Importing the Semantic Web into a lightweight formal broker : an exercise in incorporating Description Logic Programs into Logic Programs.

Antoon Goderis

2003

Abstract The first part of this thesis provides a comparative overview of UPML, DAML-S and F-X, three approaches to describe service capability on the Web. In our overview, we have special attention for the reasoning support offered by UPML, because to date it has not received much attention in the Web services literature. In the second part, we use our insights from this comparison to add functionality to F-X. In particular, we facilitate the use of domain ontologies in F-X, and we exploit such domain knowledge to support more flexible competence matching. We achieve this based on query relaxation techniques and a recent translation from a subset of Description Logics to Logic Programs.

Acknowledgements Many thanks to Dave Robertson and Stephen Potter for all the useful discussions and help. Thanks also to Andreas Abecker and Rudi Studer at the FZI/AIFB in Karlsruhe, who provided a warm welcome. A final thanks to my family and friends for their support throughout the MSc.

Contents

Chapter 1

Towards Semantic Web Services

Today, the World Wide Web is deeply affecting the way in which both humans and machines are interconnected on a global scale. With the advent of standards for describing domain knowledge and software services on the Web, there now exists a promise for software components to become available anytime, anyplace. In this chapter, we introduce the vision and technology behind Semantic Web services.

1.1 Vision

Semantic Web Services (SWS) seek to bring together effort on the Semantic Web and Web Services. In the coming sections we briefly discuss these two drivers. Here, we add some background to the vision behind SWS.

Making the Semantic Web and Web Services work crucially depends on moving beyond syntax to computational semantics. Working with semantics in a distributed setting like the Internet requires web-enabled markup languages to enable description of resources and reasoning with them. One way to view the Semantic Web is as a collection of documents, where semantics serve to describe these static resources. Another view acknowledges the dynamic nature of some kinds of resources, namely software components, and envisions a Semantic Web of services. Both views are complementary and rely on exploiting computational semantics. In this report, our focus is on languages to formally describe Semantic Web Services.

The promise is that by marking up services with such formal semantics, intelligent software agents will be in a position to discover, configure, invoke, mediate, compose, execute, monitor, recover, simulate and verify services on the Web. In addition, middleware is needed to hide complexity from users (human or machine) (Fensel & Bussler 2002). Finally, it is expected many ideas from the software agent community can be put to use (Lomuscio,

Wooldridge & Jennings 2001).

There are two ways to interpret the term " Semantic Web Services" (due to Benjamin Grosof). First, SWS can be parsed as {Semantic Web} Services. This includes, in particular, infrastructural services to support the Semantic Web, e.g. the service of providing capabilities for integration of knowledge, information and databases, for inferencing, and/or for translation between different forms of knowledge and databases. Second, SWS can be parsed as Semantic {Web Services}. This includes knowledge-based service descriptions of (parts of) general Web Services. Such descriptions of a service can be used for multiple higher-level tasks about services, such as discovery and composition.

1.1.1 Nothing new under the sun ?

Given these high promises, we should add a caution note. In particular, some worry the Semantic Web rhetoric is but a breakthrough in enthusiasm, and Web Services the name for a marketing initiative (Thompson 2002). We can add to this concern that already in 1991, the potential of knowledge services was clearly recognised, yet still is has not become reality. The vision of (Neches, Fikes, Finin, Gruber, Patil et al. 1991) sounds familiar when compared to what the Semantic Web (Berners-Lee 1999) promises today. To illustrate this, we give a quick summary of the 1991 vision below. The reader is invited to substitute the term "knowledge systems" below with the term "Web services".

With regard to end users, Neches et al. (1991) foresee that the face of information systems will change in three ways. First, knowledge systems will provide sources of information that serve the same functions as books and libraries but are more flexible, easier to update, and easier to query. Second, knowledge systems will enable the construction and marketing of prepackaged knowledge services, allowing users to invoke (rent or buy) services. Third, it will be possible for end users to tailor large systems to their needs by assembling knowledge bases and services rather than programming them from scratch.

With regard to the way developers view and manipulate knowledge systems, they envision three mechanisms that would increase their productivity. These mechanisms are very similar to the major goals formulated for the IBROW project, which we discuss in Section 2.1.2. First, there should be libraries of multiple layers of reusable knowledge bases that could either be incorporated into software or remotely consulted at execution time. At a level generic to a class of applications, layers in such knowledge bases capture conceptualizations, tasks, and problem-solving methods. Second, system construction will be facilitated by the availability of common knowledge representation systems and a means for translation between them. Finally, there is a need for tools and methodologies that allow developers to find and

use library entries useful to their needs as well as preexisting services built on these libraries. These tools will be complemented by tools that allow developers to offer their work for inclusion in the libraries.

1.1.2 Challenges for Knowledge Representation

Why has this vision not been realized yet ? The main difficulty lies in representing computational semantics. Like any other field, knowledge representation (KR) struggles with a number of fundamental research questions. These include the expressibility-tractability trade-off, the knowledge acquisition bottleneck and the issue of how to deal with heterogeneous vocabularies or different conceptualisations of the same domain. A distributed environment like the Web brings further challenges to KR — it does not solve the earlier ones. Unpredictability of how knowledge will be used, multiplicity of knowledge sources, scalability of reasoning, or issues of authority become a big part of the equation (van Harmelen 2002). Regardless of what can be said at this early stage of research on the feasibility of the Semantic Web or Web Services, "grand challenges" are valuable if only to inspire whole generations of researchers (Shadbolt 2003). There is certainly more than this to the idea of SWS, but in what follows, we will further abstain from the debate, and concentrate on the technology available today.

1.2 Web Services

Web Services are the first driver for Semantic Web Services. As a technology, they are mainly industry-driven. Web Services are defined as self-contained, self-describing, modular applications that can be published, located, and invoked across the Web (IBM Web Services Architecture Team 2000). They rely on standards like XML (Schema), SOAP, WSDL and UDDI to provide the web equivalent of remote procedure calls. In what follows, we provide an overview of these languages. We also narrow the focus of this thesis down to one particular aspect of Web service lifecycle management: brokering.

1.2.1 The Web Services language stack

The Extensible Markup Language (XML) (Bray, Paoli & Sperberg-McQueen 1998) allows disparate software agents to interoperate by modelling application boundary crossings in terms of application-specific types, secure in the knowledge that an XML-based representation of these types can be exchanged across language, process, host and vendor boundaries (Box, Skonnard & Lam 2000). In particular, XML documents that are exchanged between software agents often follow a common form or structure that is shared by many documents in a give problem domain. XML Schema (Thompson, Beech, Maloney & Mendelsohn 2001) allows to prescribe the structure of

a document class, such that all instances of such a class can be checked for their (schema-) validity. In this respect, XML Schema can be seen as syntactic (i.e. structural) metadata.

At the lowest level in the Web services "stack", XML is irrefutable as the standard for data encoding and formatting. SOAP (the acronym is now the name) (Mitra 2002) packages these data and transfers it from system to system. It consists of a very simple request/reply mechanism. One layer higher up, SOAP is bound to an actual web protocol such as HTTP. To facilitate and improve web service advertising, discovery and invocation, SOAP has spawned into two other technologies: UDDI (Universal Description, Discovery and Integration) and WSDL (Web Service Description Language) (Chinnici, Gudgin, Moreau & Weerawarana 2003). UDDI offers a forum for organisations to describe the services they offer and inquire about services made available by other organisations. UDDI is complemented by WSDL, which builds upon XML Schema to define an actual XML vocabulary for such service descriptions (Lemahieu 2001).

One problem with the WSDL/UDDI approach is that, whereas traditional distributed object technologies base their search criteria on the type of interface an object implements, this may be totally inadequate for Web service selection. Web services not only deliver some kind of return value to a request, they may also have an effect in the real world (McIlraith & Martin 2003). This effect may be a much more important search criterion; the web service's interface will only be a secondary facet. UDDI does provide a categorization mechanism according to "real world" criteria such as industry branch, product type and geographic location, but it is in no way destined at discovering services based on fine-grained specifications of what is required from the service (Lemahieu 2001). Currently, it is far from clear how these effects on the world would be represented formally. Nevertheless, the idea is there. In any case, there is a need for more expressive service descriptions, which would yield improved support for service discovery, configuration and composition. For this reason, researchers in Knowledge Representation are currently considering how to bring KR to the Web.

1.2.2 Brokering Web Services

In section 1.1.1 we mentioned many of the facets related to Web service lifecycle management. In this thesis, we assume that a human or software agent, in order to make use of a particular service, will walk through (some of) the following steps.

1. Agent formulates a request
2. Agent discovers a broker
3. Agent submits request to a broker

4. Broker discovers services, possibly through composition
5. Broker returns (a list of) (composed) services
6. Agent selects a service, or reformulates its request and goes back to Step 2
7. Agent configures service
8. Agent verifies and/or simulates service
9. Agent invokes/executes service
10. Agent monitors/recovers service

Central to this scenario is that the agent relies on a brokering agent to discover relevant services. In the following chapters, we will mainly focus on techniques to support brokerage of services. The dictionary defines a broker as follows (from Dictionary.com):

> **Broker**
> An agent employed to effect bargains and contracts, as a middleman or negotiator, between other persons, for a compensation commonly called brokerage.

The important notion here is the broker's role as middleman or negotiator. Brokers of software components will mediate components between agents and service providers. In our view, brokering can comprise service composition. Discovering whether a particular service exists or can be assembled from existing services will often require composition techniques.[1]

The framework given in (Decker, Sycara & Williamson 1997) offers a good overview of the different kinds of brokering. It uses the notion of *privacy*, or the degree to which requests and capabilities are made public, as the primary dimension to distinguish between brokering scenarios (see Figure 1.1). Note that these authors call brokers "middle agents", and that they reserve the term "broker" for the situation where both requests and capabilities are publicly known. In future chapters, we will refer back to this framework when we introduce different techniques to support brokering. Many of these techniques will, in some form or other, make the link to the Web. Therefore, in the next section we provide some background on the languages proposed for use on the (Semantic) Web.

[1] Under that same broad view of mediation, brokering could also comprise data mediation.

preferences initially known by	*capabilities initially known by*		
	provider only	provider + middle agent	provider + middle + requester
requester only	(broadcaster)	"front-agent"	matchmaker/ yellow-pages
requester + middle agent	anonymizer	broker	recommender
requester + middle + provider	blackboard	introducer/ bodyguard	arbitrator

Figure 1.1: Brokering framework (Decker et al. 1997)

1.3 The Semantic Web

The Semantic Web is a vision for the future of the Web put forward by the W3C: the idea of having data on the Web defined and linked in a way that it can be used by machines not just for display purposes, but for automation, integration and reuse of data across various applications (W3C 2001).[2]

Semantics can take many shapes, ranging from natural language descriptions to formal specifications. To enable the Semantic Web vision, the semantics or meaning of the data should become accessible for machines. Using only XML and XML Schema as web markup will not do. XML Schema remains largely restricted to modelling syntactic properties, not semantic ones. XML Schema's are a means to provide integrity constraints for information sources, whereas ontology languages are a means to specify domain theories. They allow to describe the structure of the semantics of more complex objects (Klein, Fensel, van Harmelen & Horrocks 2001).

Ontologies are designed to capture consensual understanding: they are defined as formal, explicit specifications of a shared conceptualization ((Borst 1997), based on (Gruber 1993)). As Studer, Benjamins & Fensel (1998) explain, conceptualization refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their

[2]For an illustrative overview, see `http://www.semaview.com/c/SW.html`

use, are explicitly defined. Formal refers to the fact that the ontologies should be machine readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private to some individual, but accepted by a group.

One lesson learned in knowledge representation (KR) over the years is that representation and reasoning should be seen as two sides of the same coin (Davis, Shrobe & Szolovits 1993). There exists a trade-off between expressibility and tractability, which causes reasoning with expressive notations to be hard. As illustrated by current web language proposals like OWL (see Section 1.3.2), this trade-off stays with us when we tranfer knowledge representations to the Web.

1.3.1 Resource Description Framework (RDF)

The Semantic Web builds on XML's ability to define customized tagging schemes and RDF's flexible approach to representing data.[3] RDF (Lasilla & Swick 1999) is a datamodel for objects ("resources") and relations between them. It provides a simple semantics for this datamodel, and these datamodels can be represented in an XML syntax. RDF Schema (RDFS) (Brickley & Guha 2000) extends RDF with a vocabulary for describing properties and classes of RDF resources. It includes a (somewhat controversial) semantics for generalization hierarchies of such properties and classes. In contrast to XML Schema, RDFS primarily describes resources instead of prescribing constraints to them. See (Ratnakar & Gil 2002) and (Klein, Fensel, van Harmelen & Horrocks 2000) for a further comparison of these (and other) ontology languages with XML Schema.

1.3.2 Ontology Web Language (OWL)

The Ontology Web Language (OWL) extends the vocabulary of RDF(S) for describing properties and classes: amongst others, it supports modelling relations between classes (e.g. disjointness), cardinality, equality, richer typing of properties, characteristics of properties and enumerated classes (McGuinness & van Harmelen 2003). Most of the XML Schema datatypes can be used without restriction, except for some[4] (see (Patel-Schneider & Horrocks 2003) for a discussion).

OWL provides three increasingly expressive sublanguages.

- OWL Lite supports those users primarily needing a classification hierarchy and simple constraint features, thus providing a migration path for thesauri and other taxonomies.

[3] For an entertaining account of how Google could exploit the Semantic Web using RDF, see `http://www.ftrain.com/google_takes_all.html`

[4] In particular, duration, Qname, ENTITY, ENTITIES, NOTATION, ID, IDREF(S), and NMTOKENS are either not supported or carry restrictions.

- OWL DL (Description Logics) supports those users who want the maximum expressiveness while their reasoning systems maintain completeness and decidability.

- OWL Full provides maximum expressiveness and the syntactic freedom of RDF with no computational guarantees.

Elegantly, each of these sublanguages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded (McGuinness & van Harmelen 2003). OWL is a revision of the DAML+OIL (DAML+OIL Joint Committee 2001) (Horrocks 2002) web ontology language. It can be expected that web formalisms which currently depend on DAML+OIL (including many of the ones discussed in the next chapter) will migrate to OWL DL.

With regard to querying Semantic Web Services, Horrocks & Tessaris (2002) show how DAML+OIL queries (and thus OWL queries) can be rewritten so that query answering is reduced to the problem of knowledge base satisfiability for the logic corresponding to the ontology language. This enables us to answer queries using standard reasoning techniques, and to guarantee that query answers will be sound and complete in the case that that our knowledge base satisfiability test is sound and complete. In practice, this means that one can use implemented Description Logic systems (or any other system capable of deciding knowledge base satisfiability) to provide sound and complete answers to queries. We will look deeper into the matter of discovering SWS in the coming chapter.

Chapter 2

Existing approaches to capability description

Making the vision behind Semantic Web Services work crucially depends on choosing the right (set of) language(s) to talk about service capability. This chapter highlights some of the fields that are likely to make an impact here, and reviews some of the proposals made. In this overview, we try to keep a clear distinction between the actual *service ontology* and the *object language* in which this ontology is represented.

2.1 Problem-solving methods

This section presents a brief history of and motivation for the use of problem-solving methods, and discusses a prominent approach to configuring and composing these into knowledge systems. We provide an overview of what the UPML language for describing problem-solving methods looks like, and discuss the notion of *assumptions*. We will build on these concepts in subsequent chapters.

2.1.1 Building blocks for knowledge-based systems

Recently, in the Web services literature, some authors (see (Motta, Domingue, Cabral & Gaspari 2003) and (Wroe, Stevens, Goble, Roberts & Greenwood To appear)) introduced the notion of *tasks* to facilitate capability matching. One reason for keeping tasks separated from the actual web service descriptions is a possible n-to-m mapping between them, i.e. the same web service can serve different tasks and different (competing) services can serve the same task. Fensel & Bussler (2002) give the example of an online book-selling service, which could be used both for buying books and bibliography retrieval.

The nature of tasks (also called problem types or task definitions) and

their interaction with generic inference structures (called *problem-solving methods*) has been studied extensively within the Knowledge Engineering community (KE) during the past decade.

As Musen (2000) writes, problem-solving methods (PSMs) were born out of the difficulties encountered in the development of expert systems in the early eighties. Although the majority of such knowledge-based systems (KBSs) even today continue to be built using rule-based shells, there are well-known limitations to the scalability and maintainability of KBSs that researchers began to identify nearly as soon as the first rule-based systems had been developed. There was a need for more abstract specifications for KBSs. In this way, KBSs would become scalable, maintainable and even reusable (Musen 2000).

As part of the effort to develop the means for reuse of large-scale software components when building intelligent systems, problem-solving methods became an important area of KBS-research in the nineties. PSMs can be seen as abstract algorithms for achieving solutions to stereotypical knowledge-intensive tasks (like diagnosis, classification, design, monitoring, etc.) but also have an operational dimension that provides working code (Musen 2000). They act as reasoning templates that need to be instatiated with domain knowledge for each new aplication (Crubézy & Musen 2003). A unifying theme for problem-solving methods is their reliance on search as a technique to find solutions. See Section 2.1.5 for a discussion of how this influences PSM characterization.

2.1.2 Assembling knowledge-based systems with UPML

In recent years, the research agenda on PSMs has been influenced by the common infrastructure offered by the Internet. During the IBROW project,[1] a large part of the (European) Knowledge Engineering community reached agreement on a common language (called UPML, see below) to describe the distributed libraries of PSMs that were developed over the past years. Knowledge components annotated in this language could thus be exploited for web-based brokerage.

Using UPML descriptions, the IBROW broker selects third-party PSMs available on the Web, and combines PSMs and knowledge bases in a distributed, "plug and play" fashion. As a result, reasoning services can be configured dynamically out of independent components to solve a specific task. The broker thus builds "throw-away" Web applications (Omelayenko, Crubézy, Fensel, Benjamins, Wielinga, Motta, Musen & Ding 2003).

[1] Web site: http://www.swi.psy.uva.nl/projects/ibrow/home.html

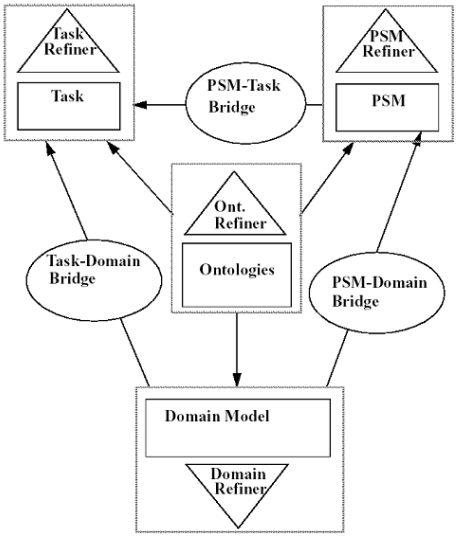

Figure 2.1: UPML architecture (Fensel et al. 2003)

Overview of UPML

UPML, or the Unified Problem-solving Method description Lanuage, plays a kernel role at each stage of the brokering process (Fensel, Motta, Benjamins, Crubézy, Decker, Gaspari, Groenboom, Grosso, van Harmelen, Musen, Plaza, Schreiber, Studer & Wielinga 2003). It is intended as a software architecture specifically designed to describe KBSs. A software architecture determines the structure of a system, which comprises software components, the externally visible properties of those components, and the relationships among them (Garlan 2001). Software architectures typically play a role as a bridge between requirements and code. The UPML architecture consists of six different kinds of elements, and additional architectural constraints and design guidelines. The dependencies between the elements are illustrated in Figure 2.1 and 2.2. These elements fulfill the following roles (from (Fensel et al. 2003)):

- A *task* defines the problem to be solved by the KBS.
- A *problem-solving method* defines the reasoning process used to solve the problem.
- A *domain model* defines the domain knowledge available to solve the problem. It consists of three elements: a characterization of properties of the domain knowledge, the domain knowledge, and (external) assumptions of the domain model.

  Each of these elements is described independently to enable reuse of

  - Task descriptions in different domains
  - Problem-solving methods for different tasks and domains
  - Domain knowledge for different tasks and problem-solving methods

- *Ontologies* provide the terminology used in the tasks, problem-solving methods, and domain definitions. Again this separation enables knowledge sharing and reuse. For example, different tasks or problem-solving methods can share parts of the same vocabulary and definitions.

- *Adapters* are necessary to adjust other (reusable) elements 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 different parts of the architecture, e.g. between a domain and a task or a task and a problem-solving method. As such, bridges change the input and output of the components to make them fit together.
  - *Refiners* can be used to express the stepwise adaptation of a single element of the specification, e.g. generic problem-solving methods or tasks can be refined into more specific ones by applying them to a sequence of refiners. Refiners may change only the internal details, e.g. the subtasks of a problem-solving method (see Section 2.1.4. Again, separating the generic and specific parts of a reasoning process enhances reusability.

Logics for UPML

To implement this software architecture, the UPML lanuage has to facilitate three aspects. First, a UPML specification must reflect the structure of Figure 2.1, describing a system of interrelated and intrarelated units. Second, the language must provide means for the declarative specification of functional aspects. Finally, it must provide means to describe control and communication aspects.

In order to achieve this, UPML envisages the use of three types of logics (for more information see (Fensel, Motta, Benjamins, Decker, Gaspari, Groenboom, Grosso, Musen, Plaza, Schreiber, Studer & Wielinga 1999, pages 50-54)):

- A logic to describe the *static aspects* (like the six architectural elements). Here first-order order logic is appropriate. Its syntactical enrichment and structuring by abstract data types (ADTs) enables to reflect the language and architectural level in the logic. Work in IBROW has used F-Logic, OCML and sorted logic for this purpose.

