Agent Society (FAS) has been proposed [Narendra, 2003] based on the Contractual Agent Society (CAS).

2.5.3 Conclusions

Semantic Web Services (SWS) is an emergent field. While there is a well defined approach to describe Semantic Web Services and some generalized standards, there is still much discussion on the operational description of services, and there are many proposals of frameworks for the composition of complex services. There are some notable similarities when comparing SWS frameworks and Agent Capability Description Languages used in open MAS. There are, however, some notable differences between both fields: While SWS are passive entities that are executed by direct invocation, agent capabilities are provided by autonomous agents that can decide autonomously whether to apply or not to apply a required capability, or the terms of commitment when accepting some request. Consequently, ACDLs can benefit only partially of SWS research, specifically, ACDLs and SWS languages share may common requirements, but differ on the operational aspects of composition: while SWS are well suited for a centralized control, autonomous agents are appropriate for a distributed control style.

SWS frameworks are designed to facilitate reuse and composition of services to achieve more complex tasks within a concrete domain, but SWS are not oriented towards domain-based reuse. Typically the knowledge used by a SWS is encapsulated or hidden, thus it is not possible to reason about the domain knowledge, neither to use a service for a new domain. Configuring a MAS on-demand from reusable capabilities and knowledge, requires a language for describing and reasoning about a MAS at an abstract level. But, although this level could be provided by semantic languages and compositional frameworks based in either SWS or TMD frameworks, there is required much work to integrate these approaches with cooperative MAS. Work on these complementary approaches is the core of this thesis.

Chapter 3

Overview of the ORCAS framework

This chapter provides an overall view of the ORCAS framework that accounts for its multi-layered structure, and highlights the outstanding points.

Now that we have reviewed the most relevant bibliography, it is time to summarize which are the open problems we are dealing with and the kind of solutions we propose; to get a view of the tree, in order to avoid getting lost when accounting for the leaves.

The main goal of this thesis is to provide a framework for open Multi-Agent Systems that maximizes the reuse of agent capabilities through multiple application domains, and supports the automatic, on-demand configuration of agent teams according to stated problem requirements.

There are some agent infrastructures relying on formal languages for describing both available capabilities and requests to solve problems using those capabilities; these infrastructures are usually based on middle agents to handle the interoperation issues. Middle agents — matchmakers and brokers— are able to match requests to advertised capabilities in order to find appropriate capability providers. However, no mechanism has been proposed to design the competence of a team in such a way that global problem requirements are satisfied. Therefore, automated design mechanisms supporting the on-demand, configuration of agent teams are required beyond existing matchmaking algorithms. Our proposal is to introduce a Knowledge Modelling Framework (KMF) to describe agent capabilities at a domain independent level, and to apply a compositional approach to design Multi-Agent Systems on-demand. The ORCAS KMF is based on the Task-Method-Domain paradigm, which distinguishes three classes of components: *Tasks*, *Problem-Solving Methods* and *domain-models*. In the ORCAS KMF we consider also three classes of components, namely agent *capabilities* (corresponding to Problem-Solving Methods), application *tasks* and *domain-models*.

The idea of the ORCAS KMF is to apply an abstract compositional architecture to configure agent-based applications on-the-fly, by selecting and connecting components that satisfy the requirements of each specific problem. To do so, we have transferred the "bottom-up design problem" [Mili et al., 1995] from the field of software reuse (§2.3) to the field of Multi-Agent Systems. As a result, we have defined the problem of designing a team as follows:

> given a set of requirements, find a set of agent capabilities and domain-models whose combined competence satisfy the requirements.

The main difficulty when considering a 'bottom-up design problem" is how to decompose the requirements in such a way as to yield component specifications. We adopt here an abstract view that approaches this problem as a search process, namely one that can be solved by a search process over the space of all possible component compositions, until one satisfying the requirements is found [Hall, 1993, Zhang, 2000]. Specifically, the composition of components is regarded as a composition of their specifications [Butler and Duke, 1998], which in our case are provided by the Knowledge Modelling Framework and consist of tasks, capabilities and domain-models.

Our proposal to configure a MAS is the separation of two layers in the configuration process: the knowledge and the operational layers. At the knowledge layer a MAS is described and configured in terms of component specifications and connections, at an abstract, implementation independent level. At the operational layer a team of agents is formed and each agent receives instructions on how to cooperate and coordinate in solving a problem according to this abstract (knowledge-level) configuration. Figure 3.1 shows the two layers MAS configuration model.

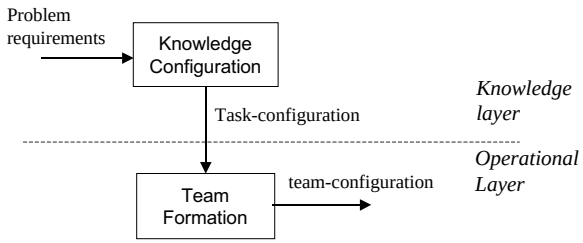

Figure 3.1: The two layers MAS configuration model.

(1) Configuration at the *knowledge layer* refers to the process of finding a configuration of components (tasks, capabilities and domain-models) adequate for the problem to be solved. We call this process *Knowledge Configuration*, and the result of the process is a *task-configuration*: a hierarchical structure of tasks, capabilities and domain-models satisfying the problem requirements. The Knowledge Configuration process is intended to take a specification of problem

requirements as input and producing a configuration of components such that the problem requirements are satisfied. The main goal of the Knowledge Configuration process is to determine which task decomposition, which competencies and which domain knowledge is required by a team of agents to solve a given problem.

(2) Configuration at the *operational layer* refers to the process of operationalizing a configuration into an executable system; in other words, to form a team of agents that is equipped with the capabilities and knowledge specified by a task-configuration (it means that the team is customized for the problem at hand). The operational configuration layer has the goal of ensuring that the multiple components involved in a task-configuration can interoperate and cooperate to solve a problem together.

There are some efforts on using a componential approach for the compositional-specification and design of intelligent agents [Decker et al., 1997a, Herlea et al., 1999, Splunter et al., 2003], and Multi-Agent Systems [Brazier et al., 2002], but these frameworks are mostly oriented to the development of agents, whereas our work is about forming and customizing agent teams on-demand, by composing components provided by already existing agents. We take the view on Multi-Agent Systems as decoupled networks of autonomous entities associated to a distributed model of expertise, regarded as a collection of specialized agents with complementary skills. According to this setting, a general framework for the Cooperative Problem Solving (CPS) process has been described [Wooldridge and Jennings, 1999] as having four stages: recognition (an agent identifies the potential for cooperation), team formation (the process of selecting team members), plan formation (collective attempts to construct an agreed plan) and execution (agents engage in cooperative efforts to solve the problem according to the agreed plan).

The result of separating the configuration of a MAS in two layers is a new model of Cooperative Problem Solving process that includes a Knowledge Configuration process before the Team Formation process. Moreover, the planning stage as presented in existing frameworks is removed from the ORCAS framework, since planning is usually associated to a particular agent architecture and is focused on internal agent processes, while our approach here is to take a macro-view, focused on the external, observable phenomena of teamwork, rather than imposing a particular agent model. In some aspects, the role played by the planning stage in the CPS process is substituted by the Knowledge Configuration process, since a task-configuration is a kind of global plan that will be used as a recipe to guide the team formation and to coordinate agents during the execution stage, that we call the Teamwork process (notice that the CPS model includes the Teamwork process within).

The ORCAS model of the Cooperative Problem Solving process comprehends four sub-processes, as showed in Figure 3.2, namely Problem Specification, Knowledge Configuration, Team Formation and Teamwork. The result of the Problem Specification process is a specification of a set of problem requirements to be satisfied, and problem data to be used during the Teamwork process.

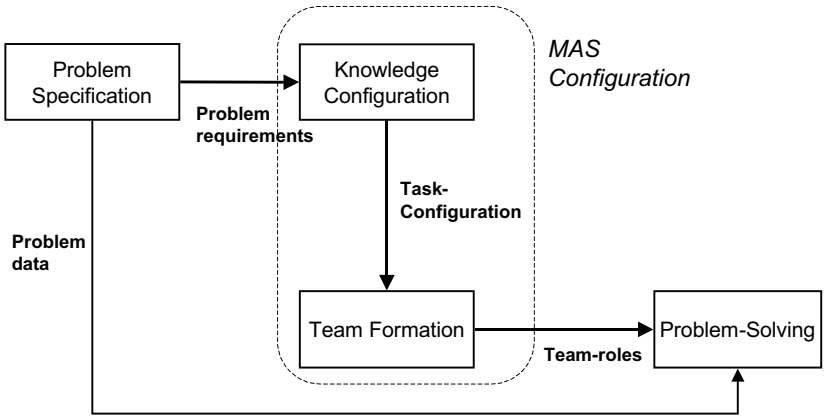

Figure 3.2: Overview of the ORCAS Cooperative Problem Solving process

The Knowledge Configuration uses the problem requirements to produce a task-configuration, which is used to guide the Team Formation process. Next, during the Teamwork process, the configured team resulting of the Team Formation stage applies the capabilities and knowledge specified in the task-configuration in order to solve the problem.

We claim that the separation between the knowledge (abstract) layer and the operational (computational) layer helps to distinguish between the static and dynamic aspects of agents to be taken into account during the configuration of a MAS. The idea is to exploit the fact that the abstract specification of agent capabilities remains stable over long periods of time, whereas there are dynamic aspects of the system or its environment that change very quickly, e.g. the agent workload or the network traffic. Therefore, it is useful to make a task-configuration in terms of a stable, abstract description of capabilities, and thereafter use the task-configuration to select the "best" candidate agents[1] according to dynamic and context-based information. While other frameworks and infrastructures focus on the task allocation stage carried on during Team Formation, the Knowledge Configuration process is situated just before Team Formation in the ORCAS Cooperative Problem-Solving model.

However, the CPS model should not be understood as a fixed sequence of steps, it is rather a process model of CPS. Thus, although the Knowledge Configuration is situated before Team Formation in the model, this does not imply that the Knowledge Configuration process should be fully completed before Team Formation begins. In fact, we have implemented strategies that interleave both activities with Teamwork, enabling distributed configuration, lazy configuration and dynamic reconfiguration on runtime (§5.7).

[1]The notion of agent goodness is specified as a criteria to be optimized, i.e. cost, speed, reliability, etc. and the possible trade-offs among them.

An interesting issue addressed by our framework concerns Team Formation in large systems: agent selection during Team Formation may involve an exponential number of possible team combinations, and a blow-out in the number of interactions required to select the members of a team. The performance of the Knowledge Configuration process brings about a task-configuration that can be used to guide the Team Formation process: only agents with capabilities selected during the Knowledge Configuration process are candidates to join a team, thus reducing the number of potential teams to be considered during the selection of team members, and drastically decreasing the amount of interaction required among agents. Throughout, the combinatorial problem is transferred (although mitigated) from the Team Formation process to the Knowledge Configuration process. Our proposal to further improve this issue is the use of Case-Based Reasoning to constrain the search over the space of possible configurations. The idea is to use past configuration problems (cases) to heuristically guide the search process over the space of possible configurations, as explained in §4.5.

Lastly, our objective is to develop an open agent infrastructure backing the on-demand configuration of Cooperative MAS according to stated problem requirements, based on both the Knowledge Modelling and the Operational Framework. Our purpose is to provide an open but trustworthy infrastructure where to test the proposed frameworks. Specifically, we propose to introduce a social mediation layer where specialized agents provide the services required to perform the different processes of the CPS model: Problem Specification, Knowledge Configuration, Team Formation and Teamwork. Since we want to avoid imposing architectural constrains over individual agents, we adopt a macro-view centered on the interaction protocols and communication language rather focusing on any particular agent architecture. Therefore, we decided to use a social oriented approach to build an open agent infrastructure, and more specifically, we adopted the *electronic institutions* (§2.4.4) approach.

An electronic institution is an infrastructure providing the mediation services required for agents to successfully interact in open environments under controlled conditions. This means that the institution imposes some constrains over the agents observable behavior. In other words, an electronic institution provides the specification of the *rules of encounter* for a successful interaction. The theoretical loss of autonomy that supposes joining an electronic institution brings, however, the advantage of allowing external agents to be more informed about other agents, since the overall system behavior becomes more predictable by virtue of the institution. The ORCAS infrastructure has been designed and implemented as an electronic institution, that we call the ORCAS e-Institution. In the ORCAS e-Institution requesters and providers of capabilities join and interact by using the services provided by institutional agents. Institutional agents are middle agents offering services beyond the usual matchmaking service. In particular, the ORCAS e-Institution includes agents that are able of: (1) keeping a repository (a library) of components available in a MAS, including application tasks, agent capabilities, and domain-models; (2) configuring the task to be achieved by a team in order to solve a given problem, according to an ab-

stract specification of problem requirements; (3) forming and instructing agent teams that are customized for each particular problem; and (4) coordinating team members behavior during the Teamwork process.

The main outcome of this work is a two-layered framework for integrating Knowledge-Modelling and Cooperative Multi-Agent Systems together, called ORCAS that stands for Open, Reusable and Configurable multi-Agent Systems. The feasibility of the ORCAS framework has been demonstrated by implementing the ORCAS e-Institution, an infrastructure for MAS development and deployment that supports the on-demand configuration of teams and the coordination of agent behaviors during teamwork, according to the ORCAS two-layered framework. The applicability of the framework has been tested by building WIM a multi-agent application running upon the ORCAS e-Institution. WIM is a configurable MAS application to look for bibliographic references in a medical domain, and is explained in Chapter 7. We show that the clear separation of layers will support a flexible utilization and extension of the framework to fit different needs, and to build other infrastructures different from the implemented ORCAS e-Institution.

These are the two layers of the ORCAS framework, namely the Knowledge Modelling Framework, and the Operational Framework:

1. The *Knowledge Modelling Framework* (KMF) (Chapter 4) is about the conceptual description of a problem-solving system from a knowledge-level view, abstracting the specification of components from implementation details. Our approach is to describe agent capabilities abstracted from implementation details in order to support the configuration of the MAS independently of the programming language and the agent platform. The configuration of a Multi-Agent System in terms of abstract (knowledge-level) descriptions is automated to enable the MAS configuration to occur on-demand, according to the requirements of the problem at hand. This layer consists of an Abstract Architecture, and Object Language and a Knowledge Configuration process:

    - The *Abstract Architecture* defines the different components used to model a MAS, which is based on the Task-Method-Domain-modelling paradigm; the features proposed to describe each component; and the functional relations that constrain the way components can be connected to become a meaningful system. The elements of the Abstract Architecture are explicitly declared as an ontology, called the Knowledge Modelling Ontology.
    - The *Object Language* is the language used to formally specify component features, and the inference mechanism used to reason about the specification of components (e.g. to determine if a capability is suitable to solve a task).
    - The *Knowledge Configuration* process is a search process aiming at finding a configuration of components (tasks, capabilities and domain-models) fulfilling the specification of the problem at hand. The result

of the Knowledge Configuration process is a hierarchical decomposition of the initial task into subtasks, capabilities bound to each task, and domain models bound to capabilities requiring domain knowledge. The result of the Knowledge Configuration process is called a *task-configuration*.

2. The *Operational Framework* (Chapter 5) describes the link between the characterization of the problem solving components and its implementation by Multi-Agent Systems. This framework describes how a task-configuration at the knowledge-level can be operationalized into a team of agents that is formed on-demand and is customized to satisfy the specific requirements of each problem. This layer introduces a *team model* based on the KMF, the ORCAS ACDL, the Team Formation process, and the Teamwork process.

   - The *Team model* describes the structure and the organization of teams according to a task-configuration as obtained by the Knowledge Configuration process. The team model establishes a mapping between KMF concepts and concepts from teamwork and Multi-Agent Systems.
   - The *Agent Capability Description Language* (ACDL) is a language for describing and reasoning about agent capabilities in such a way that enables the automatic location (by performing matchmaking), invocation, composition, and monitoring of agent capabilities. The ORCAS ACDL is defined as a refinement or extension of the Knowledge Modelling Ontology. Specifically, the ORCAS ACDL introduces an agent based formalism to describe the operational aspects of a capability: the *operational description* and the *communication*.
   - The *Team Formation* process deals with the selection of agents to join a team, and the instruction of the selected team members to solve a specified problem according to a task-configuration, as established by the Knowledge Configuration process. During the Team Formation process, the tasks required to solve a problem are allocated to suitable agents through a bidding mechanism, and selected agents receive instructions on which roles to play and how to cooperate with the rest of the team. The result is a collection of team roles and commitments of each team-member to their assigned roles that allows the team solving the overall problem.
   - The *Teamwork* process addresses the interaction and the coordination required for teams to successfully solve a problem according to requirements of a task-configuration. During the Teamwork process, the members of a team suitable for the problem (as established by the Team Formation process) engage in cooperative work until the global problem is solved, a reconfiguration of the team is needed, or it is impossible to solve the problem. Cooperation is basically a process of delegating subtasks to other agents in a team and aggregating

or distributing the results of the different subtasks at appropriate synchronization points.

