Multi Agent Systems Meet the Semantic Web - Relating LCC to OWL-S

Master's Thesis
24. August 2007

Mikkel Boje
s0675908
mikkel@diku.dk

Supervisor: David Robertson

[Abstract] There is a wish for integration of Multi Agent Systems research and Semantic Web technologies from both the MAS community and the SW community. The MAS community would benefit from the widespread infrastructure of the Web and the SW community would benefit from research results in the fields of agents and communication between them. However, several challenges exist before such an integration is possible. Firstly, the relationship between Multi Agent Systems and Semantic Web Services is at the time of writing only poorly understood. Secondly, even though Service-Oriented Architectures seem to provide a link between the two fields, services is a fairly new subject to MAS research.

This project provides an analysis of the MAS protocol language, LCC, and of the Semantic Web Service specification language, OWL-S. Furthermore, the relationship between the two is explored and the two are compared with respect to the challenges a service oriented Semantic Web poses. The comparison shows that LCC is superior to OWL-S in terms of service composition, but that LCC has no notion of services and therefore is inferior in terms of service discovery.

A notion of services for LCC protocols is therefore developed and a discovery system for discovering LCC services is also developed. Both are evaluated and compared critically with respect to OWL-S. It is concluded that LCC services are different from OWL-S services in several aspects, but that discovery of LCC services can be carried out successfully.

Master of Science - Artificial Intelligence

School of Informatics
College of Science and Engineering
University of Edinburgh

Contents

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Mikkel Boje

Preface

This report is a Master's Thesis, a part of a MSc Taught Master's programme in Artificial Intelligence at the University of Edinburgh, 2006/2007.

Its intended readers are persons with a technical background knowledge. Some acquaintance to agent communication and the Semantic Web including Description Logics (DL) is an advantage as these subjects are central to the report, but will only be introduced briefly. It has been pursued to explain new concepts with references at the point where they are introduced in the text.

As of typographic conventions, DL concepts and properties have been formatted in sans-serif, code and file names are in type-writer format, and finally examples are in italic. A terminological convention is that the two terms, class and concept will be used interchangeably and a distinction is not made. This includes any derivations of the two terms such as subclass and subconcept.

The report is structured as follows:

Introduction The project is first put into context in section 1. Specifically, LCC is placed in the Multi Agent Systems field and OWL-S' relationship to the Semantic Web is explained.

Problem The problem and the hypothesis of the project is then stated in section 2.

Comparison The relationship between LCC and OWL-S is explored in sections 3 through 5. Section 3 and 4 describe OWL-S and LCC, respectively. Section 5 then provides the comparison of the two.

Extension Section 6 extends LCC with a notion of services. Concretely the LCC syntax is extended with types.

Discovery Building on the syntax extension, sections 7 and 8 describe the development of a discovery system for LCC services.

Evaluation The proposed extension and the discovery system developed are evaluated in section 9.

Conclusion Finally, section 10 concludes on the results of the report and points out future research opportunities.

It is my hope that some of the excitement during the project can be sensed from the report. I would like to thank my supervisor, David Robertson, for his guidance and provision of several research links including the LCC interpreter used in the project.

Mikkel Boje
24th of August 2007

1 Introduction

Essentially, this project is about computer communication. Of this comprehensive field, focus will be on the World Wide Web (WWW) and Multi Agent Systems (MASs). The WWW is a well known, fully deployed technology which has had an enormous influence both on personal everyday life and in commercial situations. MASs, on the other hand, is a more academic topic with advanced features and much less deployment. The field of MASs is concerned with complex communication between autonomous and independent agents. It is a research area driven more by visions than deployed implementations [Woo02] section 1.1.

As the WWW continues to develop and the field of MASs matures, two obvious questions come to mind: can advanced research in MASs be used in the further development of the WWW; and is the WWW a suitable platform for MAS deployment [Wal07] p. 204? The concept which seems to bring the two fields closer is services [Wal07], [SH05], [SDF06]. However, services are somewhat new to MAS research and many aspects of services are still poorly understood with respect to the WWW. Needless to say, a service connection between MASs and the WWW still needs investigation [SDF06].

Services will be the pivotal point of this report. In the following they will be introduced in terms of Service Oriented Architectures before MASs and the next generation of the WWW, the Semantic Web, are introduced.

1.1 Service Oriented Architectures

A Service Oriented Architecture (SOA) is an architecture where participants (computer programs) communicate via services. A service is an entry point to a set of resources. Resources may be abstract like information or algorithms; or may be concrete like telescopes or banking systems [Wal07] p. 194. Services in an SOA are wrapped into descriptions. It is basically this wrapping which is the main advantage of an SOA. The description should include an interface describing how to use the service and semantics of the service, i.e. what it does. SOAs are usually defined through a series of requirements [SH05] p. 76, [BMN+04] section 3.1:

- Loose coupling is required because the infrastructure is expected to be slow and unreliable. Tight transactional properties between components should be replaced by abstract reasoning on information flows.
- Coarse granularity is also required as a consequence of the slow, unreliable infrastructure. Communication should consist of few, but significant messages.
- Implementation neutrality is a consequence of the description approach. It is a very important requirement as it secures interoperability between diverse systems which is one of the main advantages of SOAs.
- Flexibility ensures that components can be replaced and bound together late or even on the fly.

One other important characteristic of SOAs is the notion of a provider and a consumer or requester. Until recently, most emphasis has been on a single requester - single provider scheme which corresponds to the WWW's client - server architecture. However, one of the challenges of SOAs is to break with this scheme and allow composition of services:

- Service Invocation is the challenge of ensuring a standard, automatic and platform independent way of communicating with services. This challenge has more or less already been met through standard RPC mechanisms [Wal07] p. 195.
- Service Discovery is the challenge of dynamically finding a service of interest. It is necessary as SOAs are flexible and can be of open nature where services are not known beforehand.
- Service Composition is the least understood challenge of SOAs [Wal07] p. 196. Connecting different services to create more complex applications is one of the reasons to use an SOA in the first place.

When discussing both the requirements and challenges of SOAs, it is instructive to note a shift from closed environments to open environments. There is a trend of going from intra-business environments to inter-business environments (e.g. the Internet). This trend makes the requirements even more fundamental and the challenges even more challenging.

In this project the service connection between the WWW and MASs will be discussed through analysis of the challenges of service discovery and service composition. These two challenges will from now on be referred to as the SOA challenges.

Service Discovery The key to service discovery is service descriptions. These descriptions hold the information which makes discovery possible. To enable discovery, descriptions must include (loosely based on [BMN+04]):

- An identifier of the service
- A description of what the service does
- Description of parameters of the service, i.e. inputs and outputs
- Under what conditions the service can be invoked
- What effects invocation will have

Presuming we have descriptions with the above information, discovery follows the steps below [FGY07] section 2:

1. The user dispacthes a description of the requested service, either to a centralised yellow page directory of services, e.g. [udd], or, in a decentralised setting, to its neighbour peers [DKK+].
2. The requested service is matched against services residing either in the yellow page registry or in the registries of neighbour peers.
3. If the matched service descriptions do not include providers, the requester then needs to discover providers.

Several challenges are clear from the steps above. However, the challenge of terminological agreement might not be apparent at first sight. In open SOA environments the terms used to describe (classify) services may not be universal. The same terms may mean something different and different terms may refer to the same between different organisations. This is a serious challenge and apply generally when communication is to occur in open environments. Much research has therefore been carried out in mapping from one set of terms to another [KS05].

Service Composition is fundamental if full advantage is to be taken of an SOA. However, the challenge of service composition is not straightforward. Composition requires:

1. Description of interaction between subservices. This description must include information on how the communication is to flow (control flow) and what data connections exist between subservices (data flow).
2. Consistency between conditions and effects of subservices.
3. Some glue to adapt different datastructures between subservices.
4. Handling of exceptions.

Service composition, of course, also has to deal with terminological agreements between subservices.

1.1.1 A Service Example

To motivate the discussion, a service example will be used throughout the report. Without doubt, one of the most important uses of services is trade. Both transactions between individuals and companies, and business to business transactions are important. As explained above services are characterised by a service interface. The interface of a service selling items to either individuals or organisations is given in figure 1. The interface is defined as a function. The name

$$Shop(ProductCode, CreditCard) \rightarrow Receipt$$

Figure 1: Example of a Service

of the service is *Shop*; the inputs are *ProductCode* and *CreditCard*; the output is a *Receipt*. The meaning is that a client provides the service with a product code of some needed product and credit card details. The service will then carry out the purchase and return a receipt to the client. Obviously, the service is greatly simplified compared to real applications. Throughout the report the service will be elaborated to analyse different aspects, but the intention is not to end up with a complete, realistic trade specification.

1.2 Multi Agent Systems

The field of Multi Agent Systems derives from the area of distributed systems which dates back several decades [Woo02] section 1. The field is concerned with communication between agents. There is no single accepted definition of an agent, [Woo02] p. 15, but in this report it will suffice to note that they are autonomous computer systems situated in some environment (which is close to the definition used in [Woo02] p. 15).

A coupling between agents and services has been proposed before. Agents can be regarded as services themselves, or more accurately as providers of services. One reason to regard agents as providers rather than services is the simple fact that an agent can be imagined to provide more than one service. More important, though, is it that agents seem more flexible and complex in nature than services. Some of these differences are listed in [SH05] section 16.1:

- An agent is self-aware as opposed to services. It can therefore learn and adapt to the environment.
- Agents are communicative in an active way. Services are passive until invoked.
- Agents are autonomous whereas services, so far, have been thought of as simple parts of a larger system.
- Agents are cooperative. Coalitions of agents that achieve higher, more complex goals have been studied in MASs. The composition of services is currently poorly understood.

These differences may very well diminish as services evolve to meet the new demands of open environments. That is, as new demands to services arise, advanced features of agent research might very well be applicable to the evolution of services.