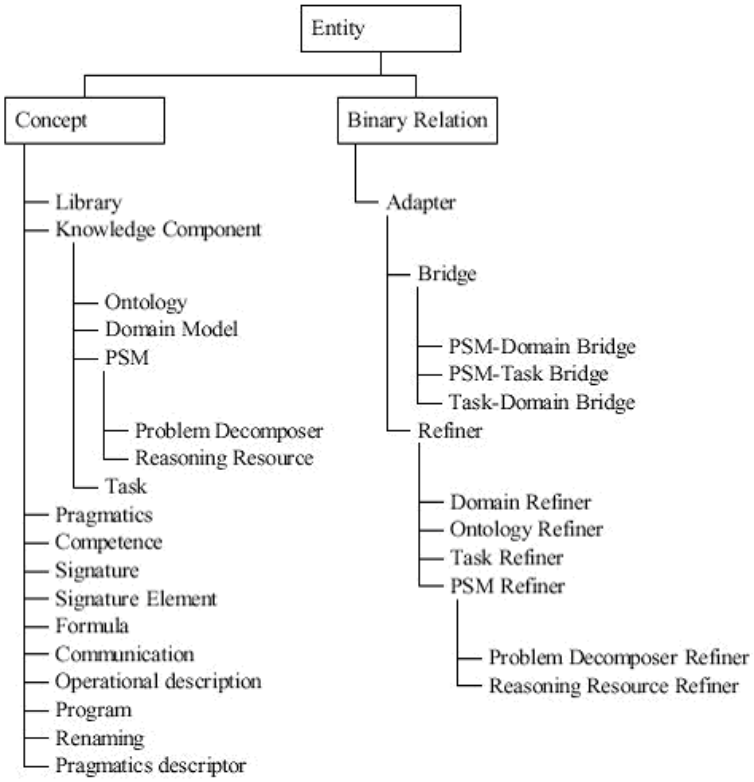

Figure 2.2: UPML class hierarchy (Omelayenko et al. 2000)

- The operational specification of the problem-solving method and its communication behavior (see Section 2.1.4) introduce the aspect of *dynamics* into the specification framework. By describing states with an algebra a smooth transition from static to dynamic aspects can be achieved. UPML proposes to use MCL for this purpose, but approaches based on dynamic logic and situation calculus would be possible as well.

- A logic to specify the *communication protocol* of a method. The standard option here is to use process algebra or finite state machines.

Now that we have an understanding of the vision behind UPML and the main architecture, we focus our attention on the parts of it that are crucial for capability brokering. In particular, in what follows we will concentrate on two key UPML elements : Tasks and Problem-solving methods.[2]

[2]We refer the reader to (Fensel et al. 1999b), (Omaleyenko et al. 2000) and (Fensel et al. 2003) for information about the other elements.

2.1.3 Tasks

A task defines the problem that is supposed to be solved by the system. The decision to make tasks first-order citizens is based on the following grounds:

- For reasons of computational complexity (see Section 2.1.5) the functionality desired by the user may differ from the functionality actually provided by the system. Distinguishing the desired functionality from the actual competence of the system provides the advantage to have an explicit notion for this gap.
- A second particular feature of a task definition is its domain-independence. This enables reuse of problem definitions in different domains. Classification tasks, diagnostic tasks, and design tasks can be defined independently from the domain in which they are reused. The (well proven) assumption is that there are problem types that appear in different domains. For example, problems solved by diagnosis appear in a broad variety of domains (electronic circuits, fluid systems, copying machines, etc.). In consequence, a task does not only introduce a goal (i.e. a notion of the desired functionality) but also a generic description of the type of domains it can be instantiated to. Its requirements on domain knowledge (also called assumptions, see Section 2.1.5) provide a "domain-independent characterization of its domain-dependency".

The Task concept (a subconcept of Knowledge Component, see Figure 2.2) specifies the task to be achieved by PSMs. The Task can also define a subtask of a Problem Decomposer (defined in the next subsection).[3]

```
Task < Knowledge Component
  uses : (Task)
  input roles : (Signature Element)
  output roles : (Signature Element)
  competence : Competence
  assumptions : (Formula)

Competence < Concept
  preconditions : (Formula)
  postconditions : (Formula)
```

[3]Omelayenko et al. 2000 use the following notation for UPML definitions. Concepts (classes) begin with upper case and the subclass relationship between two concepts is denoted by <. Each concept or relation is represented as a list of attribute-type pairs (attribute : type). Each pair from the list describes one property of the entity called attribute, whose values have as the type either a class or a primitive type STRING. Brackets around the type of an attribute denote that the attribute can appear multiple times in the concept description.

Figure 2.3: An example task (rendered with the Protégé UPML editor (Crubézy et al. 2002))

- The input roles and output roles together with the competence property define the input/output specification of the task. The Input roles specify the input of case data and the output roles specify the output of the case data.
- Competence includes preconditions restricting valid inputs and postconditions which describe the output of a task.
- The assumptions property defines requirements on the knowledge that is used to define the goal (see also Section 2.1.5). The Task can import and refine other tasks via its uses attribute.

In Figure 2.3 an example of a task is shown. This particular task allows to specify a classification problem, based on the library built by (Motta & Lu 2000). It takes as input a set of observables, candidate solutions, a match criterion and an admissibility criterion solution. As output it generates a set of possible solutions to the problem.

2.1.4 Problem-solving methods

The PSM component represents a problem-solving method, defined by its competence and communication properties.

```
PSM < Knowledge Component
  input roles : (Signature Element)
  output roles : (Signature Element)
  competence : Competence
  communication : Communication

Communication < Concept
  communication : STRING
```

- The input roles and output roles of the PSM specify its inputs and outputs, similar to their function in the Task component.
- The Competence concept represents the functional input/output specification of the Task component. Competence includes preconditions restricting valid inputs and postconditions which describe the output of a method.
- The PSM's communication property describes its interaction protocol with its environment, in particular with other PSM components. It defines the ports of each building block and for each port how the block reacts to incoming events and how it provides outcoming events.

UPML distinguishes two different types of problem-solving methods: complex problem-solving methods that decompose a task into subtasks and primitive problem-solving methods that make assumptions about domain knowledge to perform a reasoning step. The latter correspond to inference actions in the CommonKADS methodology.[4] Consequently, the PSM concept has two subclasses: Problem Decomposer (or complex PSM) and Reasoning Resource (or primitive PSM).

Complex problem-solving methods

A Problem Decomposer (also called complex PSM) decomposes a task to be solved into a set of subtasks. A complex method decomposes a task into subtasks and therefore recursively relies on other methods that process its subtasks. Such a subtask may describe a complex reasoning task that may further be decomposed by another problem-solving method or may be performed directly by a simple problem-solving method.

```
Problem Decomposer < PSM
  subtasks : (Task)
  operational description : Operational Description

Operational Description < Concept
```

[4] Website: http://www.commonkads.uva.nl

```
  intermediate roles : (Signature Element)
  programs : (Program)

Program < Concept
  program : STRING
```

Its Operational Description specifies the control structure over the subtasks and internal data flow among the subtasks. Such an operational description explains how the desired competence can be achieved. It defines the data and control flow between the main reasoning steps of the method. For this purpose it introduces intermediate roles (the stores for input and output of the intermediate reasoning steps), the procedures, and the control. As mentioned earlier, for specification of control UPML relies on MCL, a language combining features from the $(ML)^2$ and KARL KBS specification languages.

Primitive problem-solving methods

A Reasoning Resource (also called primitive PSM or simple PSM) solves a primitive step — also called subtask — of a problem provided by the problem decomposer.

```
Reasoning Resource < PSM
  knowledge roles : (Signature Element)
  assumptions : (Formula)
```

The specification of a primitive problem-solving method closely resembles the definition of a complex one, but there are two differences:

- A primitive problem-solving method does not provide an operational specification. Primitive PSMs do not have an internal structure, i.e. their internal structure is regarded as an implementational aspect of no interest for the architectural specification of the entire knowledge-based system. The knowledge roles attribute specifies the input of the (domain) knowledge to the reasoning resource.
- The definition of the competence differs slightly. Assumptions describe properties of the domain knowledge that is needed by the method. The assumptions about domain knowledge are the equivalent of the assumptions about the subtasks of a complex problem-solving method. These assumptions regarding the domain knowledge must be fulfilled in order to perform a primitive reasoning step.

2.1.5 Assumptions

The notion of assumptions appears to be where problem-solving methods differ from traditional software components. In what follows, we pay closer attention to them. We start by explaining how PSMs link to the notion of problem solving as search in AI. Using this connection, the role of assumptions in the brokering process will become clear(er).

Problem solving as search

Problem solving in general can be characterized in terms of search problems and consequently search underlies much of AI. When you are given a problem, you are usually not given an algorithm to solve it ; you have to search for a solution. For example, determining a logical entailment can be reduced to the syntactic problem of searching for a logical proof (Poole, Mackworth & Goebel 1998, page 114). Search is an enumeration of a set of potential partial solutions to a problem so that they can be checked to see if they truly are solutions, or could lead to solutions.

In order to solve a problem, you explicate the underlying search space and apply a search algorithm to that search space (also called solution space). The existence of public key encryption codes demonstrates the difficulty of search. In this example, the search space is clear and the test for a solution is given, however humans have no hope of solving this problem and computers cannot solve it in a realistic time frame. This suggests that computer agents need to exploit knowledge about special cases to guide them in a solution. This extra knowledge beyond the search space is called heuristic knowledge (Poole et al. 1998, page 115). The term heuristic search refers to the art of good guessing ; heuristic knowledge is knowledge that guides. Much of the practice of problem-solving methods is concerned with effective ways to incorporate such knowledge into reasoning systems (Stefik 1995, page 147).

Restricting the context of problem-solving methods

Problem-solving methods are used primarily to describe the reasoning steps and types of knowledge which are needed to perform a task by a KBS. With the above rationale of heuristic search in mind, Fensel (2000, page 8) advocates that PSMs should be characterized in terms of the assumptions they carry. Not unlike other AI problems, problems tackled with knowledge-based systems are often inherently complex and intractable. Fensel stresses that, when this relation between problem-solving methods and computational complexity is ignored or kept as an implicit notion, neither the selection nor the design of problem-solving methods can be performed in an informed manner.

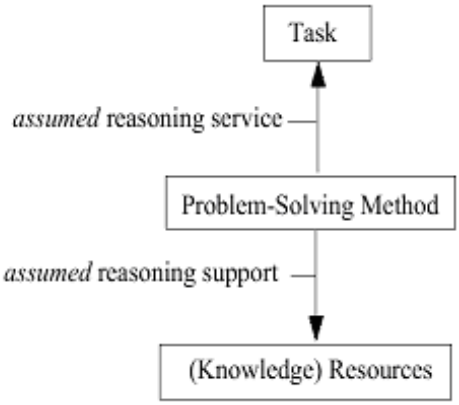

Figure 2.4: The relation between a problem-solving method and its context (Fensel 2000, p.8)

So how can problem-solving methods provide functionality if the problems are intractable in their general form? They use heuristic knowledge acquired from a domain expert to improve efficiency. The way PSMs achieve this "efficient realization of functionality" is by making assumptions. The assumptions put restrictions on the context of the PSM (Figure 2.4):

- One role of the assumptions of a PSM is to define the domain knowledge that the PSM requires. These assumptions describe the domain dependency of a PSM in domain-independent terms. Technically they can be viewed as proof obligations: missing pieces in the proof that the domain knowledge fulfills its assumed properties (i.e. one has to prove that the given domain knowledge fulfills the assumptions). As such, one could regard them as goals for the knowledge acquisition and domain modelling process.
- The second role of assumptions of a PSM is to define the task (i.e. the problem) that can be tackled successfully by the PSM. For example, the termination of a PSM may only be provable when assuming that a solution exists.

Otherwise put, assumptions play two roles: they formulate requirements on reasoning support that is assumed by the PSMs and they put restrictions on the reasoning support that is provided by the PSMs. In Section 3.1.2 they will be used as central concept in the brokering process.

Assumptions and domain properties

The properties of the domain knowledge are the counterpart of the assumptions on domain knowledge introduced by the other parts of a specifica-

tion (e.g. task or PSM) (Fensel, Benjamins, Decker, Gaspari, Groenboom, Grosso, Musen, Motta, Plaza, Schreiber, Studer & Wielinga 1999). The domain knowledge is necessary to define the task in the given application domain and necessary to carry out the inference steps of the chosen PSM.

Assumptions and preconditions

As could be seen from the definition of a PSM in Section 2.1.4, UPML makes the distinction between assumptions and preconditions. In an abstract sense, both assumptions and preconditions can be viewed as input. However, distinguishing case data, which are processed (i.e. input), from knowledge, which is used to define the goal/problem, is a characteristic feature of knowledge-based systems. UPML defines preconditions as conditions on dynamic inputs, whereas assumptions are conditions on knowledge consulted by the reasoner but not transformed. Often, assumptions can be checked in advance during the system building process, preconditions cannot. They rather restrict the valid inputs.

The assumptions ensure (together with the axioms of the ontology) that the task can always be solved for permissable input (input for which the preconditions hold). For example, when the goal of a task is to find a global optimum, then the assumptions have to ensure that such a global optimum exists (i.e. by "assuming" that the preference relation used is non-cyclic).

2.2 Description logics

Problem-solving methods offer only one way to think about component capabilities. Recently, Descriptions Logics (DLs) have been considered as a framework for service (capability) description. In this section, we provide a brief introduction to DLs, and motivate why one would use them in this context. We take a close look at the DAML-S service ontology, and make the comparison with UPML.

2.2.1 Supporting service classification

It is clear that to discover services and to configure compositions, services should first be described and classified in some way. Given that nowadays these can be published on the World Wide Web, service registries are expected to hold many service descriptions.

However, manually created classifications of services are inflexible and hard to manage when they become large, detailed and multi-axial. For example, a description should always be self-coherent and consistent with respect to others in the classification. The service classification should evolve as the descriptions evolve, for instance when changes occur in the quality of service or functionality. To resolve this issue, some advocate to keep service

descriptions, classification and constraint management tightly coupled and to treat this within the uniform framework of Description Logics (Wroe et al. To appear). Services and data types are grouped into taxonomic hierarchies, together with definitions of the relationships and constraints between classes and their service instances (McIlraith & Martin 2003).

One of the main attractions of using a Description Logic (DL) for describing services is the support this formalism brings for building large service ontologies. DLs help ontology development by checking for logically inconsistent concepts (with regard to the entire ontology), reclassifying changed descriptions, and detecting (possibly unexpected) implicit subsumption relationships. They also support ontology deployment by offering classification reasoning to external applications.

2.2.2 Description Logics in a nutshell

Description Logics are expressive subsets of First-Order Logic, which lend themselves particularly well for describing domain knowledge. As explained in (Baader, Calvanese, McGuinness, Nardi & Schneider 2003, pages 16-20), a Description Logic knowledge base is typically comprised by two components — a $TBox$ and an $ABox$. The TBox contains general information about the problem domain (intensional knowledge, "classes"), whereas the ABox contains knowledge that is specific to the individuals of the domain (assertional or extensional knowledge, "instances"). Intensional knowledge is usually thought not to change, whereas extensional knowledge is usually thought to be contingent, or dependent on a single set of circumstances, and therefore subject to occasional or even constant change.

The basic task in constructing a TBox terminology is classification, which amounts to placing a new concept expression in the proper place in a taxonomic hiearchy of concepts. Classification can be accomplished by verifying the subsumption relation between each defined concept in the hierarchy and the new concept expression. The basic reasoning task in an ABox is instance checking, which verifies whether a given individual belongs to a specified concept. This is accomplished by verifying the subsumption relation between each defined concept in the (TBox) hierarchy and the new concept expression. For further information, we refer to (Baader et al. 2003, Chapters 1 to 3) which provide an excellent introduction to Description Logics and associated reasoning.

The resulting classifications are lattices, not trees, as a description can have multiple parents. For example, in bioinformatics, a protein might be classified by what it transports, what it catalyses, the process it participates in; and where it is located. A service may be classified by its location, its cost, its inputs, its function and so on (Wroe et al. To appear).

2.2.3 DAML-S : a service ontology in Description Logic

DAML-S (DAML-S Consortium 2003) (McIlraith & Martin 2003) is an upper level service ontology specified in the (web-enabled) description logic DAML+OIL. See (Horrocks 2002) and (Baader et al. 2003, Chapter 14) for an introduction to DAML+OIL.

The use of DAML+OIL as a DL formalism for DAML-S brings three important aspects with it: rich modelling primitives as provided by the frame community, formal semantics and efficient subsumption reasoning as provided by description logics, and a standard proposal for syntactical exchange notations as provided by the Web community (i.e. XML and RDF) (Horrocks 2002) (Fensel & van Harmelen 2001).

The DAML-S service ontology builds on industry standards such as SOAP, WSDL, and WSFL (Web Services Flow Language; (Leymann 2001)) by adding rich typing and class information to describe and constrain the range of Web service capabilities much more effectively than XML Schema data types.

The upper ontology of DAML-S contains the following elements (Narayanan & McIlraith 2002): a Service Profile for service advertisements, a Service Model (process model) for describing the actual program that realises the service and a Service Grounding for describing the transport-level messaging information associated with execution of the program. In what follows, we take a closer look at these different elements. Most of the remainder of this section is copied from (Aitken & Tate 2003).

- The *Service Profile* contains the name of the *Service*, contact information, plus a textual description. It is *providedBy* an *Actor*, which in turn has a title, phone, fax, email, physical address and a URL. The Profile represents two aspects of the functionality of the service: the information transformation and the state change produced by the execution of the service (DAML-S Consortium 2003).

- A *Service* is *describedBy* a *Service Model*, being a process model which is described in terms of a process ontology. The process ontology distinguishes three subtypes of Process: *Atomic Process*, *Composite Process* and *Simple Process*. It also contains control constructs which are used to describe how component processes are combined into a composite process. Control constructs include: *sequence, split, split+join, unordered, choice, if-then-else, iterate* and *repeat-until*. These are described in text, but no official formal definition is provided (yet).

  This is partly due to the lack of intrinsic support in DAML+OIL to define process control or dataflows. Therefore, any information regarding such information cannot be used to calculate a classification structure by DAML+OIL reasoners. However, both (Ankolekar, Huch

& Sycara 2002) and (Narayanan & McIlraith To appear) have developed additional formalisations, for the purposes of verification and simulation.

- The *Service Grounding* that a *Service* supports defines the concrete means of realising the service. The abstract notions of the ontology and process models are expressed in terms of existing XML-based technologies: WSDL, SOAP and UDDI.

Both the *Service Profile* and the processes in a *Service Model* can have inputs, outputs, preconditions and effects (the so-called IOPE's) associated with them. Each of these properties is represented by a *Parameter Description*, composed of:

- *parameterName*: the name, a literal or URI
- *restrictedTo*: a restriction on the values the relation holds of
- *refersTo*: a reference to the process model

The Service Profile is primarily intended for advertising, discovery, selection and matchmaking purposes, whereas the Service Model (process model) can be used for various purposes related to invocation, execution, monitoring, recovery, etc.

The Service Model is intended to be the "home" of inputs, outputs, preconditions, and effects. The Profile is meant to derive its advertised IOPE's from the process model. So the IOPE's on the Profile are normally just a copy of what is on the corresponding process, or perhaps even just a subset of what is on the corresponding process. The idea is that a process may have inputs and outputs that are not really useful for matchmaking, so they can be omitted from the Profile.[5]

2.2.4 A closer look at DAML-S

In its vision to create a Semantic Web for services, the DAML-S consortium has expressed the ambition to support many aspects of the service lifecycle. These include service discovery, invocation, mediation, composition, execution, monitoring, recovery, simulation and verification. It should be of little surprise that getting all this right from the start is hard. It appears as if DAML-S tries to be all things to all people, being too general at some points, while being too specific at others.

In what follows, first we look in some detail at the conceptual model. We then sketch out different design choices for the process model currently under consideration. The questions asked here include what should go in the process model, and how this should be modelled.

[5]These last two paragraphs are due to David Martin (e-mail correspondence with Austin Tate dd. 4 June 2003).

Imprecise conceptual model

Work by (Sabou, Richards & Splunter 2003) has shown that DAML-S has an imprecise underlying conceptual model. There are different reasons for this (which we quote literally from the authors):

Different models within DAML-S The parts of DAML-S employ different metaphors to describe services. At the Profile level a service has four types of parameters: IOPE's. At the Process level IO's and PE's are treated conceptually differently as they emerge from two different views of a service. A service viewed as a program is defined by its IO's, while when seen as an action the PE's are important. The conceptual gap is even wider when one tries to align the DAML-S model to the WSDL model which defines services as collections of ports. These alternative conceptual models make specification of services difficult and mapping between models almost impossible. Even within DAML-S the different models can lead to inconsistencies in the specification.

Unclear links between models Several links exist between the conceptual models however they are often unclear. Insufficient descriptions are provided in the DAML-S documentation to discover the intended meaning of certain properties and in particular which properties are related to properties in one or more of the other models. The lack of precise specification of the interconnections between models and the possibility of inconsistent models is admitted by the DAML-S Coalition.

No clear correspondence with software engineering concepts Many, if not the majority, of intended users of DAML-S are software engineers. Reference to and support for SE concepts, perhaps in the form of concept mappings, would ease the understanding of DAML-S. While WSDL intuitively models different interfaces as PortTypes and allows grouping operations in ports (as the methods of an interface), it seems that DAML-S only considers the very simple function metaphor (methods). More complex concepts such as parametric polymorphism or re-use are not supported. Using the SE model would both disambiguate some of the concepts and give a shared framework for DAML-S and WSDL.

Service Model design choices

The current characterization of the Service Model is rather unwieldly, and not formally defined. However, it is still very much under discussion. Below

we summarize the major lines of thought regarding process model content and specification in DAML-S.[6]

Process model content Two kinds of needs can drive process model content.

- One way to see process modeling is as modeling an executable specification, in support of automated proof and validation. This corresponds to a 'glass box' view of a component, where the component internals are widely published.

- The other line of thinking promotes partial modeling using constraints, modelling only what a service provider wants or needs to expose. This approach is known as the 'gray box' view (see also (Fensel & Bussler 2002)). The latter approach specifies internal milestones or points at which conditions can be met or constraints can be established, as well as a simple advertisement or description of the overall results. A particular advantage of this approach is that the standardized Process Specification Language (PSL) (Schlenoff, Gruninger, Tissot, Valois, Lubell & Lee 2000) can be built over it (PSL is not aimed to specify software components, like for instance the Z language).

Process model representation Once it is known what goes inside the process model, another issue DAML-S must address is how to represent these processes in Description Logics and exploit reasoning capabilities. Recently, several alternatives have been proposed to address the issue. The following list is a summary of a recent discussion on the WWW-WS mailing list.[7]

- *Represent processes as concepts* Currently, in defining a process in DAML-S or OWL-S, DAML-S conceptualizes a process as the class of its execution instances (or, synonymously, "execution traces"). This allows not only for the specification of the process (as a class) but also for the specification of an individual execution trace (as an instance of that class). One could use subsumption to answer a question like "Is this execution trace a valid instance of that process?". However, reasoning for the ABox (i.e. on instances) is known to be problematic in DLs. Other objections against this approach include the need for OWL Full, the use of unintuitive constructs and the question whether one can think of process executions as being instances of processes.