	Architecture	*Language*	*Processes*
Knowledge Layer	Abstract Architecture	Knowledge-Modelling Ontology and Object Language	Problem Specification & Knowledge Configuration
Operational Layer	Team model	ACDL (Communication and Coordination)	Team Formation & Teamwork

Table 3.1: Summary of the two-layered framework

Table 3 is summarizes the main elements addressed at each layer, attending to three aspects: architectures, languages and processes. Furthermore, we can consider the ORCAS e-Institution as a third layer referring to a particular implementation of the former layers. We can then represent the two ORCAS layers plus the ORCAS e-Institution as a pyramid, as illustrated in Figure 3.3. The layer at the bottom addresses the more abstract issues, while upper layers correspond to increasingly implementation dependent layers. Therefore, developers and system engineers can decide to use only a portion of the framework, starting from the bottom, and modifying or changing the other layers according to its preferences and needs.

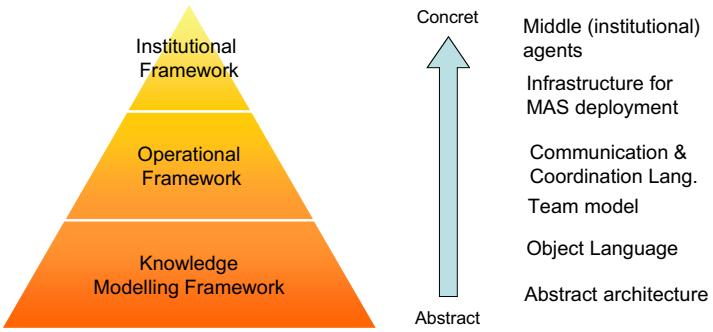

Figure 3.3: The three layers of the ORCAS framework

The structure of the thesis follows this direction from the most abstract layer to the more concrete, implementation dependent layers. Chapter 4 describes the KMF; Chapter 5 deals with the Operational Framework; Chapter 6 describes the implemented ORCAS e-Institution, and Chapter 7 demonstrates the feasibility of the framework through examples of an implemented application (WIM).

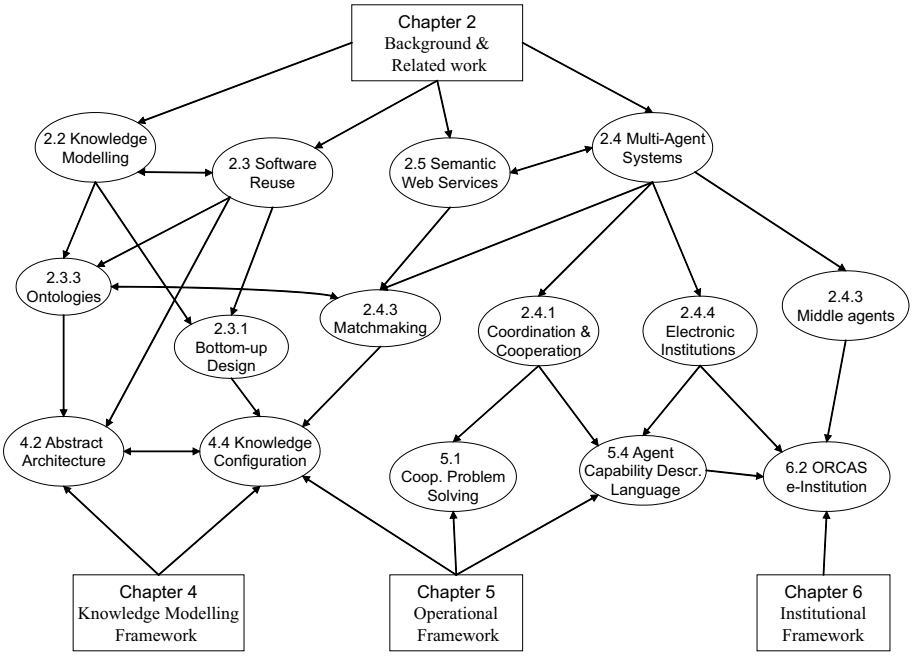

Figure 3.4: Cognitive map for the main topics involved in ORCAS

Figure 3.4 shows a map of the main topics addressed within the ORCAS framework and the relation with the subjects reviewed in the related work (Chapter 2).

Chapter 4

The Knowledge Modelling Framework

This chapter describes a framework to specify agent capabilities at the knowledge-level, which allows to reason about agent capabilities in order to design the competence of agent teams according to the requirements of the problem at hand.

4.1 Introduction

The *Knowledge Modelling Framework* (KMF) is a framework for describing and configuring Multi-Agent Systems from an abstract, implementation independent level.

The purpose of the KMF is twofold: On the one hand, the KMF is a conceptual tool to guide developers in the analysis and design of Multi-Agent Systems in a way that maximizes reuse; on the other hand, the KMF provides the basis of an Agent Capability Description Language supporting the automatic configuration of Multi-Agent Systems according to stated problem requirements.

The Knowledge Modelling Framework (KMF) proposes a conceptual description of cooperative Multi-Agent Systems at the *knowledge level*[Newell, 1982], abstracting the specification of components from implementation details. This framework is designed to maximize reuse and to support the formation and the coordination of customized agent teams during the *Cooperative Problem-Solving* process [Wooldridge and Jennings, 1999].

The KMF is the more abstract layer of the ORCAS framework for cooperative MAS. This layer consists of three main elements, namely: the *Abstract Architecture*, the *Object Language*, and the *Knowledge Configuration* process:

- The *Abstract Architecture* defines the types of components in the model, the features required to describe each component, and the relations constraining the way in which components can be connected. The ORCAS

Abstract Architecture is based on the Task-Method-Domain paradigm prevailing in existing Knowledge Modelling frameworks, which distinguish between three classes of components: *Tasks*, *Problem-Solving Methods* and *Domain-models*. In ORCAS there are tasks and domain models, while PSMs are replaced by agent capabilities, playing the same role than a PSM, but including agent specific features, like a description of a communication protocol and other features required of an Agent Capability Description Language. Nonetheless, in order to keep the KMF independent of agent details, these agent-specific aspects of ORCAS components remain unspecified here, and are described at the Operational Framework (Chapter 5).

- The *Object Language* defines the representation language used to formally specify component features. Many languages can be used as the Object Language, as far as they provide a way of specifying component features as *signatures* and *formulae*, and endorsing an inference mechanism enabling automated reasoning processes over component specifications.

- The *Knowledge Configuration* process is a search process aiming at finding a configuration of components (tasks, capabilities and domain-models) fulfilling the specification of the problem at hand. The result of the Knowledge Configuration process is a hierarchical decomposition of the initial task into subtasks called a *task-configuration*.

This chapter is organized as follows: Section §4.2 describes the components of the architecture and the relations between components that determines if two components can be connected (matching relations); Section §4.3 justifies the approach adopted here to the Object Language as a way to increase the flexibility of the ORCAS KMF, and introduces a specific Object Language; next, Section §4.4 describes the Knowledge Configuration process as a search process over the space of possible configurations; the specific approach for the Knowledge Configuration process using Case-Based Reasoning to guide the search process is described in §4.5; and finally, we end the chapter with a brief discussion on Knowledge Configuration reuse in §4.6.

4.2 The Abstract Architecture

The *Abstract Architecture* is a general modelling framework that is not specifically designed to describe Multi-Agent Systems, but to describe problem-solving (knowledge-based) systems in general. It rather plays the role of a skeleton that should be specialized or refined to deal with a concrete kind of software system, like Cooperative Multi-Agent Systems (CooMAS).

The main goal of the Abstract Architecture is to provide a way of specifying systems that maximizes the reuse of existing components and favors a compositional approach to software development. The Abstract Architecture specifies which are the components used to build an application (the "building blocks"),

and the way in which these components should be connected (the componential framework) in order to produce a valid application.

This architecture is intended to become a conceptual tool for the solution of the "bottom-up design problem" [Mili et al., 1995] and its application to the field of Multi-Agent Systems, stated as as one of main goals of this thesis (§1.1, bibliographic references in §2.3.1):

> given a set of requirements, find a set of agent capabilities and domain-models whose combined competence satisfy the requirements.

Each component in the Abstract Architecture is characterized by some features (i.e. inputs and outputs), but the particular language used to specify these features is independent of the Abstract Architecture, and belongs to the *Object Language*. The different components in the Abstract Architecture and the features characterizing them have been conceptualized and represented explicitly as an ontology, called the Knowledge Modelling Ontology (KMO). The KMO is used for analyzing, designing and describing problem-solving systems at an abstract, implementation independent level, thus it can be further refined to deal with specific architectures, including non agent-based architectures. In addition, this architecture is designed to facilitate the integration of heterogeneous systems whenever they share the same abstract architecture, i.e. using the same KMO for describing the components at the abstract level.

The Abstract Architecture enables a compositional approach supporting the on-the-fly configuration of Multi-Agent Systems at the knowledge level. A configuration is constructed by reasoning about the knowledge-level description of agent capabilities, tasks (goals) to achieve, and domain-models describing specific domain knowledge. From this approach, a system is described and configured by reusing and composing existing components. In particular, our view of components is based on the Task-Method-Domain (TMD) paradigm in Knowledge Modelling. TMD models propose three classes of components to model a knowledge system: *tasks*, *Problem-Solving Methods*(PSMs) and *domain-models*.

1. *Tasks* are used to characterize generic and reusable types of problems. This characterization is based on properties of the input, output, and nature of the operations that map the input to the output. Usually, there is a main task that describes an application problem, but tasks can be decomposed into subtasks with input/output relations between them, resulting in a task structure [Chandrasekaran et al., 1992].

2. *Problem-Solving Methods* (PSMs) specify the reasoning part of a knowledge system [Fensel et al., 1999]. PSMs are used to describe different ways of solving a task. A problem-solving method may decompose a task into subtasks or may apply a primitive inference step without further decomposing the task [Poek and Gappa, 1993]. PSMs can use domain knowledge to apply a reasoning step, create or change intermediary knowledge structures, perform actions to gather more data, etc. [Steels, 1990]. Problem-Solving Methods are described with independence of the domain knowledge

in order to maximize reuse [McDermott, 1988]. Some examples of problem solving methods are *hill climbing*, *cover and differentiate*, *propose and revise*, etc. [Breuker, 1994]

3. *Domain-models* describe domain specific knowledge that is used by Problem-Solving Methods to perform inference steps [Fensel et al., 1999]. Models can be constructed from different perspectives (for example, there are *functional models*, *causal models*, *behavioral models* and *structural models*) and represented in heterogeneous forms, like rules, hierarchies or networks [Steels, 1990].

4. *Ontologies* provide the terminology and its properties used to define tasks, problem solving methods and domain definitions [Fensel et al., 1999]. An ontology provides an explicit specification of a conceptualization, which can be shared by multiple reasoning components communicating during a problem solving process.

In our framework, tasks are used to describe the types of problems that a Multi-Agent System is able to solve. On the other hand, problem solving methods are used to specify the different capabilities agents are equipped with to solve tasks, wether they solve some task directly by applying some domain knowledge or by decomposing a task into subtasks and delegating them to other agents. While tasks are just generic problem specifications abstracted from any particular implementation, agent capabilities refer to concrete, implemented methods to solve problems that are provided by specific agents. Finally, domain-models are used to represent specific knowledge and information sources, whether the knowledge is provided by a shared repository, hold by an agent, or provided by an external information source.

The Abstract Architecture specifies the components and the way of connecting components that is necessary for the knowledge-level configuration of Multi-Agent Systems. Two kind of connections are included in our framework: on the one hand, connections between tasks and capabilities; on the other hand, connections between domain-models and capabilities. Such connections are based on the idea of *matching* and *mapping*. A matching relationship is a binary relation between two components that is verified by comparing its specifications. The verification of a matching relationship is often called *matchmaking* (e.g. matchmaking can be used to determine if two components are substitutable). On the other hand, a mapping function is an isomorphism between two specifications, in other words, a mapping is an explicit specification of the transformations required to match elements expressed in different terms —using different ontologies, with the same meaning (equivalent semantics).

In the ORCAS KMF the components —tasks, capabilities and domain-models— are described with explicit, independent ontologies [Fensel et al., 1997]. Because of this conceptual decoupling, ontology mappings may be required to match capabilities to tasks, and domain-models to capabilities when there is an ontology mismatch. Nevertheless, we will focus on the matching relations, assuming that the necessary ontology mappings are already

built, or assuming that all the components share the same ontologies. This is a reasonable assumption, since it is feasible and convenient to built the mappings beforehand, previously to make a component available for its use.

4.2.1 Components

There are three types of *knowledge components* in the Abstract Architecture (Figure 4.1), namely *tasks*, *capabilities* (playing the role of PSMs) and *domain-models*. Furthermore, each of these components is described using concepts defined at an explicit ontology, as indicated by the arrows in the figure. Herein every component can be specified using its own ontology, which entails that component specifications can be decoupled in order to maximize reuse (§4.2.1).

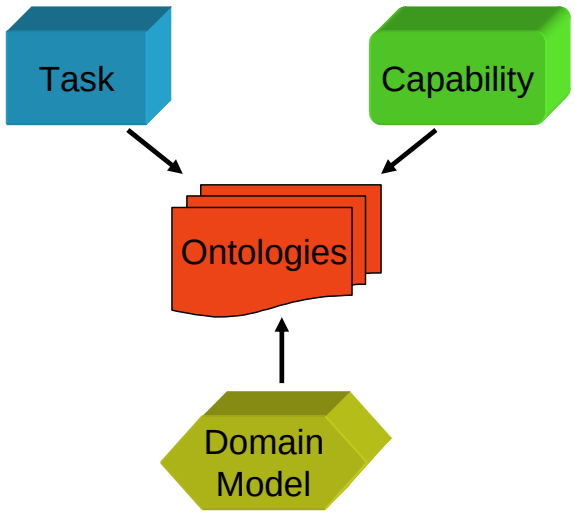

Figure 4.1: Components in the Abstract Architecture

In addition to describe component features using ontologies, the concepts used by the Knowledge Modelling Framework have been explicitly declared as an ontology, the so called Knowledge Modelling Ontology (KMO). Since the KMO is about components that are further specialized with specific component's ontologies, it can be seen a *meta-ontology* for describing software systems. Our approach is to keep a clear separation between the Abstract Architecture and the Object Language. Whilst the Abstract Architecture defines the components of the architecture and the features characterizing each component, the Object Language is used to describe component features in terms of *signature elements* and *formulae* in the Object Language.

These concepts in the KMO are organized into a hierarchy of sorts (Figure

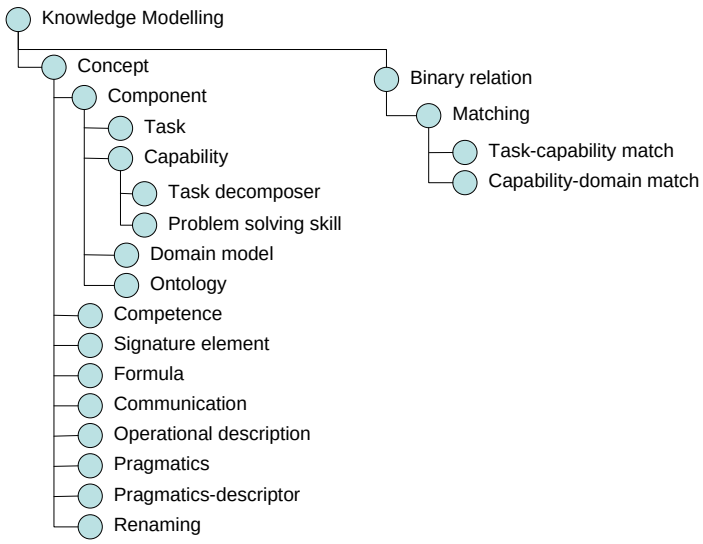

Figure 4.2: Hierarchy of sorts in the The Knowledge Modelling Ontology

4.2). There are two main sorts, namely *Component* and *Binary-Relation*, from which all the other sorts are specializations of. Most of these sorts have features that describe them, which are described by primitive types (e.g. string) or by other sorts in KMO (e.g. inputs and outputs are described with elements of the sort *Signature*).

