Extending matchmaking to maximize capability reuse

Mario Gómez and Enric Plaza
IIIA-CSIC
{mario, enric}@iiia.csic.es

Abstract

This paper describes an extension of semantic matchmaking that aims at maximizing the reuse of agent capabilities over new application domains. Our approach is to use an Agent Capability Description Language (ACDL) not only to describe the requests and the advertised capabilities, but also to describe the domain-models characterizing specific application domains. The description of tasks and capabilities is independent of any particular domain, though a capability can specify the knowledge requirements to be verified by the application domain. Therefore, capabilities can be used by a team to solve a request whenever theirs knowledge requirements are satisfied by the application domain.

1. Introduction and motivation

The usual approach to overcome the interoperability problems arising in open MAS environments is based on middle agents [5] which mediate between requesters and providers, e.g. matchmakers [6], facilitators [7, 11] and brokers[19]). 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). Matchmaking allows to verify whether a capability can solve a new type of problem, but the reuse of existing capabilities over new application domains is difficult because capabilities are usually associated to a specific application domain.

We aim at extending matchmaking in order to maximize the reuse of capabilities and tasks over new domains. Our proposal to achieve this goal is the use of a Knowledge Modelling Framework (KMF) as the basis of an Agent Capability Description Language (ACDL). Within this framework agent capabilities are described independently of any specific domain, but a capability can declare the type of knowledge it requires and the properties to be verified by the application domain in order for the capability to be sensibly used. Therefore, in addition to use a *task-capability matching* relation during the matchmaking process, we propose a new kind of matching relation called *capability-domain matching*. This matching relation compares a capability with a collection of domain-models characterizing the application domain, and has the purpose of deciding whether the capability can be sensibly applied using knowledge from that domain.

Furthermore, our work addresses the composition of capabilities in order to solve complex tasks that cannot be achieved by a single capability. This process has been designed and implemented as a search process over the space of possible capability compositions, and is called Knowledge Configuration. The proposed matching relations are used by the Knowledge Configuration process to decide whether a capability is suitable to solve a task. The result of the Knowledge Configuration process is a task-configuration, a hierarchical decomposition of a target task into subtasks, and capabilities bound to tasks according to the allowed matching relations, such that the resulting configuration satisfies the global problem requirements. A task-configuration is used as a team-design that guides the team formation process and reduces its complexity, thus allowing the team to be customized on-demand, according to the problem at hand.

Both the KMF and the Knowledge Configuration process are part of the ORCAS framework, a multi-layered framework for the design, development and deployment of Cooperative MAS. An overview of the ORCAS KMF is presented in section §2. Our proposal to extend matchmaking by introducing a capability-domain matching relation is presented in §3, and a brief description of the Knowledge Configuration process is presented next. Finally, some re-

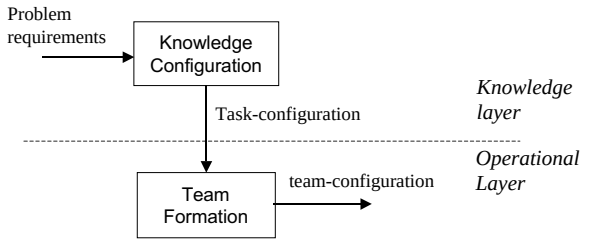

Figure 1. Two layers team formation

lated work is discussed in §5.

2. Overview of the Knowledge Modelling Framework

The ORCAS framework a new model of the team formation process at two levels: the knowledge level and the operational layer (figure 1).

- Team formation at the knowledge level refers to the design of a team in term of tasks, capabilities and domain models satisfying the problem at hand. We call this process Knowledge-Configuration, and the result is called a *task-configuration*, a hierarchical tree that decomposes a task into a subtasks, where each task is bound to a capability suitable for it, and optionally some domain-models satisfying the knowledge requirements of the capability.
- Team formation at the operational level refers to the process of forming a team of agents to solve a problem in a cooperative way, according to a task-configuration. This process is performed by allocating tasks to agents and instructing selected team members on the capabilities to use and the agents to cooperate with.