[6]For more details, see `http://lists.w3.org/Archives/Public/www-ws/2003Apr/0080.html` and `http://lists.w3.org/Archives/Public/www-ws/2003Apr/0082.html`

[7]Start from the following www-ws@w3.org messages for more information: http://lists.w3.org/Archives/Public/www-ws/2003Aug/0029.html, and http://lists.w3.org/Archives/Public/www-ws/2003Aug/0044.html

- *Represent processes as instances* This is the current focus of the DAML-S consortium. Experience has taught the DAML-S people that it is more natural to think of a process, and an execution of the process, as two different (though closely related) kinds of things, and model the relationships in some other way than class membership. If processes are to be represented as instances, the current DAML-S process ontology will have to be extended to include the specification of individual execution traces.

- *Represent processes as properties* The fundamental notion here is that properties would represent transitions between states (represented by instances). Classes, then, become preconditions of the transition (i.e., what has to be true of the state before the transition can occur).

Another important issue is how to handle variable bindings. As Aitken & Tate (2003) observe, in process workflow, it is common to state that the thing output by one process is the same thing input to another. However, Description Logics, and ontologies based on them, lack the variable binding mechanism that is normally used for this purpose. All we can do is specify the types of the inputs and outputs. A solution to this problem will come from the DAML-Rules initiative. We refer to Section 4.3 for an example of such bindings in the F-X brokering environment. See Chapter 5 for an introduction to the Description Logic Programs formalism, on which DAML-Rules will most likely rely.

2.2.5 Comparing DAML-S with UPML

In Section 2.1 we presented an overview of UPML. Given their rather similar objectives (to ease the assembly of software components over the Web), it makes sense to draw the comparison between DAML-S and UPML. Table 2.1 gives a quick overview of how the major concepts relate. In this section, we briefly compare object languages and investigate to what extent UPML notions are present in DAML-S. We look at the concept of Task (-refinement), PSM and Bridge.

Object language

The biggest difference between DAML-S services and UPML PSMs appears to be in the freedom granted when choosing an object language to represent the service ontology. This results in different kinds of reasoning support. DAML-S structures its descriptions using DAML+OIL, while leaving some room on top for additional languages. As detailed in Section 2.1.2, UPML pretty much leaves open which object language to use, as long as it is a subset of FOL.

UPML	DAML-S
Task	Service request
PSM	Service
Competence (incl. pre-/postcond.)	PE in Profile and Service Model
Operational Description	Composite Process
Pragmatics	Non-functional part of Profile
Pragmatics + Competence	Profile
Operational Description + Competence	Service Model
Assumptions	-
-	Effects

Table 2.1: UPML to DAML-S dictionary

Nevertheless, efforts have been made to represent UPML elements in OIL (the Ontology Inference Layer, (Fensel, van Harmelen & Horrocks 1999)), the precursor of DAML+OIL and OWL (see (Fensel 2002) for a history report). Here we report on the experiences gained by (Fensel, Crubézy, van Harmelen & Horrocks 2000). The authors used OIL to directly express UPML specifications. In this way, a component specification of an ontology or a task would correspond to an ontology in OIL.

In particular, they tried to model a simple task ontology and a task specification in OIL. The following problems arose:

- Important axioms could not be expressed directly. As a result, they were written down in the OIL rule base which has no semantics.
- OIL was extended with the property of being nonreflexive for slots and with cardinality constraints for classes.
- When using OIL the structure of the specification units of UPML gets lost. Things like the definition of an input role or an output role are only kept as natural language comments in the documentation slot.
- OIL does not provide the means to specify functional slots. The authors solved this by defining a cardinality constraint but this was not yet part of the language definition for slots.
- Finally, their task ontology defined sets of sets (as is possible in sorted logics). More concretely, in their given example, an instance of *findings* is a set of instances of the class *finding*. Therefore, they included a powerset operator in the OIL specification which was not part of the language definition. Such a decision may cause serious problems for the OIL semantics. However, without this operator it was not possible to capture the essence of the small and simple UPML ontology.

A number of OIL's limitations have since been addressed in DAML+OIL and its descendant, OWL DL. In particular, support for cardinality constraints and functional slots is now available. Sets of sets are still not possible. In general, given that UPML axioms are not restricted to the DL subset of FOL, there will always be areas where (DAML+)OIL or OWL DL are insufficiently expressive.

In the same paper, the authors also address the rather interesting question whether UPML can provide any help to OIL (and hence DAML+OIL). It turns out it can, as we discuss below in the section on bridges.

Tasks

One of the larger projects where the DAML-S service ontology has been put to use is *myGrid*[8], a pilot project part of the UK Grid effort. myGrid aims at building a virtual laboratory workbench to support bioinformaticians in making use of the complex distributed resources of the Grid. One important addition that was made to DAML-S in this setting was the introduction of the notion of a task (Wroe et al. To appear). In particular, the Profile was extended with a `performs_task` property. We refer the reader to Sections 2.1.3 and 4.4.1 for a discussion on how tasks are viewed in UPML and F-Comp (see Section 4.1) respectively.

Adding the notion of a task to DAML-S is also advocated in (Motta et al. 2003). The authors write that in DAML-S tasks are defined as service-seeking agents. Tasks are always application specific, so no provision for task registries is envisaged. In contrast, in the authors' UPML-based approach tasks provide the basic mechanism for aggregating services and it is possible to specify service types (i.e., tasks), independently of specific service providers.

The authors further point out that, in principle, this is also possible in DAML-S. Here a task would be defined as a service class, say S, and its profile will give the task definition. However, this solution implies that all instances of S will inherit the task profile. This approach is not very flexible, given that it makes it impossible to distinguish (and to reason about) the differences between the profile of a task (service class) and the profile of a method (specific service provider) — attributes are inherited down IS-A hierarchies.

In particular, in some cases, a method may only solve a weaker form of a task, and it is therefore important for a brokering agent to be able to reason about the task-method competence matching, to decide whether it is OK to use the weaker method in the given scenario. For instance, in a currency conversion scenario, a task specification may define currency conversion rates in terms of the official stock exchange quotes, but different service providers

[8]Web site: `http://mygrid.man.ac.uk`

may adopt other conversion rates. By explicitly distinguishing between tasks and methods (Motta et al. 2003) provide a basic framework for representing these differences and for enabling matchmaking agents to reason about them.

Both approaches mentioned suggest to represent task and service in the same service ontology. The impossibility to distinguish profiles mentioned above could be avoided if one separates out tasks and services into different ontologies. One could then select a task from the task registry, and classify this one among the available Service Profiles in the service registry. Such a modelling approach would also correspond better to the brokering scenarios skectched in Chapter 1, where both providers and requesters publish their information seperately, and where matchmaking agents/brokers can intervene.

The exchange rate example used above employs the notion of a "weaker method". Imposing such an order on components is hard, and one would have to adopt domain-specific heuristics (i.e. there are no general techniques for automatic software refinement). If one accepts this lack of generality, we see no reason why one could not do this for DAML-S as well, and organise tasks and services in a weaker-to-stronger hierarchy.

Further, the question can be raised how expressive tasks should be. Obviously, the more we know about the problem, the easier it is to solve. However, the burden on the user increases equally. Under the UPML approach, tasks clearly can get very expressive. Hence, much knowledge must be elicited from the user before attempting the PSM matching process. The approach of (Motta et al. 2003) requires extensive task knowledge, and acquires this using a methodology similar to the one described in Section 3.1.1. In this way, the task-PSM brokering effort can be kept very simple, and the focus is mostly on "task brokering".

The approach discussed in Chapter 4 takes a look at the other end of the spectrum, where tasks are considered to be simple queries. In this case, the burden is on the PSM broker, which has to decide whether it can find a (composition of) service(s) to realize the desired task competence.

Problem-solving methods

DAML-S and UPML both offer functionality to describe problem solvers. Still, there is no direct substitute for the DAML-S notions of "Service Profile" and "Service Model" in UPML. The Profile captures elements which in UPML are modelled under Pragmatics and Competence, while Service Model covers the concept of Operational Description and again Competence. We explain this in some detail below, using the UPML notions as a starting point.

Competence The elements in DAML-S that correspond closest to a PSM competence are the preconditions and effects. UPML currently proposes an

approach based on algebraic specifications (although a situation calculus-based approach would be possible as well, see Section 2.1.2). DAML-S seems to have adopted a hybrid style, borrowing elements from both algebraic specifications and action-based formalisms. We mentioned earlier that the connection between Profile and Service Model is unclear because of this, and that the Service Model currently is ill-defined.

Adding to the confusion is the fact that DAML-S services can make claims about changing the state of the real world after service execution — the so-called effects in the world. The notion of effect is not without controversy.[9] Suppose one wants in a representation language some description of effects to the world. The only reason then for having that is that to reason about those in some way. The effects should somehow affect how you would use your service or the consequences of using it. As such, these effects have to be meaningful in some sense. It remains an open question how it would be meaningful in an open Internet environment to talk about changes to the world connected to services that one knows nothing about. In addition, this happens in situations where one has no idea whether these services are even persistent or not, and certainly no such guarantees can be provided. To understand the semantics of this, notions of persistence of state over time are needed, and those have always been tricky for AI.

An alternative way to cater for these kinds of circumstances is to let some notion of contract, agent dialogue or coordination take the place of the sort of deeper reasoning about states of the world. Dialogue protocols do not yield the same sort of shared state one has when reasoning about actions of world, but it is the sort of state one can reason about if, for example, a contract over the use of some resource is needed.

To date, DAML-S has stayed silent on how to actually represent effects, and at the moment preconditions and effects are mapped to Thing (i.e. any DAML+OIL expression is allowed). A working group is currently trying to specify rules in DAML-S. Given the restrictions that DLs impose on forming rules (see also Sections 5.3.2 and 5.4.2), it will be interesting to see how expressive the preconditions and effects will prove.

Complex and primitive problem-solving methods Intuitively, complex PSMs correpond with DAML-S services that have composite processes in the Service Model. Just like complex PSMs and their subtasks, composite processes are decomposable into other noncomposite or composite processes. Decomposition of composite processes can be specified using control constructs such as Sequence and If-Then-Else.

Primitive PSMs are very closely related to DAML-S Atomic processes : Atomic Processes are directly invocable, have no subprocesses, and execute in a single step. DAML-S also has the notion of Simple Process, which

[9]The following is due to a discussion with Dave Robertson.

is not invocable and is not associated with a Grounding, but serves as an abstraction to provide a view on using some (atomic or composite) process, for purposes of planning and reasoning.

As far as reasoning support goes, UPML's Operational Description (which is exclusive to complex PSMs) relies on MCL descriptions, which can be exploited using interactive theorem proving (see Section 3.1.2). As for DAML-S, we discussed the representation of processes in Section 2.2.4, where we stressed that the Service Model is still being defined. However, as mentioned in Section 2.2.3, some formalizations are readily available.

One solution to the lack of support for dataflows or control in DAML+OIL is to ascribe process semantics to DAML-S by mapping it to the Process Specification Language (PSL) mentioned above (Schlenoff et al. 2000). PSL is a (standardized) process specification ontology described in the situation calculus (McCarthy & Hayes 1969), which is a (mostly) first-order logic language for reasoning about action and change in dynamical systems. With these semantics in hand, (Narayanan & McIlraith 2002) encode service descriptions in a Petri net formalism and provide decision procedures for service simulation, verification and composition.

Communication The UPML Communication concept corresponds roughly to the Grounding in DAML-S. The DAML-S Grounding (and DAML-S in general) makes an active effort to link into many of the upcoming industry standards, including BPEL4WS (Mandell & McIlraith 2003), UDDI (Paolucci, Kawamura, Payne & Sycara 2002) and WSDL. Recently, work started on ebXML. This is an area where UPML is underdeveloped, although some work has been done to integrate Communication with the FIPA standard for agent communication.

Pragmatics UPML Pragmatics correspond to the Profile's non-functional part. UPML pragmatics are a little different as they also relate to the notion of cost, which is absent in the standard DAML-S Profile. The costs of a problem-solving method have several different dimensions: interaction costs (with user and other aspects of the environment) and computation costs (in terms of average, worst, or typical cases).

Bridges

As Motta et al. (2003) write, the separation made in UPML between tasks and methods provides a basic model for dealing with ontology mismatches. The UPML framework assumes that the mapping between methods and tasks may be mediated by bridges (as explained in Section 2.1.2). In practice this means that if task T is specified in ontology A and a method M is specified in ontology B, which can be used to solve T, it is still possible to use M to solve T, provided the appropriate bridge is defined.

The work of (Fensel et al. 2000) discussed whether UPML could provide any help to OIL (one could imagine repeating this exercise for OWL). One important area where such help is possible concerns the facilities available in UPML for modularising ontologies. OIL provides a very simple construction to modularise ontologies. In fact, this mechanism is identical to the namespace mechanism in XML. It amounts to a textual inclusion of the imported module, where name-clashes are avoided by prefixing every imported symbol with a unique prefix indicating its original location. However, much more elaborated mechanisms are required for a structured representation of large ontologies.

As the authors further note, renaming, restructuring, and redefinition means must be applicable to imported ontologies. Here, one can make use of the adapter concept of UPML. UPML provides refiners and bridges to modify components. These adapter components of UPML can be used to integrate the need of ontology structuring into an existing architecture. When combining UPML and OIL in this way one is able to specialize the generic adapter concept of UPML for the fixed set of language primitives of OIL. The precise integration of the adaptation concept of UPML in (DAML+)OIL is still under investigation.

Assumptions

Another important concept in UPML concerns Assumptions. We postpone our discussion of assumptions to Section 4.4.1, at which point we will have explained how assumptions are exploited during the matching process (see Section 3.1.2 for this). Section 4.4.1 also extends the discussion initiated above on the relevance of tasks.

2.3 Other approaches

We now briefly mention other areas that are likely to have an impact on capability description languages for the Web.

Software specification techniques There exists a long tradition in hardware verification and proving properties of software, which has yielded a number of robust formal languages such as Z or VDM. Often these languages are indecidable or rely on theorem proving. Given the current state of the art, it is rather unlikely that they will scale to the size of the Web — or even the Intranet for that matter.

Database query languages An active field of research possibly relevant to discovery of Semantic Web Services concerns the querying of semi-structured data like XML. However, since we require a query language to

closely fit to the underlying data model, the choice for RDF as the data-representation language immediately rules out any of the XML-based query languages (Broekstra, Fluit & van Harmelen 2000). A comparison of RDF query languages is beyond the scope of this thesis. For a survey including RDF and XML based approaches, see (Magkanaraki, Karvounarakis, Anh, Christophides & Plexousakis 2002) and (Broekstra et al. 2000)).

Agent systems Agent systems often are looking for the right balance between expressivity and reasoning power to have agents work together. A lot of relevant work has been done on coordination for agent systems. One important example here is KAoS, the Knowledgeable Agent-oriented System.[10] We should also mention the work on LARKS, which offers very good matchmaking capabilities (Sycara, Widoff, Klusch & Lu 2002). Given that our focus is not on agent systems, we do not look into this any further.

Action representation formalisms We will briefly mention action formalisms in the next chapter. Again, they are outside our scope.

[10] Web site: `http://ksi.cpsc.ucalgary.ca/KAW/KAW96/bradshaw/KAW.html`

Chapter 3

Reasoning mechanisms for composing and configuring services

It is well known in logic that representation and reasoning represent two sides of the same coin. The previous chapter focussed mainly on representing service capability. Here we look in some detail at the support problem-solving methods offer in terms of service composition and configuration. We briefly mention a few other techniques as well.

3.1 Techniques for problem-solving methods

In what follows, we present three different approaches that were proposed in the context of research on Problem-Solving Methods. After each section, we insert a brief discussion on the relevance of the approach for Semantic Web Services.

3.1.1 Configuring knowledge-based systems

Building knowledge-based systems from existing components requires fitting together heterogeneous knowledge components. Crubézy, Motta, Lu & Musen (2003) propose a methodology to configure such KBSs based on UPML. Their *Internet Reasoning Service* (IRS) provides tool support for this process. There exist two versions of the IRS: one was produced by Stanford Medical Informatics based on the Protégé environment[1], the other was created at the Knowledge Media Institute.[2] The former provides strong support for mapping, extending the work of (Park, Gennari & Musen 1997) (see also (Crubézy & Musen 2003)), while the latter is intertwined with an

[1]Web site: http://protege.stanford.edu/plugins/psmtab/PSMTab.html
[2]Web site: http://kmi.open.ac.uk/projects/irs

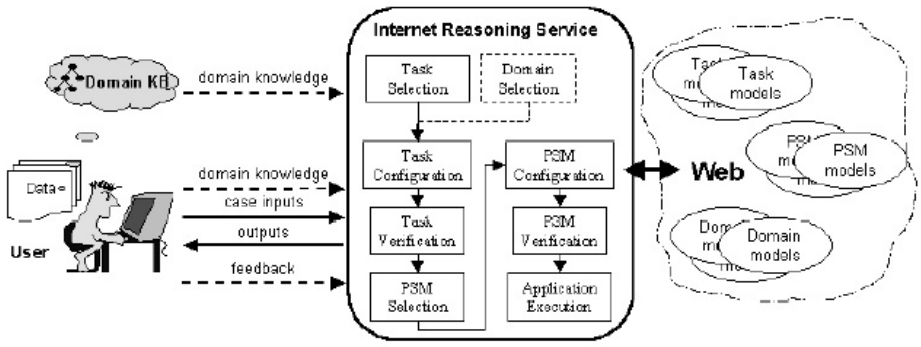

Figure 3.1: IRS use scenario and process model (Crubézy et al. 2003)

operational environment that can execute the configured system. The IRS methodology consists of eight steps and is summarized below. An overview of the IRS setup is shown in Figure 3.1.

Task selection First of all, IRS users first need to characterize the goal of their application in terms of one of the tasks known to the IRS. Currently, the IRS has been tested with a library of components for classification (Motta & Lu 2000), and a library for scheduling tasks is in the works. Task selection means adopting a particular viewpoint to impose over an application. Often in KE, the same application can be built using different task models. As such, a task model provides a conceptual viewpoint for characterizing an application and for focussing the knowledge-acquisition process. The IRS supports users in browsing and navigating UPML-based descriptions of the various tasks in the available libraries, along with examples of the ways tasks have been applied previously. Users select a task by choosing a refinement of a high-level task.

Task configuration In this crucial step, users configure the selected task for their particular domain by providing relevant domain knowledge to fill the input and output roles defined by the task. If the domain knowledge does not conform to the task ontology, the IRS supports users in constructing a mapping relation between the task role and domain knowledge. The created mappings are then stored in a task-domain bridge (see Figure 2.1).

Domain selection In this phase (which could also take place before task selection), users specify the domain knowledge that they intend to use in their application. The IRS provides a range of existing domain models stored in UPML-compliant libraries. The advantage of selecting an available domain model is that, in many cases, no configuration

effort will be needed, given that mappings (adapters) themselves are often reusable. Alternatively, the IRS provides knowledge acquisition forms to enter domain knowledge directly. Another possibility is that users provide their own knowledge base. The nice thing here about the Protégé IRS is that Protégé knowledge bases can be imported directly. This is an important feature given that Protégé (arguably) has become the de facto standard platform for building ontologies.

Task verification This step consists of checking the assumptions that the selected task defines on domain knowledge. The IRS performs this step automatically by running an assumption-checking engine on the task input roles — possibly obtained from domain inputs through mapping relations — and the assumptions of the task. However, in the general case, not all assumptions are necessarily verifiable (see Section 3.1.3 for an approach that explicitly addresses this issue), for example when one assumes that only one solution exists in the target domain.

PSM selection This step focuses on selecting a PSM that can realize the configured task. Users can always choose among the available PSMs manually, but one way the IRS can assist in finding PSMs is by checking the list of available PSM-task bridges, which contains PSMs that can realize the task. Another way the IRS could retrieve PSMs is by competence matching. In this case, candidate PSMs are selected if their postcondition statement fulfills the goal of the task, and if their preconditions do not contradict the assumptions of the task. It is not always clear what "goal" means in a UPML context. Here it makes sense to interpret the goal as the postconditions of the task, where suitable PSMs have postconditions stronger than those of the task. Crubézy et al. (2003) note that, in general, such reasoning requires full first-order logic theorem proving support, which is not part of the current implementation of the IRS. The important issue is to design contents-specification formalisms, which strike the right balance between expressiveness and efficiency in support of matching. In this respect, we refer the reader to Section 4.4.1 for a discussion on UPML's expressiveness in relation to capability matching.

PSM configuration This step is similar to the task configuration step, at the level of the selected PSM: the IRS guides users in specifying the domain entities that fill-in the input and output roles of the PSM. Some of the roles for the PSM are "inherited" from the configured task, through a corresponding PSM-task bridge. If not already provided in the library, the IRS supports the creation of such a bridge to map the inputs and outputs of the configured task to the ones of the selected PSM. In addition, the PSM may define supplemental input and output roles. Like with task configuration, mapping relations to

domain knowledge may be needed, which would then be stored in a PSM-domain bridge.

PSM verification This step focuses on verifying that the selected PSM can be applied to the task and the domain in accordance with the assumptions of the PSM and the results of the configuration process. It is essentially the same as the task verification step. The IRS notifies users about the domain inputs that do not satisfy the assumptions or preconditions of the PSM.[3] In this case, the IRS guides users back either to the task configuration step or to the PSM configuration step, to re-specify the inputs that are not satisfied.

Application execution This final step consists in running the configured PSM to realize the specified task, with domain case data entered by the user. The IRS first acquires case data from the user and instantiates the case inputs of the PSM by interpreting the various mapping relations built. The IRS also checks the preconditions of the PSM and task on the mapped case data. The IRS then invokes the PSM code with the mapped inputs, by running a code interpreter either locally or remotely. Knowledge about location and type of PSM code is stored in the pragmatics field of the UPML description of the PSM. Finally, the IRS fills in the domain outputs with the results of PSM execution, possibly transformed with domain-PSM mapping relations defined during the PSM configuration step.

Discussion

It should be clear that performing all eight steps represents the "worst case scenario": the simpler the system the more likely some steps will be trivial. The approach is a clear reminder that KBS systems are in fact complex, large software systems, that can take a lot of effort to configure, during which human intervention is indispensable. This is in contrast with the work done in F-X (see Section 4.1), which aims for as much automation as possible. We will make a short comparison between the two approaches in Section 4.4.1. Here, we also refer to recent work that recasts some of the work on the KMi IRS implementation in a Web services setting (Motta et al. 2003). We discussed some features of this prototype in Section 2.2.5.

3.1.2 Brokering as assumption discovery

The above approach layed out the different steps to take in order to select and configure UPML components from a library. The work we discuss in this section addresses how to actually build the components to fill this library.

[3] This appears to be a particular case where preconditions *can* be checked in advance during the system building process (compare Section 2.1.5).