1.2.1 Agent Communication

Multi Agent Systems include not only research on individual agents, but also on systems of agents. Much research in MASs has been concerned with agent communication. It is instructive to note that research has to a large extent been based on human communication research. Several layers of communication have been identified: low-level message passing, mid-level dialogues and high-level social norms. Message passing research includes speech-act theory application and message languages like KQML (Knowledge Query and Manipulation Language) and FIPA-ACL [Woo02] section 8. Dialogues have been implemented as dialogue frames [Wal07] p. 172. With dialogues and understanding of messages, conversations between pairs of agents are possible. For more than two agents to communicate in a proper way, something more is needed: social norms or policies. One popular technique for reasoning about social norms is Electronic Institutions (EI).

EIs are meant to correspond to real world institutions. Such an institution could be an auction. An auction consists of a bidding part and a pay part. These parts are denoted scenes. EIs consist thus of institutions divided into scenes with agents taking on roles in each scene. Social norms are implemented as constraints on: agents entering an institution; agents taking on roles in scenes; role scripts for each role and scene; and agent progress through scenes.

Protocols provide a very succesful way of implementing social norms and specifying dialogue patterns. Protocols are powerful in that they basically provide executable specifications for interactions between two or more agents. As they are models of interactions, they are also referred to as interaction models. A protocol modelling the interaction of the service in section 1.1 is shown in figure 2. The protocol shows how a client starts the interaction by sending a

$$\begin{array}{lcl} client \rightarrow & message(ProductCode, CreditCard) & \rightarrow shop \\ client \leftarrow & message(Receipt) & \leftarrow shop \end{array}$$

Figure 2: Example of a Protocol

message which includes product code and credit card details to the shop. The shop then sends back a receipt to confirm the purchase.

Light Coordination Calculus (LCC), [Rob04a], provides a state-of-the-art protocol language for defining interaction models of MASs. LCC is based on process calculus developed for reasoning about the semantics of concurrent systems. This foundation emphasises the close relationship between MAS research and distributed systems. [Wal07] p. 181 describes LCC as a sugared variant of π-calculus [Mil91]. LCC will be described further in section 4. An LCC protocol for the trade example is given in section 4.1 (figure 7).

1.2.2 MASs and the SOA Challenges

The attentive reader will have noticed the resemblance of systems of agents and service composition. That is, the SOA challenge of service composition might very well benefit from research on MASs as envisaged in the beginning of this section, cf. section 1.

The SOA challenge of discovery is not an equally well integrated part of MASs research as composition or collaboration. This is simply due to the, often taken, assumption of a closed environment where other agents are known. This assumption is not completely default and discovery has been analysed in the MAS community as witnessed by its inclusion in the agent management specification provided by FIPA (Foundation for Intelligent Physical Agents) in 1997 [fip97].

1.3 The Semantic Web

The World Wide Web (from now on simply the Web) as of today is designed to present information to humans. Web pages consist of non-structured data with a markup of content for viewing purposes. The Web has turned out to be an extremely succesful infrastructure for data communication. More advanced applications have thus been deployed through the Web. People are able to carry out net-banking and shopping using the Web. These applications are not merely presentations of information to humans. Instead the term, Web Service, has been chosen to represent such activities which involve machine processing of web content. This processing of content is the crux of the Web evolution debate. The current Web does not support structured data which is necessary for machines to carry out automatic processing. The Semantic Web is envisioned to extend the current Web by providing meaning or structure to web data [HL01]. As mentioned, the prominent benefitors of this "meaning" will be applications processing web information, i.e. Web Services, which are termed Semantic Web Services in the Semantic Web.

The Semantic Web vision has caught great attention. One of the reasons might very well be that the project is backed up by large organisations, W3C[1] being an important actor. W3C, with the Web's inventor, Tim Berners-Lee, in front, is trying to standardise the technologies necessary to fulfil the dream of the Semantic Web. These are usually presented as a layered diagram (see figure 3). The lowest layers up to and including XML are concerned with standard data formats and will not be discussed further.

[1] W3C stands for World Wide Web Consortium and is an international standardisation body for the Web [w3c].

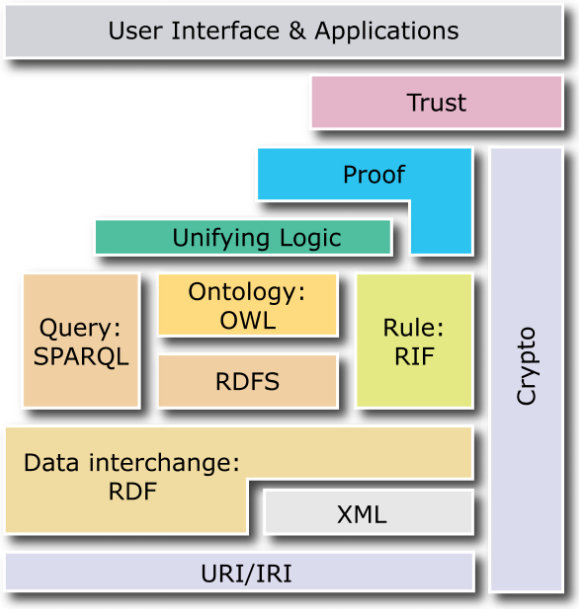

Figure 3: The Technology Layers of the Semantic Web

Figure taken from the Semantic Web project at [w3c].

1.3.1 RDF Triples and OWL

The chosen knowledge representation of the Semantic Web is key to its machine support. Knowledge representation has been extensively researched in the AI (Artificial Intelligence) area [HL01]. The basic data representation idea of the Semantic Web builts on this research and is based on triples. The rationale is that all data is expressable as binary relations between entities. A triple thus consists of a subject (first element), a predicate (second element), and an object (third element) [MMM04]. RDF (Resource Description Framework) is the formalisation of such triples [MMM04]. It makes use of URI (Uniform Resource Identifiers), [BLFIM98], for all three elements - though simpler datatypes without URIs are allowed as objects (third element). The URIs are important because they ensure openness and at the same time disambiguity on the Web.

The assumption that RDF triples are enough to represent data may be true, but it is insufficient in representing and reasoning about more complex information than data such as information about information [AvH04] . This type of information has been represented using ontologies in the AI field. Ontologies are basically a representation or conceptualisation of some domain. They are based on hierarchies of classes and properties about these classes. Properties are binary and so the ontology approach fits well with the Semantic Web triples. The approach simply adds in another level: individuals and classes. RDFS (RDF Schema) brings in basic class features like the notion of classes and properties. RDFS, however, showed limitations in terms of the usual requirements made to ontology languages [AvH04]:

- A formal semantics
- Sufficient expressive power

- Efficient reasoning support

The next layer of the Semantic Web infrastructure was then created, the ontology language layer cf. figure 3. The ontology language proposed by W3C is OWL (Web Ontology Language) [MvHe04]. The knowledge representation theory behind OWL is known as Description Logics, DL, and has been studied extensively in the AI field (see [NBB+03] for a nice introduction). The second and third aim of ontology languages turn out to be dependent on each other and three versions of OWL have been constructed to suit different needs: OWL-Lite, OWL-DL and OWL-Full. OWL-DL (OWL Description Logics) has complete (and rather efficient) reasoning support as opposed to OWL-Full, while still retaining expressive power as opposed to OWL-Lite. In this report we will only be concerned with the OWL-DL version.

1.3.2 Semantic Web Services

The data and information representation described above is a prerequisite for the semantic applications envisioned. Following the trend in computer science and lessons learned in distributed computing, [SH05], applications of the Semantic Web are expected to follow a Service Oriented Architecture [BMN+04] section 3.1. Thus, the applications are referred to as Semantic Web Services, or just Web Services where the difference is clear from the context. Web Services will make use of web information represented in OWL. It is obvious to use OWL for service descriptions as well. The OWL-S (OWL-Service), [eea04], ontology is exactly an ontology for describing Semantic Web Services using OWL. OWL-S is described in detail in section 3. The differences between Semantic Web Services and general services consist of semantic descriptions of Semantic Web Services and the fact that SWSs use the Web as infrastructure. The trade example of section 1.1 is therefore also an example of a possible Semantic Web Service. A SWS would, however, be accompanied by a semantic description; these are discussed in section 3.

1.3.3 The Semantic Web and the SOA Challenges

The SOA challenges from section 1.1 also apply to Semantic Web Services. Most of the discovery challenge is handled by semantic description of services, though the issues of matching and terminological agreement still remain open. Even though OWL-S also has the ability to describe composed services, it will become clear that many issues of Web Service composition are still unsolved, cf. section 3.4.

2 Problem Statement

Having put the project into a broader context, the next step is to state what precisely this project aims at. The project focuses on the combination of MASs and the Web with respect to services. The technique from systems of agents that will be analysed is the protocol language, LCC. Web Services will be represented by the OWL-S ontology. The aim is to compare the approaches of OWL-S and LCC in terms of interaction and service challenges. As it will turn out LCC has no support for services, an extension of LCC will be proposed. A system that deals with the service challenge of discovering LCC services will be designed, implemented and evaluated. This discovery system will test the service capabilities of the extended LCC.

2.1 Motivation

As indicated in section 1, the motivating idea behind this project is to explore and combine the fields of Multi Agent Systems and Web technology to the benefit of both. This opportunity is widely accepted; for instance in [JCI02] p. 105: "Agent and Internet technologies can complete each other and are a perfect integration for building autonomous, complex, and distributed business and engineering systems".

Specifically, as Web Services still grope for a proper model of service interaction and composition ([owlb] explicitly states that OWL-S is missing a conversational protocol), the relation to MAS interaction models - exemplified by LCC - seems like an obvious research opportunity. There is currently research ongoing in the area of OWL-S and interaction models, [NFH05], but no comparison to LCC exists. Furthermore, the Web looks like a promising platform for MAS deployment, cf. section 1. Services are expected to be the facilitator, but LCC has yet no notion of services. Another motivation is therefore to explore the relationship between LCC and services - exemplified by OWL-S. This opportunity is also recognized in [Rob04b], but to date no attempt has been made which further motivates the project.