Although the Knowledge-Modelling Framework is not dependent of any particular Object Language, the Knowledge Modelling Ontology declares two concepts that should be provided by the Object Language: *signature-elements* and *formulae*. Theses concepts are defined by the sorts *Signature-element* and *Formula* in the KMO, which should be further refined to yield a precise, interpretable meaning. In our search of a trade-off between expressive power and computational efficiency we are using *Feature Terms* as the Object Language, and subsumption as the inference mechanism (explained later, in §4.3.1).

All the components of the Abstract Architecture are subsorts of the sort Knowledge-Component. The description of elements of the sort Knowledge-Component contains a specification of pragmatics aspects of a component (e.g. name, description, creator, publisher, evaluation, etc.); and a collection of ontologies providing the terminology (the "universe of discourse") used to specify other features of a component. Moreover, the pragmatics slot can be customized by including new attributes (e.g. cost, performance measures, classification indexes, reputation, etc.) to better fit the needs or preferences of the developer.

Figure 4.3 shows the features characterizing the sort Knowledge-Component. The symbol **is-a** means subtype (subsort): *S* **is-a** *S'* means that the sort S' is a subtype (subsort) of the sort S'. The symbol → indicates the sort used to specify

Knowledge-Component **is-a** *Concept*
pragmatics → set-of *Pragmatics*
ontologies → set-of *Ontology*

Figure 4.3: The Knowledge-Component sort

a feature: f → S means that the feature f is specified as an element of sort S (an instance of S). The slot ontologies is defined as set of elements of the sort Ontology, and the pragmatics slot is described with an element of the sort *Pragmatics*. The sort *Pragmatics*, subsort of Concept (Figure 4.4) contains a number of specific features (name, creator, subject, description, publisher, etc.) specified by a String, and application descriptors, defined as a set of elements of the sort *Pragmatics-descriptor*, which are represented as attribute-value pairs (Figure 4.4). Thus, pragmatic-descriptors allow the definition and use of application specific attributes, which should be represented as attribute-value pairs.

Pragmatics **is-a** *Concept*
name → *String*
creator → *String*
subject → *String*
description → *String*
publisher → *String*
other contributor → *String*
date → *String*
format → *String*
resource identifier → *String*
source → *String*
language → *String*
relation → *String*
rights management → *String*
las date of modification → *String*
when and where be used → *String*
evaluation → *String*
application descriptors → *Pragmatics − descriptor*

Pragmatics-descriptor **is-a** *Concept*
attribute → *String*
value → *Any*

Figure 4.4: Sorts Pragmatics and Pragmatics-descriptor

Task

A *task* is a knowledge-level description of a type of problem to be solved. A task can also be seen as a set of goals characterizing "what" is to be achieve by solving some type of problem, in contrast to "how" that type of problem can be solved, which is represented as a capability. Since it is possible to have different ways of solving a problem, different capabilities might be able to solve a particular task.

The sort Task (Figure 4.5) is defined as a subsort of the sort Knowledge-Component. In addition to have a name, a description, pragmatics and ontologies, like any component, a task is described by a functional description, in terms of inputs, outputs and competence.

- The *inputs* feature represent the data required to solve the kind of problems represented by a task. This feature is specified by a set of elements of the sort *Signature* (see Figure 4.6), which is specified by a name, represented by a String; and a signature-element, of sort sort *Signature-element*). The sort *Signature-element* is kept undefined in the KMO, since it is refined by the sort Signature-element defined in the Object Language. The inputs can be constrained by preconditions in the competence.

- The *outputs* represent the type of data that is expected as a solution to the kind of problem a task represents. The *outputs*, like the inputs, are specified by a set of elements of the sort *Signature*. The outputs can be constrained by postconditions in the competence.

- The *competence* feature of a task expresses the relation between the input and the output. Since it is unfeasible to specify an input-output relation extensively (through an extensive list of input-output pairs), it is required some form of formal representation. In particular, we adopt the approach of specifying the competence as a set of *preconditions* and *postconditions* (figure 4.7). Preconditions are conditions that have to hold prior to solve a task, in order to enable it, while postconditions are the properties that have to hold after solving the task (also referred in the literature as the goals or the effects to bring about). Both preconditions and postconditions are represented as a set of elements of the sort *Formula*, which is refined by the sort Formula in the Object Language. By keeping the Formula and Signature-element sorts undefined in the KMO, the Object Language could vary without affecting the Abstract Architecture, neither the matching relations at the object level (the two levels of the matching relations are explained later, in §4.2.2).

Capability

A *capability* describes the reasoning steps required to solve a class of problems (a task), that is to say, it describes the process applied by the capability to the input in order to obtain the output of the problem, and the use of required

Task **is-a** *Knowledge-Component*
pragmatics → set-of *Pragmatics*
ontologies → set-of *Ontology*
inputs → set-of *Signature*
outputs → set-of *Signature*
competence → *Competence*

Figure 4.5: The Task sort

Signature **is-a** *Concept*
name → *String*
signature-element → *Signature-element*

Figure 4.6: The Signature sort, where the Signature-element sort is to be defined by the Object Langauge

Competence **is-a** *Concept*
preconditions → set-of *Formula*
postconditions → set-of *Formula*

Figure 4.7: The Competence sort

knowledge to solve it. There are two types of capabilities, *task-decomposers* and *skills*. Skills are used to describe primitive or atomic reasoning steps, that are not further decomposed, while task decomposers are used to describe complex reasoning processes that decompose a problem into more specialized subtasks.

As showed in Figure 4.8, the sort Capability is defined as a subsort of the sort Knowledge-Component, and as such, it has two features for defining the ontologies used by the capability and its pragmatics. In addition, a capability includes a functional description that is specified as a collection of *inputs*, *outputs* and the *competence*, which is an intensional description of the capability input/output relation. Another aspect of a capability is relative to the domain knowledge it requires to operate: in order to be domain independent a capability explicitly declares the type of knowledge it can operate with. The type of domain knowledge required by a capability is specified as a collection of *signatures* in the *knowledge-roles* feature. Moreover, a capability includes a specification of *assumptions*, which are properties that should be verified by a domain-model providing some knowledge-role, in order to be sensibly used by the capability. Furthermore, a capability includes another feature called *communication*, which is used to describe the technical aspects required to invoke and interact with a capability. Since the information provided by the communication slot depends on implementation details it is explained later, within the Operational Framework (Chapter 5).

Capability **is-a** *Knowledge-Component*
pragmatics → *Pragmatics*
ontologies → set-of *Ontology*
inputs → set-of *Signature*
outputs → set-of *Signature*
competence → *Competence*
knowledge-roles → set-of *Signature*
assumptions → set-of *Assumptions*
communication → *Communication*

Figure 4.8: The Capability sort

A more detailed description of the different features characterizing a capability (Figure 4.8) follows:

- The *inputs* feature represents the data required to apply the capability. This feature is represented by a set of elements of the sort *Signature*, which is defined by sort consisting of a name and a signature element. A signature element is defined by an element of the sort *Signature-element*, to be refined by a sort in the Object Language (Figure 4.6). *Inputs* can be constrained by the the preconditions specified in the competence feature.
- The *outputs* feature represents the type of the data that is expected as a result of applying the capability to solve the tasks it is suitable for.

The *outputs* are represented by elements of the sort *Signature* and can be constrained by postconditions in the competence feature.

- The *competence* of a capability represents the relation between the input and the output, and (as for tasks) is specified as a set of preconditions and postconditions (Figure 4.7). Preconditions are conditions that have to verify in order to enable the inference process provided by the capability, while postconditions are the new conditions produced by the application of the capability (also referred as the effects brought about), whenever the preconditions hold. Both preconditions and postconditions are specified as elements of the sort *Formula*, which is defined in the Object Language.

- The *knowledge-roles* are specifications of "inputs" to be provided by some domain knowledge. Knowledge-roles refer to concepts characterizing the application domain, and are used during the Knowledge Configuration to select domain models that are compatible with a capability. Only the domain models providing the concepts required by a knowledge-role are suitable for a capability. A knowledge-role is defined as an element of the sort *Signature*, like the inputs and outputs, and as such it is defined by the sort *Signature-element*, to be refined in the Object Language.

- The *assumptions* of a capability are necessary criteria for the achievement of the desired competence of the capability. Assumptions are conditions required from a domain model providing a particular knowledge-role to be sensibly used by the capability. Assumptions are associated to a particular knowledge-role, from those specified in the *knowledge-roles* feature of the capability. Assumptions are represented as elements of the homonym sort: *Assumptions* (Figure 4.9). An assumption is specified as a pair consisting of a knowledge-role, and a collection of conditions over the domain-model providing such knowledge-role. If a capability introduces more than one knowledge-role, then a domain-model is required to fill in every knowledge-role, and each domain-model has to verify the conditions associated to that knowledge-role.

- The *communication* slot defines technical information about the interaction protocol and the data format used to communicate with the provider of the capability. Since capabilities in ORCAS are provided by agents, the communication information is basically defined by the agent communication language and some kind of interaction protocol describing the pattern of communication between the requester and the provider of the capability. The information provided by the communication slot is required during the Cooperative Problem-Solving process to request an agent to apply a capability for solving a task. Since the communication property is closer than other aspects of a capability to the implementation details, it is avoided at the Knowledge Configuration process, and is instead presented as an operational property required for the ORCAS KMF to become

a full-fledged Agent Capability Description Language. Consequently, communication aspects are described in the Operational Framework (Chapter §5).

Assumptions
knowledge-role → *Signature*
conditions → set-of *Formula*

Figure 4.9: The Assumptions sort

Skills

A *skill* describes a primitive reasoning capability, without decomposing the problem to be solved into subproblems. The sort Skill (Figure 4.10) is defined as a subsort of the sort Capability. A skill does not need to introduce any new feature beyond the properties defined by the sort Capability. Therefore, a skill has pragmatics and ontologies inherited from the sort Knowledge-Component; and inputs, outputs, competence, knowledge-roles and assumptions inherited from the sort Capability.

Skill **is-a** *Capability*
pragmatics → *Pragmatics*
ontologies → set-of *Ontology*
inputs → set-of *Signature*
outputs → set-of *Signature*
competence → *Competence*
knowledge-roles → set-of *Signature*
assumptions → set-of *Assumptions*
communication → *Communication*

Figure 4.10: The Skill sort

Task-decomposers

A *task-decomposer* is a capability that decomposes a problem in subproblems. A task-decomposer capability describes how a task is decomposed into a number of subtasks which combined competence satisfies the postconditions of the capability, whenever the preconditions of the capability hold.

The sort Task-Decomposer (Figure 4.11) is defined as a subsort of the sort Capability. Like a capability, a task-decomposer is described with inputs, outputs, knowledge-roles, competence, assumptions and communication, but in addition a task-decomposer provides the *subtasks* in which it decomposes a task, and the

operational description of the problem-solving process, representing the control and data flow among subtasks.

- The *subtasks* feature specifies a collection of tasks the combined competence of which can achieve the competence defined by the task-decomposer. The subtasks feature is defined as a set of elements of the sort Task.
- The *operational description* feature specifies the reasoning steps applied by a capability in order to achieve its competence. In the Knowledge-Modelling Ontology, the operational description is defined as an element of the sort *Operational-Description*. Like the sort *Communication*, the sort *Operational-Description* is more closer to the implementation details than the other features characterizing a task-decomposer, and consequently it will be defined at the Operational Framework, as another element to be extend the KMF in order to become an Agent Capability Description Language (section 5.4.3).

Task-Decomposer **is-a** Capability
pragmatics → *Pragmatics*
ontologies → set-of *Ontology*
inputs → set-of *Signature*
outputs → set-of *Signature*
competence → *Competence*
knowledge-roles → set-of *Signature*
assumptions → set-of *Formula*
communication → *Communication*
subtasks → set-of *Task*
operational-description → *Operational-Description*

Figure 4.11: The Task-Decomposer sort

While other Knowledge Modelling frameworks describe the reasoning process of a capability in terms of inference steps, we prefer to describe complex capabilities in terms of a decomposition of interrelated subtasks (§5.4.3), without further specifying which capability is applied to solve each subtask (this is decided during the configuration of the task-decomposer at the Knowledge Configuration process). Therefore, an operational description should be understood as a template for decomposing a complex task into subtasks, without specifying the way each subtask is solved, only the way they are combined. This way of representing a task decomposition in terms of subtasks and not in terms of capabilities maximizes reuse and allows a flexible configuration of a task by assigning the most appropriate capabilities for each specified problem. A task-decomposer is then a problem decomposition schema that can be instantiated or configured on-demand, by selecting capabilities and domain-models suitable for a specific problem. This way of decomposing a problem into subtasks is a key element

of the Knowledge Configuration process backing the on-demand configuration of Multi-Agent Systems to satisfy specified problem requirements. We do not introduce now a concrete language for specifying the operational-description, since this feature, like the communication feature, pertains to the Operational Framework.

Both the *Communication* and the *Operational-Description* sorts are excluded from the Knowledge Configuration process and the Knowledge Modelling Ontology, thus they are described at the Operational Framework (Section §5.4). The idea is to keep the Knowledge Configuration process independent of the operational-level details of agent capabilities.

While a task decomposer corresponds to a *white-box* model of a capability, a skill is described using a *black-box* model. This consideration implies a weak notion of the difference between task-decomposers and skills according to the two ways of modelling a process: white-box vs black-box, i.e. choosing a skill to represent a capability does not necessarily mean that a very simple capability is being represented, but it can obey to some interest in hiding the details of a complex reasoning process. The purpose of introducing this consideration is to support a more flexible approach to the operational framework. For example, an agent may prefer to hide the details of a capability in order to keep it under local control. Therefore, autonomous agents may decide to declare a capability as a task-decomposer or as a skill according to its own preferences or to some conditions.

Domain-models

A *domain-model* (DM) specifies the concepts, relations and properties characterizing the knowledge of some application domain.

The sort Domain-Model (Figure 4.12) is defined as a subsort of the sort Knowledge-Component, and as such a domain-model has a name, a textual description, pragmatics and ontologies. Moreover, the sort *domain-model* introduces some new features, namely *knowledge-roles*, *properties* and *meta-knowledge*.

- The *knowledge-roles* define the concepts provided by the domain-model ontologies to describe domain knowledge that can be used by a capability. The knowledge-roles are specified as a set of elements of the sort Signature-element, to be refined by the sort Signature-element in the Object Language.

- The *properties* feature is used to represent properties verified by the knowledge characterized by the domain-model.

- The *meta-knowledge* feature represents properties of the knowledge that are assumed to be true though they cannot be verified.

Domain-models are used to explicitly describe the knowledge required by a capability to solve a problem. Therefore, the concepts defined by a domain-

model ontology should be understood by a capability in order to use the knowledge characterized by that domain-model, or there must exist a mapping between the concepts defined in the capability ontology and equivalent concepts in the domain-model ontology. Moreover, in addition to fill in the knowledge-roles of a capability with domain-models, the assumptions of a capability must be satisfied by the domain-models of choice. These assumptions can be verified by either the domain-model's properties or the meta-knowledge.

Domain-Model **is-a** *Knowledge-Component*
pragmatics → *Pragmatics*
ontologies → set-of *Ontology*
knowledge-roles → set-of *Signature-element*
properties → set-of *Formula*
meta-knowledge → set-of *Formula*

Figure 4.12: The Domain-Model sort

Ontologies

Concerning ontologies, we agree with [Guarino, 1997b] about the potential role of explicit ontologies to support reuse. Although a definition of what ontologies are is still a debated issue, it is a topic of active research in the AI community, and has been declared as a key issue in maximizing reuse [Fensel, 1997a, Fensel and Benjamins, 1998b]. From that view, the main goal of an ontology is to make knowledge explicit and sharable. Below follows a concrete definition that expresses worth the role played by ontologies in our framework.

> Ontologies are shared agreements about shared conceptualizations. Shared conceptualizations include conceptual frameworks for modelling domain knowledge; content-specific protocols for communicating among interoperating agents; and agreements about the representation of particular domain theories. In the knowledge sharing context, ontologies are specified in the form of definitions of representational vocabulary [Guarino, 1997b].