Fensel, Schönegge, Groenboom & Wielinga (1996) present a formal approach for the specification and verification of UPML-based KBSs. Formal specifications of UPML elements are first built and subsequently matched with others using an interactive theorem prover. As mentioned in Section 2.1.2, UPML uses abstract data types and a variant of dynamic logic as the formal means to specify the different architectural elements. This entails that, as pointed out in the previous section, a first or higher order theorem prover will be needed to match tasks with PSMs.

Architectural constraints

A consistent specification is built based on architectural constraints. These help to ensure that the different elements, when taken together, define a consistent system. The architectural constraints of UPML consist of requirements that are imposed on the intra- and interrelationships of the different parts of the architecture. They can ensure either:

- A valid part (for example, a task or a problem-solving method) by restricting possible relationships between its subspecifications
- A valid composition of different elements of the architecture (for example, there are constraints on connecting a problem-solving method with a task)

The constraints on well-defined components apply for tasks, domain models, and PSMs. The constraints for composition are introduced by constraints that apply to bridges. An example of a architectural constraint for a task ontology is that:

$$axioms \cup preconditions \cup assumptions$$

must have a model; i.e. a task ontology must be consistent (see (Fensel et al. 2003, pages 28-33) for a list of all constraints). As we shall explain below, *these architectural constraints act as proof obligations* for the theorem prover. Because the different elements in UPML can be reused, not all of these proof obligations will have to be repeated for every application though.

Discovering assumptions

These architectural constraints provide a starting point for specifying knowledge components. Fensel et al. (1996) employ the Karlsruhe Interactive Verifier (KIV) as theorem prover. As described on the KIV website,[4] at the heart of the KIV system is a tactical theorem prover in the tradition of the

[4]Web site: `http://i11www.ira.uka.de/~kiv/`

Edinburgh LCF system. Construction of proofs is done by applying *tactics*[5], thereby reducing goals to subgoals. The selection of tactics is mostly done by the heuristics implemented in the KIV system, which leads to a high degree of automation. If all (selected) heuristics fail to succeed, the user may interact by selecting tactics or heuristics, backtracking, pruning the proof tree or introducing lemmas.

In our particular case, KIV is used to *discover* where in the specification *extra assumptions* need to be introduced in order to arrive at a consistent specification, either at the component level or at the KBS level. KIV supports this discovery by analysing failures of proof attempts based on the architectural constraints. Gaps found in a failed proof provide initial characterizations of missing assumptions. They appear as sublemmas that were necessary to proceed with the proof. An assumption that implies such a sublemma is a candidate for becoming a new assumption to the component.

Fensel et al. (1996) give the following example. Consider a PSM which performs *hill-climbing* search with the competence to find a local minimum in a graph. Suppose we have a task under concern which requires to select an optimal element from a set. In this case, there are two possibilities to close the gap between task and PSM:

- Strenghten the domain assumptions: one can introduce additional requirements (assumptions) on domain knowledge that enable hill-climbing to find a global optimum.
- Weaken the application functionality : one can weaken the task definition by introducing extra assumptions on it, in such a way that the PSM competence is now sufficient to realize the task.

In the example, using KIV would indeed make clear that the task assumptions are too weak to guarantee equivalence of task definition and PSM competence. As such, one could then decide to add extra assumptions over the input of the task.

Discussion

Fensel et al. (2003, page 35) note that assumptions found in this way are sufficient to guarantee the correctness of the proof, but often they are neither necessary for the proof nor realistic in the sense that application problems will fulfill them. This remains a problem for this approach to address.

When comparing the KIV approach with other work, Fensel et alea point out that the UPML approach is more general than, for instance, the AMPHION system, which uses deductive synthesis of programs.[6] Pro-

[5] A tactic is a function from goals to subgoals but it is also more than that. It provides a way of constructing a proof data structure.

[6] Web site: `http://ase.arc.nasa.gov/docs/amphion.html`

grams constructed with KIV are combinations and instantiations of domain-independent PSMs, rather than simply a sequence of calls of subroutines from a library. We will mention the use of deductive synthesis again at the end of this chapter, when we discuss assembly of Grid applications. In Section 4.4.1, we briefly compare the brokering used in the F-X system with the KIV approach.

3.1.3 Configuration as parametric design

A third proposal for configuring knowledge-based systems can be found in (ten Teije, van Harmelen, Schreiber & Wielinga 1998). This approach has no intention to discover hidden assumptions in components. Rather, it acknowledges that some assumptions are simply too hard to satisfy anyway, for instance due to inherent incompleteness of data or knowledge (e.g. lack of good heuristics) in AI-problems. Based on this insight, the work presents a framework to think about additional aspects of the PSM lifecycle, in particular automated PSM construction, validation and monitoring.

Constructing problem-solving methods

Say we have a library of problem-solving methods. Which problem-solving method is optimal for a given problem type ? In general, the choice of an appropriate PSM will depend on the goal of problem solving, and on characteristics of the specific input. As a result, PSMs must be selected or, as first considered by (ten Teije et al. 1998), *constructed*. In the former case, methods are selected from a predefined set, while in the latter case parts of existing methods or newly defined parts are combined to construct a new method.

Such a selected or constructed method possibly does not guarantee the satisfaction of all the intended goals, for example due to lack of sufficient knowledge about when to apply a PSM, or due to incompleteness of data or knowledge inherent to AI-problems. Because the intended goals are not guaranteed, one has to *validate* the constructed method. If this validation fails, we can *iterate* the selection and construction process, using the results of the validation.

In (ten Teije et al. 1998) and (ten Teije & van Harmelen 2003), the authors (semi-) automatically construct PSMs by exploiting certain restrictions in the shape of the configuration search space. The approach is based on the correspondence between the construction of problem solvers and parametric design.

Configuration of PSMs as a parametric design task

The approach of (ten Teije et al. 1998) to automated configuration of problem solvers relies on a uniform representation of (the functionality of) problem-

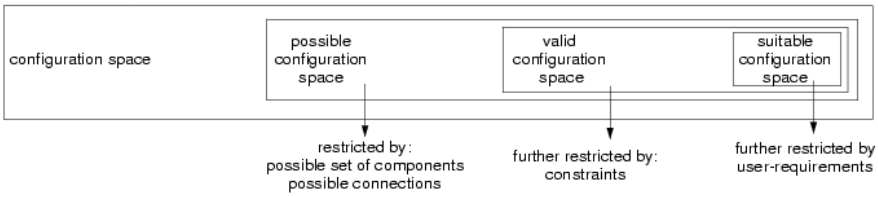

Figure 3.2: Configuration as search (ten Teije et al. (1998))

solving methods. Below we largely follow their exposition. The central idea of the uniform representation is that the functionality of a class of PSMs is captured in a single schematic formula. Some of the predicates and terms from that formula are regarded as parameters that must be further instantiated to capture different members of the class. Thus, different members of that class correspond to different definitions for the parameters occuring in the formula.

A configuration task can be seen as the task of building configurations which satisfy all the given requirements. A configuration consists of a set of components and a description of the connection between the components in the set. The basic inputs of the configuration task in the context of configuring PSMs are:

- The set of possible definitions of the components in the schematic formula. These possible definitions are the building blocks of the configuration and are given beforehand.
- The schematic formula which represents the PSM class. The mapping to a configuration problem is possible, because of the uniform representation available for PSMs.
- Constraints between PSM components and constraints between underlying assumptions of the components.
- Static and dynamic goals (see the next subsection) that have to be fulfilled.

Under this approach, a *possible* configuration is a method that contains a definition for each component of the general method schema. A *valid* configuration is a method that expresses a PSM and has no conflicts with the assumptions under which the method must operate. A *suitable* configuration is a method that satisfies the desired goals (see Figure 3.2).

Because the schema is fixed and the possible definitions of each component can be considered as the range of the parameters in a given formula, this configuration task can be interpreted as parametric design.

Static and dynamic configuration

The goal of automated construction of methods is to construct a method that produces acceptable solutions for a given problem under particular assumptions and desired goals. The approach is to first configure and then validate a method, and, if this validation fails, to iterate the configuration step.

The construction before validation is called static configuration ("Which PSM is optimal?") and configuration using the validation results results in dynamic configuration ("What should be done if the PSM does not give the desired solution?"). Using the distinction between static and dynamic configuration, static and dynamic goals can be identified. *Static* goals are requirements that can be guaranteed only on the basis of the description of the method. *Dynamic* goals are requirements of the solution that can only be validated after executing the method. This distinction between static and dynamic goals is not fixed. It depends on the knowledge that is available about methods.

Solving the configuration task

To solve the task of configuring the PSM, the authors use a PSM which is known as $Propose\text{-}Critique\text{-}Modify$ (PCM). Characteristic of a PCM method is that when a configuration is not suitable, the configuration process does not continue with a completely new configuration, but uses the test results for determining a new configuration instead of generating a new one from scratch. PCM family (Chandrasekaran 1990) methods consist of four steps: propose, verify, critique and modify. When applying the PCM method in the current context, it acts as follows:

- The *propose step* proposes a configuration. It starts by giving a partial or complete configuration. The parameters in the schematic formula are proposed based on the required static goals. These partial proposals are then completed by proposing values for the remaining parameters, and we end up with an instance of the general schema that we use for representing PSMs.

- The *verify step* involves checking that the proposed configuration satisfies the constraints and the user-requirements. Constraints on the components and on the assumptions can be directly calculated or estimated by means of domain specific formulae ("static" verification). Required "behaviour" however can be derived only by validating the method by execution ("dynamic" verification). Based on the result of these tests the dynamic goals can be verified. In the case that not all goals are met, the results of verification contains the failing goals, as well as the method that failed to meet these goals.

- The *critique step* is an analysis of why the verification failed, or why the method is not appropriate. It comprises solving the diagnostic problem of mapping from undesired behaviour to the parts of the configuration which are possibly responsible for this behaviour (i.e. assign blame). To do this it needs information about how the structure of the device contributes to the desired behaviour. In our case, this is knowledge of how properties of the components of the PSM schema relate to properties of the complete schema. In this phase, one can use diagnostic knowledge about goal violations and repairs.

- The *modify step* uses the repair information from the critique step to find an appropriate modification and executes the repair action. It changes the configuration to get closer to the specifications. In our case, this is the adaptation of the PSM. A modification action can consist of modifying an individual component, an entire method, or tuning components so that they become more compatible.

Discussion

Much of this work can be recast in terms of Semantic Web Services. Service libraries are expected to vary in dimensions of generality, formality and granularity. A central question for SWS would then become "Which service is optimal given a particular task?". In particular, an agent will be looking to solve some task, and has a few options to choose from in a UDDI-like service library. If a service is selected and it matches, we are done. In contrast, if the match is not immediate, rather than give up, an iterative brokering process should determine whether an appropriate new service can be constructed, albeit closely related to the current service.

In addition, the distinction between static and dynamic goals stands. In the context of SWS, one could think of the task of booking a trip to Germany. An example of a static goal for a travel agency service could be whether it offers trips to Germany. A dynamic goal could be to (try to) book on-line a room under 60 euros with a selected partner.

The main motivation for (ten Teije et al. 1998) using PCM as a brokering method was to the need to prevent expensive tests of dynamic goals (in casu, performing diagnosis). The same condition seems to hold for SWS : checking a dynamic goal could require access to other people's systems and could even imply rollback operations, which can be costly (in terms of time and resources required).

ten Teije & van Harmelen (2003) report that the main difficulty in realising the PCM-based approach lies in the identification of knowledge for the critique and revise steps. If certain goals are not achieved by the current task definition, which knowledge must then be exploited to identify and repair it in order to improve the match ? For SWS to be able to use PCM-

style brokering, in addition, one would need to be able to identify generic classes of web services. The early experiences of (Wroe et al. To appear) in the myGrid project seem to suggest that this should be doable once the application domain is determined. It is less clear how domain-independent classes of web services would be characterised.

Finally, the authors use their scheme to construct one predifined problem-solving method. One could argue that, if a schematic formula for a composed service were built, with different possible instantiations, then this approach could equally serve as a way to perform (guided) service composition. We refer the reader to the brokering section in Section 4.4.1 for a discussion on further synergies with the F-X framework.

3.2 Other approaches

Although our focus for this thesis is on brokering techniques used for PSMs, here we refer to two additional techniques which in our opinion carry a lot of potential for service composition.

3.2.1 Planning

Planning is nowadays a popular way to think about service composition. The idea is to conceive each Web service as an action. The advantage of exploiting an action metaphor is that it lets us use all earlier research on reasoning about action. We refer to (Wickler 1999, Chapter 9) for a good overview of brokering strategies with regard to action formalisms.

3.2.2 Deductive synthesis

As noted in the previous chapter, DAML+OIL is not, by itself, sufficiently rich to describe programs. In the context of the Grid and e-Science, (Bundy & Smaill 2002) have recently started work on a logic that is sufficient for deductive synthesis, by either extending DAML+OIL or by translating it into a richer logic. Recent work is investigating the potential of event logic for this purpose (Bundy, Smaill & Yang 2003). Event logic is a mathematical structure based on constructive Type Theory designed to express the key features of a distributed computing system at the level of abstraction appropriate for specifying interactive behaviours.

The aim is to support rapid (re-)assembly (i.e. composition) of a dependable, high quality Grid application from available Grid services. As (Bundy & Smaill 2002) explain, in deductive synthesis, a program is built automatically as a side effect from a constructive proof that this program's specifications can be met. An automated reasoning system is required to prove a conjecture of the form:

$$\forall inputs.\exists output.spec(inputs, output)$$

where *spec* is the specification of a relationship between the inputs and the output. Provided that the logic is constructive, i.e. excludes pure existence proofs, then the proof will implicitly define the construction of this output from all possible inputs (Bundy & Smaill 2002).

Once synthesised, it becomes possible to re-synthesise or revise the program, e.g. with a slightly different specification. Even if the original synthesis proof had required human interaction, this new process might be completely automated via the use of analogy, since many of the previously hard steps may now be simply transcribed to the new proof (Bundy & Smaill 2002). One application here could be the automated recovery from failure of a Grid application, for example because of one particular Grid service going down.

In the next chapter, we focus on a lightweight alternative for the previously mentioned approaches. We will confront our insights from PSM brokering with this new approach and look for synergies.

Chapter 4

F-X: brokering lightweight formal capability descriptions

In previous chapters, an array of different languages possibly relevant to service description was displayed. This chapter introduces yet another language for describing service capabilities, but one which stands out by the particular angle chosen. The aim of the continuing work on the F-X system at Edinburgh is to design a service specification language which is as lightweight as possible, while retaining those elements that are essential to *automated* service composition. In what follows, we describe the rationale and technology behind F-X, and analyse in detail the differences and similarities with UPML and DAML-S (see Chapter 2), two important proposals on the current Semantic Web Services scene.

4.1 Rationale

The F-X system (Robertson 2001) aims to provide a general architecture for formal knowledge management systems, spanning the entire knowledge management lifecycle. F-X considers a knowledge management system to be a large and distributed collection of components capable of expressing knowledge in some form. The knowledge captured inside these components can be pretty much anything : knowledge bases, programs, text, interfaces with humans, ...

From this general definition, it follows that different kinds of automation will be required, and a unifying framework to cater for them should be in place. To this end, F-X proposes a standard, lightweight notation for describing knowledge components, a brokering algorithm tho exploit these descriptions, and a "low level" architecture for communicating between components. These various elements are explained below.

4.2 Technology

To kickstart our overview of F-X technology, first we describe F-Comp's language tenets. Subsequent subsections zoom in on the brokering support offered for services and how brokering results are communicated. We conclude with an example. The reader should note that these subsections basically summarize work described in (Robertson 2001) and Potter (2003*a*, 2003*b*), but that we add our own comments and figures throughout the text.

4.2.1 F-Comp : capability descriptions for F-X

The aim of F-X's component representation language, dubbed "F-Comp", is to describe knowledge components in as simple a style as possible, allowing communication between components to be achieved while imposing few constraints on the internal design of the individual component (Robertson 2001).

There is a clear tension between keeping the number of language primitives down and the language simple, while at the same time trying to support efficient reasoning across the range. In its design, F-X makes no claims of having resolved this issue. However, unlike other proposals, it does make this trade-off explicit and provides a clean starting point to experiment with possible extensions.

F-Comp is inspired by and combines work on coordination between distributed agents (Robertson, Silva, Vasconcelos & de Melo 2000) and efforts on the Unified Problem-solving Method description Language (Fensel et al. 2003) (see also Section 2.1).

F-Comp distinguishes between ontologies, domain descriptions, tasks, and problem-solving methods. Figure 4.1 below provides an architectural overview of F-Comp. It also provides some insights as to how F-Comp differs from UPML (compare with Figure 2.1 in Section 2.1). In this subsection we limit ourselves to F-Comp's elements. Section 4.4 will contrast F-Comp with its main "contenders".

Ontologies

An ontology in F-X is of the form:

$$ontology(Name, \{T_1, \ldots, T_n\}, \{A_1, \ldots, A_2\})$$

where:

- $Name$ is a constant used to identify the ontology uniquely.
- Each T describes the type of an element of the ontology.
- Each A is an axiom of the form $U \rightarrow E$, where

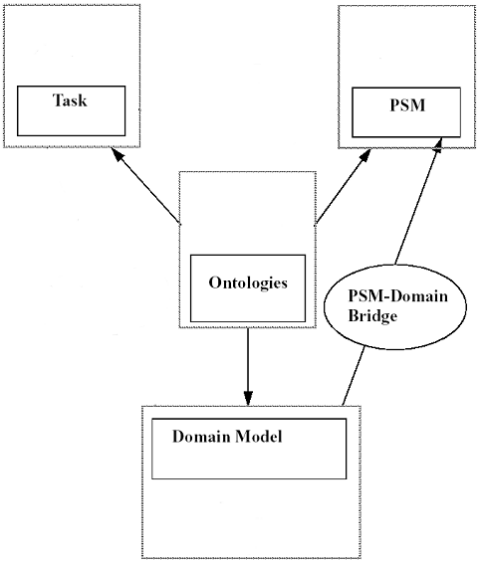

Figure 4.1: F-Comp architecture (adapted from (Fensel et al. 2003))

- U is a unit term which is valid in the ontology
- E is any expression constructed from the expressions of the ontology and the normal logical connectives.

For example, $breathes(X) \rightarrow animal(X) \wedge alive(X)$ would be a structurally correct axiom. As hinted at in this example (with two terms in the head of the clause), F-X is intended as an architecture that can take any First Order Predicate Calculus (FOPC) fragment. However, to date, most experiments with F-X have concentrated on the Horn logic case (where the number of clauses in the head is limited to one).

Domain descriptions

The Domain description introduces domain knowledge, merely the formulas that are then used by problem-solving methods and tasks. The essential components of a domain description in F-Comp are: the name of the ontology it uses; the properties of the knowledge expressed in the domain model; and the domain knowledge itself. A domain description thus is of the form:

$$domain(Name, O, \{P_1, \ldots, P_n\}, D\})$$

where:

- $Name$ is a constant used to identify the domain description uniquely.
- O is the name of an ontology.

- Each P specifies a property of the domain model, described as a FOPC expression in which each unit term is valid in the domain's ontology. The properties (synonym for theorems) can be derived from the domain knowledge and they are visible to and directly used by the broker.
- D is the domain model itself, which may be in any of a number of knowledge representation languages. As mentioned above, the core F-X architecture is neutral to the choice of representation language in D.

Problem-solving methods

The essential components of a problem-solving method in F-Comp are: the name of the ontology it uses; the competences[1] which it provides; and the problem-solving mechanism delivering these competences. A problem-solving method in F-X is therefore of the form:

$$psm(Name, O, \{C_1, \ldots, C_n\}, M)$$

where:

- $Name$ is a constant used to identify the problem-solving method uniquely.
- O is the name of an ontology.
- Each C_x specifies a competence of the problem solver, of the form

  $$c(C, I, O, C_e)$$

  where:

  - C is a competence specification of the form $G \leftarrow P$, where G is a goal which is claimed to be satisfiable by the problem solver, given the conditions P.
  - I is a set of constraints on variables in C which are required to hold before we attempt to utilize that competence.
  - O is a set of constraints on variables in $C \leftarrow P$ which should hold after we have successfully utilised that competence.
  - C_e is a set of competence specifications which must be satisfied externally to the problem solver. Normally this is achieved by utilising the competences of other problem solvers.
- M is the problem-solving mechanism, which may be in any programming language. The core F-X architecture is neutral to the choice of programming language for M.

[1] In this report, the term *competence* is used interchangeably with the term *capability*.

Potter (2003*a*, 2003*b*) reformulates the F-Comp form for problem-solving methods into "Semantic Web Services speak". This version is not compatible with the one used above however. As such, we recast it in the above format:

$$service(AgentName, AgentURI, AgentOntology, \{C_1, \ldots, C_n\})$$

where:

- $AgentName$ is a constant used to identify the service (in conjunction with the AgentURI).
- $AgentURI$ contains the location of the service.
- $AgentOntology$ is the name of an ontology against which the agents competences are to be understood.
- Each C_x specifies a competence of the web service, of the form

  $$c(C, InputRoles, OutputRoles, ExternalCompetences)$$

  where:

  - C is a competence specification of the form $Name \leftarrow Preconditions$, where $Name$ is a goal which is claimed to be satisfiable by the web service, given the conditions $Preconditions$.
  - $InputRoles$ is a set of constraints on variables in C which are required to hold before we attempt to utilize that competence.
  - $OutputRoles$ is a set of constraints on variables in $Name \leftarrow Preconditions$ which should hold after we have successfully utilised that competence.
  - $ExternalCompetences$ is a set of competence specifications which must be satisfied externally to the problem solver.

The reader will notice that a new term AgentURI was added to indicate where the service resides on the network. This now caters for the situation where services have the same AgentName. This situation is commonplace in for example bioinformatics, where multiple mirrors can exist of the same service.

More fundamentally, one can also see that the problem-solving mechanism M is missing. This makes sense, as this term provides no support whatsoever during the F-X brokering process (see the next subsection). We will revisit this issue later, but, for now, suffice it to say that if F-X wishes to position itself as a lightweight brokering environment for service lifecycle management, it makes sense to strip away most information about service pragmatics and operation.

Bridges

Knowledge components may use different ontologies so there must be translation between terms in these different ontologies if the components are to communicate. A bridge defines this translation. A bridge in F-X is of the form:

$$bridge(Name, O_f, O_t, \{T_1, \ldots, T_n\})$$

where:

- $Name$ is a constant used to identify the bridge uniquely.
- O_f is the name of the ontology from which a translation is being made.
- O_t is the name of the ontology to which a translation is being made.
- Each T specifies a translation from a unit term of O_f to a unit term of O_t, of the form $t(X_f, X_t, C)$ where:
  - X_f is a unit term used in O_f.
  - X_t is a unit term used in O_t.
  - C is a constraint describing the conditions under which the translation is considered valid.