The ORCAS Knowledge Modelling Framework proposes a conceptual description of Multi-Agent Systems at the *knowledge level* [18], abstracting the specification of components from implementation details. The purpose of the *Knowledge Modelling Framework* (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 capability reuse across different domains; on the other hand, the KMF provides the basis for an Agent Capability Description Language (ACDL) supporting the automatic, on-demand configuration of agent teams according to stated problem requirements.

The ORCAS KMF consist 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 matching relations constraining the way in which components can be connected.
- The *Object Language* defines the representation language used to formally specify component features. Several languages can be used as the Object Language, as far as they endorse an inference mechanism enabling automated reasoning over component specifications.
- The *Knowledge Configuration* process is a search process aiming at finding a configuration of components (tasks, capabilities and domain-models) such that the requirements of a problem are satisfied. The result of the Knowledge Configuration process is a hierarchical decomposition of a target task into subtasks called a *task-configuration*.

The Abstract Architecture specifies which are the components used to build an application (tasks, capabilities and domain-models), and the way in which these components can be connected (the matching relations) in order to produce a valid application.

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* (PSM) 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 the agent communication language and the interaction protocol required to communicate with an agent. Adopting such a KMF we expect the ORCAS Abstract Architecture to provide an effective organization for constructing libraries with large "horizontal cover"[1], thus maximizing reusability and avoiding the brittleness of traditional, monolithic libraries[17].

Figure 2 shows the components we use to describe a MAS, and table 1 summarizes the main features characterizing each component. Tasks are used to describe the types of problems that a Multi-Agent System is able to solve, while capabilities are different methods agents are equipped with to solve tasks. While tasks are generic problem specifications abstracted from any particular implementation, agent capabilities refer to concrete, implemented methods to solve problems. Finally, domain-models are used to represent application domain knowledge, whether the knowledge is provided by a shared repository, hold by an agent, or provided by an external information source.

1 Horizontal cover refers to the range of problem solving behaviors supported by a library. Actually, several libraries of problem-solving components have been described using this approach, like search, classification, diagnosis and parametric design [1, 25, 16]

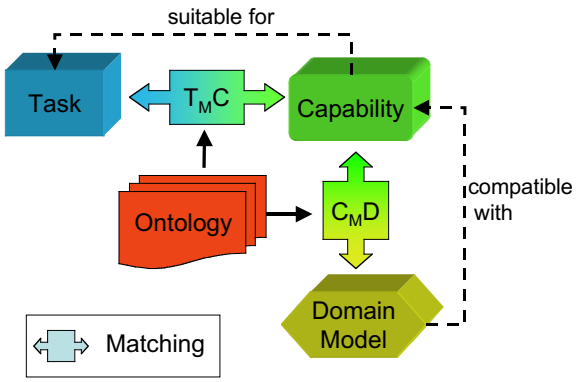

Figure 2. The ORCAS Abstract Architecture

There are three types of components at the knowledge-level, namely: *task*, *domain-model* and *capability*, which have two further subtypes: *skill* and *task-decomposer*. The description of any component contains a customizable specification of pragmatics aspects (e.g. name, description, creator, publisher, evaluation, etc.) and the ontologies providing the terminology used to specify component features, plus a variable number of component specific features. Table 1 sums up the features used to specify each component, where 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 $\mathbb{O}$; and $prop, mk$ are properties and meta-knowledge respectively, also specified by formulae in $\mathbb{O}$. There are other features that have not been included here, like the communication and the operational description of a capability, since they play no role in our definition of the matching relations.

Task	T = $\langle in, out, pre, post \rangle$
Capability	$C = \langle in, out, pre, post, asm, kr \rangle$
Task-decomposer	$D = \langle in, out, pre, post, asm, kr, st \rangle$
Skill	$S = \langle in, out, pre, post, asm, kr \rangle$
Domain Model	$M = \langle kr, prop, mk \rangle$