2.2 Hypothesis and Goal

The hypothesis of this project is that LCC broadly corresponds to the ServiceModel of OWL-S, but, unkike OWL-S, without any compliance to Service Oriented Architectures.

The theoretical goal of the project is to support the above hypothesis. The practical goal is to extend LCC with a notion of services, implement a discovery system based on this notion, and evaluate the extension and the system.

2.3 Method

The method to accomplish the goals is given by the following report outline:

- Describe OWL-S and LCC - section 3 and 4
- Compare the two - section 5
- Analyse LCC with respect to services and through this analysis propose an extension of LCC - section 6

- Design a discovery system for LCC services - section 7
- Implement the system - section 8
- Evaluate the extension and the system - section 9
- Conclude on theoretical and practical outcome of project - section 10

In the description and comparison of OWL-S and LCC, the SOA challenges and accompanying issues from section 1.1 will be used as a framework for discussion.

2.4 Scope of Project

The project is limited to a discussion of interactions between entities in terms of OWL-S and LCC. That is, other candidates for technology transfers between the domain of MASs and the Web will not be considered. Furthermore, the project is positioned at an abstraction level where low-level communication is not considered. As a consequence of this abstraction level, the SOA challenge of service invocation will not be discussed in detail.

The SOA issue of terminological agreement will only be handled indirectly. That is, using semantic markup of terms will be considered a solution and issues regarding ontology management will not be analysed in detail.

3 OWL-S

The following description of OWL-S is based on a technical overview of OWL-S [eea04], submitted for W3C as a proposed standard for Web Service description. Besides this white paper, the ontology (and subontologies) itself has been used [owl04]. When the overview has been unprecise or differences between overview and ontology have occured, the ontology has been regarded as definitive. At the time of writing, a 1.2 pre-release of OWL-S exists. Both ontology and documentation is incomplete, and - if not noted otherwise - OWL-S is discussed, in this report, with respect to its 1.1-version [eea04].

As mentioned in section 1.3.2, OWL-S is written in the ontology language OWL. In this report space does not allow us to describe OWL in further detail, instead see [MvHe04] for a more comprehensive introduction.

We will start out with a description of the ontology before analysing its capabilities. The top concept is Service, the name reflecting that the ontology describes services. This concept has three properties, presents, supports and describedBy. The properties, respectively, refer to: what the service provides described in concept ServiceProfile, how the service works in concept ServiceModel and how invocation of the service should be carried out in concept ServiceGrounding. Each of these concepts will be described in turn.

3.1 OWL-S Service Profile

An important job of the ServiceProfile is to facilitate discovery of services. The properties can be divided into three parts: who provides the service, how the service functions and what the service is [eea04]. The organisation behind the service is described by:

- serviceName gives the name of the service. Could be an identifier.
- textDescription a human-oriented description of the service.
- contactInformation provides contact information to the organisation behind the service in a human or machine readable format.

It might first look as if a description of the functionality in the ServiceProfile is redundant as it is already provided in the ServiceModel-part of the OWL-S ontology. This is indeed true and is the reason why the following properties are recommended to refer to concepts in the ServiceModel [eea04]. The reason to also include them in the ServiceProfile is that input, output, preconditions and effects might be useful information for discovering a service, cf. section 1.1. The properties are:

- hasParameter links some parameter to the service.
- hasInput, a subproperty of hasParameter which denotes an input to the service.
- hasOutput, a subproperty of hasParameter which denotes an output of the service.
- hasPrecondition describes a precondition for the service to be applicable.
- hasResult denotes the result of the service which is a structured value consisting of a condition, effects and output. Several results can then be associated with the service, all with different conditions.

The third part of the ServiceProfile is essential for discovery as it describes what kind of service the service is. These are the properties:

- serviceCategory has range ServiceCategory which is a categorisation class with more properties.
- serviceParameter has range ServiceParameter which is a class that is supposed to provide more information about the service, so the requester agent can make a choice between services with same service category.
- serviceClassification classifies the service; the range is OWL ontologies.
- serviceProduct denotes the product of the service; the range is OWL ontologies.

The serviceCategory property and the serviceClassification and serviceProduct properties are very similiar and essentially only differ in the allowed ranges of ontology-references. That is, some of the properties of the ServiceCategory concept have a relaxed range such as the union of literals, URIs and URLs, which is opposite the restricted range of the two other properties.

3.2 OWL-S Service Grounding

The ServiceGrounding part of OWL-S is aimed at providing a low-level interface to a service, enabling concrete communication. In other words, a description of the interaction between requester and provider. As of version 1.1 of OWL-S, a generic Grounding class has not been significantly developed. In recognition of WSDL (Web Services Description Language [CCMW01]) being the most widespread Web Service interface language, a WsdlGrounding class has been elaborated instead. An instance of this class maps concepts from the ServiceModel to concepts of WSDL. By making the WSDL interface a part of the OWL-S description, advantage can be taken of OWL-S' classification power for parameter types etc. The mapping between OWL-S and WSDL consists of:

- Atomic OWL-S processes[2] to WSDL operations
- Inputs and outputs of OWL-S-processes to WSDL messages
- OWL-S IO-types to WSDL abstract types

This mapping needs to be stated in the ServiceGrounding.

The ServiceGrounding part of OWL-S is supposed to deal with the challenge of a concrete communication scheme. The overlap between OWL-S and WSDL and the complexity of both the OWL-S ServiceModel (see 3.3) and WSDL which are linked together, makes the OWL-S communication scheme rather comprehensive. This is one of the reasons, tools to support OWL-S editing has been developed [owla].

3.3 OWL-S Service Model

The ServiceModel is the most comprehensive part of the OWL-S ontology. It should be noted that Process Model is also used to denote the same ontology. A process is simply another way

[2]Atomic processes will be described in 3.3.

to view a service [eea04]. When discussing the ServiceModel, the term process will be used instead of service to follow the OWL-S convention. The model describes both non-composed processes and composed processes.

The Process class which is a subclass of ServiceModel has, besides the properties shared with ServiceProfile, the hasParticipant and name properties[3]. The Process class is meant to be an abstract superclass of the three disjoint classes, AtomicProcess, SimpleProcess and CompositeProcess. The SimpleProcess class is an abstract class covering the two others which will not be considered further here.

The ServiceModel properties can be divided into three parts: properties shared with ServiceProfile, properties concerned with control of compositions and properties describing data flows in compositions. The first part of properties is common to both AtomicProcess and CompositeProcess and will be discussed next. The two other parts only pertain to CompositeProcess and will be dealt with in that context, i.e. in section 3.3.3.

3.3.1 Parameters, Conditions and Results

The properties listed in section 3.1 have the concept ranges of Condition, Result, and Parameter with subconcepts: Input, Output, and Local (not mentioned in section 3.1). The most important job for the Parameter concept and its subconcepts is typing of variables through use of the parameterType property. The Local concept is introduced to accomplish a kind of scoping.

Outputs often tend to be outputs of some function. OWL-S therefore provides two ways of specifying an output, valueData and valueFunction. valueData is a property linking an output to some static data value. valueFunction has an XMLLiteral as range which means a dynamic function can be described as output. That is, no concept has been developed to cover functions - which is needed as they are outside the expressive power of OWL. This means there are no variableBinding properties available to link function variables to the parameters of the process! These bindings need to be specified as part of the XMLLiteral. It also means that terminological agreement regarding the functions is non-existent. How the function is to be interpreted is unknown and meaning of terms used in the function description and thereby meaning of the function as a whole is not known.

Condtions Preconditions and their range, Condition, which are logical formulas, are also outside the expressive power of OWL. Unlike functions, however, a development of a supplementary ontology, Expression [4] has begon. Condition is a subclass of Expressions and has a property, expressionBody, whose range is an (XML) string. That is, logic formulas are basically written as (XML) strings. Expression has another property, expressionLanguage which has a range of LogicLanguage. This gives some information about the string and how to interpret it. Like in the case of functions, we still miss: variable bindings, terminological agreement, and an understanding which would enable OWL reasoners to reason about preconditions of services. Some second parsing beyond OWL is proposed in [eea04] section 5.1.

[3] hasLocal is actually also only part of the Process concept, but is a subproperty of hasParameter.

[4] See [owl04].

Results The outputs and/or effects of a service might be different according to different circumstances (i.e. different conditions). This is a necessary complication which is taken care of with the Result concept in OWL-S. It has the following properties: inCondition which states the precondition for the result; hasResultVar declares variables in precondition (recall that conditions are written in another language than OWL and a variable mapping is necessary); withOutput refers to binding of some output parameter of the service to this particular result; hasEffect points to an effect, which is represented by the Expression concept (upper concept of Condition). As for preconditions of services, reasoning about results of services is problematic because of the necessary logical formulas of result conditions and effects.

An important issue is the relationship between the conditions and effects of results and the preconditions and effects of the service as a whole. My personal experience from a project using Protégé with OWL-S[5] was that preconditions and effects of services were much harder to recognise than conditions and effects attached to results. Taking as example the conditions, we might ask whether the preconditions of services are the union of all result conditions? The answer is no, since results can be imagined happening under mutual exclusive circumstances. Service preconditions were originally most likely introduced from studies of planning, [eea04], but their application to services needs to be further examined and in particular their connection to result conditions.

3.3.2 Atomic processes

Atomic processes are instances of the AtomicProcess concept. This concept has the same properties as its upper concept, Process. An atomic process always has two participants, TheClient and TheServer. The AtomicProcess is the 'leaf' process type which is directly invokable using its corresponding ServiceGrounding.

3.3.3 Composite processes

A composite process is a description of a way a client can accomplish some desired results. The description is a composition of subprocesses which may be AtomicProcesses or CompositeProcesses. The description can broadly be divided into two parts:

- A control description, i.e. when to carry out each subprocess.
- A dataflow description describing how the inputs and outputs of the subprocesses and the top process are connected.