4.2.2 Reasoning with F-Comp

Assembling knowledge components automatically

Robertson (2001) formulates the vision for the F-X broker as follows. If we have an open knowledge management architecture then we must assume that our set of available knowledge components can change and that there will be too many of these to search manually. We therefore need some form of automated brokering mechanism to identify the assemblies of knowledge components appropriate to a task we wish to achieve. The purpose of the the broker is to do this using the information in the F-Comp specifications.

A four-step brokering process

The way F-X regards brokering is as a means of orchestrating problem solving among agents, where each agent contains one or more knowledge components and is capable of utilising these to solve problems. The brokering task is then to identify, from all the available agents, a subset which contains collections of knowledge components appropriate to our task.

Ideally we would like this set to contain precisely and only those agents which will perform the task. In practice this is unlikely because the only

way to be sure a task can be performed is for the agents to run the appropriate confederation of knowledge components. If we were able to prove appropriate properties of all our components then we might not be in this situation but such extensive use of verification appears impractical.

A more realistic approach is, by reasoning about the specification of knowledge components, progressively to reduce the size of the set of agents which appear appropriate until we reach the point where the set is sufficiently small that we have a high probability that the components selected from it will be able to perform the task.

To support this idea, F-X brokering — in the most general case — consists of the following steps:

1. Find components with ontology signatures that matches to those of the task. The components found in this step are using the same ontology (possibly with translation/bridging).

2. Within those componenents, find signature matches with consistent ontological axioms. The remaining components are consistently using the same ontology (possibly with translation/bridging).

3. Match competences of tasks and components. The resulting components offer the competences necessary to solve the problem.

4. Run the problem solvers. This makes clear which components were actually sufficient to solve the problem.

Agent-broker interaction

Agents advertise their competences simply by sending these to the broker, which records the competences and the agents who claim to be able to supply them. In the next stage, another agent sends a query to the broker. The broker constructs from its competence descriptions (now detached from the agents) its internal description, which is called a "brokerage structure" of how the query might be answered based on those competences. It then translates its brokerage structure into a sequence of KQML-like performative statements describing the messages which it thinks should enable the query to be satisfied by requesting appropriate agents to discharge their competences. The brokerage structure itself is built in a Definite Clause Grammar (DCG) style, where the grammar is used to generate the sequence of terminal symbols, corresponding to performatives, by unpacking the brokerage structure (see Appendix and (Robertson 2001) for details). We provide an example of a brokerage structure in the Section 4.3. Note that the sequences constructed using this approach are constrained to be linear, in that there will be no alternative paths within a sequence: disjunctions are represented through distinct complete sequences.

In the final stage this performative information is used by the agent which sent the query to select which agents to contact; to send appropriate messages to them; and await appropriate responses.

The method does not prescribe that the agents submitting queries must be different from the agents supplying competences. It also does not prescribe that there must physically be a single, centralised broker to which all the agents advertise and to which all queries must be addressed. It would be possible to use the method to in a decentralised brokering, provided that each agent had a copy of the brokering mechanism and there was a way of broadcasting competences to groups of agents.

We provide a concrete example of a brokering scenario below. See (Robertson 2001) for an example from the system dynamics domain and (Potter 2003*b*) for a ticket-office scenario.

4.2.3 Communication in F-X

For transporting service requests from a component to the broker and for returning brokering results to a component, F-X relies on the AKT Bus. The AKT Bus, built at the University of Aberdeen, allows for communication between problem solvers. As Robertson (2001) writes, the Bus uses a Linda server to manage concurrent access and adaptation of a tuple space in Prolog (where tuples can be considered as Prolog facts).[2]

4.3 F-X in action

Now that all elements have been described and the supporting reasoning mechanism explained, we demonstrate how F-X works in practice, drawing on a trip-booking example.

4.3.1 F-X implemented in Prolog

Before we walk through the example in the next section, we need to understand how F-X gets translated to Prolog. In particular, a service description

$$service(AgentName, AgentURI, AgentOntology, \{C_1, \ldots, C_n\})$$

contains a number of competences C_x, each of the form

$$capability(Name \leftarrow Preconditions, InputRoles, OutputRoles,$$

$$ExternalCompetences)$$

In Prolog, the different elements can be interpreted as follows (slightly adapted from (Potter 2003*a*)):

[2]For further information about the AKT Bus, see http://www.csd.abdn.ac.uk/research/akt/aktbus/aktbus-1.0/

Competence name The name can be thought of as a Prolog goal, plus the (input and output) variables associated with it.

Preconditions The preconditions describe some state of affairs (in terms of a list of Prolog goals) that must be known (or, if not known, then proven) to be true before this goal can be discharged.

Input and output roles The goal will contain input variables (i.e., in Prolog terms, instantiated variables) and output variables (uninstantiated variables which will, on successful execution of the competence, become instantiated appropriately). The input and output roles are two lists which serve to explicitly declare which variables are inputs and which outputs respectively, as well as providing some typing (against either the service ontology or some assumed common type ontology (providing integer, string, float, etc.)) of their expected values.

Required external competences A list of goals which are required by this agent during execution for the successful discharge of this competence, but which can not be achieved internally: the agent must invoke the competences of a second agent if this competence is to be provided, and hence, at execution time, the environment must contain an agent capable of supplying each of these external competences for this particular competence to be considered.

Note that in the version of the brokering algorithm we are using for our example below, capabilities that have external competences are actually separated out into a different construct *p_capability*.

A query agent requests a service by sending the desired goal description (containing instantiated or uninstantiated variables, or some combination of the two) to the broker. Upon receipt of a query, the broker attempts to generate one or more sequences of performatives, which, if executed, will result in the successful discharge of this goal.

The brokering algorithm involves locating a service that matches the current goal, determining that there are additional services available that will satisfy both the preconditions and the external calls of this service, and adding the information necessary to invoke these services in the appropriate order into the current sequence. This algorithm is repeated until no more unique sequences are found; these sequences are then returned as alternative solutions to the current query (Potter 2003*a*). The complete algorithm in Prolog code can be found in Appendix A.

4.3.2 A trip-booking example

Our example describes the following simple scenario. A user (machine or human) is looking for information on making a return trip, which includes

a return flight and a hotel stay at the destination. In the scenario, this user relies on a travel agent to make this happen. The travel agent is able to offer trip information and to facilitate booking the trip.

There are three agents to support this process, which all publish their competences with the broker.[3]

Airline agent *aa*

One actor in this scenario is the airline agent *aa*, which is able to inform other agents of flight availability, based on time period, departure point and destination. Here we restrict ourselves to one agent, but obviously multiple airline agents would be possible. The agent also accepts bookings, on the condition that payment follows. The agent has the following capability descriptions. We use the *capability* construct to represent competences that do not require external agents to achieve their competence.

$$capability(aa, (informFlight(D, A, LD, RD, F) :-$$

$$departure(D), arrival(A), leavingDate(LD), returnDate(RD), flight(F))).$$

$$capability(aa, (bookFlight(F) :- flight(F), payFlight(F))).$$

$$capability(aa, departure(D)).$$

$$capability(aa, arrival(A)).$$

$$capability(aa, leavingDate(LD)).$$

$$capability(aa, returnDate(RD)).$$

$$capability(aa, flight(F)).$$

$$capability(aa, payFlight(F)).$$

The first capability requires that all variables are correctly typed through its input and output roles. It imposes preconditions nor postconditions.

aa also publishes a second capability. As before, the capability requires for the output to be properly typed (i.e. flight(F)). In addition, it imposes as a postcondition that when a booking of a flight F is made, payment follows. We simplify the scenario by not including personal identification data for payment (which is plausible if, for instance, the airline has a list of partner travel agents that have a running account). Alternatively, we could include an additional credit-checking agent. The remaining capabilities provide typing support.

[3]The reader should note that once again we will use the terms *capability* and *competence* interchangeably. The brokering algorithm we use (given in Appendix A) is slightly out of touch with the latest F-X web services notation, but the core functionality remains the same.

Hotel agent *ha*

The second agent, *ha*, is capable of informing interested parties of available hotel accommodation in a number of locations, based on the desired period. Its competence also includes accepting hotel bookings. For any practical application, one would have multiple hotel agents publishing with the broker.

$$capability(ha, (informHotel(L, ID, OD, HS) :-$$

$$inDate(ID), outDate(OD), location(L), hotelStay(HS))).$$

$$capability(ha, (bookHotel(HS) :- hotelStay(HS), payHotel(HS))).$$

$$capability(ha, inDate(ID)).$$

$$capability(ha, outDate(OD)).$$

$$capability(ha, location(L)).$$

$$capability(ha, hotelStay(HS)).$$

$$capability(ha, payHotel(HS)).$$

The first capability carries the same kind of typing restrictions as the previous *aa* agent. Also analogous to *aa*, the next capability demands that payment comes through after booking. The remaining capabilities again support correct typing.

Travel agent *ta*

Finally, there is a travel agent *ta*, which can either inform the user of trip availability, or book a trip with the other agents on the user's behalf.

To achieve this, the travel agent relies on the other agents. Consequently, this is the part where the broker will have to combine the capability of different agents in order to satisfy a query. The *p_capability* construct represents a (partial) capability that depends on competences offered by other agents. It is of the form:

$$p\_capability(K, C, E)$$

denoting that agent K can deliver capability C if external capability E is available.

Trip information In our example, the first partial capability of *ta* describes what it expects in order to offer trip information:

$$p\_capability(ta, (informTrip(D, A, LD, RD, F, HS) : -departure(D)),$$

$$(informFlight(D, A, LD, RD, F), informHotel(A, LD, RD, HS))).$$

In order to discharge successfully, both informFlight and informHotel must be satisfied. We use LD (leaving date) and RD (return date) in informTrip (instead of informHotel's InDate and OutDate) since those two dates will always be marking the start and end of a trip (see below).

The broker algorithm we work with requires a precondition on a partial capability, and for this reason we insert departure(D). This condition is redundant, as it would be checked by *aa* anyway. See (Robertson 2001) or Appendix A for the broker algorithm used.

For simplicity, we assume that, when flying, the traveller still arrives at her destination the same day. In addition, we assume hotel check-in is on the same day as when the outbound leg arrives, and check-out occurs on the day of the inbound leg. Therefore, we can reuse the informTrip variables LD and RD again when instantiating the informHotel variables ID (in date) and OD (out date).

Nevertheless, these are different concepts, and one could think of scenarios where they do not correspond so well. In such a case, one would specify their relationship explicitly using a bridge, to improve de-coupling and maintainability. Similarly, L (location) and A (arrival) could be mapped explicitly using a bridge. A bridge has the following structure in the brokering algorithm:

$$corresponds(K_1, C_1, K_2, C_2, G) \leftarrow P$$

where C_1 is a capability in agent K_1 which corresponds to capability C_2 in agent K_2 with the constraint G restricting the acceptable substitutions for variables in C_1 and C_2. The precondition P is an optional conjunction of other correspondences upon which the main correspondence depends. We refer to (Robertson 2001) for an implemented example of a bridge.

Trip booking The second partial capability of *ta* publishes trip-booking capability and has the following structure :

$$p\_capability(ta, (bookTrip(D, A, LD, RD, F, HS) : -departure(D)),$$

$$(informFlight(D, A, LD, RD, F), informHotel(A, LD, RD, HS),$$

$$bookFlight(F), payFlight(F),$$

$$bookHotel(HS), payHotel(HS))).$$

This specifies that, if both flight and hotel are available (through other agents), then we will move forward with their respective booking and payment.[4]

Running the example

Based on these capability descriptions, the broker can deduce which (group of) agent(s) are most likely to execute a query successfully.

A simple query to ask the broker might be:

$$? - brokerable(informFlight(karlsruhe, edinburgh,' 210803',' 230803'$$

$$Flight), Result).$$

This results in the following (hard to read) result (which we have shortened a bit). Below we show an example of a set of performatives such a result is translated to. Note that this result does not yield an answer to the question of which flight to take. Rather, the broker returns information on which constellation of agents to contact that together will be able to answer this question. We refer the reader to (Robertson et al. 2000) for a more detailed exposition of the algorithm (based on the solution structures)

$$Result = c(aa, dq(informFlight(karlsruhe, edinburgh,' 210803',' 230803', Flight),$$

$$c(conj, co(c(aa, departure(karlsruhe)), c(conj, co(c(aa, arrival(edinburgh)),$$

$$c(conj, co(c(aa, leavingDate(...)), c(conj, co(...)))))))))$$

A more difficult query might be to ask the broker how to achieve a booking for a trip from Karlsruhe to Edinburgh (including a hotel stay):

$$? - brokerable(bookTrip(edinburgh, karlsruhe,' 210803',' 230803', Flight,$$

$$HotelStay), Result).$$

We obtain the following (again shortened) answer:

$$Result = c(ta, pdq(bookTrip(edinburgh, karlsruhe,' 210803',' 230803', Flight, HotelStay),$$

$$c(aa, departure(edinburgh)), c(conj, co(c(aa, dq(informFlight(edinburgh,$$

$$karlsruhe,' 210803',' 230803', Flight),$$

$$c(conj, co(c(aa, departure(...)), c(conj, co(...))))))),$$

$$c(conj, co(c(ha, dq(informHotel(karlsruhe,' 210803',' 230803', HotelStay),$$

$$c(conj, co(...)))), c(conj, co(c(aa, dq(...)), c(conj, co(...)))))))))$$

[4]In the scenario, we ignore the issue of locking database transactions.

This result gets translated into a more readable series of KQML-like performatives that show how to contact the different agents and in which order:

$$Performatives = [ask(aa, departure(edinburgh)), ask(aa, departure(edinburgh)),$$
$$ask(aa, arrival(karlsruhe)), ask(aa, leavingDate('210803')),$$
$$ask(aa, returnDate('230803')), ask(aa, flight(Flight)),$$
$$ask(aa, (informFlight(edinburgh, karlsruhe,' 210803',' 230803',$$
$$Flight) : -departure(...), ..., ...)),$$
$$ask(ha, inDate('210803')), ask(ha, outDate(...)), ask(...)|...], ...$$

We can see from this example that both the *aa* and *ha* agent will be heavily involved in the problem-solving process.[5]

4.4 F-Comp compared

Now that we have a fair idea of the inner workings of the broker, we show how F-Comp compares to the approaches we reviewed earlier. First we study the relationship between F-Comp and UPML, which is followed by F-Comp versus DAML-S. Based on this analysis, in the next chapter we will propose a number of changes to F-X.

4.4.1 F-Comp and UPML

In this section we analyze how F-X differs from UPML. We compare the respective interpretations of tasks, problem-solving methods and assumptions. We also contrast the differing approach to brokering, and study possible synergies.

Tasks

For Robertson (2001), tasks and PSMs in UPML relate to each other as follows.

> Task descriptions in UPML are similar in structure to problem-solving methods. Both are declarative descriptions of problem solving but the former is intended to describe the problem to be solved while the latter is intended to be able to solve problems with a given set of competences. Thus, essentially, task descriptions are PSMs which do not commit to a specific method,

[5] One can also see that departure(edinburgh) is tested twice. This is due to the redundant precondition we had to insert earlier. We could easily add an extra clause in the brokering algorithm in Appendix A to remedy this situation.

defining only the competences necessary to complete a task. For simplicity, F-X assumes that competences are only advertised when methods exist to utilise them, therefore we do not use task descriptions. It is, however, possible that task descriptions may be re-introduced later.

As a result, in F-X at present, the notion of a task is equal to a run-time query in Prolog. By rejecting the notion of (stored) tasks, F-X both gains and loses.

F-X gains mainly in terms of simplicity and understandability. The query-based approach of the broker provides a lean and a well-understood architecture.

Below we provide a few arguments that could be made in favour of using stored tasks. The notion of tasks has potential for KBS construction and for matching Web services. It must be said that it is hard to assess the value of these arguments as to date, very few task-based systems have actually made it into practice.

Supporting KBS construction In Knowledge Engineering, the different problem types are well known. However, often people also know these problems cannot be solved, for example due to their computational complexity. When a task competence cannot be achieved for sure, one would still like a way to express this, if only to give a broker clues into how to "build bridges" toward more doable PSM competences — this was the first argument that was made in Section 2.1.3.

The second argument in that section argued that tasks, other than introduce desired functionality, also introduce a generic description of the type of domains they can be instantiated to — i.e. they provide templates. As such, one can guide the user in building sensible tasks (see also (ten Teije & van Harmelen 2003)).

Supporting Web services The above arguments generalize only partly to general software components. Unlike a broker for knowledge engineering, a broker for general software components will never be able to anticipate all possible types of problems. Nevertheless, it is clear that in some fields general classes of tasks do exist, where templates could be instantiated. For instance, bioinformaticians often perform a number of standard computational tasks which they execute in a rather strict order (see (Wroe et al. To appear)).

In addition, analogous to knowledge engineering, in some cases, clearly insoluble competences (e.g. time or cost based) would still get advertised, and a broker might use this information to come up with proposals of more realistic competences.

More generally, by dropping the concept of a task, F-X excludes itself from all brokering scenarios where requesters publish their "tasks" in public registries (see Section 1.2.2).

Finally, getting rid of the task concept (obviously) means there is no way to map between tasks and services. This prohibits the n-to-m matching scenario for tasks and services suggested in Section 2.1.1. In F-X, currently "task names" (i.e. query names) have to match service competence names exactly. Clearly, one cannot hope to automate the mapping problem, but that is not to say there is no room for improvement here.

Problem-solving methods

Contrary to UPML, F-X makes no distinction when it comes to complex and primitive PSMs. The only information that is published about the reasoning process of a PSM is the external competences required. These roughly correspond to the notion of subtasks used in UPML's complex PSMs (but one has to take into account the simplistic notion of a task in F-X). F-X PSMs do specify a problem-solving mechanism, but this mechanism is not used during the brokering process. In our view, it should therefore be removed, in the same way that F-X PSMs do not incorporate the notion of pragmatics.

Interestingly, PSMs specified in UPML each publish only one competence, whereas PSMs in F-X offer multiple competences per PSM. One reason for this might be that F-X stems from an agent tradition, whereas UPML describes software methods, which typically perform a specific function.

Finally, as far as PSM communication goes, both F-X and UPML remain agnostic about how to implement the communication layer, and show little interest to hook up with industry efforts. F-X implemented the AKT Bus to support communication, while within IBROW some work was done using FIPA.

Assumptions and domain knowledge

The notion of assumptions, which is so prominent in UPML, is discarded in F-X. In F-X, constraints formulated on inputs through input roles are meant to capture both the dynamic and static aspects of such restrictions.

Evidently, this means that in F-X no assumptions can be formulated on domain knowledge. In UPML, domain descriptions (called Domain Models) consist of three elements: properties, meta-knowledge, and domain knowledge itself. Meta-knowledge is meant to capture the implicit and explicit assumptions made while building a domain model of the real world. Meta-knowledge is assumed to be true, i.e. it has not been proven or cannot be proven. Since F-Comp has no notion of assumptions, no notion of meta-knowledge is available in its Domain descriptions. What remains are the

properties and domain knowledge.

Brokering

We now contrast F-X with the three approaches to PSM brokering we reviewed in the previous chapter.

Configuring knowledge-based systems The first brokering mechanism used the IRS methodology (Crubézy et al. 2003). On the one hand, F-X cannot compare to the support for mapping present in the Protégé Internet Reasoning Service. On the other hand, one cannot compare IRS to the competence matching functionality present in F-X. Thus, it seems natural to think about ways to link both approaches.

One way to achieve this would be to use F-X as a plugin to the IRS in order to handle competence matching. In this scenario, F-X could still reside on a remote location, but one would outsource all IRS competence matching to it.

The reverse — IRS as a plug-in to F-X — sounds less plausible, but improved mapping support would make a welcome addition to F-X. In this context, we note that Stanford Medical Informatics are about to release a stand-alone mapping tool.[6] Such a tool would bring valuable support for building bridges and refiners. At the moment, there is only limited experience available in the F-X setting with regard to the first two steps of the four-step brokering process outlined in Section 4.2.2. Improved mapping support would help remedy this situation.

It is unclear whether KMi's classification PSM library (Motta & Lu 2000) could be used in F-X in a non-trivial way. The way this library is constructed makes the PSM selection process rather trivial — all the intelligence is in configuring a task, whereas F-X is more about matching tasks with PSMs.

Brokering as assumption discovery In Section 3.1.2 we mentioned some difficulties that arise when discovering assumptions using theorem proving. F-X stays clear of such issues, and leans more towards the practical, engineering side. F-X shows little interest in formal software verification techniques. Rather, it provides support for building an environment of problem solvers of a reasonable scale (for instance, the F-X broker is being considered to serve as central broker for the AKT project, linking together many separate efforts).

Configuration as parametric design Currently, F-X offers no support to adapt the behaviour of the broker depending on the results returned. To

[6] Details at `http://protege.stanford.edu/workshop_vi/Monica_Crubezy_PWS-PSMLibrarian-Jul03.pdf`

some extent, this is to be expected, since there are no general principles how to rank results, and F-X aims for generality. However, F-X should at least offer the framework in which it would be easy to specify such heuristics. One strong point of the "Configuration as parametric design"-approach is that it is rooted in the well-founded reflection architecture of (ten Teije & van Harmelen 1996). In addition, in contrast to F-X, this approach can handle both dynamic and static goals as part of an interactive brokering process. In our opinion, F-X would benefit from a similar mechanism to influence the number of results yielded during and between different "brokering cycles". The case-study we present in the next chapter can be considered a first step toward this goal.

4.4.2 F-Comp and DAML-S

We limit our comparison between F-Comp and DAML-S to those features that are different from the earlier "UPML versus DAML-S"-exercise of Section 2.2.5. We concentrate on the respective object languages and the form of each service ontology.

Object language F-Comp service descriptions are specified in Horn clauses, whereas DAML-S relies on the Web-enabled Description Logic DAML+OIL and thus relies on $SHOQ(D)$. Horn clauses are well known and well understood. In terms of the established user base, SHOQ(D) DL is much less of a standard.

The specification of services in F-Comp is flat, whereas DAML-S allows to specify services in a lattice, and to have service classification occur dynamically based on properties. The case-study in the next section introduces some notion of hierarchy in F-Comp, based on generalization relationships between competence descriptions.