Table 1. Types of knowledge components and their main features

In this framework, both capabilities and tasks are described independently of a particular application domain. The point is that a capability declares the type of knowledge and the assumptions to be verified by the application domain, therefore any knowledge base providing the required type of knowledge and verifying the assumptions of the capability can be sensibly used by the capability. It should be remarked that domain-models do not contain knowledge, they are mere descriptions of knowledge types and properties verified or assumed to be true by a particular knowledge-base. Let's see an example from WIM (the Web Information Mediator) [12], a configurable MAS that is able to perform complex information search tasks in the Web. There is a capability called Generalize-query that takes a query as input and produces a collection of more general queries using hypernyms of the keywords in the input query. The hypernyms used by the capability are not included as an input, because they are considered domain knowledge, and as such they will depend on the specific application domain. The capability is then independent of any application domain, like medicine or engineering, but it declares the type of knowledge it requires, a thesaurus. In addition, this capability imposes two assumptions to be verified by a thesaurus: providing hypernyms and having no cycles (in general a thesaurus may be a graph). Any thesaurus is a potential candidate, but only those thesaurus complying with the assumptions of the capability can be sensibly used. For instance, the MeSH thesaurus satisfies those properties for the medical domain.

Each component in the Abstract Architecture is characterized by some features, but the particular language used to specify these features is independent of the Abstract Architecture, and is called 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). Although the KMO is not dependent of any particular Object Language, it declares two concepts that should be further refined by the Object Language to yield a precise, computer interpretable meaning: *Signature-element* and *Formula*. Two examples of expressive languages that can be used as the Object Language are DAML-OIL and OWL. However, the decision on which language to use should take into account not only the expressiveness, instead a trade-off between expressiveness and efficiency is preferable so as to use the language in practice.

In the ORCAS KMF, components can be described using their own, independent ontologies [10]. Because of this conceptual decoupling, ontology mappings may be required to match components when there is a ontology mismatch between two specifications. Nevertheless, we focus here on the matching relations, assuming that either the necessary ontology mappings are already built or all components share the same ontologies. This is a reasonable assumption, since it seems feasible and convenient to build the mappings beforehand, previously to make a component available for its use. Actually, existing agent infrastructures for

open MAS also rely upon this assumption.

3. Matching relations

In addition to provide a Knowledge Modelling Ontology for describing tasks, capabilities and domain-models, the Abstract Architecture imposes some architectural constraints, specified as *matching relations*. These relations restrict the way components can be connected when building a task-configuration during the Knowledge Configuration process.

Matchmaking is the process of verifying wether a matching relation (also referred as a "match") between two components holds. Figure 2 shows the components in the Abstract Architecture and the matching relations than can be established among components. Matching relations are verified by comparing two specifications, and have 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 fits the requirements of another. We extend the usual approach to matchmaking in two ways: firstly, in addition to a *task-capability matching* we include also a *capability-domain matching*; and secondly, we introduce a Knowledge Configuration process that goes beyond basic component matchmaking to provide the basis for an automated design of agent teams satisfying stated problem requirements. These are the two types of matching relations:

- 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 corresponding 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 application domain knowledge. 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 every knowledge-role.

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 using an Object Language, matching relations should be further refined in term of a basic relation between elements (signature-elements and formulae) expressed in 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 OR-CAS Abstract Architecture can use the matching relations as defined at the abstract level. We will focus here on the abstract-level matching.
- 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 specific *object relation* that is defined among elements (signature-elements and formulae) in the Object Language.

Our approach to component matching is based on a combination of *signature matching* [26] and *specification matching* [27], that we prefer to call *competence matching*. Signature matching relations compare the interface of two components in terms of the types of information they use (inputs and knowledge-roles) and produce (output). Competence matching relations compare the preconditions and postconditions of two components to determine wether two components are substitutable, or wether a component satisfies the requirements of another.

3.1. Task-capability matching

We define a *Task-capability match* as the conjunction of a *Generalized Type Match* over the input signature specification, a *Specialized Type Match* over the output signature specification [26], and a *Plug-in Match* [27] over the competence specification.