In ORCAS ontologies are used to explicitly declare the concepts used to specify the features characterizing a component, that are described in terms of two sorts provided by the Object Language: *Signature-element* and *Formula*. We adopt the usual approach to represent ontologies as hierarchies of concepts and relations. Specifically, we use the *Feature Terms* [Armengol and Plaza, 1997, Arcos, 1997] formalism to describe the sorts defined by an ontology, as described in §4.3.

In Feature Terms, concepts are organized as a hierarchy of *sorts*, and both descriptions and individuals as represented as collections of functional relations.

Any ontology used to specify components in ORCAS provides two basic sorts from which all the other sorts are descendant: the sort *Signature-element* and the sort *Formula*.

The sort Ontology is defined as a subsort of the sort Knowledge-Component (Figure 4.13), and as such it inherits the pragmatics feature, and the ontologies feature. But the information characterizing an ontology is mainly provided by a hierarchy of sorts below two main sorts: *Signature-element* and *Formula*. Therefore, an ontology includes a collection of sorts defined as subsorts of *Signature-element*, and a collection of sorts defined as subsorts of *Formula*. Furthermore, an element (an instance) of the sort *Ontology* can import other ontologies through the "ontologies" feature, inherited from the sort Knowledge-Component, and specified as a collection of elements of sort Ontology.

Ontology **is-a** *Knowledge-Component*
pragmatics → *Pragmatics*
ontologies → set-of *Ontology*
signature → set-of *Signature-element*
formulae → set-of *Formula*

Figure 4.13: The Ontology sort

Figure 4.14 shows the sorts defined by the ontology used to describe the tasks and capabilities in the WIM application (Chapter 7). The sort *Signature-element* is used to describe the inputs and outputs of a task or capability, and the knowledge-roles of a capability or a domain-model. The sort *Formula* is used to specify the preconditions and postconditions of a task or capability, the assumptions of a capability, and the properties and meta-knowledge of a domain-model.

The explicit, declarative and shared nature of component's ontologies make them appropriate to annotate components with semantic information; thus enabling to compare component specifications on a *semantic matching* basis [Guarino, 1997a, Paolucci et al., 2002]. Semantic matching has been defined as an operator that takes two graph-like structures (e.g., database schemas or ontologies) and produces a mapping between elements of the two graphs that correspond semantically to each other [Giunchiglia and Shvaiko, 2003].

4.2.2 Matching relations

This section describes the different relationships that may be established between components in the Abstract Architecture and the role these relations play in the Knowledge Configuration process.

The ORCAS KMF describes a system as a collection of tasks, capabilities and domain-models. Nevertheless, an enumeration of classes of components is not enough to describe a system, some structuring principle is needed. In

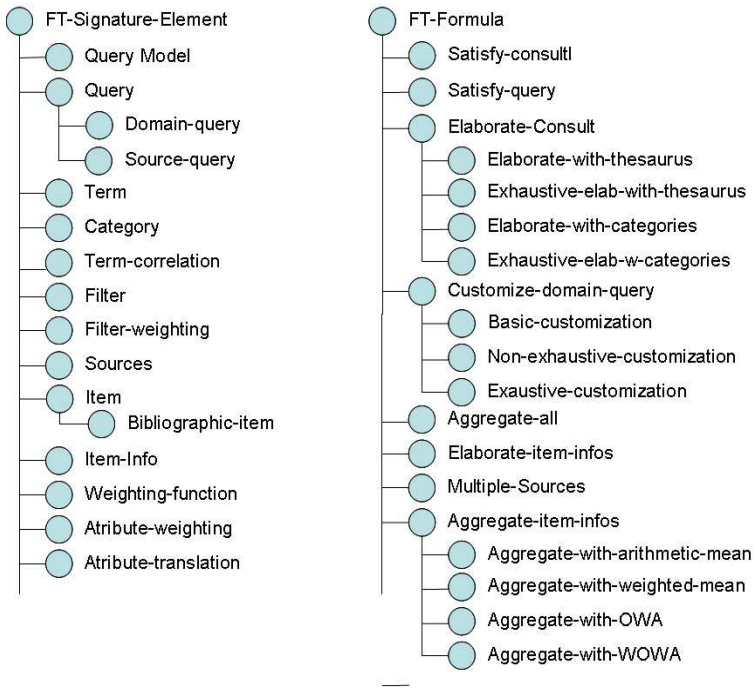

Figure 4.14: Hierarchy of sorts in the ISA-Ontology (from the WIM application, Chapter 7)

ORCAS this structure derives from the functional relations that can be established among components in the Abstract Architecture. Specifically, the ORCAS KMF includes two types of binary relations between components, namely *Task-Capability matching* and *Capability-Domain matching*.

- A *Task-capability matching* relation is defined between a task and a capability. Intuitively, a task-capability matching denotes a *suitability* relation: a task-capability relation is verified (is evaluated as true) when the capability is suitable for the task. In other words, a task "matches" a capability if the capability is able to solve the type of problems defined by the task. This relation compares the inputs, outputs and competence of a task against the homonym features of a capability to determine wether the application of the capability is able to achieve the postconditions of the task, whenever the preconditions of the task hold.
- A *Capability-domain matching* relation is defined between a capability and a collection of domain models characterizing the knowledge required by the capability. Since a capability may include many knowledge-roles, then a domain-model would be required to fill in each knowledge-role. Intuitively, a capability-domain matching denotes a relation of *satisfiability*: a capability "matches" a set of domain-models when the knowledge characterized by those domain-models satisfies the knowledge requirements (the *assumptions*) of the capability for each knowledge-role. This relation is defined in terms of knowledge-roles and capability assumptions that are satisfied by the properties and meta-knowledge of the set of domain-models.

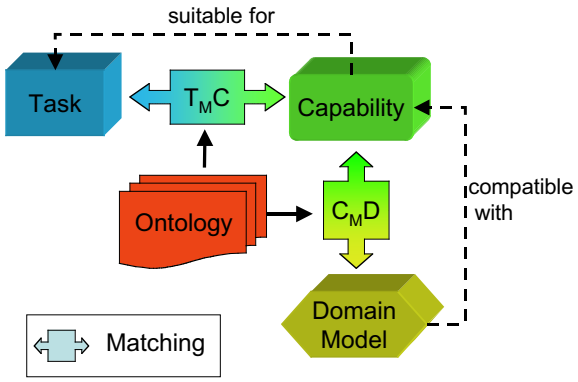

Figure 4.15: Matching relations in the Abstract Architecture

Figure 4.15 shows the components in the Abstract Architecture and the matching relations than can be established among components.

Understanding the matching relations is important because the operations to be performed during the Knowledge Configuration process are based on these

relations, and also because these relations are necessary to drive the analysis and the knowledge acquisition phases of software development.

The second point will be realized progressively through the rest of the chapter, but the main idea is using the matching relations to constrain the selection of components during the Knowledge Configuration process.

A matching relation (also referred as a "match") is determined by comparing two specifications, and has the goal of determining whether two software components are related in some way, e.g. two software components "match" if they are substitutable or if one component can be adapted to fit the requirements of another one.

The definition of a matching relation between components is built upon the definition of a more basic relation between component features. Since component features are specified in terms of the Object Language, matching will be defined upon a relation between elements (signature-elements and formulae) of the Object Language. Hence, in order to maximize the reuse of the Abstract Architecture over different Object Languages, we introduce two levels in the definition of a matching relation: the *abstract-level matching* and the *object-level matching*.

- The *abstract-level matching* is situated at the level of the Abstract Architecture. Matching relations at this level are based on an *abstract relation* between component features. Therefore, any system using the ORCAS Abstract Architecture can use the matching relations at the abstract level.

- The *object-level matching* is concerned with the Object Language. Matching relations at this level are defined as a refinement of the matching relations at the abstract level. This refinement is achieved by replacing the abstract relation by an *object relation* that is defined between elements (signature-elements and formulae) in the Object Language.

Before providing specific definitions of the ORCAS matching relations we will give some basic definitions concerning matching. Our approach to component matching is based on a combination of *signature matching* [Zaremski and Wing, 1995] and *specification matching* [Zaremski and Wing, 1997], that we prefer to call *competence matching*. Signature matching relations compare the interface of two components in terms of the types of information (and knowledge) they use (inputs and knowledge-roles) and produce (output). Competence matching relations compare the features characterizing the competence (preconditions and postconditions) of two components to determine if two components are substitutable, or if a component satisfies the requirements of another (e.g. a capability that is suitable for a task must satisfy all postconditions of that task).

In general, a matching relation is defined between two component specifications as follows:

$$Match(S, S') = match_{sig}(S, S) \wedge \ match_{comp}(Q, S)$$

where S, S' are the specification of two components, $match_{sig}$ is a *signature matching* and $match_{comp}$ is a *competence matching*.

The particular definition of *signature matching* and *competence matching* will be different for the different types of matching relation (Task-Capability matching or Capability-Domain matching).

Task-capability matching

A *Task-capability match* is defined between a task T and a capability C to determine wether C is suitable for the T (i.e. C can be applied to solve the type of problems characterized by T).

We define a *Task-capability match* as the conjunction of a *Generalized Type Match* over the input signature specification, and a *Specialized Type Match* over the output signature specification [Zaremski and Wing, 1995], and a *Plug-in Match* [Zaremski and Wing, 1997] over the competence specification.

Definition 4.1 ***(Task-capability match)***
$match(T, C) = match_{gen}(T_{in}, C_{in}) \wedge match_{spec}(T_{out}, C_{out}) \wedge match_{plugin}(T, C)$

where T is a task and is a capability, $match_{gen}$ is a *Generalized Signature Match*, $match_{spec}$ is a *Specialized Signature Match*, and $match_{plugin}$ is a *Plug-in match* defined over the competence specification (preconditions and postconditions).

The *Generalized Signature Match* over the input signatures means that the capability has an input signature C_{in} equal or more general than task input signature T_{in}.

$$match_{gen}(T_{in}, C_{in}) = \ (T_{in} \geq C_{in})$$

Inversely, the *Specialized Signature Match* over the output signatures means that the capability output is of the same type or of a more specific type than the task output.

$$match_{spec}(T_{out}, C_{out}) = \ (T_{out} \leq C_{out})$$

This combination of generalized and specialized matchings has the following justification:

- On the one hand, a capability with a more general input than a task implies the capability can extract from the task input all the information it requires. However, if we select a capability with an input more specific than a task (with more information), then the capability cannot obtain all the information it uses as input, and this fact could result on a bad capability operation.
- On the other hand, a capability with an output signature more specific means that it is able to provide all the information characterizing the output signature of a task, which is not true if the output of the capability is more general (with less information) than the output of the task.

Moreover, the *Plug-in Match* ($match_{plugin}$) [Zaremski and Wing, 1997] requires a capability C to have equal or weaker preconditions than a task T and equal or stronger postconditions than T:

$$match_{plugin}(T, C) = \ (T_{pre} \Rightarrow C_{pre}) \wedge (C_{post} \Rightarrow T_{post})$$

The reason to use that kind of matching is the following: we want to use capabilities that are suitable for (able to solve) a task, thus we want that whenever the preconditions specified by the task hold, the application of the capability guaranteed that the postconditions of the task will hold after its application.

The demonstration of the former property from the definition of the plug-in match is as follows: T_{pre} entails that C_{pre} holds, due to the first conjunct of the Plug-in Match, and C guarantees that $C_{pre} \Rightarrow C_{post}$; consequently, it's assured that C_{post} will hold after executing C, which entails also T_{post}, due to the second conjunct ($C_{post} \Rightarrow T_{post}$).

These definitions will be refined later in terms of object-level matching relations on signatures and on formulae (§4.3.3).

Capability-domain matching

Matching of domain-models and capabilities is slightly different, since a domain-model does not include a competence specification, neither an input nor an output. However, a capability may introduce more than one knowledge-role, consequently many domain-models would be required to match a single capability.

If a capability introduces only one knowledge-role, then we can say that a capability matches a domain model when the domain-model provides the kind of knowledge characterized by that knowledge-role, and satisfies the assumptions established by the capability for that knowledge-role. Moreover, if a capability specifies more than one knowledge-role, then for each knowledge-role there should exist one domain-model matching the specification of the capability .

Let's define the set of knowledge-roles of a capability as $C_{kr} = \{C^i_{kr} : i = 1 \ldots n\}$, and let's represent the set of assumptions of a capability over a particular knowledge-role as C^i_{asm}. A matching relation between a knowledge-role of a capability ($C^i_{kr} \in C_{kr}$) and one domain-model (M) is called a *partial capability-domain match*. A partial capability-domain match is defined such that there is at least one knowledge-roles in the domain-model (M_{kr}) that is equivalent or more specific than the knowledge-role of the capability (C^i_{kr}), and the assumptions of the capability for that knowledge-role C^i_{asm} are satisfied by the union of the properties and meta-knowledge of the domain-model ($M_{prop} \cup M_{mk}$), namely

$$Match_{partial}(C, M, C^i_{kr}) = \ match_{esp}(C^i_{kr}, M_{kr}) \wedge (M_{prop} \cup M_{mk} \Rightarrow C^i_{asm})$$

where C^i_{kr} is a knowledge-role of a capability and M is a domain-model; $match_{spec}$ is a *Specialized Signature Match*; M_{prop} and M_{mk} are the *properties* and *meta-knowledge* of M, and C^i_{asm} are the assumptions of capability C for the knowledge-role C^i_{kr}.

In this definition, a partial *capability-domain match* is expressed as a combination of a *Specialized Type Match* between the signature specification of a capability knowledge-role and the specification of the knowledge-roles of a domain-model, and a special kind of *Plug-in Match* defined between the specification of the assumptions of a capability for a single knowledge-role, and the properties and meta-knowledge of the domain-model.

The *Specialized Type Match* between a knowledge-role and a domain-model is defined as follows:

$$match_{esp}(C^i_{kr}, M_{kr}) = (C^i_{kr} \leq M_{kr})$$

The reason to use a *Specialized Type Match* here is that we must ensure the knowledge-roles characterized by a domain-model M can provide at least all the information required by a knowledge-role of a capability C^i_{kr}. This condition is guaranteed when the signature specification of the knowledge-roles of the domain-model (M_{kr}) is equal to or specializes the signature specification of the capability knowledge-role (C^i_{kr}). If M_{kr} was more general than C^i_{kr}, then it may occur than some of the information required by C^i_{kr} cannot be provided by the knowledge characterized by M_{kr}, and thus the capability cannot use that knowledge appropriately.

Moreover, in order for a capability to use the information characterized by a knowledge-role (C^i_{kr}), the domain-model providing that knowledge-role should guarantee that the assumptions of the capability over that role (C^i_{asm}) are satisfied by M. The specification of a domain-model is divided in two parts called *properties* (M_{prop}) and *meta-knowledge* (M_{mk}), consequently we define that the assumptions of a capability for a knowledge-role C^i_{asm} are satisfied by a domain-model when these assumptions are implied by the union of both the properties and meta-knowledge of the domain-model ($M_{prop} \cup M_{mk}$), as follows:

$$(M_{prop} \cup M_{mk} \Rightarrow C^i_{asm})$$

Now we can define a matching relation between a capability and a collection of domain models $\mathcal{M}$ that satisfy C as a conjunction of matching relations between pairs consisting of a knowledge-role (a signature element) and a domain model that matches it, such that there is a domain-model matching every knowledge-role specified by the capability.

Definition 4.2 ***(Capability-domain match)***

$match(C, \mathcal{M}) = \forall C^i_{kr} \in C_{kr} : \exists M \in \mathcal{M} | Match_{partial}(C, M, C^i_{kr})$
$= \forall C^i_{kr} \in C_{kr} : \exists M \in \mathcal{M} |\ (C^i_{kr} \leq M_{kr}) \wedge (M_{prop} \cup M_{mk} \Rightarrow C^i_{asm})$

where C is a capability, $\mathcal{M}$ is a set of domain-models; $match_{spec}$ is a *Specialized Signature Match*; M_{prop} and M_{mk} are the *properties* and *meta-knowledge* of M, and C_{asm} are the assumptions of capability C.

Ontology mappings

Since ORCAS components declare its conceptualizations (the "universe of discourse") as ontologies, and two components may be declared using different ontologies, then it can be necessary to establish a mapping between the concepts of the two ontologies, what is called a *ontology mapping*.