It is well known that ABox reasoning does not perform well in DLs. The way this is currently "solved" in DAML-S Profile matching is that instances are classified as concepts among the concepts when doing service discovery (see (Li & Horrocks 2003) for an example). It is likely this restriction will turn against DAML-S once many service instances are published in a service ontology, unless more efficient techniques for A-Box reasoning are proposed (see (Motik, Maedche & Volz 2003) for a promising approach in this respect). Modelling services in a Logic Programming setting (e.g. Prolog) is more attractive from this perspective : instance reasoning can become more efficient (In contrast to DLs, LP can rely on techniques such as bottom-up computation and "magic sets" (see (Volz, Motik, Horrocks & Grosof Submitted) and (Motik et al. 2003) for more details)).

Service ontology F-Comp roughly corresponds to the functional part of the DAML-S Profile. It has little correspondence with the Process Model,

except that it publishes external competences. External competences indicate some aspect of service behaviour, and hence in DAML-S those would get modelled in the Process Model. Given that the result of a brokering cycle in F-X is described in terms of a sequence of performatives, in theory one could publish this result in the Process Model as a new composite service, based on the sequence construct. Since these services are generated dynamically, one would consider such an option only for caching purposes.

F-Comp preconditions can be any Prolog constraint. This is in contrast with the limitations in place for specifying rules in DAML+OIL, which we already mentioned in Section 2.2.4. There is no deep conceptual difference between preconditions and postconditions in F-Comp. Since F-X adopts an algebraic specification view of the world, these conditions are just constraints, which claim nothing about states of the world. In addition, F-Comp has no notion of effects in the world.

To conclude, below we provide a comparative overview of all three languages. In the next chapter, we will build on the lessons we learned from this comparison, and try to combine the best of both worlds. In particular, we will extend the F-X environment with a notion of hierarchy and more flexible competence matching.

	UPML	F-Comp	DAML-S
View capability as program	Yes (PSM Competence)	Yes (PSM Competence)	Yes (functional part of Profile Model)
Preconditions	Yes	Yes	Yes
Postconditions	Yes	Yes	Yes
Architectural guidelines and assumptions	Yes	No	No
View capability as action	No	No	Yes (Service Model)
Effects in the world	No	No	Yes
Tasks	Stored seperately (task=psm)	Run-time query (task=psm)	Can be stored seperately (request = advert)
Refinement	Refiners	No	No
Service dependencies	Subtasks in complex PSMs	External competences	Atomic processes in composite process
Ontology mapping constructs	Adapters (Bridges and Refiners)	Bridges	DAML+OIL roles (same class/instance)
Object language	F-Logic, sorted logics	Horn logic	DAML+OIL (SHOQ(D))
Discovery matching technology	Interactive theorem proving / parametric design	Unification	Subsumption reasoning on Profile
Type of composition supported	Semi-automated	Automated	Automated (with planning)
Composition technology	Interactive theorem proving / parametric design	Backchaining	Map Service Model to formalism and do planning
Non-functional information	Pragmatics	No	Non-fuctional parts of Profile Model
Interoperation	Communication model	AKT Bus	Grounding Model

Chapter 5

Towards more flexible competence matching using query relaxation

In this chapter, we use a query relaxation mechanism to add flexibility to the F-X competence matching process. Part of the presented technique relies on the use of OWL domain ontologies, which we import into F-X and subsequently exploit for matchmaking.

In Section 4.3.2 we presented an example scenario where a travel service helps to book a trip using two other services. These additional services offered hotel and flight information as well as booking facilities. Here we will further build on this example to illustrate the ideas.

5.1 Current situation in F-X

At the moment, in F-X we can match competences between services through exact string matching between competence names, and using bridges.

Just to remind the reader, in UPML, to warrant the use of the term "bridge", one must be connecting different types of elements of the software architecture. Refiners on the other hand connect elements of the same kind. In F-X, a bridge is typically used to overcome differences in *vocabulary* between competences. The use of a bridge in F-X could thus be seen as bridging between competences (with the same core functionality) and different domain models (i.e. a kind of domain-PSM bridge).

Further, in UPML, if the functionality of one PSM is more specific than that of the other, one can try to connect these using *PSM refiners* (see Section 2.1.2). However, automated software refinement is not doable in the general case. Indeed, in AI, a whole subculture is working on software component generalization. Given F-X's aim to be as general as it is lightweight, no claims are made toward refiners.

5.2 Relax and obtain better results

Having said this, in practice it could still be useful to have some (less general) notion of refinement available. Our proposal in this section makes few claims with respect to general applicability, but does provide an additional handle on how to relate competences in practice. The notion of relaxation used here could thus be regarded as adding an operational dimension to refinement of problem solvers, with an application range limited to logic programming.

One important reason to introduce relaxation is to enable better management of the number of results yielded by the broker. Currently, such a notion of control is underdeveloped in F-X. We would like to arrive at a scheme similar to the one described in Section 3.1.3, where the PSM lifecycle was extended with monitoring. This issue is not orthogonal to the current brokering architecture, since it involves more than simply awaiting a broker run, and then reformulating a query based on those (final) results. Rather, an adaptation of the core brokering algorithm is needed.

To give a concrete example, if during the brokering process the broker cannot find a particular competence, it could attempt to generalize the current query. If successful, the broker should incorporate these additional results as part of the returned brokering structure.

Conversely, a user may not be interested in getting back thousands of results after several days of waiting. Rather, approximate results may be sufficient — and more relevant. Hence, if too many competences match during brokering, one needs a possibility to draw in the reins and introduce search heuristics. One would then continue the remainder of the brokering process based on a reduced subset of the solution space.

5.3 Query relaxation based on taxonomic information

Since the brokering algorithm we focus on is implemented in Prolog, we turn our attention to relaxation techniques for logic programming (LP). In particular, we draw on work that has its origins in database query relaxation, but which was adapted for LP. Gaasterland, Godfrey & Minker (1991) and Gaasterland (1997) introduce a method they call *relaxation* for expanding deductive database and logic programming queries. The resulting query-answering system "answers not just the user's literal query, but the intended query".

The authors discuss three ways to relax a query:

- Rewriting a predicate into a more general predicate
- Rewriting a constant (term) into a more general constant
- Breaking a join dependency across literals in the query

Below we focus on the first two options: predicate generalization and term rewriting. These are realised in a uniform framework based on so-called *taxonomy clauses*. In F-X, these clauses would either be added within the broker environment in a separate module (see Figure 5.1 on page 5.1), or offered by an external agent, which the broker would then have to consult during the brokering process.

5.3.1 Predicate generalization

We apply the technique of Gaasterland et al. (1991) for relaxing predicates of competences within F-X. We first discuss the role of taxonomy clauses, then move on to reciprocal clauses and conclude the section with an adapted version of their relaxation algorithm.

Taxononomy clauses

In our trip booking example, one example of a predicate generalization would be the following (to keep our example readable, we abstract from preconditions and input/output constraints):

```
informTravel(D, A, LD, RD, Travelticket) :-
        refiner(Travelticket,Flightticket),
        informFlight(D, A, LD, RD, Flightticket).

informTravel(D, A, LD, RD, Travelticket) :-
        refiner(Travelticket,Trainticket),
        informTrain(D, A, LD, RD, Trainticket).
```

These taxonomy clauses describe what the authors call an "implicit" taxonomy. Here we have an implicit taxonomy with informTravel ← informFlight and informTravel ← informTrain. In general, a taxonomy clause contains the following atoms:

- The body atom in a taxonomy clause that would appear in the hierarchy is called a *key* atom. In our case this would be informFlight.
- The non-key predicates serve to relate variables appearing in the clause. Taxonomy clauses capture type relations over constants as well as over predicates. To rewrite a predicate, one must know how the parameters of predicates are related. These may differ in type, intent, order and number. In our setting, it seems to make sense to call these non-key predicates domain refiners. Domain refiners are adapters that map a domain to another domain, and here we map TravelTicket with the

Tickets for the other modes of travelling. Somewhat surprisingly, we thus model a domain refiner within a PSM refiner.[1]

Reciprocal clauses

For each taxonomy clause, a reciprocal "relax" clause is added automatically to the knowledge base, where:

- The head of a reciprocal clause contains the key atom of a taxonomy clause.
- The head atom in the taxonomy clause appears in the body of the reciprocal clause.
- The non-key predicates of the taxononomy clause are in the body of the reciprocal clause.

In our example, the reciprocal clauses look as follows:

```
relax(informFlight(D, A, LD, RD, FlightTicket)) :-
        refiner(TravelTicket,FlightTicket),
        informTravel(D, A, LD, RD, TravelTicket).

relax(informTrain(D, A, LD, RD, Trainticket)) :-
        refiner(Travelticket,Trainticket),
        informTravel(D, A, LD, RD, Travelticket).
```

Relaxation process

Drawing on the taxonomy and reciprocal clauses, a relaxation process can now infer the "more general" predicates / competences. A competence may be relaxed if it unifies with any *key* in any taxonomy clause.

Before relaxation, a query must be *variable substituted* so that the constants are moved out into separate atoms. This enables the handling of each kind of rewrite uniformly. Variable substitution replaces constants and repeated variables in a query with new unique variables. Then it equates the original constants and variables to the new variables.[2]

The relaxation process is closely intertwined with deduction. To perform deduction and relaxation together, (Gaasterland et al. 1991) define a meta-interpreter that takes as input a variable-substituted query and uses a knowledge base of rules, facts and taxonomy clauses to return first direct answers and then relaxed answers. It processes the query in two steps: finding the query's next relaxation, and then finding all answers for the relaxed query.

[1] The UPML documentation stays silent on the kind of relationships that are possible between various refiners.

[2] See the coming section on term rewriting for a brief example.

We use the following meta-interpreter to implement this overall process. We adapted this from (Gaasterland et al. 1991), removing support for processing a list of goals. `relax_solve` works on every competence description that is invoked when the brokering algorithm runs.

```
relax_solve(Q) :- clause(relax(Q), T), solve(T).
```

The `relax` step finds a new relaxed query by looking for a taxonomy clause that matches the input query. The `solve` step performs deduction. Note that we could also be interested in restraining the query, rather than relax it, in which case it would be helpful to mark those clauses with a different name, e.g. restrict(). It would be up to the meta-interpreter to process these clauses differently.

We did not implement our adapted version of the broker, nor did we experiment with the performance degradation these extra checks bring with them. We also did not implement an example of heuristics to order brokering results (these would have to be domain dependent since a general solution to this is not available).

The notion of user preferences/constraints would also make a welcome addition. User preferences are somewhat related to the notion of stored tasks, since typically such preferences would be kept over time. They can heavily influence the competences searched for. Gaasterland (1997) describes how to handle both user constraints and result ordering, based on the above framework. Given her results, we expect these extensions will not to be too hard to implement.

Representing relaxation clauses in OWL

Since taxonomy and reciprocal clauses rely on the notion of hierarchy, we could consider using OWL DL as a Web-enabled and standardized KR formalism to describe these clauses, and possibly share them.

However, since OWL DL is a Description Logic at heart, the above rules are very hard to represent. One issue at play here is that the arity of taxonomy clauses may be higher than 2, whereas DL concepts correspond to predicates with arity one (e.g. father(X)), and DL properties translate to predicates with arity two (e.g. fatherOf(X,Y)) (see (Grosof, Horrocks, Volz & Decker 2003) for details). Still, we could consider rearranging the predicate into a number of properties that relate to a central concept.

More seriously though, in DL one cannot express queries with arbitrary structure (from (Motik et al. 2003) and (Grosof et al. 2003)). They have the so-called tree-model property, which says that a DL class C has a model (or interpretation) iff C has a tree-shaped model, i.e. the interpretation of properties defines a tree-shaped directed graph.

This implies that all Description Logic concept descriptions can be represented as rules where predicates chained according to variables appearing

in them form a tree. Yet, this also means that a concept such as Oedipus-ComplexPerson (that is, a concept containing all people having the same mother and wife) cannot be expressed in DL. It can easily be expressed in a deductive database however:

```
OedipusComplexPerson(X) :-
       Person(X), hasMother(X, Y), hasWife(X, Y).
```

This rule cannot be represented as a tree: starting from the Person, one must traverse the hasMother link and the hasWife link, and should then arrive at the same object.

5.3.2 Rewriting terms

Other than relax the predicates of our F-X competences, we also want to be able to generalize terms. Gaasterland et al. (1991) give the following example of what they understand under term generalization.

```
cousin(jack, steve).
```

rewrites to

```
cousin(jack, X), male(X).
```

In this case, the constant `steve` can be replaced with a variable whose domain is restricted to `male`. As such, a constant is replaced with a variable whose domain includes constants related to the original somehow. If all we know about Steve is that he is male, we can loosen to consider other males if `steve` fails.

Term relaxation is again handled within the framework of taxonomy clauses. We do not repeat the above discussion here, but limit ourselves to an example of a term rewrite in our trip-booking example. Say that, if the departure place X has no airport, we should look what region this place is a part of, and check whether another city (NearbyCity) in that region does have an airport (called NearbyAirport). The concrete relaxation in this case might look as follows:

```
relax(hasAirport(X,Y)) :- partOf(X,Z), wholeOf(Z,NearbyCity),
                          hasAirport(NearbyCity,NearbyAirport)).
```

Flight booking services often lack such a feature in practice. For instance, Kelkoo.co.uk is a service that collects flight information from various airlines.[3] At Kelkoo, a user can check for flight information by entering a departure point. However, when this departure point is not an airport,

[3]Web site: `http://www.kelkoo.co.uk`

the system comes back to the user empty-handed. Since one may not know which airport is the closest by (for example being foreign abroad), with a relaxation clause there is room for improvement here.

Note that, for this to work, we expect there to be either a `partOf` clause in the broker's knowledge base, or that an ontology server publishes as a competence that it can determine whether a particular departure place is part of a bigger geographical entity (see the next section for more detail). This entity subsequently may well prove to have an airport at its disposal. So, the idea behind this is that when the inputted departure does not result in valid flight information, we look for a part-of property that covers a bigger entity.

We would thus be relying on taxonomic knowledge to still find us some answers, whereas before the query would simply fail. This brings us back to the idea of using ontologies to represent (part of) the information captured in the competence descriptions.

Earlier we remarked that representing rules is not a stronghold of DL ontologies. This still applies for rewriting terms: we could think of rules that DLs would not be able to represent. Nevertheless, DLs are very good for specifying definitions of domain concepts and properties, and to organize these in a clear way: using taxonomies. For this reason, in the following section we experiment in building and accessing an OWL domain ontology in support of our travel scenario.

5.4 Importing OWL ontologies in F-X

In the previous, we focussed our attention on a technique for more flexible competence matching. We now build a subset of a travel ontology in OWL, containing the basic vocabulary used in the competences and competence relationships introduced above. Using a translation mechanism, we are then able to access this OWL ontology from our Logic Programming setting.

We give an overview of our approach in Figure 5.1. From a workflow point of view, the work described in the current section happens (at least partly) before specifying the relaxation clauses. Of course, establishing a feedback loop between these two steps will be useful.

5.4.1 Motivation

As we discussed in Section 1.2, ontologies are touted as the key to reusable domain knowledge on the Web. It is expected that the OWL language(s) will serve as the central KR formalism here. With the move towards the Semantic Web, it will be useful for F-X to have access to this worldwide repository of knowledge. Hence, it makes sense to think about what (at the slogan level) we might call "importing the Semantic Web into F-X". In particular, we could define certain terms in the competences based on publicly

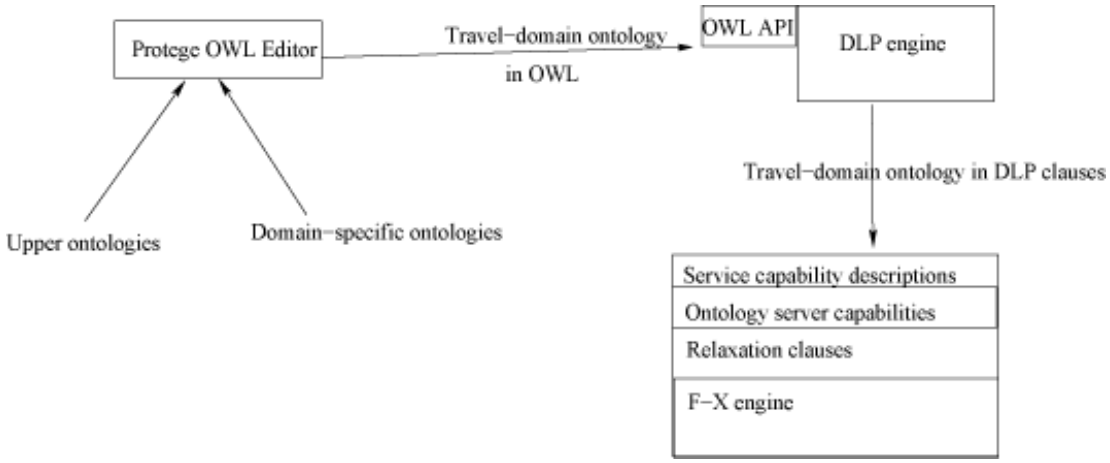

Figure 5.1: Overall architecture and workflow

available OWL ontologies. These would typically include the typing restrictions on variables (i.e. input and output roles). Examples of already freely available OWL ontologies include (part of) the MIT Process Handbook for business processes (Bernstein & Grosof Submitted) and the Gene Ontology for molecular biology.[4]

Importantly, using such an approach opens up the way to exploit the structure present in these ontologies. In particular, as mentioned at the end of the previous section, we would like to use this taxonomic knowledge to help steer the brokering process.

5.4.2 From Description Logics to Logic Programs

Ideally, we have a complete translation from Description Logic to Logic Programs. We already saw in the previous section that this will be impossible, however. Instead, recent work has explored the *intersection* between the two formalisms. Grosof et al. (2003) made an initial translation from (a limited subset of) DL to LP, resulting in the Desciption Logic Programs (DLP) formalism. This formalism was recently extended by Volz et al. (Submitted), making "a commonly used subset of OWL" available for use in Horn Logic. The translation starts from the correspondence of both OWL DL and Horn Logic with First-Order Logic. In Table 5.1 we show a few of the logical sentences that can be used in specifying our domain ontology while staying compatible with LP. DLP can also express minimum and maximum cardinality of one (i.e. functional restrictions), existential property restrictions, and nominals (classes which are defined by direct enumeration of its

[4]For the Gene Ontology, see `http://gong.man.ac.uk/` and `http://protege.stanford.edu/plugins/owl/owl-library/index.html`

instances) (see Volz et al. (Submitted) for details).

DL	FOL
$\top$	$true.$
$a : C$	$C(a)$
$\langle a, b \rangle : P$	$P(a, b)$
$C \sqsubseteq D$	$\forall x.C(x) \rightarrow D(x)$
$C_1 \sqcap \ldots \sqcap C_n$	$C_1(x) \wedge \ldots \wedge C_n(x)$
$\exists P.C$	$\exists y.(P(x, y) \wedge C(y))$
$\forall P.C$	$\forall y.(P(x, y) \rightarrow C(y))$

Table 5.1: Part of the DL FOL equivalence (from Volz et al. (Submitted)). C represent a class, P stands for a property, and a is an instance.

The reader will notice that negation and disjunction are not included in this overview. Volz et al. (Submitted) explain why a general DL-to-LP translation of negation and disjunction is not possible. Negation used in Description Logics is classical FOL negation, whereas logic programming usually provides negation-as-failure. LP systems typically assume something to be false if it cannot be proven to be true. This is related to the closed-world assumption: a logic program assumes that it knows all relevant facts and eveythings else is assumed to be false, whereas a Description Logic assumes it does not know all the facts, so it must be explicitly told which facts are false (open-world assumption). Translating disjunction is equally problematic in Logic Programs, given that negation and disjunction are closely related by De Morgan's Law: $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$.

Recent work by (Motik et al. 2003) tackles this limitation by moving the translation to a disjunctive databases setting. The authors are able to translate all of ALC, which is the foundation of almost all other more expressive description logics. Since here we are mostly interested in using OWL ontologies within the Prolog broker environment, we do not experiment with this translation. Current work by Boris Motik and Ulrike Sattler extends this translation to disjunctive databases by trying for $SHIQ$, an expressive DL very close to OWL DL.

5.4.3 Building a domain ontology

Our goal in building the ontology is not to come up with an extensive conceptualisation of all things concerning travel. Rather, the goal is to show in a simple way how we could support the trip booking scenario. Even building a small ontology takes a lot of effort. Here we report on our major findings during this process.

Ontology construction by reuse

In order not to reinvent the wheel, first we try to reuse as much of the existing body of knowledge as possible. There are two approaches to ontology modelling: bottom-up, starting with the most specific concepts, and top-down based on upper ontologies.

Bottom-up ontologies It is hard to find existing bottom-up ontologies for the travel domain. Both the OntoLingua repository and the Protégé ontology library have no detailed ontologies available.

Top-down ontologies One excellent candidate here is OpenCyc, "the world's largest and most complete general knowledge base and commonsense reasoning engine".[5] OpenCyc was recently released to the public free of charge. Before we can reuse some of the Cyc concepts, the first job is to understand the slightly different notation it uses for denoting classes (*collections*) and properties (*properties* if they are unary predicates, and *relationships* if binary predicates). IS-A relationships have similar counterparts (*gnls* for classes and *genlpreds* for properties).

Our next task is to learn to navigate the Cyc knowledge base in order to retrieve reusable concepts. Navigation in Cyc is quite straightforward, with many forward and backward references. Still, in places, Cyc can be rather unpredictable and incomplete. For example, it knows that Departure belongs in the collection LeavingAPlace, but does not know about the Arrival concept. Moreover, it has no knowledge of the concept of Hotel, Booking, or Flight.

While disappointing, completeness is not what commonsense knowledge bases are about. We do find Cyc to be very good at specifying abstract notions such as Place, Time or Movement. For instance, Cyc defines Place as follows:

> "A specialization of both SpatialThing-Localized and SomethingExisting. Each instance of Place is a spatial thing which has a relatively permanent location. Thus, in a given microtheory, each Place is stationary with respect to the frame of reference of that microtheory."

Modelling a return trip in Cyc In order to promote reusability across applications, we should try to think about a conceptualizing a trip in terms of Cyc's fundamental concepts. Searching and navigating the Cyc knowledge base teaches us that a trip can be modelled as a translation: a motion of a body in which every point of the body moves parallel to and the same

[5] Web site: `http://www.opencyc.org`

distance as every other point of the body. However, because we are talking about a round trip, things get quite subtle. A round trip really is a translation with no location change, because at the end of the trip we end up in the same position.