Definition 1 ***(Task-capability match)*** $\mu(T, C) = (T_{in} \geq C_{in}) \wedge (T_{out} \leq C_{out}) \wedge (T_{pre} \Rightarrow C_{pre}) \wedge (C_{post} \Rightarrow T_{post})$*, where* T *is a task and* C *is a capability,*

The *Generalized Signature Match* over the input signature ($T_{in} \geq C_{in}$) means that the capability has an input signature C_{in} equal or more general than the task input signature T_{in}. Inversely, the *Specialized Signature Match* over

the output signature ($T_{out} \leq C_{out}$) means that the capability has an output signature more specific than the task output signature. This combination of generalized and specialized match has the following justification: on the one hand, a capability C with a more general input than a task T implies that all the information required by C_{in} can be obtained from T_{in}. However, if C_{in} is more specific than T_{in} (with more information), then C is not assured to obtain all the input information from T_{in}, which can result on a capability being incorrectly applied. On the other hand, a capability with an output signature C_{out} more specific than T_{out} means that C_{out} is able to provide all the information specified by T_{out}, which is not guaranteed when C_{out} is more general (with less information) than T_{out}.

Moreover, the *Plug-in Match* ($(T_{pre} \Rightarrow C_{pre}) \wedge (C_{post} \Rightarrow T_{post})$) requires a capability C to have equal or weaker preconditions than a task T and equal or stronger postconditions than T. 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 holds, the selected capability guarantees that the postconditions of the task will hold after applying the capability.

The demonstration of the former property from the definition is follows: if T_{pre} holds then C_{pre} holds (because of the first conjunct of the Plug-in Match). Since we interpret C to guarantee that $C_{pre} \Rightarrow C_{post}$, we can infer that C_{post} will hold after executing C. Finally, since, the second conjunct is $C_{post} \Rightarrow T_{post}$, then we are assured that T_{post} will hold after executing C.

3.2. Capability-domain matching

Whilst task-capability matching is defined between a task and a capability, capability-domain matching is defined between a task and one or several domain-models. A capability may introduce more than one knowledge-role, and each knowledge-role may be filled in by a different domain-model, thus several domain-models would be required to satisfy the knowledge requirements of a single capability.

If a capability introduces only one knowledge-role, then we can say that a capability matches a domain model when: (1) the domain-model provides the kind of knowledge characterized by the capability knowledge-role, and (2) satisfies the assumptions established by the capability. However, if a capability specifies more than one knowledge-role, there should exist at least one domain-model matching the specification of the capability for each knowledge-role.

Let's represent 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 single knowledge-role of a capability ($C^i_{kr} \in C_{kr}$) and a domain-model M is called a *partial capability-domain match*, and is defined such that the signature specification of the domain-model knowledge-roles (M_{kr}) is equivalent or more specific than the signature specification of the capability knowledge-role (C^i_{kr}), and the assumptions of the capability for that role C^i_{asm} are satisfied by the union of the properties and meta-knowledge specifications of the domain-model ($M_{prop} \cup M_{mk}$), as follows:

Definition 2 ***(Partial Capability-domain match)*** *$\mu_p(C, M, C^i_{kr}) = (C^i_{kr} \leq M_{kr}) \wedge (M_{prop} \cup M_{mk} \Rightarrow C^i_{asm})$, where C^i_{kr} is a knowledge-role of a capability C, and M is a domain-model; M_{prop} and M_{mk} are the* properties *and* meta-knowledge *of M, and C^i_{asm} are the assumptions of 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 knowledge roles ($C^i_{kr} \leq M_{kr}$), and a new kind of matching 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 ($M_{prop} \cup M_{mk} \Rightarrow C^i_{asm}$).

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 is equal or specializes the signature specification of the capability knowledge-role ($C^i_{kr} \leq M_{kr}$). If M_{kr} was more general than C^i_{kr}, then it may occur that some of the information required by C^i_{kr} cannot be provided by the knowledge characterized by M_{kr}, and thus the capability cannot sensibly use that knowledge.

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 can be inferred from the union of the properties and meta-knowledge of the domain-model ($M_{prop} \cup M_{mk} \Rightarrow C^i_{asm}$).

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 for every knowledge-role specified by the capability.

Definition 3 ***(Capability-domain match)***

$\mu(C, \mathcal{M}) = \forall C^i_{kr} \in C_{kr} : \exists M \in \mathcal{M} | \mu_p(C, M, C^i_{kr})$, *where* C *is a capability,* $\mathcal{M}$ *is a set of domain-models;* M_{prop} *and* M_{mk} *are the properties and meta-knowledge of* M*, and* C_{asm} *are the assumptions of* C*.*

4. Capability composition: the Knowledge Configuration process

The *Knowledge Configuration* process has the goal of finding a configuration or composition of application tasks, agent capabilities and domain-models, in such a way that the requirements of the problem at hand are satisfied.