Control description Composite processes are processes which must have the composedOf property defined. The range of this property is a subclass of the ControlConstruct class. These subclasses denote some type of control:

- Sequence, subprocesses are to be done in order. Completes when all subprocesses have terminated.
- Split, subprocesses are to be executed in parallel. Completes when all subprocesses have been initiated.

[5]See problem formulation in [mas].

- Split+Join, subprocesses are to be executed in parallel. Completes when all subprocesses have terminated.
- Any-Order, subprocesses can be executed in any order. Completes when all subprocesses have terminated.
- Choice, a single subprocess is executed from a bag of subprocesses. Completes when the subprocess terminates.
- If-Then-Else, then subprocesses are executed if condition holds. Otherwise, else subprocesses are executed. Terminates when subprocesses have terminated.
- Iterate, an abstract superclass of the following two classes.
- Repeat-While, checks a condition, executes subprocesses if condition holds, then loops.
- Repeat-Until, executes subprocesses, then checks condition and loops if condition holds.

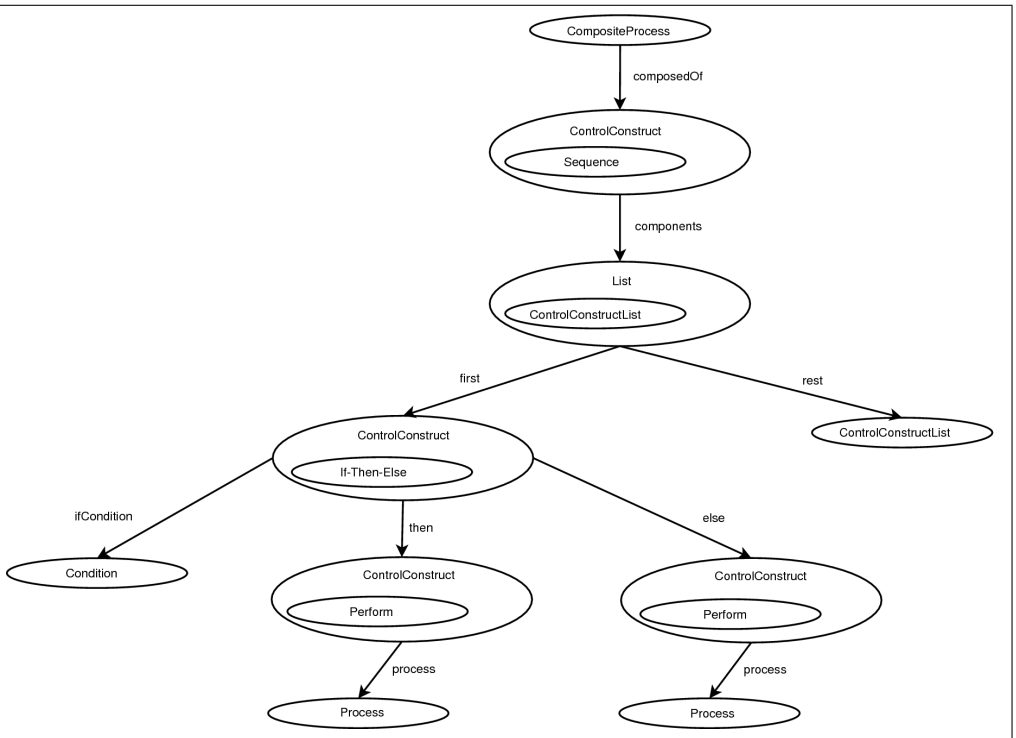

Figure 4: Example of OWL-S Control Description

Subprocesses should be understood as potentially including more control constructs. That is, the subprocesses of the above control concepts are actually wrapped in ControlConstructs. To handle situations with more than one sub-control construct, collections of control constructs are used. The unconditioned control classes from the above list have a components property with an appropriate range (with respect to its domain). The range of the components property is restricted in each ControlConstruct subclass, usually to some sublcass of &shadow-rdf;#List. The conditioned control classes have special properties corresponding to components, which point to single control constructs in contrast to collections of them.

The control construct design enables a kind of tree structure where leaves are reached by letting the sub-control-construct refer to Perform. Perform is also a subclass of ControlConstruct, but the process property is necessary and has the Process class as range. An example is given in figure 4.

The reason Perform has been introduced (instead of just using Process) is to distinguish between a description of a process and the execution of it. The description is stateless whereas an executing process has a state with values assigned to its variables.

Conditional constructs like If-Then-Else and Repeat-While have properties linking them to the Condition concept like preconditions, cf. section 3.3.1. As mentioned in the discussion of preconditions, this means that an OWL-reasoner cannot be expected to reason about conditional constructs in composite OWL-S services.

Dataflow description Having described the control flow of the process, we still need to describe how inputs and outputs of the different processes are linked together. As for the control description, processes are referred to as performs in dataflow descriptions. It has been decided to use a pull perspective. It is thus the perform where the dataflow ends that carries the description. Figure 5 shows the structure of the concepts used for dataflow modelling in OWL-S. The pull perspective is indicated by the property name, hasData**From**. As the figure shows, dataflows between processes are described with the Binding concept and the valueSource property. The range concept of valueSource, ValueOf, has two properties to state where the value comes from: fromProcess is the process and theVar specifies the process' parameter.

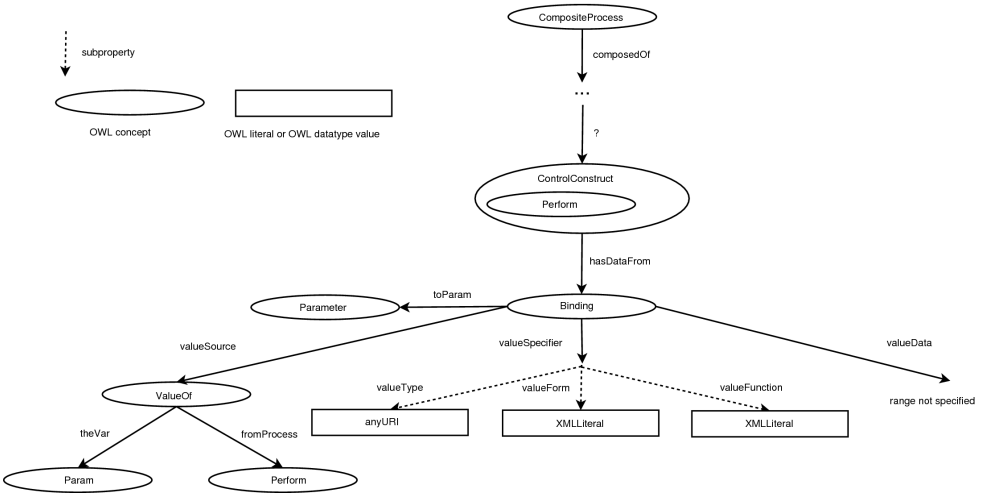

Figure 5: Domains and Ranges of Dataflow Properties

Values can also be described in other terms than processes. This is what the properties, valueData, valueSpecifier superclass of: valueFunction, valueForm, valueType are for. As mentioned in the ontology comments [Mar05], valueData would ideally also be a subclass of valueSpecifier [6]. Only one value descriptor should be used, even though this is not enforced by the ontology. It is not clear why this is recommended with respect to valueType which simply describes the

[6]It is actually not clear why this is not possible as there are no inconsistency concerning domains or ranges.

type of the value (which should be consistent with the type of the parameter) and does not seem to be incompatible with a property describing the value of the value. valueData is meant to be used for constant data as opposed to valueForm and valueFunction. The distinction between valueForm and valueFunction is not completely clear, though.

References to the service itself (the top service) are made using the special concept, TheParentPerform (replaced by ThisPerform).

A last complication is posed by conditional dataflows. Because the control description includes conditions, a source of data might also be conditioned. This is solved with respect to the overall process, ThisPerform, by introducing a class, Produce and a corresponding property, producedBinding. A Produce is expected to be inserted in the control flow every time a potential (conditioned) output of the overall process is produced. The producedBinding provides the binding between the data source and the output of the parent perform. It is not discussed anywhere how one should deal with conditioned data sources between the subprocesses.

Like for conditions in the control description, OWL is not expressive enough to deal with the specifications of functions (or even constant data) of values.

3.4 OWL-S and the SOA Challenges

OWL-S has been specifically designed to meet the challenges of Web Services on the Semantic Web [eea04]. Its basis in OWL and DL makes it ideal for providing a description of services in general terms, i.e. ServiceProfile.

The Challenge of Discovery The service profile facilitates service discovery and, together with ServiceGrounding, service invocation. The discovery requirements to service descriptions are met as follows:

- An identifier of the service is given by the serviceName property of ServiceProfile.
- Description of what the service does is described in detail by the properties: serviceCategory, serviceParameter, serviceClassification, and serviceProduct, cf. section 3.1.
- Parameters are described with use of the Parameter concept and corresponding properties of ServiceProfile, cf. section 3.1.
- Conditions and effects of the service are included in the ServiceProfile, but issues regarding the two are evident. Firstly, OWL is not expressive enough for their nature and work-arounds are needed. Secondly, their relationship to results of the service needs to examined.

In spite of the issues regarding conditions and effects, OWL-S must be regarded as a good candidate for service descriptions when it comes to discovery.

The Challenge of Composition Composition is described in the ServiceModel of OWL-S. It is important to emphasise that the ServiceModel only provides a non-executable specification of service compositions [eea04] section 5, [Rob04b] section 10. One consequence of this is that requirement in section 1.1 of glue-logic is not catered for by OWL-S. It has no direct features to handle exceptions although one can handle them by using different results.

Requirement of consistency between preconditions and effects is also not addressed since OWL-S is merely a specification language. OWL-S does contain a description of compositions, but as mentioned it is highlevel and non-executable. The hierarchical structure of the description provides good abstraction. It is, however, likely that this hierarchy is the cause of problems with data flow specification. Especially the Produce construct must be regarded as a weak point in the framework which is also noticed in [NFH05].