If we model this in OWL using the Protégé plugin[6], we obtain the deeply nested structure illustrated in Figure 5.2. Many properties have little direct bearing on our actual problem scenario. For instance, hasFromLocation and hasToLocation have to be modelled as being equal, since the movement starts and ends at the same place. We invite the reader to have a look at the other properties that are needed. The same wealth of concepts arises when trying to model concepts such as Place and Time.

It should be clear from the above that using all these concepts is really overkill. It adds dimensions to the scenario that have little relevance for our problem (e.g. from physics). To our knowledge, there are no workarounds for this in Cyc, e.g. in the form of different views on the ontology.

In our setting, we find Cyc to be useful mainly as a way to confront our own ideas about the problem domain. For instance, how does Airport relate to City or Place ? Cyc models Airport as FixedStructure under Place, as opposed to City, which is a GeographicalRegion under Place. For us this is a useful distinction to make. Unfortunately, there is no easy way to export these concepts to other environments. Recently, OpenCyc announced to support the DAML initiative, which could remedy this situation in the future.

A simple domain ontology in OWL

In what follows, we limit our focus to properly modelling one particular domain concept (departure place) in support of the travel booking scenario. From this, we show how an ontology can support relaxation based on its concept and property definitions. The example should be read in conjunction with the earlier section where we were looking for ways to support more flexible matching for competences. The relaxation clause presented there aims to find a nearby airport in case the current departure place has no airport.

In our ontology, the Place class has two subclasses, GeographicalRegion and FixedStructure. GeographicalRegion has Country, CountrySubsidiary (e.g. a county) and City as its subclasses. The complete place ontology is shown in Figure 5.3. Figure 5.4 shows a couple of instances. In what follows we explain how we went about modelling the part-whole relationships in our small ontology.

[6]Web site: `http://protege.stanford.edu/plugins/owl`

Figure 5.2: Modeling a return trip based on top-level Cyc concepts (rendered with the Protégé OWL plugin)

Figure 5.3: Modeling a small ontology for places

Modelling part-whole relationships As far as semantics are concerned, it would be wrong to model the relationship between City and CountrySubsidiary as a subclass relationship. Their relationship is one of part-whole, and subsumption could easily be abused to model this. From a formal point of view however, subsumption can only be used to indicate that all instances of the subclass are *necessarily* instances of the superclass. Every subclass that can exhibit the properties of the superclass *must* exhibit these properties (Guarino & Welty 2002). In our example, the essential properties of CountrySubsidiary (for instance, has a regional capital) do not apply to City, and hence City is not a subclass. Likewise, CountrySubsidiary is not a kind of Country.

In the context of OWL, the following approach to modelling part-whole relationships has been proposed (from a presentation by Guus Schreiber[7]):

- Create a subclass part-whole property as a subclass to the OWL property metaclass
- State for each property denoting a part-whole relation that it is an instance of the part-whole metaclass e.g. parts such as feet of a piece of antique furniture
- Attach the appropriate semantics to the part-whole metaclass

At the time of writing, the Protégé OWL editor did not allow to play with the OWL metaclasses. As a result, we modelled our part-whole relationships with partOf and wholeOf properties. An example of this can be seen on Figure 5.3, where CountrySubsidiary has a partOf property with a role (i.e. slot or property) value restriction of "Country". CountrySubsidiary also has a wholeOf property, which links it to City (see Figure 5.4). City and Country link back to CountrySubsidiary using the reverse properties.

5.4.4 Dealing with DLP domain ontologies in the broker

Armed with our place ontology, we can now think again about supporting the relaxation clause introduced in Section 5.3.2. To achieve this, as illustrated in Figure 5.1 on page 77, first we have to make the place ontology available in a format the broker can handle. Once this is done, we can ask the question how much of this domain ontology should actually get published in F-X.

Importing the place ontology into the broker Importing the place ontology into the broker consists of two steps. First, we translate the ontology from OWL to DLP. We start by exporting the ontology to an OWL/RDF format using Protégé (see Appendix C for a listing). Next, we make use of

[7]Web site: `http://homepages.cwi.nl/~lynda/WGI/jun2002/schreiber.ppt`

Figure 5.4: A few instances of CountrySubsidiary

the OWL API and DLP engine developed at the Universities of Manchester and Karlsruhe to generate the Horn clauses.[8]

As a result, the place ontology is available as a collection of Horn clauses (shown in Appendix D and detailed below), ready for use in the broker.

The role of domain knowledge in F-X The question then becomes : "How much of this domain information should be made available within the broker environment ?" We see two ways the broker might access domain knowledge:

Using T-Box and A-Box knowledge One alternative is to have all ontology information stored within the F-X broker as a separate module. In this case, one would import both the T-Box (concepts and properties) and the A-Box (instances) into the broker. The advantage of this approach is that less service calls will be made during service execution, since the broker would already have solved some part of the execution (i.e. the queries concerning domain knowledge). This approach raises questions of how to do subsumption within logic programs. At least for the DLP fragment of OWL, all subsumption can be reduced to Prolog query answering (see (Volz et al. Submitted) for a step-by-step guide).

Using only T-Box knowledge Another alternative chooses to keep the broker as slim as possible. Here, one would introduce an additional agent, the ontology agent *oa*, which is capable of telling other agents whether, for instance, X is a departure place, whether X hasAirport Y, or whether X is a partOf Z. *oa* would publish such capabilities with the broker. In this approach, all T-Box elements, i.e. the concepts and properties, get published as capabilities. Other agents can then use *oa* to check whether their particular input and output variables adhere to the ontology. The ontology agent would have (private) knowledge of the A-Box (i.e. instances), which would be used at execution time (i.e. *after* the broker returns its sequence of performatives).

The former approach seems in contradiction with the design rationale of F-X. One should remember that the broker is not intended to solve actual problems: rather it indicates which composition of services together will be able to solve such a problem. In what follows, we will stick to the first approach, and assume that an agent *oa* publishes the T-Box in the form of capabilities.

The place ontology in DLP Below we show the T-Box part of the place ontology returned by the DLP engine. We refer to Appendix D for

[8]Web site: `http://kaon.semanticweb.org/alphaworld/dlp/view`

a complete listing of the output (including the A-Box), and an explanation of some of the "extra's" that this tool outputs. As explained there, the "_ext"-predicates shown below have to do with the support for Datalog, and do not change the core behavior — they just add another step in the flow from concepts to instances.

```
airport(X) :- airportof(X,Y).
geographicalregion(Y) :- airportof(X,Y).
geographicalregion(X) :- wholeof(X,Y).
geographicalregion(X) :- partof(X,Y).
geographicalregion(X) :- hasairport(X,Y).
airport(Y) :- hasairport(X,Y).
countrysubsidiary(X) :- countrysubsidiary_ext(X).
partof(X,Y) :- partof_ext(X,Y).
airport(X) :- airport_ext(X).
airportof(X,Y) :- airportof_ext(X,Y).
country(X) :- country_ext(X).
city(X) :- city_ext(X).
hasairport(X,Y) :- hasairport_ext(X,Y).
wholeof(X,Y) :- wholeof_ext(X,Y).
geographicalregion(X) :- geographicalregion_ext(X).
geographicalregion(X) :- country(X).
fixedstructure(X) :- airport(X).
place(X) :- fixedstructure(X).
geographicalregion(X) :- city(X).
geographicalregion(X) :- countrysubsidiary(X).
place(X) :- geographicalregion(X).
countrysubsidiary(X1) :- wholeof(X,X1), country(X).
countrysubsidiary(X2) :- partof(X,X2), city(X).
city(X3) :- wholeof(X,X3), countrysubsidiary(X).
country(X4) :- partof(X,X4), countrysubsidiary(X).
```

To remind the reader, in the relaxation example of Section 5.3.2, we used the predicates hasAirport, partOf and wholeOf. We can see here that hasAirport(X,Y), partOf(X,Y) and wholeOf(X,Y) are all part of the DLP translation. In the next section, these will get published as competences of *oa*, in the same way as how we specified agent capabilities in Section 4.3.

5.5 Trip-booking scenario revisited

Now that all building blocks for our scenario are in place, we put them together and provide a summary of the different steps in the scenario. To this end, we refer back to the architectural overview on page 77. In a

nutshell, we have facilitated the use of domain ontologies in F-X, and used such knowledge to support more flexible competence matching.

In Section 4.3 we introduced a scenario in which a user could book a return trip via a travel agent. In this chapter, we started from the observation that there may exist more flexible ways to competence matching than string matching. To this end, we introduced the notion of relaxation. Relaxation covers both predicate relaxation and term rewriting within competence descriptions. We illustrated the concept in our trip booking scenario via a term rewriting example. The idea is that rewrite rules can be specified on the basis of an ontology by using some of the ontology concepts in these rules, and that this same ontology is later used again when the relaxing rules are fired during competence matching to test the rules. To this end, the ontology needs to be available to the broker, and we argued why having the T-Box published as competences of an ontology agent is the cleanest way to handle this.

5.6 Related and future work

We conclude this chapter by formulating what work is related to ours here, and by pointing out a few directions that may be worthwile to investigate in the future.

5.6.1 Related work

The work on query relaxation by (Gaasterland 1997) does not address the issue of importing ontologies. She presumes all information will be readily available in the knowledge base. The author also is not concerned with service description : her examples are about querying knowledge bases.

Recent work by (Grosof & Poon 2003) (Working paper dating July 12 2003) also explores using DLP in a service context. However, the authors are mainly interested in accessing simple concepts from the DLP ontology (which contains predicates from the MIT Process Handbook), and do not exploit the ontology's structure for complicated query answering. They do not consider service composition or relaxation.

The work by (McIlraith & Son 2002) is very close to ours, since they also use a Prolog environment (Golog to be exact) to do service composition. In addition, they employ user preferences to steer the process, which we, at the moment, do not. They do not consider the use of DLP to import service vocabulary, however, and they also do not take relaxation into account.

5.6.2 Future work

Future work would be useful in the following areas:

- It would be useful to extend the work of (Gaasterland 1997) with control knowledge that can actually handle "constraining relaxations" as part of the knowledge base. Her approach foresees that only user contraints would be limiting the set of possible solutions.

- As opposed to the current divide, it would be cleaner to have a unified reflection framework which captures both the internal relaxation steps (captured by relaxation clauses) that occur *during* the brokering cycle, and the revise and modify knowledge that is necessary to guide the broker *between* various broker cycles (as explained in Section 3.1.3).

- In addition, as we remarked multiple times throughout the text, there is a divide between brokering approaches that can handle tasks, but offer poor automated reasoning support, and the approaches that are able to automate brokering but have a simplistic notion of a task. It would be worthwile to further explore the spectrum between queries, user constraints and stored tasks, and the trade-offs making a choice brings with it.

- We think it would be worthwile to investigate if and how interoperability between services specified in different world views (i.e. service as action versus program) can be ensured. This is an issue that plagues DAML-S at the moment, but one which probably will not go away easily.

- Finally, we would like to explore further whether DLP ontologies are able to capture most of the logical constructs needed for specifying service ontology vocabulary. This is an empirical question, and would need to be answered through experience with real-world examples.

Chapter 6

Conclusions

This thesis started with an overview of Semantic Web Services, a new concept indicating the convergence of two major drivers on today's Internet: Web services and the Semantic Web. We cautioned for over-optimism and made the comparison with an earlier similar vision.

We then developed a comparative framework which contrasts two major approaches to brokering service capabilities, namely UPML and DAML-S. We identified the major strenghts and weaknesses in both approaches and we showed how they are similar and where they differ. UPML and DAML-S both use a great deal of language constructs. UPML is even more general than DAML-S: it puts no restrictions on the object language and explicitly disentangles the concepts of Task, Problem-Solving Method and Domain model using the notion of an Adapter. This is explained by UPML's background: it was conceived to support the construction of knowledge-based systems, whereas DAML-S is a more recent phenomenon inspired by modular Web services.

Next, we provided a comparison of how UPML and DAML-S relate to the F-X lightweight formal brokering system. F-X aims at fully automated brokering and makes a number of clear design choices because of this.

We also reviewed brokering techniques developed for problem-solving methods based on UPML, and made the link with Semantic Web Services for each technique. We contrasted these brokering techniques with the strategy used by F-X, and identified a number of synergy areas.

Based on this analysis of different approaches to capability brokering, we isolated one area to improve the F-X broker. A prototype was developed which extends the F-X brokering system with a more flexible way to do competence matching. To this end, we combined work from deductive databases and the Semantic Web area. In particular, we applied work on relaxation to specify generalization relationships between competence descriptions.

We imported a OWL ontology into the F-X brokering environment using a recent translation mechanism (Description Logic Programs), which is

based on the intersection of Description Logics and Horn Logic. In doing so, we introduced a notion of hierarchy in this otherwise flat Logic Programming environment. The ontology's vocabulary is used to specify the competence descriptions and competence relationships, as well as to answer ontological queries during a later phase. In a nutshell, we have facilitated the use of domain ontologies in F-X, and used such knowledge to support more flexible competence matching.

Among the areas we identified to do further work in, we think the need for interoperability between action-based formalisms (which includes some of the DAML-S descriptions) and service capabilities based upon algebraic specifications (such as the ones specified in F-X) could become pressing in the future.

Further empirical experimentation would be required to evaluate whether the DLP formalism is sufficient for representing the majority of (domain vocabulary used in) services.

Appendix A

F-X brokering algorithm in Prolog

See Section 4.2.2, (Robertson et al. 2000) or (Robertson 2001) for details.

```
brokerable(Q, c(S,Q)) :-
    capability(S, Q).
brokerable(Q, c(S,dq(Q,QC))) :-
    capability(S, (Q :- C)),
    brokerable(C, QC).
brokerable(Q, c(S1,pdq(Q,QC,QP))) :-
    p_capability(S1, (Q :- C), P),
    brokerable(C, QC),
    ext_brokerable(P, S1, QP).
brokerable((Q1,Q2), c(conj,co(CQ1,CQ2))) :-
    brokerable(Q1, CQ1),
    brokerable(Q2, CQ2).
brokerable(Q2, c(S2,cn(Q2,Constraint,c(S1,BQ)))) :-
    corresponds(S1, Q1, S2, Q2, Constraint),
    brokerable(Q1, c(S1,BQ)).

ext_brokerable(Q, Sn, c(S,Q)) :-
    capability(S, Q),
    \+ S = Sn.
ext_brokerable(Q, Sn, c(S,dq(Q,QC))) :-
    capability(S, (Q :- C)),
    \+ S = Sn,
    brokerable(C, QC).
ext_brokerable(Q, Sn, c(S1,pdq(Q,QC,QP))) :-
    p_capability(S1, (Q :- C), P),
    \+ S1 = Sn,
```

```
    brokerable(C, QC),
    ext_brokerable(P, S1, QP).
ext_brokerable((Q1,Q2), Sn, c(conj,co(CQ1,CQ2))) :-
    ext_brokerable(Q1, Sn, CQ1),
    ext_brokerable(Q2, Sn, CQ2).
ext_brokerable(Q2, Sn, c(Sn,cn(Q2,Constraint,c(S1,BQ)))) :-
    corresponds(S1, Q1, Sn, Q2, Constraint),
    brokerable(Q1, c(S1,BQ)).

% The following code unpacks the above structure
% into performatives using Definite Clause Grammar.

assemble(c(S,dq(Q,QC))) --> !,
    {dependent_queries(QC,DQ)},
    assemble(QC),
    [ask(S,(Q :- DQ))].
assemble(c(S1,pdq(Q,QC,QP))) --> !,
    {dependent_queries(QP,DQ)},
    assemble(QC),
    assemble(QP),
    [ask(S1,(Q :- DQ))].
assemble(c(conj,co(CQ1,CQ2))) --> !,
     assemble(CQ1),
     assemble(CQ2).
assemble(c(S,cn(Q,C,CQ))) --> !,
     assemble(CQ),
    [test(C), tell(S,Q)].
assemble(c(S,Q)) -->
    [ask(S,Q)].

dependent_queries(c(_,cn(Q,_,c(_,_))), Q) :- !.
dependent_queries(c(conj,co(CQ1,CQ2)), (DQ1,DQ2)) :- !,
    dependent_queries(CQ1, DQ1),
    dependent_queries(CQ2, DQ2).
dependent_queries(c(_,dq(Q,_)), Q) :- !.
dependent_queries(c(_,Q), Q).
```

Appendix B

Prolog code for trip-booking scenario

See Section 4.3.2 for more information.

```
capability(aa, (informFlight(D, A, LD, RD, F) :- departure(D),
   arrival(A), leavingDate(LD), returnDate(RD), flight(F))).
capability(aa, (bookFlight(F) :- flight(F), payFlight(F))).

capability(aa, departure(D)).
capability(aa, arrival(A)).
capability(aa, leavingDate(LD)).
capability(aa, returnDate(RD)).
capability(aa, flight(F)).
capability(aa, payFlight(F)).

capability(ha, (informHotel(L, ID, OD, HS) :- inDate(ID),
   outDate(OD), location(L), hotelStay(HS))).
capability(ha, (bookHotel(HS) :- hotelStay(HS), payHotel(HS))).

capability(ha, inDate(ID)).
capability(ha, outDate(OD)).
capability(ha, location(L)).
capability(ha, hotelStay(HS)).
capability(ha, payHotel(HS)).

p_capability(ta, (informTrip(D, A, LD, RD, F, HS) :- departure(D)),
   (informFlight(D, A, LD, RD, F), informHotel(A, LD, RD, HS))).
p_capability(ta, (bookTrip(D, A, LD, RD, F, HS) :- departure(D)),
   (informFlight(D, A, LD, RD, F), informHotel(A, LD, RD, HS),
      bookFlight(F), payFlight(F), bookHotel(HS), payHotel(HS))).
```

Appendix C

Place ontology as OWL/RDF

This is the Place ontology as exported by Protégé. For clarity, we provide the abbreviated version of the OWL/RDF format the tool uses. The actual ontology fed into the OWL API contains the full syntax.

```
<rdf:RDF
    xmlns:j.0="http://owl.protege.stanford.edu#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource=
            "http://protege.stanford.edu/plugins/owl/protege"/>
  </owl:Ontology>
  <owl:Ontology/>
  <owl:Class rdf:ID="Airport">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#FixedStructure"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Country">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#wholeOf"/>
        </owl:onProperty>
        <owl:allValuesFrom>
          <owl:Class rdf:about="#CountrySubsidiary"/>
        </owl:allValuesFrom>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
```

```
    <owl:Class rdf:about="#GeographicalRegion"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Place">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:resource=
              "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"
         rdf:type="http://www.w3.org/1999/02/22-rdf-syntax-ns#List"/>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
<owl:Class rdf:ID="GeographicalRegion">
  <rdfs:subClassOf rdf:resource="#Place"/>
</owl:Class>
<owl:Class rdf:ID="FixedStructure">
  <rdfs:subClassOf rdf:resource="#Place"/>
</owl:Class>
<owl:Class rdf:ID="City">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:FunctionalProperty rdf:about="#partOf"/>
      </owl:onProperty>
      <owl:allValuesFrom>
        <owl:Class rdf:about="#CountrySubsidiary"/>
      </owl:allValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#GeographicalRegion"/>
</owl:Class>
<owl:Class rdf:ID="CountrySubsidiary">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#wholeOf"/>
      </owl:onProperty>
      <owl:allValuesFrom rdf:resource="#City"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:FunctionalProperty rdf:about="#partOf"/>
```

```
      </owl:onProperty>
      <owl:allValuesFrom rdf:resource="#Country"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#GeographicalRegion"/>
</owl:Class>
<owl:ObjectProperty rdf:ID="wholeOf">
  <rdfs:domain rdf:resource="#GeographicalRegion"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="AirportOf">
  <rdfs:domain rdf:resource="#Airport"/>
  <rdfs:range rdf:resource="#GeographicalRegion"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasAirport">
  <rdfs:domain rdf:resource="#GeographicalRegion"/>
  <rdfs:range rdf:resource="#Airport"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="hasName"
   rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
  <rdfs:range rdf:resource=
                  "http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Place"/>
</owl:DatatypeProperty>
<owl:FunctionalProperty rdf:ID="partOf"
   rdf:type="http://www.w3.org/2002/07/owl#ObjectProperty">
  <rdfs:domain rdf:resource="#GeographicalRegion"/>
</owl:FunctionalProperty>
<j.0:Country rdf:ID="Instance_14"
   j.0:hasName="Germany">
  <j.0:wholeOf>
    <j.0:CountrySubsidiary rdf:ID="Instance_6"
       j.0:hasName="Badem-Wurttemberg">
      <j.0:partOf rdf:resource="#Instance_14"/>
      <j.0:wholeOf>
        <j.0:City rdf:ID="Instance_13"
           j.0:hasName="Karlsruhe">
          <j.0:partOf rdf:resource="#Instance_6"/>
        </j.0:City>
      </j.0:wholeOf>
      <j.0:wholeOf>
        <j.0:City rdf:ID="Instance_7"
           j.0:hasName="Baden-Baden">
          <j.0:partOf rdf:resource="#Instance_6"/>
        </j.0:City>
```

```
      </j.0:wholeOf>
      <j.0:wholeOf>
        <j.0:City rdf:ID="Instance_8"
           j.0:hasName="Mannheim">
          <j.0:partOf rdf:resource="#Instance_6"/>
          <j.0:hasAirport>
            <j.0:Airport rdf:ID="Instance_12"
               j.0:hasName="Frankfurt">
              <j.0:AirportOf rdf:resource="#Instance_14"/>
              <j.0:AirportOf>
                <j.0:GeographicalRegion rdf:ID="Instance_11"
                   j.0:hasName="Badem-Wurttemberg">
                  <j.0:hasAirport rdf:resource="#Instance_12"/>
                </j.0:GeographicalRegion>
              </j.0:AirportOf>
            </j.0:Airport>
          </j.0:hasAirport>
        </j.0:City>
      </j.0:wholeOf>
      <j.0:hasAirport rdf:resource="#Instance_12"/>
      <j.0:hasAirport>
        <j.0:Airport rdf:ID="Instance_5"
           j.0:hasName="Koln">
          <j.0:AirportOf rdf:resource="#Instance_14"/>
          <j.0:AirportOf rdf:resource="#Instance_6"/>
        </j.0:Airport>
      </j.0:hasAirport>
    </j.0:CountrySubsidiary>
  </j.0:wholeOf>
</j.0:Country>
<j.0:Airport rdf:ID="Instance_20"
   j.0:hasName="Edinburgh">
  <j.0:AirportOf>
    <j.0:Country rdf:ID="Instance_10">
      <j.0:hasName>United Kingdom</j.0:hasName>
    </j.0:Country>
  </j.0:AirportOf>
  <j.0:AirportOf>
    <j.0:CountrySubsidiary rdf:ID="Instance_15"
       j.0:hasName="Scotland">
      <j.0:partOf rdf:resource="#Instance_10"/>
    </j.0:CountrySubsidiary>
  </j.0:AirportOf>
</j.0:Airport>
```

```
<rdf:Description>
  <rdf:rest rdf:parseType="Resource">
    <rdf:rest rdf:parseType="Resource">
      <rdf:rest rdf:parseType="Resource">
        <rdf:rest rdf:parseType="Resource">
          <rdf:rest rdf:resource=
              "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
        </rdf:rest>
      </rdf:rest>
    </rdf:rest>
  </rdf:rest>
</rdf:Description>
<j.0:CountrySubsidiary rdf:ID="Instance_16"
   j.0:hasName="Bayern">
  <j.0:partOf rdf:resource="#Instance_14"/>
</j.0:CountrySubsidiary>
<rdf:Description>
  <rdf:rest rdf:resource=
               "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
</rdf:Description>
<j.0:Country rdf:ID="Instance_9"
   j.0:hasName="Belgium"/>
<rdf:Description>
  <rdf:rest rdf:parseType="Resource">
    <rdf:rest rdf:resource=
              "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
  </rdf:rest>
</rdf:Description>
<rdf:Description>
  <rdf:rest rdf:parseType="Resource">
    <rdf:rest rdf:parseType="Resource">
      <rdf:rest rdf:parseType="Resource">
        <rdf:rest rdf:resource=
              "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      </rdf:rest>
    </rdf:rest>
  </rdf:rest>
</rdf:Description>
<j.0:City rdf:ID="Instance_22"
   j.0:hasName="Koln">
  <j.0:partOf rdf:resource="#Instance_6"/>
  <j.0:hasAirport rdf:resource="#Instance_5"/>
</j.0:City>
<rdf:Description>
```

```
    <rdf:rest rdf:parseType="Resource">
      <rdf:rest rdf:resource=
                "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
    </rdf:rest>
  </rdf:Description>
  <j.0:CountrySubsidiary rdf:ID="Instance_19"
     j.0:hasName="England">
    <j.0:partOf rdf:resource="#Instance_10"/>
  </j.0:CountrySubsidiary>
  <j.0:CountrySubsidiary rdf:ID="Instance_18"
     j.0:hasName="Wales">
    <j.0:partOf rdf:resource="#Instance_10"/>
  </j.0:CountrySubsidiary>
  <j.0:Country rdf:ID="Instance_17"
     j.0:hasName="France"/>
  <j.0:City rdf:ID="Instance_21"
     j.0:hasName="Frankfurt">
    <j.0:partOf rdf:resource="#Instance_6"/>
    <j.0:hasAirport rdf:resource="#Instance_12"/>
  </j.0:City>
</rdf:RDF>
```