The input for the Knowledge Configuration process is twofold: on the one hand it takes a *query* specifying the problem requirements and the application domain, and on the other hand it uses a *library* of tasks and capabilities as a "yellow-pages service". 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 knowledge is bound to domain-models satisfying its assumptions, and c) the whole configuration complies to the problem requirements.

A query specifies the application task that better characterizes the type of problem to be solved, a set of domain-models characterizing the application domain, and a collection of problem requirements: input and output signatures describing the type of data available and the type of data expected, preconditions that are stated to be true, and post-conditions to be achieved.

Definition 4 ***(Query)*** *A query* $Q \in \mathcal{Q}$ *is represented as a tuple* $Q = \langle T_0, in, out, pre, post, dm \rangle$*; where* T_0 *is the application task;* in, out *are the input and output signatures,* pre, post *are preconditions and postconditions respectively, and* dm $\subset \mathcal{M}$ *is the set of domain-models characterizing the application domain.*

Notice that the domain knowledge required by an application is not provided within the query, which contains just an abstract specification of available domain knowledge, as specified by the domain models in the query, Q_{dm}. This specification is sufficient to perform capability-domain matchings, but in addition, the agent providing a certain capability must have real access to that knowledge in order to apply the capability. This issue is not addressed here, since we are dealing with matchmaking and capability composition at the knowledge level, while the way an agent access a particular knowledge-base is considered an operational issue, and thus it is kept out of the KMF.

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 5 ***(Library)*** *A library* $\mathbb{L}$ *is a repository of components specified as a tuple* $\mathbb{L} = \langle \mathcal{T}, \mathcal{C}, \mathbb{O} \rangle$*, where* $\mathcal{T}$ *is a set of tasks,* $\mathcal{C}$ *is a set of capabilities, and* $\mathbb{O}$ *is the Object Language.*

Since a task-configuration is a complex structure we need first to define its constituent elements, called configuration schemas. We note $\kappa \in \Bbbk$ as a configuration schema and the set of all configuration schemas; moreover, we 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 6 ***(Binding)*** *A binding* $(T \doteq U)$ *is a pair with a task* $T \in \mathcal{T}$ *in the* head *and either a capability* $C \in \mathcal{C}$ *or a configuration schema* $k \in \Bbbk$ *in the* tail*:* $U \in \mathcal{C} \cup \Bbbk$

A binding is *valid* iff two matching relations hold: (1) a task-capability match $\mu(T, C)$, and (2) a capability-domain match $\mu(C, Q_{dm})$, where Q_{dm} is a set of domain-models specified in the query.

Definition 7 ***(Configuration schema)*** *A configuration schema* $\kappa \in \Bbbk$ *is a pair* $\langle (T \doteq C), \{(T_i \doteq \kappa_{j_i})\}_{i=1...n} \rangle$ *where* $T, T_1, \ldots, T_n \in \mathcal{T}$*,* $C \in \mathcal{C}$*, and* $\kappa_{j_1}, \ldots, \kappa_{j_n} \in \Bbbk$*, 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 (valid) 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 configuration schemas as follows:

Definition 8 ***(Configuration relation)*** $\mathbb{R}(\kappa, \kappa') \Leftrightarrow \exists (T_i \doteq \kappa') \in tail(\kappa)$*; 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 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. We will note $K \in \mathcal{K}$ a task-configuration and the set of all the task-configurations. 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 and valid configuration is found.

A task-configuration K is *complete* when all schemas belonging to K are complete, and K is *valid* if all the task-capability bindings are valid. K is valid for a particular query Q if, in addition to have all the task-capability bindings valid, the root schema of K has the application task T_0 in the binding of its head (K is a configuration of T_0 and all the requirements specified by Q are satisfied by K).