An *ontology mapping* is a declarative specification of the transformations required to match elements of one ontology to elements of another ontology. An example of a mapping is a *renaming*, but a mapping can include any kind of syntactic or semantic transformation: *numerical mapping*, *lexical mapping*, *regular expression mapping* and others [Park et al., 1998].

Nevertheless, in this thesis we focus on the matchmaking process (the process of verifying if a matching relation holds) assuming that all the components share the same ontologies or the required ontology mappings are already built.

4.3 The Object Language

This section presents a specific Object Language to be used within the ORCAS KMF to specify component features, and a particular relation (subsumption) to compare the specification of components in order to verify whether a matching relation holds.

We need an Object Language $\mathbb{O}$ in which to write the specification of capabilities, tasks and domain-models. Moreover, the Object Language has to be used also to represent other objects required by the Knowledge Configuration process, including the specification of problem requirements, task-configurations (the result of the Knowledge-Configuration process), and search states (Knowledge Configuration is approached as a search process over the space of possible configurations).

The Object Language provides a formalism for defining the terminologies used to specify component features. As we have introduced when describing the Abstract Architecture (§4.2.1), most of the features used to describe a component are specified by either signature elements or formulae in the Object Language: The sort SIGNATURE-ELEMENT is used to describe the input and output of tasks and capabilities, and the knowledge-roles of capabilities and domain-models as well. The sort FORMULA is used to specify the preconditions and postconditions of tasks and capabilities, the assumptions of a capability, and the properties and meta-knowledge of a domain-model[1]. In ORCAS the terminologies used to specify component features are represented by ontologies, thus the elements of the Object Language are specified as sorts of some ontology.

Figure 4.14 shows an example of an Object Language onotology used to specify tasks and capabilities in the WIM application (Chapter 7). Notice that

[1] Usually, knowledge-modelling languages describe these kind of specifications with a formal language, some examples are LOOM [Gaspari et al., 1998], OCML [Motta, 1999] and KARL [Studer et al., 1996]

every sort is a specialization (a subtype) of either the sort Signature-element or the sort Formula.

Many languages could be used as the Object Language while keeping the ORCAS Abstract Architecture, thanks to a clear separation of two levels in the definition of a matching relation: the *abstract level* matching and the *object level* matching.

We have already defined the matching relations at the *abstract level*, now we are going to describe the *Feature Terms* formalism to be used as the Object Language. Following, the ORCAS matching relations will be rewrited using subsumption between feature terms as the object-level matching relation..

4.3.1 The Language of Feature Terms

The ORCAS Object Language is based on the Feature Terms formalism to represent ontologies and component features. *Feature Terms* (also called feature structures or ψ-terms) are a generalization of first order terms. The difference between feature terms and first order terms is the following: a first order term, e.g. $f(x, y, g(x, y))$ can be formally described as a tree and a fixed tree-traversal order. In other words, parameters are identified by position. The intuition behind a feature term is that it can be described as a labelled graph i.e. parameters are identified by name.

Specifically, we use a concrete implementation of feature terms as embodied in the NOOS representation language [Arcos, 1997], in which several Case Based Reasoning systems have been described and implemented [Arcos et al., 1998, Arcos, 2001]. This formalism organizes concepts into a hierarchy of *sorts*, and represents descriptions and individuals as collections of features (functional relations) called *feature terms*. The attributes used to describe a component (signature-elements and formulae) will be specified as feature terms in the Feature Terms formalism.

Before to define Feature Terms formally we need to introduce the following elements:

1. a signature $\Sigma = \langle \mathcal{S}, \mathcal{F}, \leq \rangle$ (where $\mathcal{S}$ is a set of sort symbols that includes $\perp$; $\mathcal{F}$ is a set of feature symbols; and $\leq$ is a decidable partial order on $\mathcal{S}$ such that $\perp$ is the least element),

2. and a set ϑ of variables;

Succinctly, we can now define a feature term ψ as an expression of the form:

$$\psi ::= X : s[f_1 \doteq \Psi_1 \ldots f_n \doteq \Psi_n] \tag{4.1}$$

where X is a variable in ϑ that is called the *root* of the feature term, s is a sort in $\mathcal{S}$ (the root sort), $f_1 \ldots f_n$ are features in $\mathcal{F}$, $n \leq 0$, and each Ψ_i is a set of Feature Terms and variables. When $n = 0$ we are defining a term without features. The set of variables occurring in ψ is noted as ϑ_ψ.

Sorts have an informational order relation ($\leq$) among them, where $\psi \leq \psi'$ means that ψ has less information than ψ' (or equivalently that ψ is more general than ψ'). The minimal element ($\bot$) is called *any* and represents the minimum information. A feature with an unknown value is represented as having the value *any*. All other sorts are more specific than *any*.

4.3.2 Subsumption

A basic notion of the Feature Terms formalism is that of *subsumption*, which we use as the inference mechanism. Subsumption is an order relation among terms built on the top of the $\leq$ relation among sorts.

Intuitively, we say of two Feature Terms $\psi, \psi' \in \Phi$ that ψ subsumes ψ' ($\psi \sqsubseteq \psi'$) when all that is true for ψ is also true for ψ'. A more formal definition of subsumption is introduced below, but first we need to introduce some notation and basic definitions:

- $Root(\psi)$ is defined as a function that returns the root of a term (a variable X).
- $Sort(X)$ is defined as a function that returns the sort of the variable X.
- A *path* $\rho(X, f_i)$ is defined as a sequence of features going from the variable X to the feature f_i. There is a *path equality* when two paths point to the same value. Path equality is equivalent to variable equality in first order terms.
- We use a "dot notation" to reference a particular feature within a term, thus $\psi.f$ refers to the feature f of the term ψ.

Now we can define subsumption as follows:

Definition 4.3 *(Subsumption) Feature term ψ subsumes ψ' ($\psi \sqsubseteq \psi'$) if:*

1. $Sort(Root(\psi)) \leq Sort(Root(\psi'))$*, i.e. the root sort of ψ' is the same sort or a subsort of the root sort of ψ.*
2. $\forall f \in \mathcal{F} : \psi.f \neq \bot \Rightarrow \psi'.f \neq \bot$*, i.e. every defined feature in ψ is also defined (has a value different from $\bot$) in ψ'.*
3. $\forall f \in \mathcal{F} : \psi.f = v \neq \bot \Rightarrow v \sqsubseteq v' = \psi'.f$ *if v is a singleton,*
4. *otherwise if $\psi.f$ is a set $\psi.f = V = \{v_1 \ldots v_m\}$ then $\forall v_i \in V : \exists v'_j \in V' = \psi'.f : v_i \sqsubseteq v'_j)$, i.e. there is a subsumption mapping between the sets.*
5. $(\rho(Root(\psi), f_1) = \rho((Root(\psi), f_2) = v) \Rightarrow (\rho(Root(\psi'), f_1) = \rho((Root(\psi'), f_2) = v' \wedge v \sqsubseteq v')$*; i.e., path equality is satisfied downwards.*

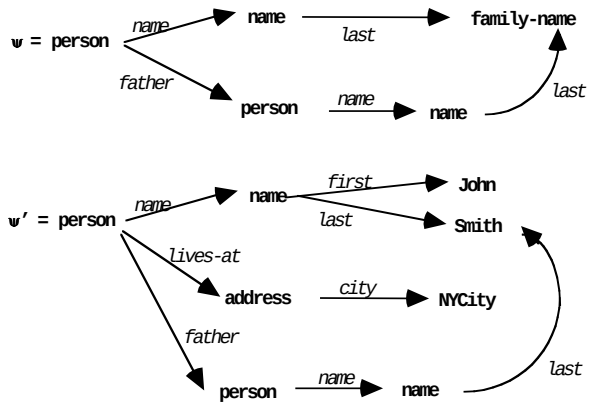

Figure 4.16: Representation of feature terms as labelled graphs

Notice that step 4 of Definition 4.3 provides a concrete interpretation of the subsumption between two sets: thus given two sets of feature terms Ψ, Ψ' we say that $\Psi \sqsubseteq \Psi'$ if step 4 of Definition 4.3 is verified.

Figure 4.16 shows an example of two features that verify a subsumption relation. This picture shows feature terms as labelled directed graphs: for each variable $X : s$ there is a node q labelled with sort s, while an arc from q to another node q' is labelled by f, for each feature f defined in q with feature value q'.

In this example there are two feature terms ψ, ψ' with the same root sort, person. Notice that ψ subsumes ψ', $\psi \sqsubseteq \psi'$, since ψ' contains more information than ψ (ψ' is a specialization of ψ, which entails that all that is true for ψ' is also true for ψ, $\psi \Rightarrow \psi$): Both ψ and ψ' are of sort person. ψ has two features, *name* and *father*, specified as elements of sort name and person respectively. The feature *name* is specified as terms of sort name, which has a feature called *last* representing the father's name. For both terms the father's name has a single feature called *name* which is the same (there is path equality) than the *last* name of the person. However, ψ' specializes ψ in the following aspects: the *last* name of ψ is specified as a term of sort family-name, and the last name of ψ' is John, which identifies a term of sort family name, and thus it is just a specialization of the last name of ψ'. Finally, ψ' has another feature not existing in ψ (*lives-at*) that contains a partial description of their home address. ψ' is a specialization of *psi*, since ψ' contains more information than ψ, and thus we conclude than ψ' is subsumed by ψ.

4.3.3 Matching by subsumption

Subsumption can be now used to define the matching relations at the object-level. Both signatures and formulae in our Object Language are represented using *terms* in the Feature Terms formalism. Moreover, each term ψ has an associated root sort ($Sort(\psi)$), and these sorts are organized into a hierarchy of sorts in the component's ontology. As a consequence, terms without any feature can be directly evaluated upon the informational partial order relation ($\leq$) over their root sorts.

In general a signature or a formula can be expressed as a full fledged Feature Terms structure, and therefore the relations ($\leq$ and $\Rightarrow$) used to define the matching relations should be reformulated using Feature Terms concepts, namely the informational partial order relation $\leq$ and the implication $\Rightarrow$ relation. These relations will be specialized by the subsumption relation among Feature Terms (Definition 4.3) as follows.

- On the one hand, if a term ψ subsumes another term ψ' ($\psi \sqsubseteq \psi'$), then the root sorts for these terms verify a partial order relation (Step 1 of Definition 4.3): $\psi \sqsubseteq \psi' \Rightarrow Sort(\psi) \leq Sort(\psi')$. Intuitively, the subsumption relation can be regarded as a generalization of the informational order relation ($\leq$) to compare terms having no empty features (features with a value different of $\bot$) with structures, in place of single sorts. Therefore, the partial order relation ($\leq$) introduced at the abstract-level matching is replaced by the subsumption relation at the object-level matching.

- On the other hand, the implication relation ($\Rightarrow$) introduced at the abstract-level matching boils down to subsumption over Feature Terms at the object-level matching. Intuitively, a term ψ subsuming another term ψ' ($\psi \sqsubseteq \psi'$) means that ψ' is more specific or has more information than ψ, and this implies that all that is true for ψ is also true for ψ': $\psi \Rightarrow \psi'$.

Now we can express signature matching over inputs and outputs using *subsumption* over Feature Terms, as follows:

$$match_{sig}(T, C) = T_{in} \sqsubseteq C_{in} \wedge T_{out} \sqsubseteq C_{out}$$

where subsumption ($\sqsubseteq$) is among sets (step 4 of Definition 4.3).

Similarly, we can reformulate the definition of a *Plug-in Match* between a task and a capability as follows:

$$match_{plugin}(T, C) = T_{pre} \sqsubseteq C_{pre} \wedge C_{post} \sqsubseteq T_{post}$$

Now that we have defined the basic matching relations between signature-elements and formulae expressed at the object-level, we are ready to reformulate the whole definition of a Task-capability match (Definition 4.1) to the language of Feature Terms using subsumption:

Definition 4.4 ***(Task-capability match by subsumption)***
$match(T, C) = T_{in} \sqsubseteq C_{in} \wedge C_{out} \sqsubseteq T_{out} \ \wedge C_{pre} \sqsubseteq T_{pre} \wedge T_{post} \sqsubseteq C_{post}$

Intuitively, this definition is justified as follows: If the input of the capability subsumes the input of the task ($C_{in} \sqsubseteq T_{in}$) then the input for the task will provide at least all the information required by the capability, since the input of the task is more specific or has more information that the input of the task. Complementarily, the output of the capability subsumes the output of task ($C_{out} \sqsubseteq T_{out}$) implies that the output of the capability is more specific than the output of the task and, consequently, the application of the capability can provide at least all the information required by the task output.

Concerning the competence, the fact that the preconditions of the capability subsume the preconditions of the task ($C_{pre} \sqsubseteq T_{pre}$) means that all the preconditions of the capability are guaranteed by the equal or more specific preconditions of the task, and viceversa: the fact that the postconditions of the task subsume the capability postconditions ($T_{post} \sqsubseteq C_{post}$) means that all task postconditions are guaranteed by the capability postconditions.

Similarly, we can specialize the definition of a Capability-domain match (Definition 4.1) to the language of Feature Terms by using subsumption as the basic matching:

Definition 4.5 ***(Capability-domain match)***

$$match(C, \mathcal{M}) = \forall C^i_{kr} \in C_{kr} : \exists M \in \mathcal{M} | Match_{partial}(C, M, C^i_{kr})$$
$$= \forall C^i_{kr} \in C_{kr} : \exists M \in \mathcal{M} |\ (C^i_{kr} \sqsubseteq M_{kr}) \wedge (C^i_{asm} \sqsubseteq M_{prop} \cup M_{mk})$$

where C is a capability, $\mathcal{M}$ is a set of domain-models; C_{kr} are the knowledge-roles, and C^i_{asm} are the assumptions of the capability for the ^ith knowledge-role, specified as a set of signature-elements and formulae respectively; and M_{prop}, M_{mk} are the properties and meta-knowledge of a domain-model, specified as formulae. Both signature-elements and formulae are specified by feature terms.

In this definition, two conditions are imposed for establishing a match between a capability and a collection of domain-model:

1. the specification of knowledge-role signatures by the domain-models must provide at least as much information as required by the signature specification of the knowledge-role of the capability. This condition is ensured if for each signature specifying a knowledge-role of the capability, there is a specification of domain-model signatures that refines or is more specific than it, which in feature terms is expressed by the idea of being subsumed ($C^i_{kr} \sqsubseteq M_{kr}$);

2. the assumptions of the capability for each knowledge-role C^i_{kr} must be satisfied by the properties and meta-knowledge of a domain-model that in addition verifies the matching between the signature specification: this is assured if all that is true for the domain-model is also true for the assumptions of the capability ($(M_{prop} \cup M_{mk}) \Rightarrow C^i_{asm}$). Using feature terms this condition can be expressed using subsumption as $C_{asm} \sqsubseteq (M_{prop} \cup M_{mk})$.

4.4 Knowledge Configuration

We have defined the MAS configuration process as being performed at two layers: the Knowledge Configuration process, that is situated at the knowledge-layer (this chapter), and the Team Formation process, that is carried on at the operational-layer (Chapter 5).

During the previous sections of the chapter the Abstract Architecture and a concrete Object Language for the ORCAS KMF have been described. In addition, we have formally defined the matching relations to be used during the Knowledge Configuration process in order to verify whether a capability is suitable for a task, and whether a domain-model satisfies a capability. Hence, we have described the basic concepts required to explain the Knowledge Configuration process itself, which is the aim of this section.

The *Knowledge Configuration* process has the goal of finding a configuration in terms of a composition of application tasks, agent capabilities and domain-models, in such a way that the requirements of the problem at hand are satisfied. This process actually follows a *Problem Specification* process, the main purpose of which is the characterization of the problem at hand in order to later select the most appropriate components during the Knowledge Configuration process. This characterization of a problem is formalized by a specification of *problem requirements* to be satisfied by a Task-Configuration.

After introducing some basic definitions and notation (§4.4.1) we will describe the Problem Specification process (§4.4.2) and the Knowledge Configuration process (§4.4.3). Next, three strategies for the Knowledge Configuration process (§4.4.4) are presented.

4.4.1 Notation and basic definitions

We have already seen the components defined in the Abstract Architecture and the different relations constraining the way in which components can be connected, which have been defined as matching relations. This section summarizes the specification of components in the Abstract Architecture and introduces some basic definitions.