The incomplete handling of functions and conditions must be regarded as a serious shortcoming for real-world deployment of the model. However, future releases of OWL-S might present a satisfactory work-around to the lack of expressive power.

Concluding on how OWL-S meets the SOA challenges, we note that discovery is handled by the ServiceProfile[7] and composition is handled by the ServiceModel. OWL-S seems well equipped to meet the discovery challenge, though issues with respect to conditions and results exist. The challenge of composition is not completely handled by OWL-S. First and foremost, does OWL-S not provide an executable specification to carry out the composed service. Secondly, the ServiceModel is still deficient in several regards and it yet remains to be determined how suited a final version of OWL-S will be for service composition in real-world settings.

[7]Service invocation is handled also with the service profile and in addition with the ServiceGrounding.

4 LCC

As mentioned in the introduction, cf. section 1.2.1, LCC is a language designed to specify interaction models. Its terminology comes from its inspiration of Electronic Institutions. Its formalism is due to its foundation in process calculus. LCC will be described by delving right into its syntax and an example. The semantics of LCC models will then be presented before process calculus aspects are identified in the language.

4.1 LCC Syntax

In accordance with EIs, LCC views communication between agents as a description of roles performing some actions such as communicating with other roles. An interaction model thus consists of a list of roles or clauses, each containing a pair of an agent description and a perform description. An agent description (from now on simply an agent when there is no risk of ambiguity) is a pair of a role type and an identifier. A perform construct is recursively built up of sequences (`then`) of performs and/or choices (`or`) of performs. The leaf performs consist of, possibly constrained, empty-actions, role shifts, and dispatch and reception of messages. The entire LCC syntax is described in BNF in figure 6, very much like in [Rob04a] and [RWB+06]. Notice that the precedence of message operators, $\Rightarrow$ and $\Leftarrow$, have higher precedence than the

$Model$	::=	{ $Clause$, ... }
$Clause$	::=	$Agent :: Perform$
$Agent$	::=	`a`($Type$, ID)
$Perform$	::=	$Agent \mid Agent \leftarrow Constraint \mid Message \mid$ `null` $\leftarrow Constraint \mid$ $Perform$ `then` $Perform \mid Perform$ `or` $Perform \mid Perform$ `par` $Perform$
$Message$	::=	$M \Rightarrow Agent \mid M \Rightarrow Agent \leftarrow Constraint \mid$ $M \Leftarrow Agent \mid Constraint \leftarrow M \Leftarrow Agent$
$Constraint$	::=	$Term \mid \neg Constraint \mid Constraint \wedge Constraint \mid Constraint \vee Constraint$
M	::=	$Term$
$Type$	::=	$Term$
ID	::=	$Constant \mid Variable$
$Term$	::=	$Constant \mid Variable \mid Constant(Term, ...)$
$Constant$	::=	Lower case alphanumeric sequence possibly followed by mixed case alphanumeric sequence
$Variable$	::=	Upper case alphanumeric sequence possibly followed by mixed case alphanumeric sequence

Figure 6: Syntax of LCC Protocols

constraint operators, $\leftarrow$ and $\rightarrow$. At first it might appear as if the performs of roles 'just' send and receive messages. However, the constraints in the language provide a rather powerful way of describing agent behaviour during interaction.

The use of constraints and LCC in general is best illustrated with an example. Figure 7 shows an LCC protocol of the trade example from section 1.2.1. A requester takes on the *client*-role. The product code of the product the requester needs is the input of this role. Constraints of the *client*-role are used to access credit card details as well as looking up the shop to contact.

Just as in the generic protocol example in section 1.2.1, a message is sent to the shop. LCC describes roles rather than just interaction and so the *client*-role also specifies the reception of a receipt-message.

The *shop*-role is straightforward: it receives a *buy*-message from the requester and then, provided the client can afford the purchase and the order can be completed, a receipt is returned to the client. Notice that _ is used to denote a variable which is not of interest, i.e. a wild-card variable[8].

$$
\begin{array}{l}
\{ \\
\quad a(client(ProductCode), C) :: \\
\qquad buy(ProductCode, CreditCard) \Rightarrow a(shop, S) \leftarrow cc(CreditCard) \wedge lookup(S)\ then \\
\qquad receipt(Receipt) \Leftarrow a(shop, S). \\
\quad , \\
\quad a(shop, S) :: \\
\qquad buy(ProductCode, CreditCard) \Leftarrow a(client(\_), C)\ then \\
\qquad receipt(Receipt) \Rightarrow a(client(\_), C) \leftarrow enough\_credit(CreditCard, ProductCode) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad complete\_order(ProductCode, CreditCard, Receipt). \\
\}
\end{array}
$$

Figure 7: Example of an LCC Protocol

The example shows how constraints are used both as connections to resources outside the protocol (e.g. the *cc*-constraint) and as conditions for the interaction flow (e.g. *enough_credit*).

4.1.1 Constraints

In [Rob04a] constraints are divided into proaction and reaction constraints. These two categories correspond to constraints and consequences in [FGY07] respectively. The constraint type in [FGY07] is attached to dispatch of messages and role shifts[9]. Consequences are attached to reception of messages. What is the semantic difference between the two? According to [Rob04a], a proaction constraint describes under what circumstances a message can be sent (or a role shift can happen). A reaction constraint describes what will be true after reception of a message. In the semantic framework given in [Rob04b], proaction constraints ($satisfied(C)$ are interpreted as:

> "$satisfied(C)$ is true if C can be solved from the agent's current state of knowledge."

Reaction constraints ($satisfy(C)$) are interpreted as:

> "$satisfy(C)$ is true if the agent's state of knowledge can be made such that C is satisfied.

The difference between these two interpreations is rather subtle and perhaps of no practical significance. This is substantiated by the facts that: 1) an agent needs to be able to satisfy

[8]Such wild-card variables are also used in functional languages like ML and in Logic Programming languages like Prolog.

[9]In [Rob04a] constrained role shifts are not (yet) mentioned.

both types of constraints to continue expansion of a clause [Rob04a]; 2) the same mechanism is used to satisfy both types of constraints in the interpreter, which, among others, has been used in the development of LCC and is used in this project (see appendix D).

In the present report, the two types of constraints will be referred to as proaction and reaction constraints. Their semantic meaning will be regarded as identical except in terms of timing. The semantics is defined as:

> "A constraint is true if it can be solved from the current state of the agent and the protocol."

The issue of timing influences the meaning of 'current' in the above definition. Proaction constraints have the timing policy of being satisfied *before* the accompanying construct, whereas reaction constraints are to be satisfied *after* the accompanying construct.

4.2 LCC Interaction Model

So far, the syntactic constructs of LCC and an example have given an intuition of what LCC is. It is important, however, to have a formal semantics of the language. This is the only way we can formally reason about the interactions between agents. The semantics of LCC is provided by analysing the interaction model from a state changing perspective. The state of the interaction can either be maintained centrally (as part of the messages being sent around) or locally within each agent participating section 2.3 [RWB+06]. The advantage of local states is a more flexible system with agents possibly reacting in parallel. A centrally maintained state of the interaction provides a linear model which is easier to reason about and provides an opportunity to use something like CLP[10] on variables in the interaction [Rob04b].

The state of a linear system is provided by the triple $m(\mathcal{S}, \mathcal{P}, K)$, where $\mathcal{S}$ consist of clauses unfolded by agents participating (to be explained), $\mathcal{P}$ is the protocol under consideration and K is shared knowledge between the agents (e.g. variable bindings). Assuming signals between agents include a role, an identifier and a state triple m, an agent reacts (if it is correctly identified) to a signal by choosing a clause corresponding to its role. The clause is first looked for in the set of clauses already unfolded and, if not found, in the protocol definition. Once a suitable clause is found, the clause is *unfolded* by expanding the clause following a collection of rewrite rules (see [Rob04b] or the interpreter in appendix D). For instance, rewrite rule 1 in [Rob04b] says that expansion of a clause is done by expanding the perform-part of the specific clause. Each expansion uses a set of messages received, $Mrec$ and the protocol defintion, $\mathcal{P}$ and outputs a set of unprocessed, remaining received messages, $Mrem$, and a set of messages to be sent, $Mout$. A series of expansions (i.e. state transitions) is usually noted like this [Rob04b]:

$$\langle C_i \overset{Mrec_i, Mrem_i, Mout_i, \mathcal{P}}{\longrightarrow} C_{i+1} \ ... \ C_{n-1} \overset{Mrec_{n-1}, Mrem_{n-1}, Mout_{n-1}, \mathcal{P}}{\longrightarrow} C_n \rangle$$

If C_n in the above series can't be expanded more, it's either because the interaction is stuck (e.g. a role ends with a constraint that cannot be satisfied) or because the clause is closed. The notion of closed clauses is rather important as it denotes whether a clause has been processed [Rob04b]. This is essential to understanding the total state of an interaction and the progress

[10] CLP(FD) is an acronym for Constraint Logic Programming (over Finite Domains) and is a well known technique implemented in several Prolog variants [sic].

of it. Closures of performs travel up perform trees as perform leaves are closed. Perform leaves are closed when they have been executed and any possible constraints have been satisfied.

After the expansion of a clause, $\mathcal{S}$ is updated and the agent will send any messages generated by the expansion (i.e. $Mout$).

4.3 Process Calculus

As noted in the introduction (section 1), LCC can be seen as a sugared variant of π-calculus. Space does not allow to elaborate on process calculi, so only a simplified form of π-calculus is given in figure 8. Composition in π-calculus is either sequential or parallel [Mil93]. In LCC

$Action, A$	::=	$\bar{x}z.P$	send z along channel x
		$xy.P$	receive any y along channel x
$Process, P$	::=	$A_1 + ... + A_n$	alternative action ($n \geq 0$)
		$P_1 \mid P_2$	composition
		vyP	restriction
		$!P$	recursion

Figure 8: A Simple Form of π-calculus Taken from [Mil93]

roles are to be executed either in sequence or parallel - this is left open, cf. section 4.2. We can thus equate roles to processes. Subprocesses correspond to subparts of roles; both are executed either in sequence or in parallel indicated by | in π-calculus and `then` and `par` in LCC. π-calculus recursion is achieved in LCC by role shifts to the same role (see protocol example in section 9.2.2). Channels equate messages in LCC.