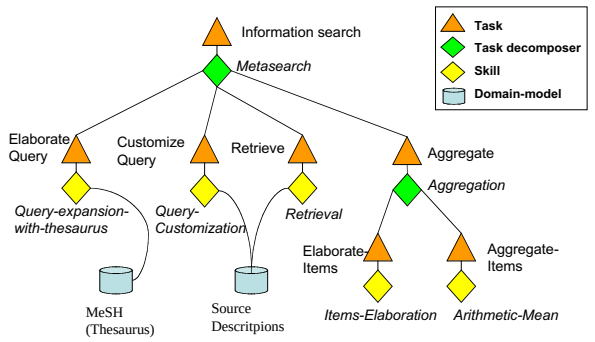

Figure 3. Task-configuration example

Figure 3 shows an example of a task-configuration for the Information-Search task, which is used within the WIM application. This task is being decomposed into four tasks by the Meta-search task-decomposer: Elaborate-query, Customize-query, Retrieve and Aggregate, which is further decomposed by the Aggregation capability in two subtasks: Elaborate-items and Aggregate-items. The example shows some skills requiring domain knowledge, e.g. the Query-expansion-with-thesaurus requires a thesaurus (e.g. *MeSH*, a medical thesaurus), and the Retrieval and Query-customization skills require a description of the information source to be queried.

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). This search process adds configuration schemas until a complete configuration K is reached, and then checks whether K satisfies the requirements in: if correct then a solution has been found and the process terminates, otherwise the search algorithm proceeds exploring other branches.

Three strategies have been implemented for the Knowledge Configuration process: the *Search and Subsume* configuration mode implements a depth first strategy for searching; The *Constructive Adaption* strategy applies a *best-first* search process in the state space [20] using Case-Based Reasoning (CBR) (a measure of similarity between the current state and past configuration cases); and the *Interactive Configuration*, that uses CBR to suggest the best options to the user who is then free to decide the next step.

5. Related work and discussion

Most of the languages used for describing agent capabilities in open environments are based on logical deduction languages like Prolog. Two well known examples are the Interface Communication Language (ICL) used in the Open Agent Architecture [13, 2], and LDL++, used in the InfoSleuth infrastructure [19]. These languages support inferences about whether an expression of requirements matches a set of advertised capabilities, but they do not take into account the reuse of capabilities over new domains. Some steps to overcome this limitation have been introduced with LARKS, the language used within the RETSINA framework [24]. LARKS incorporates application domain knowledge in agent advertisements and requests, specified as local ontologies in the concept language ITL (the concept language is equivalent to our notion of the Object Language). The ORCAS KMF goes a step beyond by stating a clear separation of tasks and capabilities from the domain, as proposed by the KMF community, and introducing a new type of matching between the capabilities and the application domain.

With the introduction of the knowledge level [18] in the development of Knowledge-Based Systems, the knowledge acquisition phase turns from a knowledge transfer approach to a model construction approach [3, 23]. Knowledge Modelling Frameworks propose methodologies, architectures and languages for analyzing, describing and developing knowledge systems [22, 14, 21, 9]. The goal of a 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 [4] and to facilitate reuse [8].

However, KMFs have rarely been applied in the field of MAS to deal with the reuse and interoperation issues in open environments. The ORCAS framework explores the use of a KMF for describing and composing agent capabilities with the aim of maximizing capability reuse. The ORCAS KMF is a conceptual tool for the solution of the “bottom-up design problem” in the field of cooperative Multi-Agent Systems. The “bottom-up design prob-

lem" is an open issue of software composition defined as: given a set of requirements, find a set of components within a software library whose combined behavior satisfies the requirements [15]. In our approach this problem is more precisely defined as: given a set of requirements and domain-models, find a combination of agent capabilities whose aggregated competence (and knowledge) satisfies the requirements. The solution to this problem enables a compositional approach to software development in general and agents teams in particular. The ORCAS KMF is currently being used as an ACDL within the ORCAS agent infrastructure, which achieved a third price in the Agentcities Agent Technology Competition (Infrastructure category).

References

[1] J. Breuker and W. V. de Velde, editors. *COMMONKADS Library for Expertise Modelling*. IOS Press, 1994.

[2] A. Cheyer and D. Martin. The open agent architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1):143–148, March 2001. OAA.