We deal with three main types of knowledge-level components, namely: *task*, *domain-model* and *capability*, which have two further subtypes: *skill* and *task-decomposer*.

Knowl.-Component	X	
Task	$T < X$	$T = \langle in, out, pre, post \rangle$
Capability	$C < X$	$C = \langle in, out, pre, post, com, asm, kr \rangle$
Task-Decomposer	$D < C < X$	$D = \langle in, out, pre, post, com, asm, kr, st \rangle$
Skill	$S < C < X$	$S = \langle in, out, pre, post, com, asm, kr \rangle$
Domain Model	$M < X$	$M = \langle kr, prop, mk \rangle$

Table 4.1: Types of knowledge components an their main features

Table 4.1 sums up the hierarchy of sorts used to describe the components in the Abstract Architecture, and the features used to specify each component, where $A < B$ means that A is a subsort (subtype) of B, st are subtasks ($st \subset \mathcal{T}$), in, out, kr are inputs, outputs and knowledge-roles, specified as signature-elements in the Object Language $\mathbb{O}$, $pre, post, asm$ are preconditions, postconditions and assumptions, specified as formulae in the same language $\mathbb{O}$, com, od are the communication aspects and the operational description of a capability respectively, and pro, mk are the properties and the metaknowledge of a domain model, specified by formulae in $\mathbb{O}$.

We will note an element of a tuple specification as subscript, e.g. T_{in} is the input signature of task T and T_{post} are the postconditions of T.

However, since the Knowledge Configuration process requires a repository of components as an input to choose from the components of a task-configuration, we will introduce the idea of a repository of components or a *library*.

A *Library* is a collection of tasks and capabilities specified using some Object Language. A Library is independent of the domain because both tasks and capabilities are described in terms of their own ontologies, and not in terms of the domain ontology.

Definition 4.6 ***(Library)***
$\mathbb{L} = \langle \mathcal{T}, \mathcal{C}, \mathbb{O} \rangle$,
where

- $\mathcal{T}$ *is a set of tasks,*
- $\mathcal{C}$ *is a set of capabilities,*
- *and* $\mathbb{O}$ *is the Object Language.*

In the following definitions we will note $\mathcal{T}$ and $\mathcal{C}$ as the set of tasks and capabilities in the library used by the Knowledge Configuration process.

The Knowledge Configuration process takes a specification of stated problem requirements and a library of components as input and produces a task-configuration as output. Since a task-configuration is a complex structure we need first to define its constituent elements, called configuration schemas. We will note $\kappa \in \mathbb{k}$ as a configuration schema and the set of all configuration schemas; moreover, we will note $(T \doteq U) \in \mathcal{B}$ as a *binding* and the set of all bindings, where a binding is a link between a task and a capability or a configuration schema that is selected to solve that task. More formally:

Definition 4.7 ***(Binding)*** *A binding $(T \doteq U)$ is a pair with a task $T \in \mathcal{T}$ in the* head *and a capability $C \in \mathcal{C}$ or a configuration schema $k \in \mathbb{k}$ in the* tail*: $U \in \mathcal{C} \cup \mathbb{k}$*

Definition 4.8 ***(Configuration schema)*** *A configuration schema $\kappa \in \mathbb{k}$ is a pair $\langle (T \doteq C), \{(T_i \doteq \kappa_{j_i})\}_{i=1\ldots n} \rangle$ where $T, T_1, \ldots, T_n \in \mathcal{T}$, $C \in \mathcal{C}$, and $\kappa_{j_1}, \ldots, \kappa_{j_n} \in \mathbb{k}$, and $T_1, \ldots, T_n \in C_{st}$.*

A configuration schema specifies in the *head* of the pair a binding between a task T and a capability C ($T \doteq C$). The *tail* of the configuration schema is a set of bindings from C_{st} (the subtasks of C) (which will be empty if C is a skill, since skills have no subtasks) to other configuration schemas. A configuration schema can be complete or partial, defined as follows:

$$Complete(\kappa) \Leftrightarrow \forall T_i \in C_{st}\ \exists \kappa_{j_i} : (T_i \doteq \kappa_{j_i}) \in tail(\kappa)$$

i.e. if all subtasks of C are bound to another schema in the tail; otherwise κ is *partial*.

We define a *configuration relation* $\mathbb{R}$ among schemas as follows:

$$\mathbb{R}(\kappa, \kappa') \Leftrightarrow \exists (T_i \doteq \kappa') \in tail(\kappa)$$

i.e. two schemas are related if one of them is bound to a subtask in the tail of the other.

Noting $\mathbb{R}^*$ the closure of $\mathbb{R}$ we can now define a *task-configuration* as follows:

Definition 4.9 ***(Task-configuration)*** *A task-configuration is defined in terms of configuration schemas $Conf(\kappa) = \{\kappa' \in \Bbbk | \mathbb{R}^*(\kappa, \kappa')\}$. A task-configuration $Conf(\kappa)$ can be complete or partial: $Complete(Conf(\kappa))$ iff $\forall \kappa' \in Conf(\kappa) : Complete(\kappa')$; otherwise $Conf(\kappa)$ is partial.*

Thus, a *task-configuration* is a collection of interrelated configuration schemas (starting from a root schema κ). A task-configuration is *complete* when all schemas belonging to it are complete.

We will note K a task-configuration and $\mathcal{K}$ the set of all the task-configurations ($K \in \mathcal{K}$).

4.4.2 The Problem Specification process

The Problem Specification process aims at characterizing the problem to be solved in terms of a task to be solved and a set of requirements the configured system must comply to. We are not interested on the particular strategy to carry on this process; actually we are interested in the result of the Problem Specification process, since this is an input (the other input is a library of components) for the Knowledge Configuration process (Figure 4.17). Nonetheless, we will succinctly describe a iterative approach to specify problem requirements by interacting with the user and using a matching relation between problem requirements and tasks to find out which tasks are representative of the problem at hand.

The Problem Specification process starts with the specification of some initial requirements that are used to select the *application task* T_0. The idea is that a configured system is an application that results of assembling existing components such that the resulting composition satisfies the requirements of the problem at hand. We call this class of application that is designed and assembled on-demand by reusing and existing components over a (probably new)

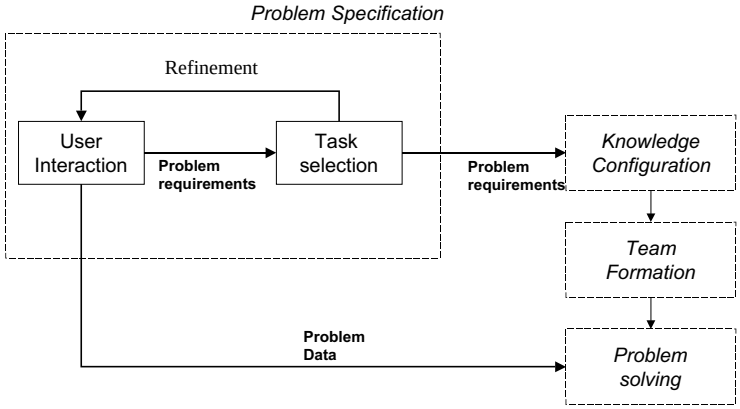

Figure 4.17: Problem Specification process

specific domain, an *on-the-fly* application. The task that is decided as the better characterization of the problem at hand is the starting point for the Knowledge Configuration process and is called the *application task*.

Since many tasks may satisfy the initial problem requirements, these requirements can be further refined and modified to better characterize the problem at hand, according to the needs and preferences of the requester (a human user or a software agent).

Problem requirements are used by two processes: Problem Specification and Knowledge Configuration. On the one hand, problems requirements are used as both input and output of the Problem Specification process; on the other hand the final output of the Problem Specification process is a collection of requirements that becomes an input for the Knowledge Configuration process.

The Problem Specification process starts with a query containing some problem requirements, and proceeds by selecting a set of tasks characterizing the problem at hand. Afterwards the user may select a task he thinks is a better characterization of his problem (the application task T_0) and may add or delete requirements from the query to be satisfied by the configuration of that task.

Definition 4.10 ***(Query)*** *A query Q is a tuple $Q = \langle in, out, pre, post, dm \rangle$*

where in, out are the query input and output signatures, which specify the type of the data provided to solve a problem, and the type of data expected as a result to that problem; pre are preconditions, which characterize the circumstances holding before starting the problem solving process, $post$ are postconditions, or the effects to bring about after solving the problem; and $dm \subset \mathcal{M}$ is a set of domain-models characterizing the application domain..

Notice that the specification of a query is like the specification of a task, but a query adds in a collection of domain-models characterizing the application domain.

In order to find out which tasks in the library satisfy the requirements stated by a query, it is necessary to establish a matching relation between a query Q and a task ($T \in \mathcal{T}$). We define a *query-task matching* relation following the idea of a *task-capability match* (Def. 4.4), but in this case the query plays the role of the task, and the task plays the role of the capability.

$$match(Q,T) = match_{gen}(Q_{in}, T_{in}) \wedge match_{spec}(Q_{out}, T_{out}) \wedge match_{plugin}(Q,T)$$

We specialize this definition for the case of using Feature Terms as the Object Language, by using the *subsumption* ($\sqsubseteq$) relation as the basic matching relation.

$$match(Q,T) = T_{in} \sqsubseteq Q_{in} \wedge T_{out} \sqsubseteq Q_{out} \ \wedge T_{pre} \sqsubseteq Q_{pre} \wedge Q_{post} \sqsubseteq T_{post}$$

The idea of this matching relation is to assess whether a task is representative of a problem. A task is considered to be representative of a problem if, given an input of the type specified by the query, and assumed that the preconditions stated by the query holds, the solution to the task will provide all the information required by the query output signature, and guarantees that the query postconditions will hold afterwards.

The *task-selection* activity can be described as function ($\mathcal{Q} \times \mathcal{T} \rightarrow \mathcal{T}$) that retrieves from a library a set of tasks that are representative of the problem at hand, where representativeness is decided upon the verification of a query-task match. This function takes a query and the tasks in a library as input and return those tasks that verify a matching relation with the query. We can now define the result of the task-selection activity as a set $\mathcal{T}_Q$ of tasks satisfying a matching with Q, formally:

$$\mathcal{T}_Q = \{T_i \in \mathcal{T} | match(Q, T_i)\}$$

At the end of the Problem Specification process one task is selected by the user as the application task: $T_0 \in \mathcal{T}_\mathcal{Q}$. Otherwise, all the tasks satisfying (matching) the query can be used as legitimate starting points for the Knowledge Configuration process.

The final result of the Problem Specification is a collection of problem requirements and problem data. The resulting specification of problem requirements (Figure 4.18) includes an application task (the task to be configured), a specification of the kind of inputs provided to and outputs expected from the Problem Solving process, a collection of preconditions that are assumed to hold, a collection of postconditions to be satisfied, and a specification of domain-models to be used. It does not matter whether the task is straightforwardly selected by the user, by an automated service, or through an interactive problem-specification support tool, like the interactive broker presented in §4.4.4.

Problem requirements are specified by the following features (Figure 4.18):

- *Application task*: the task characterizing the type of problem to be solved. This task characterizes the type of problems to which the current problem is supposed to belong to.

Problem-Requirements
application-task → *String*
ontology → *Ontology*
inputs → set-of *Signature*
outputs → set-of *Signature*
preconditions → set-of *Formula*
postconditions → set-of *Formula*
domain-models → set-of *Domain-Model*
configuration-options → set-of *Configuration-Options*

Figure 4.18: Features used to specify problem requirements

- *Inputs* and *outputs* are specified as signature elements in the Object Language. These signatures characterize the kind of data provided as an input to the problem and the kind of data expected or required as a solution. These signatures verify the signature matching relation between of a query-task matching with respect to the application task T_0. More formally, $T_{in} \sqsubseteq Q_{in} \wedge T_{out} \sqsubseteq Q_{out}$.

- *Preconditions* are also used to select a task during the initial problem specification, but can be extended to better characterize the problem in order to refine the collection of candidate tasks. Finally, the preconditions included as problem requirements must guarantee that the preconditions of the application task hold, which is achieved by the condition $T_{pre} \sqsubseteq Q_{pre}$ applied during the task-selection activity.

- *Postconditions* are the most outstanding element of problem requirements, since they characterize the goals of the problem. Postconditions are used twice: once during the selection of the application task; and afterwards, during the Knowledge Configuration process, to verify whether a task-configuration is valid (satisfies the requirements). The application task must assure that the postconditions of the query will hold after solving that task, which is endorsed when all that is true for the postconditions of the task is also true for the postconditions of the query. This condition is represented by the conjunct $Q_{post} \sqsubseteq T_{post}$ in the specification of a query-task matching relation.

- *Domain-models* are used to restrict the domain knowledge that can be used by the selected capabilities during the Problem Solving process. Only domain-models included in the query are considered during the Knowledge Configuration process.

- *Configuration Options* are used to setup the Knowledge Configuration process. The main configuration option is the strategy to be used for the Knowledge Configuration process (§4.4.4). Three strategies have been implemented: one based on a deep-first search strategy, another one based

on a best-first strategy guided by past-configuration cases, and one more that is guided by the user though the user is provided with a heuristic help to decide which components are used. Other options control parameters that are specific of a particular configuration strategy.

The problem is specified using the same ontology used by the application task or there is an ontology mapping between the problem requirements ontology and the task ontology.

Figure 4.19 shows a screenshot of the problem specification interface as presented to the user when using the *Interactive Configuration* strategy (§4.4.4. This screen shows the user the collection of signature-elements and formulae that can be chosen by the user in order to specify the inputs, preconditions, postconditions characterizing his problem.

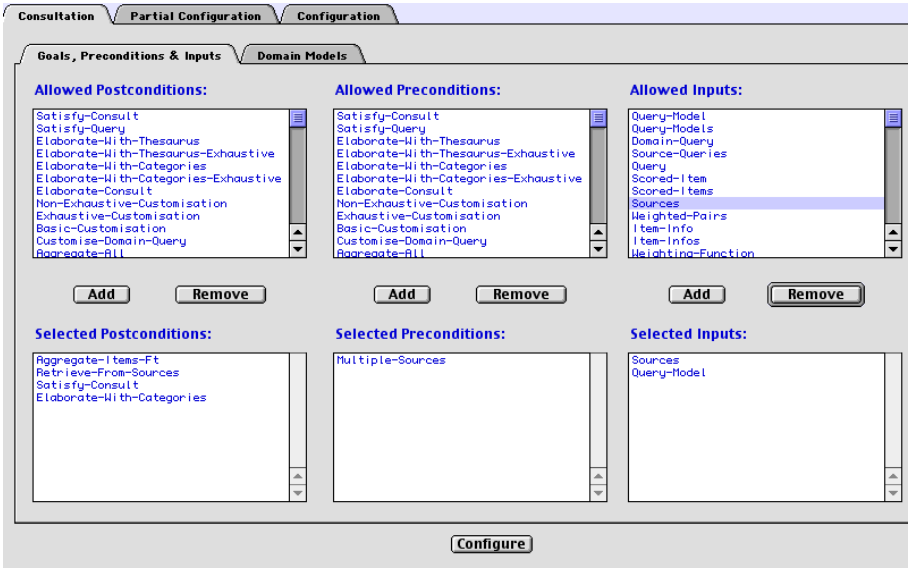

Figure 4.19: User Interface used to specify problem requirements.

Figure 4.20 shows a screenshots of the problem specification interface where the user can define the domain-models he is interested in.