Even though differences exist between LCC and π-calculus: e.g. LCC constraints have no direct counterpart in π-calculus, it is instructive to view LCC from a π-calculus perspective. An LCC protocol can thus be seen as a composed process with subprocesses (roles) running sequentially or in parallel. Each subprocess can further be broken down into subsubprocesses also running sequentially or in parallel (with choice as a third option).

4.4 LCC and the SOA Challenges

The main obstacle to evaluate LCC with respect to the SOA service challenges is the fact that LCC does not have a service perspective. An often used approach is to view each agent as a service. That is, each role in an LCC protocol corresponds to a service [Wal07] p. 210. The inclusion of a client role is a consequence of LCC's interaction perspective and does not fit well into the usual service view, but is named 'client service' in [Wal07] p. 210. The SOA challenges will be discussed from this analogy. The term, subservices, will be used for non-client roles when more than one other independent[11] role besides the client role exists in the protocol.

The Challenge of Discovery By inspection of the LCC syntax, it should be clear that the role types present the only information readily available for discovery. A role type could perhaps equate the type of service the protocol deals with. Which one is the right one to choose?

[11] An independent role is taken to be a role which is not a subrole of some other role.

The client role type is often just indicating that this is the role to be taken on by the client or customer. It is, on the other hand, not possible to identify one other role which the client communicates with as it might interact with several other roles. Thus, it seems that no single role type represents the type of service and it must be concluded that LCC provides no means of service (or protocol) classification. The other information requirements of discovery cannot be satisfied without reasoning about extraction of information from LCC protocols. This will be deferred to section 6. For the discovery challenge we must then conclude that LCC has no means to readily meet this challenge.

The Challenge of Composition LCC has a formally well grounded, yet simple, way of describing interactions between client, services and subservices. Inherent in the interaction model are both a control flow and a data flow description.

Consistency between preconditions and effects of subservices does not apply to LCC as it has no notion of preconditions and effects of roles, but only constraints (see section 6.3 for a more detailed discussion).

LCC is an executable description of service interaction and provides glue-logic through constraints to facilitate the interaction between client and subservices. This is an important advantage of LCC.

LCC has no notion of exceptions, but, like OWL-S, LCC provides choices in the code that can be used to make protocols robust.

Concluding on the relationship between LCC and the SOA challenges, LCC has no support for the discovery challenge. However, LCC seems to provide some neat techniques to the challenge of composition, although LCC has no proper notion of services.

5 LCC and OWL-S

In section 4.4 it was concluded that LCC only concerns itself with the SOA challenge of composition. Also, in section 3.4 it was concluded that it is the ServiceModel of the OWL-S ontology which attempts to meet the challenge of composition. It should be clear from these conclusions that LCC can be juxtaposed solely with the ServiceModel and not the rest of the OWL-S ontology. The rest of OWL-S addresses other issues such as discovery and invocation which LCC does not address.

5.1 The OWL-S ServiceModel and Process Calculus

The process calculus foundation of LCC was briefly discussed in section 4.3. If the OWL-S ServiceModel could also be understood in terms of process calculus, this might ease the comparison of LCC and OWL-S. In [RKM04] it is mentioned that the service model indeed can be adapted to π-calculus. It turns out, however, that a detailed investigation of the service model and process calculus is out of the scope of this project. The Cashew project, [NFPH5], has at its aim exactly to provide a formal semantics (i.e. process calculus) of Web Services based on the OWL-S ontology. [NFH05] shows how translation from the OWL-S service model to a derived formal language and further on to a process calculus which is an extension of π-calculus can be achieved. The intermediate language is required as some minor issues (primarily regarding the data flow part) of the service model seem to cause difficulties. In the following, these technicalities will not be mentioned.

5.2 LCC and the ServiceModel

The service model describes the interaction between atomic processes in a hierarchical way. Atomic processes correspond to encapsulated entities in π-calculus, like the CAR in the mobile example in [Mil91]. It should be noted that the service model's hierarchical approach is not in contrast with π-calculus. In OWL-S, layers of control constructs explain how control of the overall process flows. In a way this can be viewed as if the service model describes possible interaction pathways in a tree structure. It is interesting to note that this form of description resembles what is known as orchestration[12]. Orchestration is described in [LM07] as a work flow description from the view of a single participant.

There is no straightforward way to identify atomic processes in LCC. Possible suggestions are: roles and 'perform-leaves' (cf. section 4.1). One of the simplest 'perform-leaves' is the reception of a message. It is hard to see how this construct provide any abstraction of interest. Roles seem more promising. Taking roles to correspond to atomic processes, LCC's simple structure of all atomic processes as a bag of parallel processes is in sharp contrast to the orchestration structure of the service model. The interaction of the atomic processes is, nevertheless, not less complicated described in LCC. But the information resides in a description of how each atomic process interacts with the other atomic processes. Again, this view has been recognised as a general scheme in the SOC-community[13] and is denoted choreography. [BGG+05] presents

[12]This is backed up by the fact that [SH05] places the OWL-S service model at an orchestration level in its figure 1.1.

[13]SOC is an acronym for Service-Oriented Computing, see [SH05] for a nice introduction.

a choreography language to:

> ... "describe the behaviour of a system by defining the involved roles and their interactions" ... [BGG+05] section 1.

One concrete consequence of the difference between LCC's choreography view and OWL-S' orchestration view is the client subprocess in LCC protocols which is not present in the service model. OWL-S models a single process, the service process, whereas LCC models the interaction between participants, including the client. The interaction with the client in OWL-S is described by the interface to the composed service through the service model and the service grounding[14].

Another consequence of OWL-S' orchestration approach might be the issue of the Produce-construct (cf. 3.4). This can be regarded as a work-around to overcome interaction deficiencies of orchestration approaches compared to choreography approaches.

5.2.1 Orchestration vs. Choreography

It is out of the scope of this project to go into details on the relationship between orchestration models and choreography models. In fact, the relationship is a heated topic in the Web Service community at the time of writing this report. The orchestration scheme is used by big standardisation bodies like W3C (OWL-S service model) and OASIS (WS-BPEL [AAA+06]); choreography is represented by W3C's WS-CDL [KBR+04]. The debate between the choreography approach of WS-CDL on one side and the orchestration approach of WS-BPEL on the other side is partly due to a poor understanding of the relationship between orchestration and choreography. [BGG+05] presents a recent approach to provide a formal language in process calculus of the both and combine them.

5.3 The SOA Challenge of Composition

LCC and the OWL-S service model has so far been compared in a theoretical way. It is instructive to also compare the two with respect to requirements of the composition SOA challenge of section 1.1. From sections 3.4 and 4.4 we note that there are three important differences:

- The ServiceModel has a notion of services which LCC does not (and it thereby also has no notion of preconditions and effects).
- LCC provides an executable specification where the ServiceModel is only descriptive.
- LCC provides glue-logic to aid the interaction between entities; the OWL-S ServiceModel does not.

5.4 Conclusions on LCC and OWL-S

It has been argued that LCC corresponds to the service model of OWL-S. The other parts of OWL-S provides extra functionality for the SOA challenges which LCC does not possess. LCC

[14]The interaction information residing in the service model is duplicated in the service profile.

and the ServiceModel has a common foundation in the theory of process calculus. However, process calculus supports two different methodologies for describing communicating processes: a single process view termed orchestration, and an interaction view with multiple participants termed choreography. The relationship between these two views is currently a lively topic in the Web Service community.

With respect to the SOA requirements for composition, LCC seems superior to OWL-S except from the important fact that it does not conform to the concept of services. This remark might very well generalise to the relationship between orchestration and choreography.

6 Extending LCC

So far, this report has been focused on the differences between LCC and OWL-S. We now turn to the motivating aim of bringing the two together to take advantage of the strengths of each. This will require an extension of the LCC syntax.

It was concluded in section 4.4 that LCC is a strong tool for composition, but has no notion of services and in particular does not meet the challenge of discovery. The aim is to extend LCC so as to meet the challenge of discovery. First, a more general analysis of the relationship between LCC and services will be presented. Then a notion of service, although somewhat different than usual service notions, will be developed for LCC. Requirements for an extended syntax which can meet the challenge of discovery will be specified. Finally, an extended syntax for LCC, fulfilling the requirements, is proposed.

6.1 MASs and Services

A critical difference between MASs and services lies in the way agents communicate compared to services. Agent communication is long-lived and more complex, thus the use of protocols instead of interface specifications [Wal07] section 7.4.1. [Wal07] section 7.4.2 lists three ways to overcome this gap:

- The Agent Stub approach takes the rationale that complex interaction is only possible through advanced agent communication. It thus keeps the idea of agents rather than agent services communicating and then equips each agent with a service interface to the outside. Viewing this the other way around, services must have an agent stub to interact with other services. This clearly breaks the SOA as interoperability depends on agent interfaces rather than service descriptions.

- The Gateway approach also acknowledges the fact that agent communication is too complex to be catered for through services. The approach introduces a gateway between agent systems and services, keeping the two separate. MASs have access to services through the gateway treating it as an agent. Services can access MASs through the gateway treating it as a service. This approach is the one used in WSIG (Web Service Integration Gateway) for JADE (The Java Agent DEvelopment Framework) [GBR05] and [SDF06]. The main disadvantage of the approach is its centralised architecture. The gateway might suffer from downtime or overload.

- The Service approach considers, like the two other approaches, advanced communication capabilities of agents necessary. Agents consist of a service interface and one or more proxies. The proxies are used for communication between agents and the service interface ensures SOA compliance. The exact relationship between proxies, agent stubs and gateways is vague. However, it seems proxies have the advantage over gateways that they are of decentralised nature.