Appendix D

Place ontology as Horn clauses

This appendix provides the exact output as generated by using the OWL API and DLP-engine. It should be stressed that the DLP tool is still in a prototype phase, and hence the output has not been tailored to each particular output format. Currently, it caters for translation to RuleML, XSB and disjunctive databases. The output should be read with the following in mind:

- Protégé uses an internal naming scheme for denoting instances (e.g. instance_16 stands for Bayern). Currently, because of this naming (datatype) information gets lost in the translation.
- The table-clauses are due to the support for XSB. It can be safely disregarded for our purposes.
- The hu() clauses give the Herbrand Universe, which we do not exploit in our brokering scenario, but which can be useful when working with disjunctive databases to handle negation.
- The DLP engine is designed for use in Datalog, which explains why concepts and properties are extended with an "_ext" version. This has little implications for our scenario in Prolog. Datalog relies on the distinction between EDB (Extensional DataBase) and IDB (Intensional Data Base). The extra predicates are used to guide bottom-up computation.

```
:- table airport/1.
:- table geographicalregion/1.
:- table countrysubsidiary/1.
:- table partof/2.
```

```
:- table airportof/2.
:- table country/1.
:- table city/1.
:- table hasairport/2.
:- table wholeof/2.
:- table fixedstructure/1.
:- table place/1.

airport(X) :- airportof(X,Y).
geographicalregion(Y) :- airportof(X,Y).
geographicalregion(X) :- wholeof(X,Y).
geographicalregion(X) :- partof(X,Y).
geographicalregion(X) :- hasairport(X,Y).
airport(Y) :- hasairport(X,Y).
countrysubsidiary(X) :- countrysubsidiary_ext(X).
partof(X,Y) :- partof_ext(X,Y).
airport(X) :- airport_ext(X).
airportof(X,Y) :- airportof_ext(X,Y).
country(X) :- country_ext(X).
city(X) :- city_ext(X).
hasairport(X,Y) :- hasairport_ext(X,Y).
wholeof(X,Y) :- wholeof_ext(X,Y).
geographicalregion(X) :- geographicalregion_ext(X).
geographicalregion(X) :- country(X).
fixedstructure(X) :- airport(X).
place(X) :- fixedstructure(X).
geographicalregion(X) :- city(X).
geographicalregion(X) :- countrysubsidiary(X).
place(X) :- geographicalregion(X).
countrysubsidiary(X1) :- wholeof(X,X1), country(X).
countrysubsidiary(X2) :- partof(X,X2), city(X).
city(X3) :- wholeof(X,X3), countrysubsidiary(X).
country(X4) :- partof(X,X4), countrysubsidiary(X).

airport_ext(instance_20).
airport_ext(instance_5).
airport_ext(instance_12).

partof_ext(instance_18, instance_10).
partof_ext(instance_13, instance_6).
partof_ext(instance_7, instance_6).
partof_ext(instance_19, instance_10).
partof_ext(instance_8, instance_6).
partof_ext(instance_6, instance_14).
```

```
partof_ext(instance_16, instance_14).
partof_ext(instance_15, instance_10).
partof_ext(instance_21, instance_6).
partof_ext(instance_22, instance_6).

hasairport_ext(instance_21, instance_12).
hasairport_ext(instance_22, instance_5).
hasairport_ext(instance_11, instance_12).
hasairport_ext(instance_6, instance_12).
hasairport_ext(instance_6, instance_5).
hasairport_ext(instance_8, instance_12).

countrysubsidiary_ext(instance_18).
countrysubsidiary_ext(instance_19).
countrysubsidiary_ext(instance_6).
countrysubsidiary_ext(instance_15).
countrysubsidiary_ext(instance_16).

city_ext(instance_21).
city_ext(instance_22).
city_ext(instance_13).
city_ext(instance_7).
city_ext(instance_8).

geographicalregion_ext(instance_11).

country_ext(instance_17).
country_ext(instance_10).
country_ext(instance_14).
country_ext(instance_9).

hu(instance_20).
hu(instance_21).
hu(instance_10).
hu(instance_22).
hu(instance_11).
hu(instance_12).
hu(instance_5).
hu(instance_6).
hu(instance_13).
hu(instance_14).
hu(instance_7).
hu(instance_15).
hu(instance_8).
```

```
hu(instance_9).
hu(instance_16).
hu(instance_17).
hu(instance_18).
hu(instance_19).

airportof_ext(instance_12, instance_11).
airportof_ext(instance_5, instance_14).
airportof_ext(instance_12, instance_14).
airportof_ext(instance_20, instance_10).
airportof_ext(instance_20, instance_15).
airportof_ext(instance_5, instance_6).

wholeof_ext(instance_6, instance_13).
wholeof_ext(instance_14, instance_6).
wholeof_ext(instance_6, instance_7).
wholeof_ext(instance_6, instance_8).
```

Bibliography

Aitken, S. & Tate, A. (2003), Applying DAML languages and ontologies in CoSAR-TS, Technical report, University of Edinburgh.

Ankolekar, A., Huch, F. & Sycara, K. (2002), Concurrent execution semantics for DAML-s with subtypes, *in* 'The First International Semantic Web Conference (ISWC)', Sardinia, Italy.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D. & Schneider, P. P. (2003), *The Description Logic Handbook : Theory, Implementation and Applications*, Cambridge University Press.

Berners-Lee, T. (1999), *Weaving the Web : The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*, Harper, San Francisco.

Bernstein, A. & Grosof, B. N. (Submitted), Beyond monotonic inheritance: Towards semantic web process ontologies.

Borst, W. (1997), Construction of Engineering ontologies, PhD thesis, University of Twente, Enschede.

Box, D., Skonnard, A. & Lam, J. (2000), *Essential XML. Beyond Markup*, Addison-Wesley.

Bray, T., Paoli, T. & Sperberg-McQueen, J. (1998), 'Extensible markup language (xml) 1.0. w3c recommendation', W3C Web site.

Brickley, D. & Guha, R. (2000), Resource Description Framework (RDF) Schema Specification. W3C Candidate recommendation.

Broekstra, J., Fluit, C. & van Harmelen, F. (2000), 'The state of the art on representation and query languages for semistructured data', On-To-Knowledge Project Deliverable 8.

Bundy, A. & Smaill, A. (2002), 'The application of deductive synthesis techniques to the rapid assembly and re-assembly of grid applications', eScience and the Grid Grant proposal, Case for Support.

Bundy, A., Smaill, A. & Yang, B. (2003), The application of deductive synthesis techniques to the rapid (re-)assembly of grid applications, Technical report, University of Edinburgh.

Chandrasekaran, B. (1990), 'Design problem solving: A task analysis', *AI Magazine* **11**, 59–71.

Chinnici, R., Gudgin, M., Moreau, J.-J. & Weerawarana, S. (2003), 'Web services description language (wsdl) version 1.2 w3c working draft', W3C Web site.

Crubézy, M., Motta, E., Lu, W. & Musen, M. (2003), 'Configuring online problem-solving resources with the internet reasoning service', *IEEE Intelligent Systems* **18(2)**, 34–42.

Crubézy, M. & Musen, M. (2003), *Handbook on Ontologies in Information Systems*, Springer, chapter Ontologies in Support of Problem Solving.

DAML-S Consortium (2003), 'DAML-S and related technologies', DAML Web site `http://www.daml.org/services`.

DAML+OIL Joint Committee (2001), 'Daml+oil', DAML Web site `http://www.daml.org/2001/03/daml+oil-index.html`.

Davis, R., Shrobe, H. & Szolovits, P. (1993), 'What is a knowledge presentation?', *A.I. Magazine* **14**(1), 17–33.

Decker, K., Sycara, K. & Williamson, M. (1997), Middle-agents for the internet, *in* 'Proceedings of the 15th International Joint Conference on Artificial Intelligence', Nagoya, Japan.

Fensel, D. (2000), *Problem-Solving Methods*, LNAI 1791Springer-Verlag, Berlin Heidelberg.

Fensel, D. (2002), Language standardization for the semantic web: The long way from oil to owl, *in* J. Plaice, P. G. Kropf, P. Schulthess & J. Slonim, eds, 'Distributed Communities on the Web (DCW), 4th International Workshop', Vol. Lecture Notes in Computer Science 2468, Springer, Sydney, Australia.

Fensel, D., Benjamins, V., Decker, S., Gaspari, M., Groenboom, R., Grosso, W., Musen, M., Motta, E., Plaza, E., Schreiber, G., Studer, S. & Wielinga, B. (1999), The component model of upml in a nutshell, *in* 'Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)', San Antonio, Texas.

Fensel, D. & Bussler, C. (2002), 'The web service modeling framework WSMF', *Electronic Commerce Research and Applications* **1**(2).

Fensel, D., Crubézy, M., van Harmelen, F. & Horrocks, I. (2000), OIL and UPML: a unifying framework for the knowledge web.

Fensel, D., Motta, E., Benjamins, R., Crubézy, M., Decker, S., Gaspari, M., Groenboom, R., Grosso, W., van Harmelen, F., Musen, M., Plaza, E., Schreiber, G., Studer, R. & Wielinga, B. (2003), 'The unified problem-solving method development language', *Knowledge and Information Systems* **5**(1).

Fensel, D., Motta, E., Benjamins, R. V., Decker, S., Gaspari, M., Groenboom, R., Grosso, W., Musen, M., Plaza, E., Schreiber, G., Studer, R. & Wielinga, B. (1999), The unified problem-solving method development language UPML, Technical report, IBROW Deliverable 1.1.

Fensel, D., Schönegge, A., Groenboom, R. & Wielinga, B. (1996), Specification and verification of knowledge-based systems, *in* 'Proceedings of the ECAI'96 Workshop on Validation, Verification and Refinement of Knowledge-Based Systems at the 12th European Conference on Artificial Intelligence (ECAI-96)', Budapest.

Fensel, D. & van Harmelen, F. (2001), 'Oil: an ontology infrastructure for the semantic web', *IEEE Intelligent Systems* .

Fensel, D., van Harmelen, F. & Horrocks, I. (1999), OIL: A standard proposal for the semantic web, Technical Report 0, OnToKnowledge EU IST Project.

Gaasterland, T. (1997), 'Cooperative answering through controlled query relaxation', *IEEE Expert* pp. 48–58.

Gaasterland, T., Godfrey, P. & Minker, J. (1991), Relaxation as a platform for cooperative answering, Technical Report CS-TR-2818, University of Maryland, College Park.

Garlan, D. (2001), *Software Architecture. Encyclopedia of Software Engineering*, John Wiley and Sons.

Grosof, B. N., Horrocks, I., Volz, R. & Decker, S. (2003), Description logic programs: Combining logic programs with description logic, *in* 'WWW 2003', Budapest, Hungary.

Grosof, B. & Poon, T. (2003), SweetDeal : Representing agent contracts with exceptions using XML rules, ontologies and process descriptions.

Gruber, T. (1993), Toward principles for the design of ontologies used for knowledge sharing, *in* Guarino & Poli, eds, 'Formal Ontology in Conceptual Analysis and Knowledge Representation', Kluwer.

Guarino, N. & Welty, C. (2002), 'Evaluating ontological decisions with ONTOCLEAN', *Communications of the ACM* **45(2)**, 61–65.

Horrocks, I. (2002), 'Daml+oil: a description logic for the semantic web', *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* **25**(1), 4–9.

Horrocks, I. & Tessaris, S. (2002), Querying the semantic web: A formal approach, *in* I. Horrocks & J. Hendler, eds, 'Proceedings of the First International Semantic Web Conference', Springer, Sardinia, Italy.

IBM Web Services Architecture Team (2000), 'Web services architecture overview', IBM Web site.

Klein, M., Fensel, D., van Harmelen, F. & Horrocks, I. (2000), The relation between ontologies and schema-languages: Translating OIL-specifications in XML-schema, *in* V. R. Benjamins, A. Gomez-Perez & N. Guarino, eds, 'Proceedings of the Workshop on Applications of Ontologies and Problem-solving Methods, 14th European Conference on Artificial Intelligence (ECAI 2000)', Berlin, Germany.

Klein, M., Fensel, D., van Harmelen, F. & Horrocks, I. (2001), 'The relation between ontologies and xml schemas', *Linköping Electronic Articles in Computer and Information Science* **6**(4).

Lasilla, O. & Swick, R. (1999), 'Resource description framework (rdf) model and syntax specification. w3c recommendation', W3C Web site.

Lemahieu, W. (2001), Web service description, advertising and discovery: Wsdl and beyond, *in* J. Vandenbulcke & M. Snoeck, eds, 'New directions in Software Engineering', Leuven University Press, pp. 135–152.

Leymann, F. (2001), 'Web services flow language (wsfl 1.0)', IBM Software Group Web site.

Li, L. & Horrocks, I. (2003), A software framework for matchmaking based on semantic web technology, *in* 'In Proc. Of the Twelfth International World Wide Web Conference (WWW 2003)'.

Lomuscio, A. R., Wooldridge, M. & Jennings, N. R. (2001), A classification scheme for negotiation in electronic commerce, *in* F. Dignum & C. Sierra, eds, 'Agent-Mediated Electronic Commerce: A European Perspective', Springer Verlag, pp. 19–33.

Magkanaraki, A., Karvounarakis, G., Anh, T. T., Christophides, V. & Plexousakis, D. (2002), Ontology storage and querying, 2002 308, Research and Technology Hellas, Institute of Computer Science, Information Systems Laboratory.

Mandell, D. & McIlraith, S. (2003), A bottom-up approach to automating web service discovery, customization, and semantic translation, *in* 'WWW 2003 Workshop on E-Services and the Semantic Web ESSW' 03'.

McCarthy, J. & Hayes, P. (1969), Some philosophical problems from the standpoint of artificial intelligence, *in* B. Metlzer & D. Michie, eds, 'Machine Intelligence', Edinburgh University Press, pp. 463–502.

McGuinness, D. L. & van Harmelen, F. (2003), 'Web ontology language (owl): Overview. w3c working draft', W3C Web site.

McIlraith, S. A. & Martin, D. L. (2003), 'Bringing semantics to web services', *IEEE Intelligent Systems* .

McIlraith, S. & Son, T. (2002), Adapting golog for composition of semantic web services, *in* 'Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR)', Toulouse, France.

Mitra, N. (2002), 'Simple object access protocol (soap) version 1.2 part 0: Primer. w3c candidate recommendation', W3C Web site.

Motik, B., Maedche, A. & Volz, R. (2003), Optimizing Query Answering in Description Logics using Disjunctive Deductive Databases, *in* '10th International Workshop on Knowledge Representation meets Databases (KRDB-2003)', Hamburg, Germany.

Motta, E., Domingue, J., Cabral, L. & Gaspari, M. (2003), IRS-II: A framework and infrastructure for semantic web services, *in* 'Proceedings of the 2nd International Semantic Web Conference (ISWC 2003)'.

Motta, E. & Lu, W. (2000), A library of components for classification problem solving, *in* 'PKAW 2000: The 2000 Pacific Rim Knowledge Acquisition Workshop', Sydney, Australia.

Musen, M. (2000), Ontology-oriented design and programming, *in* J. Cuena, Y. Demazeau, A. Garcia & J. Treur, eds, 'Knowledge Engineering and Agent Technology', IOS Press.

Narayanan, S. & McIlraith, S. (2002), Verification and automated composition of web services, *in* 'Eleventh International World Wide Web Conference (WWW2002)', pp. 77–88.

Narayanan, S. & McIlraith, S. (To appear), 'Analysis and simulation of web services', *Computer Networks* .

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R. et al. (1991), 'Enabling technology for knowledge sharing', *AI Magazine* **12(3)**.

Omelayenko, B., Crubézy, M., Fensel, D., Benjamins, R. V., Wielinga, B., Motta, E., Musen, M. & Ding, Y. (2003), *UPML: The Language and Tool Support for Making the Semantic Web Alive*, The MIT Press, Cambridge, MA, pp. 141–170.

Paolucci, M., Kawamura, T., Payne, T. & Sycara, K. (2002), Importing the semantic web in UDDI, *in* 'Proceedings Web Services, E-Business and Semantic Web Workshop, CAiSE 2002'.

Park, J., Gennari, J. & Musen, M. (1997), Mappings for reuse in knowledge-based systems, Technical Report 97-0697, Stanford Medical Informatics.

Patel-Schneider, P. F. & Horrocks, I. (2003), 'Web ontology language (owl) abstract syntax and semantics. w3c working draft', W3C Web site.

Poole, D., Mackworth, A. & Goebel, R. (1998), *Computational Intelligence: A Logical Approach*, Oxford University Press.

Potter, S. (2003*a*), Broker description document, Technical report, University of Edinburgh.

Potter, S. (2003*b*), 'Knowledge brokering in AKT. Presentation for the PEA-POD workshop on synthesis'.

Ratnakar, V. & Gil, Y. (2002), A comparison of (semantic) markup languages, *in* 'Proceedings of the 15th International FLAIRS Conference, Special Track on Semantic Web', Pensacola, FL.

Robertson, D. (2001), 'F-X: A formal knowledge management system', (unpublished).

Robertson, D. S., Silva, F. S. C. D., Vasconcelos, W. W. & de Melo, A. C. V. (2000), A lightweight capability communication mechanism, *in* R. Loganantharaj & G. Palm, eds, 'Intelligent Problem Solving, Methodologies and Approaches, 13th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE', Springer, New Orleans, Louisiana, USA, pp. 660–670.

Sabou, M., Richards, D. & Splunter, S. V. (2003), An experience report on using DAML-s, *in* 'Workshop on E-Services and the Semantic Web (ESSW '03) at theTwelfth International World Wide Web Conference'.

Schlenoff, C., Gruninger, M., Tissot, F., Valois, J., Lubell, J. & Lee, J. (2000), 'The process specification language (PSL): Overview and version 1.0 specification', NISTIR 6459, National Institute of Standards and Technology, Gaithersburg, MD.

Shadbolt, N. (2003), 'Grandly challenged', *IEEE Intelligent Systems* .

Stefik, M. (1995), *Introduction to Knowledge Systems*, Morgan Kaufmann, San Francisco, California.

Studer, R., Benjamins, R. & Fensel, D. (1998), 'Knowledge engineering, principles and methods', *Data and Knowledge Engineering* **25**, 161–197.

Sycara, K., Widoff, S., Klusch, M. & Lu, J. (2002), 'Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace', *Autonomous Agents and Multi-Agent Systems* **5**(2).

ten Teije, A. & van Harmelen, F. (1996), 'Using reflection techniques for flexible problem solving (with examples from diagnosis)', *Future Generation Computer Systems, Special issue : Reflection and Meta-level AI Architectures* **12**, 217–234.

ten Teije, A. & van Harmelen, F. (2003), 'Task and method adaptation', IBROW Project Deliverable 12a.

ten Teije, A., van Harmelen, F., Schreiber, A. T. & Wielinga, B. J. (1998), 'Construction of problem-solving methods as parametric design', *International Journal of Human-Computer Studies, Special issue on problem-solving methods* **49**(4).

Thompson, H. (2002), 'Web services and the semantic web: Separating hype from reality', Global Grid Forum presentation, Edinburgh.

Thompson, H., Beech, D., Maloney, M. & Mendelsohn, N. (2001), 'Xml schema part 1: Structures. w3c recommendation', W3C Web site.

van Harmelen, F. (2002), 'How the semantic web will change kr: challenges and opportunities for a new research agenda', *The Knowledge Engineering Review* **17**(1).

Volz, R., Motik, B., Horrocks, I. & Grosof, B. (Submitted), Description Logics Programs: An evaluation and extended translation.

W3C (2001), 'Semantic web activity statement', W3C Web site.

Wickler, G. (1999), Using Expressive and Flexible Action Representations to Reason about Capabilities for Intelligent Agent Cooperation, PhD thesis, University of Edinburgh.

Wroe, C., Stevens, R., Goble, C., Roberts, A. & Greenwood, M. (To appear), 'A suite of daml+oil ontologies to describe bioinformatics web services and data', *International Journal of Cooperative Information Systems* .