4.4.3 Overview of the Knowledge Configuration process

The input for the Knowledge Configuration process is twofold: on the one hand it uses a collection of problem requirements, as described above, and on the other hand it uses a repository or *library* of component specifications to build up a configuration. The result of the Knowledge Configuration process is a *task-configuration* that, if complete and correct verifies the following: a) each task is bound at least to one capability that can achieve it, b) each capability requiring

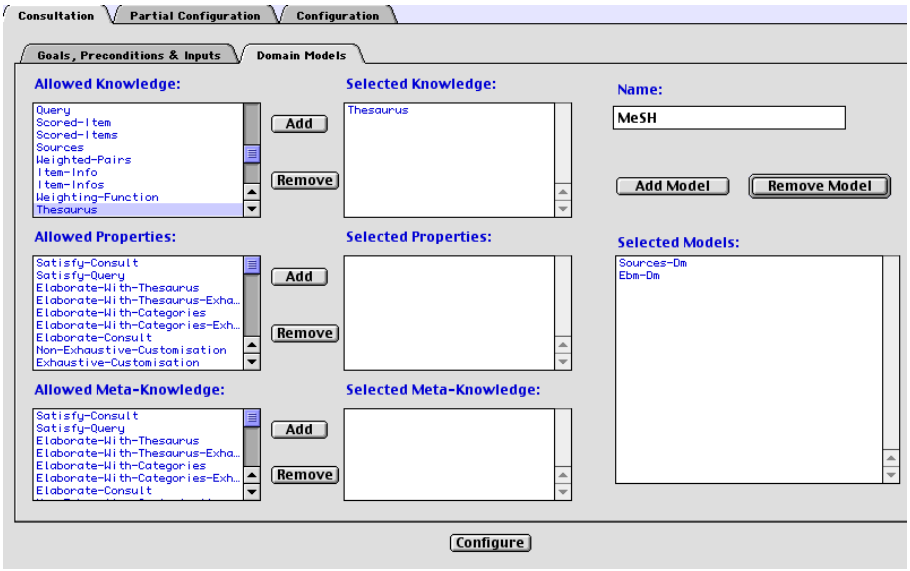

Figure 4.20: User Interface where the user defines the domain-models to be used during the Knowledge Configuration process.

knowledge is assigned a set of domain-models satisfying its assumptions, and c) the whole configuration complies to the problem requirements.

The Knowledge Configuration process has been designed and implemented as a search process in the space of partial configurations, where each state represents a partial configuration of a task.

The main information to be represented in a state is the set of task-capability *bindings* used in a partial configuration, but it also holds information about the requirements (inputs, preconditions and postconditions, domain-models and assumptions), those ones already satisfied and the ones yet to be satisfied.

The Knowledge Configuration process uses the problem requirements to generate an initial state. From the initial state, new states are generated by binding capabilities to tasks, where the bindings are realized through the verification of the matching relation between tasks and capabilities. The generation of new states continues until one of the new states is considered a final state; which occurs when the following conditions occur simultaneously: all tasks are bound to suitable capabilities, there are domain models satisfying the requirements of every capability included, and all the problem requirements are satisfied.

Figure 4.21 shows the main activities performed during the Knowledge Configuration process: *task-configuration*, *capability-configuration* and *verification*.

1. *Task-configuration*: The Knowledge Configuration process starts with an initial task and chooses a capability suitable for it (i.e. verifying a task-capability match).

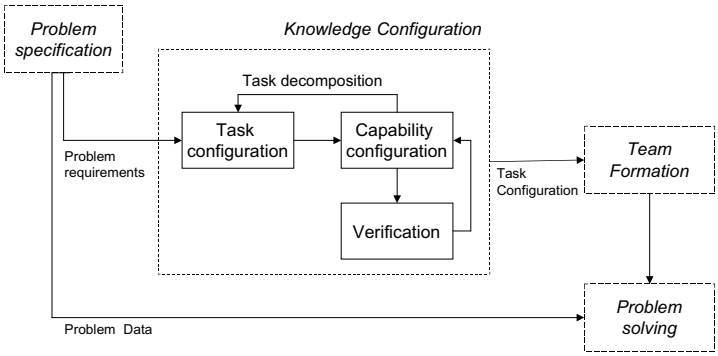

Figure 4.21: Main activities of the Knowledge Configuration process

2. *Capability-configuration*: The selected capability is configured by selecting domain-models compatible with it (verifying a capability-domain match). If a capability is a task-decomposer, then the task-configuration process should be performed for each subtask, which then becomes a recursive activity.

3. *Verification*: finally, verification is the process of checking whether the global problem requirements are met or not. If the global problem requirements are met —configuration is correct— and the configuration is complete, then the Knowledge Configuration ends, and the resulting task-configuration can be used to guide the Team Formation process. The point is that a verified task-configuration can be considered as design-level description of an application to be performed by a team of agents.

A task-configuration is recursively defined in term of a capability-configuration that boils down to a task-configuration for each subtask of the task-decomposer (a configuration schema), until all tasks are bound to a capability and thus there are no more subtasks to spawn from. The result is a tree of tasks that are bound to capabilities, and capabilities bound to domain-models, as shown in Figure 4.22.

4.4.4 Strategies for the Knowledge Configuration process

We have already introduced the notion of the Knowledge Configuration as a search process among the space of possible configurations: from an initial state the Knowledge Configuration process explores successor states according to some order until reaching a final state. We have implemented three different strategies for the search process, depending on the kind of user and the availability of past configuration cases, namely.

We differentiate between experts users, which are knowledgeable of the ORCAS KMF and the Knowledge Configuration process (usually the knowledge

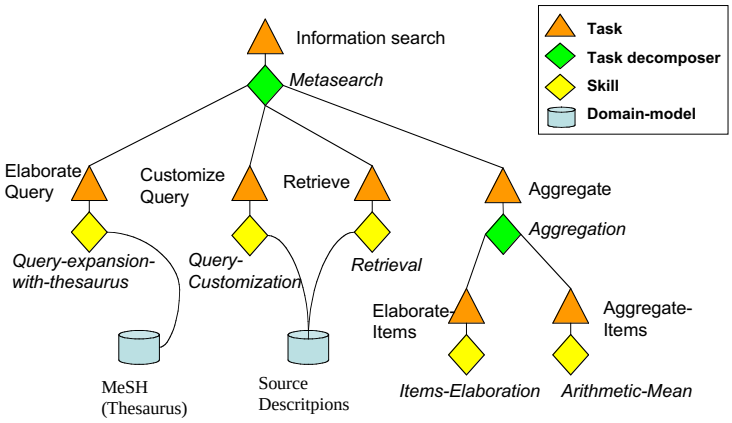

Figure 4.22: Example of a task-configuration

engineer), and the final user, which does not have such a knowledge.

The three strategies implemented for the Knowledge Configuration process are the following, namely: *Search and Subsumes*, *Constructive Adaption* and *Interactive Configuration*.

- *Search and Subsume* is appropriate for non expert users whether past configurations are not available or are not desired. The *Search and Subsume* configuration mode implements a depth first strategy for searching. This strategy uses subsumption for retrieving capabilities that match a task. For each retrieved capability, a new state (called successor state) is generated. The new states generated are added to a stack of open states; and the next state to be explored is always the head of this stack.

- *Constructive Adaption* is appropriate when the user (expert or not) wants to use *Case-Based Reasoning* (CBR) to drive the search process. The *Constructive Adaptation* strategy follows a *best-first* search process in the state space [Plaza and Arcos, 2002]. There is a heuristic function that assesses which state is "best", based on problem requirements previously used to configure a system (i.e. previous configurations here used as cases). This strategy uses past configurations to order the successor states according to a measure of similarity to the current state. A similarity measure called SHAUD [Armengol and Plaza, 2001] is used to evaluates the similarity between Feature Terms structures. The result is a ranking of past configuration cases that are used to order the states. Then, once we have reordered the states, the search process selects to expand the state with a higher heuristic value based on case-based similarity.

- *Interactive Configuration* is appropriate for expert users who want to have more control over the Knowledge Configuration process. This strategy in-

terleaves the specification and the configuration phases until a complete configuration is found. The *Interactive Configuration* strategy operates through a graphical user interface that shows the partial configuration, the available components and other information. Once the user selects a task to be configured, the interface presents the capabilities that are suitable for that task (those matching that task). These capabilities are ranked with the similarity measure used in the case-based strategy (Constructive Adaptation) in order to guide the user about which are the capabilities recommended by the system upon past configuration experience, but the user is free to select any capability based in its own criteria. If the selected capability is a task-decomposer, then the capability selection process is repeated for every subtask. In order to facilitate the user decision, the interface presents some extra information about the components and information about the current state of the configuration. The configuration service interleaves the problem specification and the task-configuration activities, and brings the user the possibility of deciding the next state to follow by selecting the capability to bind to the current task (from the set of suitable capabilities for that task). This mode is also useful for a knowledge engineer to build a initial case-base of configuration examples, thus allowing the end-user to use the constructive-adaptation strategy.

Figure 4.23 shows an example of a partial configuration as presented by the interface of the Interactive Configuration mode. The left part shows the current configuration of a task; and the rest of the interface shows from left to right and up to down the following information: first row shows available capabilities that are suitable for the task at hand, ranked according to the similarity of the current problem to past configuration cases, and a description of the capability currently selected by the user; the second row includes the pending preconditions, assumptions and goals (postconditions), and the unavailable or missing knowledge-roles; and finally, the bottom row of the interface shows the already achieved preconditions, postconditions, assumptions and knowledge-roles. In particular, this is an example from the WIM configurable application, which is described in Chapter 7. In this example, the Information-Search task is being configured. Only the configuration of the subtask Aggregate-Items remains unconcluded. The user has to select one of the four aggregation capabilities that are suitable for that task: the *Average*, the *Weighted-mean*, the *OWA* or the *WOWA*.

4.4.5 Searching the Configuration Space

The Knowledge Configuration process is approached as a search process over the space of possible configurations in order to find a configuration that is complete and satisfies all the requirements of the problem at hand.

The search space is $\mathcal{K}(\mathbb{L})$, the set of possible (partial and complete) configurations given a component library $\mathbb{L}$ and a query containing the requirements of the problem (Q). Moreover, a configuration $K \in \mathcal{K}$ can be a solution for the

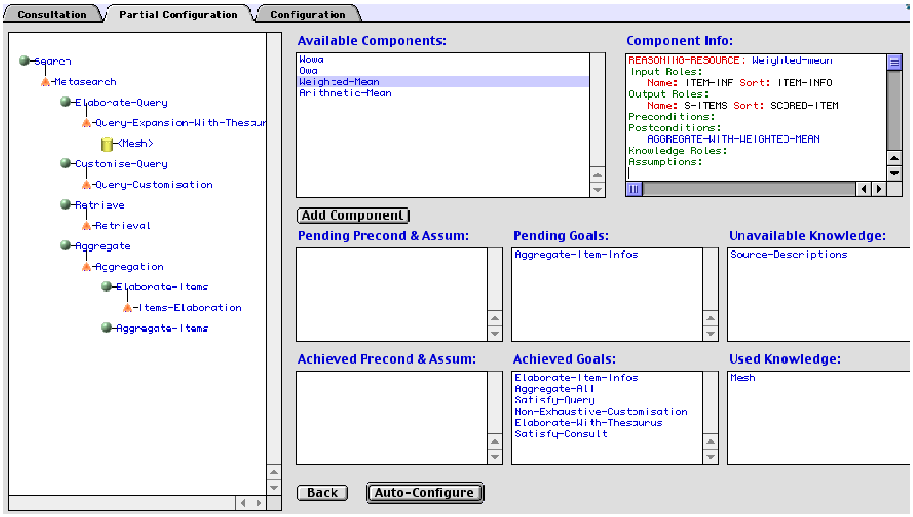

Figure 4.23: Interface that shows a partial task-configuration

query only if K is complete.

Let's start defining a query as a collection of problem requirements and optionally a selected application task.

Definition 4.11 ***(Query)*** *A query* $Q \in \mathcal{Q}$ *is a tuple* $Q = \langle T_0, in, out, pre, post, dm \rangle$

where T_0 is the application task, which characterizes the type of problem the application will be configured for, in, out are the input and output signatures describing the type of data available and the type of data expected as a result of the Cooperative Problem-Solving process, preconditions *pre* are properties that are stated to be true, postconditions *post* are properties to be achieved by the "execution" of the configuration (the performance of a CPS process based on that configuration), and $dm \subset \mathcal{M}$ is the set of allowed domain-models.

The Knowledge Configuration search process starts with a query Q and an empty configuration, and searches new states that model more detailed configurations by adding configuration schemas and recursively configuring them until a complete configuration is found. The *Constructive Adaptation* model tells us that we can improve the construction of the solution by using cases to decide which states to explore first—i.e. which configuration schemas to add to a given partial configuration. This search process adds configuration schemas until a complete configuration K is reached, and then checks if this K satisfies the query Q: if correct then a solution has been found and the process terminates; otherwise the search algorithm proceeds exploring other branches.

We turn now to consider how to represent a *state* in the Knowledge Configuration process. The main issue to be represented in a state is the set of

task-capability bindings ($T \doteq C$) in effect within a partial configuration. For this purpose the state needs not to represent the whole configuration K but only the subset of tasks included in a configuration $T_K \subset \mathcal{T}$ that are bound to a capability.

The second important issue for a state is determining which pre- and post-conditions of the query Q are satisfied by the components involved in a partial configuration—and which are not yet satisfied. Finally, we are interested in states that represent valid configurations, so states have to satisfy the task-capability matching relation for every task-capability binding ($T \doteq C$) in a state.

Definition 4.12 *(State) A* state *Z for a configuration K in a Knowledge Configuration process with a query Q is a tuple* $Z(K,Q) = \langle cpre, cpost, opre, opost, okr, ckr, ot, tc\rangle$

where *cpre* and *cpost* are the closed preconditions and postconditions respectively (those in query Q that are satisfied in Z), *opre* and *opost* are the open pre- and post conditions (those in query Q that are not satisfied in Z), *okr* and *ckr* are the open and closed knowledge-roles (those required by capabilities bound to a task), *tc* is a set of task-capability bindings ($T \doteq C$), and *ot* is the set of open tasks (those tasks in the configuration K that are not bound to any capability).

A state Z is *valid* iff $\forall(T \doteq C) \in Z_{tc} \Rightarrow match(T,C) = true$.

State
open-postconditions → set-of *Formula*
open-preconditions → set-of *Formula*
closed-postconditions → set-of *Formula*
closed-preconditions → set-of *Formula*
open-knowledge-roles → set-of *Signature*
closed-knowledge-roles → set-of *Signature*
open-tasks → set-of *Task*
tc-bindings → set-of *Task-Capability-binding*

Figure 4.24: Features characterizing a state

Figure 4.24 shows the features characterizing a *state*: *open-postconditions* and *open-preconditions* are those postconditions and preconditions not yet satisfied in the current state; *closed-postconditions* and *closed-preconditions* are the preconditions and postconditions satisfied by the partial configuration; *open-tasks* attribute holds the tasks that have no capability bound to them; *open-* and *closed-knowledge-roles* refer to the knowledge-roles to be filled or already filled respectively, by a domain-model (from those established by the query); and *tc-bindings* is a collection of task-capability bindings (i.e. pairs composed of a task bound to a capability ($T \doteq C$)).

Next we are going to explain how the search proceeds for the Knowledge Configuration process. For this purpose we are going to answer with the following questions:

1. how initial states are generated from the library of components (§ 4.4.5);
2. how succesor states are generated (§ 4.4.5);
3. how the state search is guided by case-based retrieval (only for the Constructive Adaptation strategy, described in § 4.5);
4. and how final states (solutions) are detected (§ 4.4.5).

Initializing the search process

The initialization process takes the specification of problem requirements or query Q and a library of components $\mathbb{L}$ as inputs and produces the initial state as output (initial-states: $\mathcal{Q} \times \mathbb{L} \rightarrow \mathcal{Z}$).

The first state is generate according to the application task T_0 specified in the query Q:

$$Z^0 = \langle \emptyset, \emptyset, Q_{pre}, Q_{post}, T_0, \emptyset \rangle$$

The application task T_0 is a starting point for the search process, thus this task becomes an open task, since there is no capability bound to it yet. The pre-conditions and post-conditions of the query Q become open pre- and post-conditions, and there are neither closed pre- and post-conditions, nor task-capability bindings.

If there is no application task selected, then all tasks satisfying the query $\mathcal{T}_Q = \{T_i \in \mathcal{T} | match(Q, T_i)$ are established instead as legitimate starting points for the search process. Consequently, for each task $T_Q^i \subseteq \mathcal{T}_\mathcal{Q}$ an initial state is generated as follows:

$$Z^i = \langle \emptyset, \emptyset, Q_{pre}, Q_{post}, T_Q^i, \emptyset \rangle$$

Successor states

The process of generating *successor* states from a given state (the successors function) is basically the addition of a new task-capability binding to those present in the *tc-bindings* of the given state (Z_{tc}). We are interested in retrieving from the library a capability that matches one of the open tasks (Z_{ot}).