  The difference between the agent stub approach and the service approach probably lies in the way agent communication occurs. In the agent stub approach *all* services were equipped with an agent stub to enable interaction. In the service approach only services which need to take part in complex communication with more than just two peers, make

use of proxies. Their proxies join in closed MASs using protocols. From the outside, these MASs encapsulate agent communication and the assertion is that the SOA is then intact. An objection to this assertion is that, if communication between more than two peers is the norm, all services must be equipped with a proxy and the approach seems to degrade to the Agent Stub approach.

Even though none of the above approaches seem completely satisfactory, one will have to be chosen to combine MASs and Web Services. The Service approach seems to be superior and will be assumed from now on. The architecture is shown in figure 9. As shown in the architecture, a

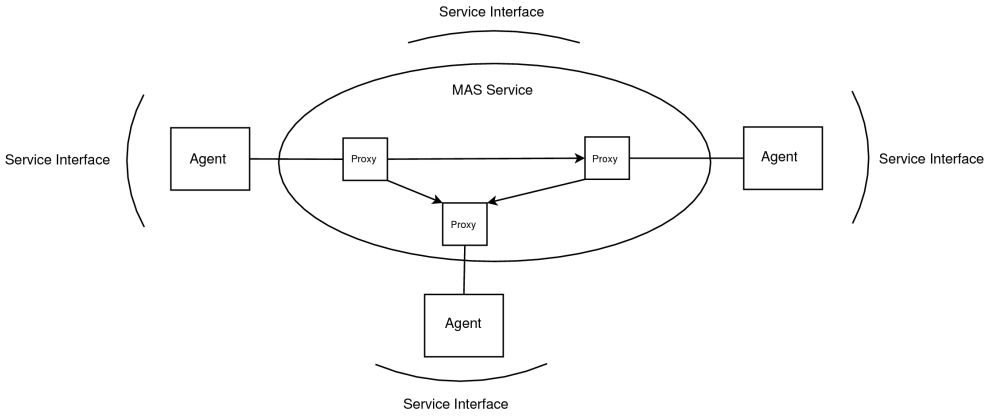

Figure 9: MASs and Services

Close to figure 7.8, p. 208 in [Wal07].

composition of proxies is named a MAS Service. The idea is that not only do each agent present a Web Service, but the composition of agent proxies also present a Web Service: a composed Web Service. This scheme, however, requires that compositions of agent proxies provide a service interface to comply with the SOA. The service interface to MAS Services will enable the important challenge of discovery of composed Web Services. Requirements that enable a MAS Service interface is basically what the sought extension of LCC is about.

6.1.1 Composition of MAS Services

An obvious question which comes to mind when regarding the proposed architecture is whether composition of MAS Services is possible. The MAS Service interface which is sought for in the proposed extension is one half that is needed for this meta-composition. The other half consists of a means to refer to MAS Services from within MAS Services, i.e. within LCC protocols. A need for these references is also recognised in [Wal07] section 7.4.3. Here constraints are referred to as Web Services. If constraints comply to MAS Service interfaces they can act as references to other MAS Services. There are, however, some issues attached to this model:

- The constraint solver needs to be extended to not only solve constraints locally, but also search for and execute MAS Services which match constraints not locally solvable.
- Most important is the fact, though, that the meta-composition makes no use of interaction models. The meta-composition can thus only be applied where simple RPC-

mechanisms are adequate. LCC's justification in the first place was to assist in service composition because RPC-mechanisms are usually *not* adequate.

6.2 LCC and Services

It is not a new thing to view LCC protocols as one composed service. The idea of LCC as a composed MAS Service comes, as mentioned above, from [Wal07] section 7.4.3. The OpenKnowledge project, [ope], introduces OpenKnowledge Components, OKCs, which are encapsulated LCC protocols [DKK+]. The encapsulation consists of three parts: the LCC protocol itself, a semantic description of what service the OKC provides, and some code which executes the OKC picking the correct client role. The OKCs take no advantage of information already present in the protocol code. For instance, parameters of the service described in the semantic description of the OKC are not mapped to any parts of the protocol code.

The disadvantages of the OKC approach are: 1) advantage is not taken of information possibly already present in the protocol code; 2) it ignores the inherent symmetry of LCC protocols as opposed to SOAs. SOAs usually have the notion of a requester and a provider; LCC regards all participants equal. With OKC's asymmetric view, an OKC is needed for each independent role[15] in a protocol. The hypotheses of this project are:

1. Information useful for service description can be extracted from the protocol code.
2. The symmetric nature of LCC can be conserved.

6.2.1 LCC Methods

The LCC service view of this project is inspired by [Wal07]'s "client service" concept and Walton's view of LCC roles as methods [Wal07] figure 7.10. So far LCC clauses have been recognised as subservices. The key objection to this view is the client role. In this report a double view of each role will be promoted. A role can either be seen as a subservice as hitherto *or* as a method to achieve some task. The idea is that each participating agent sees the roles it plays as methods and other roles it interacts with as subservices. This double view basically enables an asymmetric view of LCC protocols.

With the notion of methods and by annotating LCC roles, there will be no need to encapsulate protocols. To find a composed service protocol, a user simply searches protocols for annotated methods matching the task to be achieved. As all roles equally well can be viewed as methods and because method information is annotated in each role, the symmetry of LCC is conserved. With each method in mind the double view can thus be regarded as giving a symmetric set of asymmetric views.

There is a subtle difference between the concept of a method and that of a service. A method is part of a composed service description. It represents an entry point to the composed service. A method is thus more like an executable interface to a service. It is hard to distinguish between the interface of a service and the service itself. In the rest of this report the following terminology will be strived for:

- A method is an executable interface to a service part of a LCC protocol.

[15] See section 4.4 for a definition of independent role.

- A method description corresponds to a service interface.
- A task is a description (complying with method descriptions) of a service a requester would like to invoke.

Sometimes the distinction between a service and its interface (i.e. method description) will not matter and the two terms will be used interchangeably.

Finally, it is worth noting that a protocol with only one subservice (one other role than the method role) is not really a composed a service. LCC is thus capable of representing both non-composed services (equal to the atomic processes of OWL-S) as well as composed services.

6.3 Method Annotation

The method annotation should support extraction of a satisfactory service interface (method description). It has been decided that a satisfactory service interface of LCC protocols must enable discovery, cf. section 6. The features that enable discovery will simply be taken from the analysis of OWL-S section 3. It was concluded that the ServiceProfile is the datastructure of OWL-S which enables discovery, cf. section 3.1. The service profile was divided into: who provides the service, an interface of the service, and what it does. A number of simplifications will be made compared to the OWL-S approach. Because discovery of a service and discovery of a provider will be divided into two separate steps, cf. section 1.1, the "who-provides-the-service" will not be part of LCC service interfaces. To indicate what the service does, a simple service type will be adopted instead of the comprehensive category and classification system of OWL-S[16]. Finally, a service parameter to enable choice between similar services is not considered. This leaves us with the following information pieces:

- Service type
- Service preconditions
- Service effects
- Service results
- Service parameters

Each of these requirements for information that must be extractable from an annotated method will be considered in turn. Any annotation requirements to methods will be specified for each case.

6.3.1 Method Type

An obvious suggestion for the method type is the role type of the role which the method belongs to. However, it is important to note that methods are a new concept and different from that of roles. Role types in LCC are concerned with the type of peer that is expected to play the role rather than what the role achieves. We thus need another kind of type on roles which reflects what roles accomplish. The two different kinds of typing will be referred to as method typing and peer typing of roles.

[16]It is interesting to note that the category and classification system seems to have been simplified in the 1.2-version of OWL-S compared to the 1.1-version.

The need for peer typing is not clear and there are thus two ways to achieve method typing: change peer typing into method typing; or extend the syntax with an extra typing feature. The second option is chosen to keep a conservative extension of LCC and also to avoid term confusion.

Summarising, all roles should be assigned a second kind of role type which reflects what they accomplish rather than who should play the role.

$$\textit{Requirement}: \quad \textit{Typing of methods} \tag{1}$$

6.3.2 Method Preconditions

[SPW+04] section 3 defines preconditions as "things that must be true of the world in order for an agent to execute a service." It was noticed in section 3.3.1 that preconditions of services are hard to recognise when building Web Services and indeed they are in LCC as well. One suggestion is the constraints of protocols. There are, however, important differences:

- Constraints are used to control the interaction flow. I.e. it is the intention that some constraints should at certain times be true and at other times false. In both cases execution of the protocol should succeed.
- Constraints are determined at run-time.

As mentioned in 3.3.1 preconditions have not yet, in my opinion, been examined satisfactory in terms of Web Services. So instead of going with the tide and introducing planning preconditions to LCC, we will develop a notion of preconditions more natural to LCC. To execute a method it is required of an agent to *support* all constraints belonging to that method. By supporting a constraint the agent is expected to know *how* to solve the constraint. Whether it can be satisfied or not depends on the run-time values of the constraint's arguments. The check of support should be handled by a constraint manager as envisioned in [DKK+]. Such method preconditions require annotation of method constraints. It has been decided that the type of constraint is needed as well as the types of the constraint's arguments. As LCC is not typed, the following two requirements apply:

$$\textit{Requirement}: \quad \textit{Typing of constraints} \tag{2}$$

$$\textit{Requirement}: \quad \textit{Typing of constraint arguments} \tag{3}$$

A note should be made on the requirement that *all* constraints of the method role should be supported.

Firstly, roles contain choices and so not all constraints need to be satisfied or even supported for a succesful result to be achieved. This complication will simply be ignored with the rationale that an agent cannot know beforehand how the interaction will proceed. This is by no means a definitive argument. A counter-argument is that an agent might have searched for a method with a specific result and thereby is only interested in fulfilling the constraints attached to this result. A more sophisticated solution would be to have sets of constraints to be supported. One set for each execution path. Notice that this solution might also be interesting for the general relationship between result-conditions and preconditions of Web Services and in planning in general.