[3] W. Clancey. The knowlede level reinterpreted. *Machine Learning*, 4:285–291, 1989.

[4] W. de Velde. Issues in knowledge level modelling. In J. David, J. Krivine, and R. Simmons, editors, *Second generation expert systems*, pages 211–231. Springer Verlag, 1993.

[5] K. Decker, K. Sycara, and M. Williamson. Middle-agents for the internet. In *Proceedings the 15th International Joint Conference on Artificial Intelligence*, pages 578–583, 1997.

[6] K. Decker, M. Williamson, and K. Sycara. Matchmaking and brokering. In *Proceedings of the 2nd International Conference in Multi-Agent Systems*, 1996.

[7] T. Erickson. An agent-based framework for interoperability. In J. M. Bradshaw, editor, *Software Agents*. AAAI Press, 1996.

[8] D. Fensel. An ontology-based broker: Making problem-solving method reuse work. In *Proceedings Workshop on Problem-solving Methods for Knowledge-based Systems at IJCAI'97*, 1997.

[9] D. Fensel, V. Benjamins, E. Motta, and B. Wielinga. UPML: A framework for knowledge system reuse. In *International Joint Conference on AI*, pages 16–23, 1999.

[10] D. Fensel, S. Decker, E. Motta, and Z. Zdrahal. Using ontologies for defining task, problem-solving methods and their mappings. In *Proceedings European Knowledge Acquisition Workshop*, Lecture Notes in Artificial Intelligence, 1997.

[11] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7), 1997.

[12] M. Gomez, C. Abasolo, and E. Plaza. Problem-solving methods and cooperative information agents. *International Journal on Cooperative Information Systems*, 11(3-4):329–354, 2002.

[13] D. L. Martin, A. J. Cheyer, and D. B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, January-March 1999. OAA.

[14] J. McDermott. Toward a taxonomy of problem-solving methods. In S. Marcus, editor, *Automating Knowledge Acquisition for Expert Systems*, pages 225–256. Kluwer Academic, 1988.

[15] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *Software Engineering*, 21(6):528–562, 1995.

[16] E. Motta. *Reusable Components for Knowledge Modelling*, volume 53 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 1999.

[17] E. Motta, D. Fensel, M. Gaspari, and A. Benjamins. Specifications of knowledge components for reuse. In *Proceedings of SEKE '99, 1999.*, 1999.

[18] A. Newell. The knowledge level. *Artificial Intelligence*, 28(2):87–127, 1982.

[19] M. Nodine, W. Bohrer, and A. Ngu. Semantic brokering over dynamic heterogeneous data sources in infosleuth. In *ICDE*, pages 358–365, 1999.

[20] E. Plaza and J. L. Arcos. Constructive adaptation. In S. Craw and A. Preece, editors, *Advances in Case-Based Reasoning. Proceedings 6th ECCBR*, volume 2416 of *Lecture Notes in Artificial Intelligence*, pages 306–320, 2002.

[21] A. Schreiber, B. J. Wielinga, J. Ackermans, W. V. D. Velde, and R. D. Hoog. CommonKADS: A comprehensive methodology for kbs development. *IEEE Expert*, 9(6):28–37, 1994.

[22] L. Steels. Components of expertise. *AI Magazine*, 11(2):28–49, 1990.

[23] R. Studer, V. R. Benjamins, and D. Fensel. Knowledge engineering: Principles and methods. *Data Knowledge Engineering*, 25(1-2):161–197, 1998.

[24] K. P. Sycara, M. Paolucci, M. V. Velsen, and J. A. Giampapa. The RETSINA MAS infrastructure. Technical report, Robotics Institute, Carnegie Mellon University, 2001.

[25] A. Valente, W. V. de Velde, and J. Breuker. The commonkads expertise modelling library. In J. Breuker and W. V. de Velde, editors, *CommonKADS Library for Expertise Modeling*, volume 21 of *Frontiers in Artificial Intelligence and Applications*, pages 31–56. IOS-Press, 1994.

[26] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.

[27] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.

Acknowledgements

The authors would like to thank the Spanish Scientific Research Council for their support. This work has been developed under the IBROW project (IST-1999-190005).