We take, for a state Z, a task from open tasks $T \in Z_{ot}$ and retrieve[2] a collection $\mathcal{C}_T$ of capabilities such that they match with T. In addition, only capabilities which knowledge-roles can be filled in by domain-models included

[2]The particular retrieval method (based on subsumption) that we use has been described in [Arcos and López de Mántaras, 1997].

in the query Q_{dm}, and which assumptions are guaranteed for all its knowledge-roles, are allowed. Therefore, we can now define the set of capabilities $\mathcal{C}_T$ that can be bound to a task, as follows:

$$\mathcal{C}_T = \{\ C \in \mathcal{C} \mid match(T, C) \wedge match(C, Q_{dm})\}$$

where $match(T, C)$ is a task-capability match (definitions 4.1 and 4.4) and $match(C, Q_{dm})$) is a capability-domain match (definitions 4.2 and 4.5), and Q_{dm} is the set of domain-models specified in the query Q.

A new successor state of Z is generated for each capability $C^h \in \mathcal{C}_T$. A successor state Z^h has no longer T as an open task and has a new binding $(T \doteq C^h) \in \mathcal{C}_T$. In addition, incorporating a new capability C^h achieves some new postconditions that were not yet achieved in Z; therefore the generation of the successor state updates the pre- and post-conditions that are open and closed.

When C^h is a task decomposer, it will introduce new subtasks that are to be considered now as open tasks—and, since a new open task can introduce new pre- and postconditions, the open and closed pre- and postconditions have to be revised accordingly. Thus, the successor state is

$$succ(Z, C^h) = \langle cpre^h, cpost^h, opre^h, opost^h, Z_{ot} \cup C^h_{st}, Z_{tp} \cup (T \doteq C^h)\rangle$$

and $cpre^h$, $cpost^h$, $opre^h$, $opost^h$ are the task-decomposer open and closed pre- and postconditions.

Another source of variability depends on the domain-models that can be sensibly used by the capability C^h. Let us consider the set of domain-models in the query $\mathcal{M}^h \subseteq Q_{dm}$ that satisfy the signature specification of the capability knowledge-roles C^h_{kr}. If there is a one to one mapping from the domain-models in $\mathcal{M}^h$ to the signatures in C^h_{kr} then the pair $(C^h, \mathcal{M}^h)$ is unique. However, if there is a many to one (non-injective) mapping from the domain-models in $\mathcal{M}^h$ to the signatures in C^h_{kr} then there may be several pairs $(C^h, \mathcal{M}^h_i)$ where $\mathcal{M}^h_i \subset \mathcal{M}^h$ has a one to one mapping to the signatures in C^h. In this situation one successor state is generated for each pair.

Final states

The verification of whether a state Z^G is a solution to the configuration problem (the `goal-test` function) has to test whether a (task) configuration is *complete* and *valid*:

1. A configuration is *complete* if all tasks are bound to some capability, thus all we need to check is whether there are no open tasks, i.e. whether $Z^G_{ot} = \emptyset$.

2. A configuration is *valid* if all the problem-requirements are satisfied, that is verified when there are no open-postconditions in the state, i.e. whether $Z_{opost} = \emptyset$

A more formal definition of a valid configuration is based on the verification of a satisfiability relation between the configuration obtained in state Z^G and the problem requirements imposed by the query Q ($sat(Q, Z^G)$) as follows:

$$sat(Q, Z^G) \Longleftrightarrow (Q_{pre} \Rightarrow Z^G_{pre}) \wedge (Z^G_{post} \Rightarrow Q_{post})$$

The same definition specialized with the subsumption relation over Feature Terms is the following:

$$sat(Q, Z^G) \Longleftrightarrow (Q_{post} \sqsubseteq Z^G_{post}) \wedge (Z^G_{pre} \sqsubseteq Q_{pre})$$

That is to say, the state Z^G satisfies the initial query Q when all postconditions imposed by the query are satisfied (subsumed) by the postconditions of state Z^G and when all preconditions required by the capabilities included in the configuration of the state Z^G are satisfied (subsumed) by the preconditions established by the query.

We know, by construction, that the completeness condition is assured by $Z^G_{ot} = \emptyset$ and the validness condition is assured by $Z_{opost} = \emptyset$.

4.5 Case-based Knowledge Configuration

The *Constructive Adaptation* strategy views the Knowledge Configuration process as a search process guided by case-base information.

In addition to develop a Knowledge Configuration algorithm as a search over the space of possible configurations, we are going to use a case-based retrieval approach to rank the successor states according to the similarity of the current problem to past configuration problems. This is the reason to introduce the notion of a *configuration case* as a pair composed by a query and a past solution; specifically a configuration case is a pair (Q, K) where Q is the query, that contains some problem requirements and K is a configuration of components that satisfies Q.

Definition 4.13 ***(Configuration Case)*** *A case is a pair (Q, K) where $Q \in \mathcal{Q}$ and configuration $K \in \mathcal{K}$ is a complete configuration. A configuration case is* valid *if K satisfies Q.*

A case is thus a pair composed by a specification of problem requirements (the query to the system) and the task-configuration that is a solution (that satisfies) for the input query. A case base B is a collection of configuration cases $B = \{(Q, K)\}_{1...N}$ and B is a *valid* case base if all cases are valid.

Constructive Adaptation uses information of similar cases to guide the search process. Since in the ORCAS Knowledge Configuration process the main step is to choose a new capability hypothesis to include in a successor state, the retrieved cases are used to decide the selection of capability hypothesis. Let us suppose that we are trying to find the configuration for a query Q, that $CB = \{(Q_i, K_i)\}$

is a case base (a collection of cases), and that $Sim(Q, Q_i)$ is a similarity measure that assesses the relevance of cases in CB with respect the problem query Q.

Since a new successor state of Z is generated for each capability in $\mathcal{C}_T$ let us call $\mathcal{Z}_T$ the set of those states. Moreover, at any point in the search process there is a set of states that are "open", i.e. states from which successor states have yet to be generated; let us call $\mathcal{Z}_{open}$ the set of those states. We will use similar cases to order the set of all open states $\mathcal{Z}_T \cup \mathcal{Z}_{open}$; the search process will use this ordering to decide the next state from which successor states are spawned. Therefore, the search process follows a first-search strategy where a similarity assessment of states against past configuration cases is used as an heuristics to decide the "best" successor state.

We use the similarity measure $Sim(Q, Q_i)$ to rank the cases in the case base $CB = \{(Q_i, K_i)\}$ with respect to their similarity to our current problem Q. Once they are ordered, we need to transfer this ordering to the the set of all open states $\mathcal{Z}_T \cup \mathcal{Z}_{open}$. For this purpose, let us define two new elements:

- $\mathcal{C}_{K_i}$ is the set of capabilities used in some task on configuration K_i.
- C_{Z_j} is the new capability that has been introduced as hypothesis when state Z_j was generated.

Now, the new ordering over states is computed by transferring similarity S over cases to a similarity S_Z over states. Specifically, we define state similarity $S_Z(Q, Z_j)$ as follows:

$$S_Z(Q, Z_j) = \max_{Q_i \in CB} \{Sim(Q, Q_i) | C_{Z_j} \in \mathcal{C}_{K_i}\}$$

That is to say, for each open state $Z_j \in \mathcal{Z}_T \cup \mathcal{Z}_{open}$ we consider the newly added hypothesis (C_{Z_j}), then we check in which cases the capability C_{Z_j} appears as part of the configuration ($\mathcal{C}_{K_i}$), and we take the most similar (to Q) as the degree of similarity for state Z_j.

Thus, Constructive Adaptation proceeds by constructing the solution guided by the information of cases embodied by the similarity measure S_Z. At every decision step, Constructive Adaptation selects the state Z_{max} w.r.t S_Z as the best node to expand the search tree. Or, in other words, the best-first process uses S_Z in the cost function that decides whether a state Z is better than another state Z', as follows

$$\textit{Similarity-fn}(Z, Z') = S_Z(Q, Z) > S_Z(Q, Z')$$

.

Figure 4.25 shows the relation between the problem space and the solution space according to a similarity (distance) assessment. In ORCAS the problem space is defined by the set of possible problem requirements, while the solution space is the set of possible configurations. Given a problem that is similar to the current problem (R distance), then the solution of the previous case will be near to the solution for the current problem (A distance).

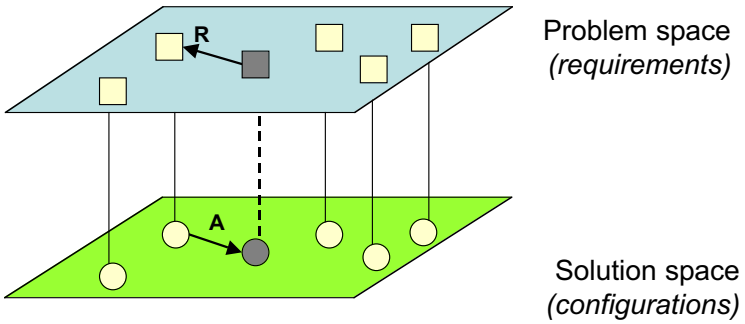

Figure 4.25: Relation between similarities in the problem space and the solution space

4.6 Configuration as reuse

One of the motivations of this work can be summed up in the idea of *reuse*, which is closely related to the notion of *domain independence*. We aim at providing a framework that maximizes the reuse of agent capabilities over new application domains. But what we mean by "reuse"?

Looking at the literature we have realized that there are two main approaches to the reuse issue, though they can adopt different nomenclatures and may be representative of different research lines, yet we can classify them in two general categories, that we call *engineering-oriented reuse* and *execution-oriented reuse*.

- *Engineering-oriented reuse*: this category includes those efforts devoted to the design of new applications by retrieving and adapting existing components. In general, we consider this class of reuse when components cannot be automatically composed and connected, but they rather require some adaptation in order to be executed. Most of the work carried over in the field of Knowledge Engineering and CBSD fall within this category. In general, there are very general components (like the so called "Off-the-Shelf" components) that can be parameterized to fit the requirements of a new problem, or it is necessary to modify the code of some components and then recompile. Usually there is a single application that is configured of tailored for a specific application by a software or knowledge engineer. This approach allows to develop very specific applications with reduced development costs, and enables the interoperation of fairly heterogeneous components, but requires the participation of an expert (the knowledge or software engineer).

- *Execution-oriented reuse*: this category includes those systems where components can be connected and executed in a very dynamic way, with minimal human guidance. The point is that the final user should be able

to drive the reuse of existing components without a deep knowledge of the components. The notions of "on-the-fly or "plug-and-play" applications fall within this category. The idea is that there exist a library of components that can be straightforward connected to interact, without adaptation or modification of the code. Therefore, there are many possible configurations that can be configured on-demand to better fit the problem at hand, rather than configuring a single application for a very specific problem.

The ORCAS framework maximizes the reuse of existing components in both the above approaches to reuse. The key to maximize reuse is the decoupling of the different types of components in the Abstract Architecture, even at the semantic level, by allowing them to be specified using independent ontologies providing the *semantics* of the components.

However, the Knowledge Configuration process as discussed here falls within the second approach to reuse (Execution-oriented reuse), since it is an automated process that allows a non expert user to specify the requirements of the problem at hand, and is able to obtain a configuration of components that can be then operationalized on runtime by forming a team of agents customized for that configuration. We assume that all the components share the same ontologies and use the same infrastructure to communicate; otherwise, it cannot be guaranteed that a configuration is found, or that two agents selected for a team can interact. Further work can be started here to work upon the consideration of different ontologies and the application of ontology-mappings to avoid ontology mismatches.

The engineering aspects about the use of the ORCAS KMF (e.g. the use of ontology mappings) are out of the scope of this work, that is focused on the automatic configuration of MAS based applications, though they are briefly addressed in the chapter about methodological guidelines (Chapter **??**).

Concerning the issues of reuse and configurable applications, we give now some definitions to better characterize the idea of reuse as used in the Knowledge Configuration process. First of all we will introduce the notion of a *library* and an *on-the-fly application*, and then we will introduce the notion of a *configurable application*.

An *On-the-fly Application* is a particular configuration of problem-solving components (tasks and capabilities) that are able to solve a task T_0 in some application domain, according to some problem requirements. An on-the-fly application is the result of connecting the tasks and capabilities of a particular task-configuration to the domain-models characterizing an specific application domain.

Definition 4.14 ***(On-the-fly Application)*** *An on-the-fly application* $\mathbf{A}$ *is a tuple* $\mathbf{A}(Q, \mathbb{L}) = \langle \mathrm{T}, \mathrm{C}, \mathrm{M}, K, \mathbb{O} \rangle$

where $\mathbb{L}$ is a library of domain-independent problem-solving components (tasks and capabilities, represented as $\mathcal{T}$ and $\mathcal{C}$) Q is a query containing problem

requirements, including an application task $T_0 \in \mathcal{T}$, and a collection of domain-model characterizing the application domain Q_{dm}; K is a task-configuration for the application task T_0; T $\subseteq \mathcal{T}$ is the set of tasks used in K, C $\subseteq \mathcal{C}$ is a set of capabilities included in K; M is a set of domain-models to be used by the capabilities in C; and $\mathbb{O}$ is the Object Language.

A *Configurable application* is the result of linking a library of components (tasks and capabilities) to a collection of domain-models characterizing some application domain. In a configurable application there are multiple possible configurations over the components of the library. While an on-the-fly application is a particular configuration of components to solve a specific task, a configurable application is potentially able to solve many types of problems (each task defines a problem type), and the same type of problems can be solved in different ways, using different task-configurations. A configurable application have multiple, alternative capabilities to solve the same class of problems, or there are multiple domain-models available to choose from.

Definition 4.15 ***(Configurable Application)*** *A configurable application is a tuple* $\mathbf{A}(\mathbb{L}, \mathcal{M}) = \langle \mathcal{T}, \mathcal{C}, \mathcal{M}, \mathcal{K}, \mathbb{O} \rangle$

where $\mathbb{L}$ is a library of problem-solving components (tasks and capabilities), $\mathcal{M}$ is a collection of domain-models characterizing the application domain, $\mathcal{T}$ is the set of tasks in $\mathbb{L}$, $\mathcal{C}$ is the set of capabilities in $\mathbb{L}$, $\mathcal{M}$ is a set of domain-models to be used by $\mathcal{C}$ (all the components share a common ontology or there exist a collection of mappings between ontologies such that they be treated as sharing a single ontology), and there exist a collection of task-configurations over the components $\mathcal{K}$ ($\mathcal{K} \in \mathbb{K}$). A configurable application is not an application in the classical sense, it is rather a collection of components that can be connected and configured "on-the-fly", by finding one of the possible configurations in $\mathcal{K}$ that satisfies the requirements of a specific problem, as far as the problem can be characterized by one the tasks in $\mathcal{T}$. In other words, a configurable application can be seen as a collection of potential on-the-fly applications sharing the same application domain.

An example of a configurable application is shown in Chapter **??**, WIM.

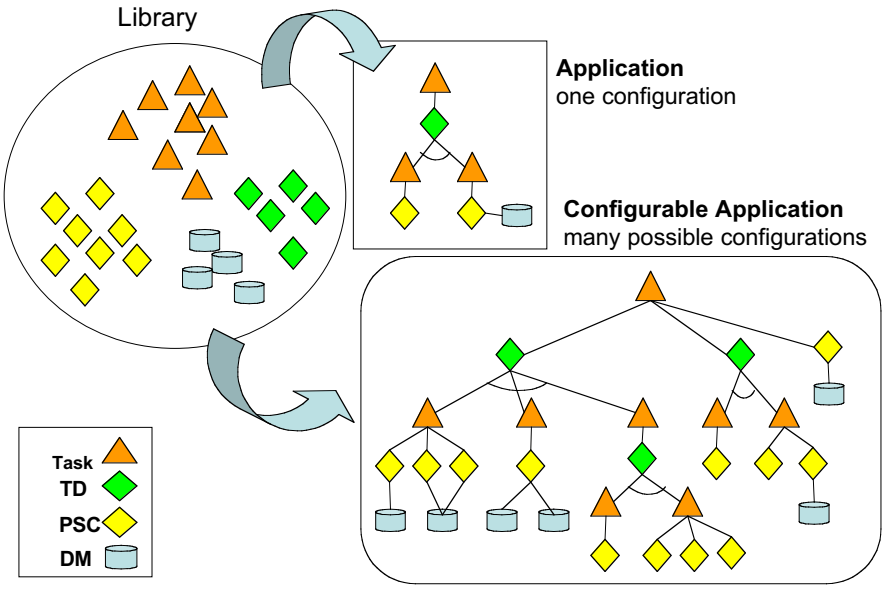

Figure 4.26: Library, application and configurable application