Secondly, constraints were divided into proaction and reaction constraints in section 4.1.1. One might suggest that only proaction constraints need to be supported. However, as mentioned in 4.1.1, a peer needs to satisfy both types of constraints of a role in order to execute the role successfully - they merely present a difference in timing.

Finally, all constraints of a method must include both constraints of the method role and any constraints of roles to which a shift might occur from the method role. These roles are denoted subroles. Constraints of both toprole and subroles will be denoted deep constraints.

6.3.3 Method Effects

[SPW+04] p. 4 defines service effects as: "physical side-effects that execution of a Web-service has on the world.". Just as it was the case for preconditions, this definition seems more suitable for planning domains with a closed world with definite many objects than for Web Services being carried out in the real world. The problem of acting in a real world is that possible specifications of action effects are hard to enforce. The MAS communication language FIPA-ACL, [Woo02] p. 175, is grounded on a semantics based on notions of beliefs, uncertain beliefs, and desires. ACL thus specifies what effects certain interactions should have. FIPA-ACL does *not*, however, provide any means to enforce this semantics. Neither does OWL-S have any means to enforce precondition and effect proclamations in service profiles.

The issue of effects has also been studied in another field, namely computer science. One important cause to the difficulty of reasoning about computer programs is that they may have side-effects in most programming languages. Pure functional and logical languages provide an alternative, but most of these languages also provide ways to circumvent the restrictions that ensure avoidance of side-effects which are often necessary in real world applications (for instance where IO is needed).

The design of LCC acknowledges the fact that it is hard to enforce effect guarantees in distributed systems. LCC protocols therefore assume nothing about how the participating agents solve constraints, send messages, receive messages, and what side-effects these actions may have.

This view that participants are autonomous agents whose actions cannot be guaranteed by protocol specifictations is so fundamental to LCC that effects will *not* be annotated in LCC methods.

6.3.4 Method Results

Results consist of conditions, outputs and effects. Results may turn out to be at least as important for the Web Service challenges as preconditions and effects of services. For instance, a client might be more interested in searching for a service with a result with a specific effect than searching for some overall effect. It then also needs to check if it can fulfil the conditions of this result.

Because neither preconditions (in their normal sense) nor effects are annotated in the extended version of LCC, results will not either.

6.3.5 Method Parameters

An interesting observation is made in [SPW+04] p. 4, that inputs and outputs of a service can be regarded, respectively, as knowledge preconditions and effects of a service. This view fits well with LCC, at least concerning inputs. It is easy to include the LCC 'perform-leaf':

```
null ← inputs(I1,...,In)
```

in the beginning of each role. An output-constraint needs to be added as a consequence of the last 'perform-leaf' of each choice-branch[17]. These knowledge constraints will, however, be considered constraints and distinguished from ('real') method parameters.

Another suggestion for parameters in LCC protocols are the messages of the method role. Recall that the other roles besides the method role (and its subroles) are considered subservices, cf. section 6.2. It is therefore straightforward to think of the communication between the method and the subservices as inputs and outputs of the composed service (the subservices combined) like in [Rob04a]. The messages dispatched by the method role would then equate inputs and received messages would equate outputs. There are two issues concerning this model. Firstly, variables sent by the method role might originate from constraints of the method role which we termed knowledge preconditions and distinguished from 'real' inputs. Secondly, message variables sent to one subservice from the method role might originate from another subservice (using the method role simply as a gateway). This reflects the advanced communication scheme LCC provides compared to those of SOAs.

What we are interested in are the parameters an agent has to provide and will get by taken on the method role (besides those provided and received through constraints). Such parameters are actually already present in LCC. All roles can be supplemented with arguments. The arguments of method roles will also be considered parameters of the method, i.e. of the service interface.

The division of parameters into inputs and outputs is rather hard in LCC because it has a relational nature. It has been decided to provide a conservative extension and the parameters will not be divided into the two classes. The single important information about parameters are their types, section 3.3.1. LCC is not typed, thus the requirement for method parameters is:

$$\textit{Requirement}: \quad \textit{Typing of role arguments} \tag{4}$$

Note that arguments of all roles in an LCC protocol should be annotated (to allow discovery of all methods in the protocol). However, parameters of any given method only amounts to the parameters of its toprole and *not* its subroles. The only parameters connecting an agent and an LCC protocol are the role arguments of the toprole the agent plays.

6.4 Extension of Syntax

All the annotation requirements in the above sections consist of LCC typing. The concrete extension of the LCC syntax to enable the required typing will now be proposed.

[17] This is strictly not possible if the last perform is a `null`-constraint.

First, the placement and nature of the type annotation has to be decided. For instance, service interface information of a method could either all be annotated in the role head of the method clause or in both the role head and the body of the clause. In this project a simple and robust annotation extension is sought. As a design guideline this means that information available further 'down' the protocol structure should not be pushed up. An example is a constraint type in a clause. This type should not be annotated in the clause head, but rather in the body in close contact with the constraint.

This is a design choice. Advantages are robustness, simplification, but perhaps most important the absence of duplication of information (e.g. information of what constraint some argument type belongs to). Avoiding duplication of information is a well-known design principle in computer science. The main disadvantage of the design choice is that it complicates possible extraction procedures.

Following the guideline of keeping annotation close to annotated elements, the syntax in figure 10 is proposed. As shown annotations consist of a constant and a annotation 'connector', `--`. The syntax will be referred to as XLCC. Notice that annotation can be empty, i.e. non-

$Model$	::=	$\{\ Clause, \ldots\ \}$
$Clause$	::=	$Annotation\ Role :: Perform$
$Role$	::=	`a(`$Type$, ID`)`
$Perform$	::=	$Role \mid Role \leftarrow Constraint \mid Message \mid$ `null` $\leftarrow Constraint \mid$ $Perform$ `then` $Perform \mid Perform$ `or` $Perform$
$Message$	::=	$M \Rightarrow Role \mid M \Rightarrow Role \leftarrow Constraint \mid$ $M \Leftarrow Role \mid Constraint \leftarrow M \Leftarrow Role$
$Constraint$	::=	$Annotation\ Constant \mid Annotation\ Constant(Annotation\ Term, \ldots) \mid$ $\neg Constraint \mid Constraint \wedge Constraint \mid Constraint \vee Constraint$
M	::=	$Term$
$Type$	::=	$Constant \mid Constant(Annotation\ Term, \ldots)$
ID	::=	$Constant \mid Variable$
$Annotation$	::=	$Constant$ `--` $\mid [EMPTY]$
$Term$	::=	$Constant \mid Variable \mid Constant(Term, \ldots)$
$Constant$	::=	Lower case alphanumeric sequence possibly followed by mixed case alphanumeric sequence
$Variable$	::=	Upper case alphanumeric sequence possibly followed by mixed case alphanumeric sequence

Figure 10: Extended Syntax of LCC Protocols

existent.

6.4.1 XLCC Protocol Example

Figure 11 is an example of an annotated LCC protocol. It is a partly annotated version of the trade LCC protocol in section 4.1 (figure 7).

First, notice that the client role has been slightly altered. Another argument to the role has been added: *Receipt*. This makes sense since the client role is now regarded as a method with role arguments acting as method parameters. The *Receipt* indeed seems like an obvious choice for a method parameter (an output of the method). It is interesting to note that XLCC

protocols provide several options for design. Another choice for the XLCC protocol in figure 11 could be to *not* alter the original protocol at all; this is also legal according to the extended syntax. A third choice could be to add the consequence $save(Receipt)$ to the reception of the receipt-message.

Besides the alteration described above and the annotation, the protocol is identical to the original one from section 4.1. The annotation consists of method type and method parameter types for the *client*-role. Also, the constraints of this method have been annotated. The option of leaving out annotation has been chosen for the *shop*-role.

With the annotation chosen it would be possible to extract a service interface of the *buy*-method. The requester would also be able to check whether he supports the constraints used by the method.

This is a rather simple example of a one-view service. An extended example with more than one annotated method is described in section 9.2.2.

It is interesting to note the difference between method types and peer types as anticipated in section 6.3.1. The first role has peer type, *client*, but method type *buy*. The role accomplishes the task of buying and is expected to be played by a client peer.

Protocols written with the XLCC syntax will be referred to as xprotocols.

$$
\begin{array}{l}
\{ \\
\quad buy \texttt{ -- } a(client(product \texttt{ -- } ProductCode, receipt \texttt{ -- } Receipt), C) :: \\
\qquad buy(ProductCode, CreditCard) \Rightarrow a(shop, S) \leftarrow \\
\qquad\qquad\qquad getcc \texttt{ -- } cc(cc \texttt{ -- } CreditCard) \wedge lookup \texttt{ -- } lookup(shop \texttt{ -- } S)\ then \\
\qquad receipt(Receipt) \Leftarrow a(shop, S). \\
\quad , \\
\quad a(shop, S) :: \\
\qquad buy(ProductCode, CreditCard) \Leftarrow a(client(\_, \_), C)\ then \\
\qquad receipt(Receipt) \Rightarrow a(client(\_, \_), C) \leftarrow enough\_credit(CreditCard, ProductCode) \wedge \\
\qquad\qquad\qquad complete\_order(ProductCode, CreditCard, Receipt). \\
\}
\end{array}
$$

Figure 11: Example of Annotation in LCC Protocols

---

7 Design

Section 6 described an extension of LCC which allows extraction of method descriptions (i.e. service descriptions) from XLCC protocols. These method descriptions include type of method, parameter types and constraint descriptions.

The aim is now to construct a system which implements extraction of protocol information and is capabable of LCC service discovery, i.e. method matching against requested tasks. The system is divided into two layers: a Protocol Layer and a Discovery Layer. The overall architecture is shown in figure 12[18]. The Protocol Layer basically handles the XLCC code.

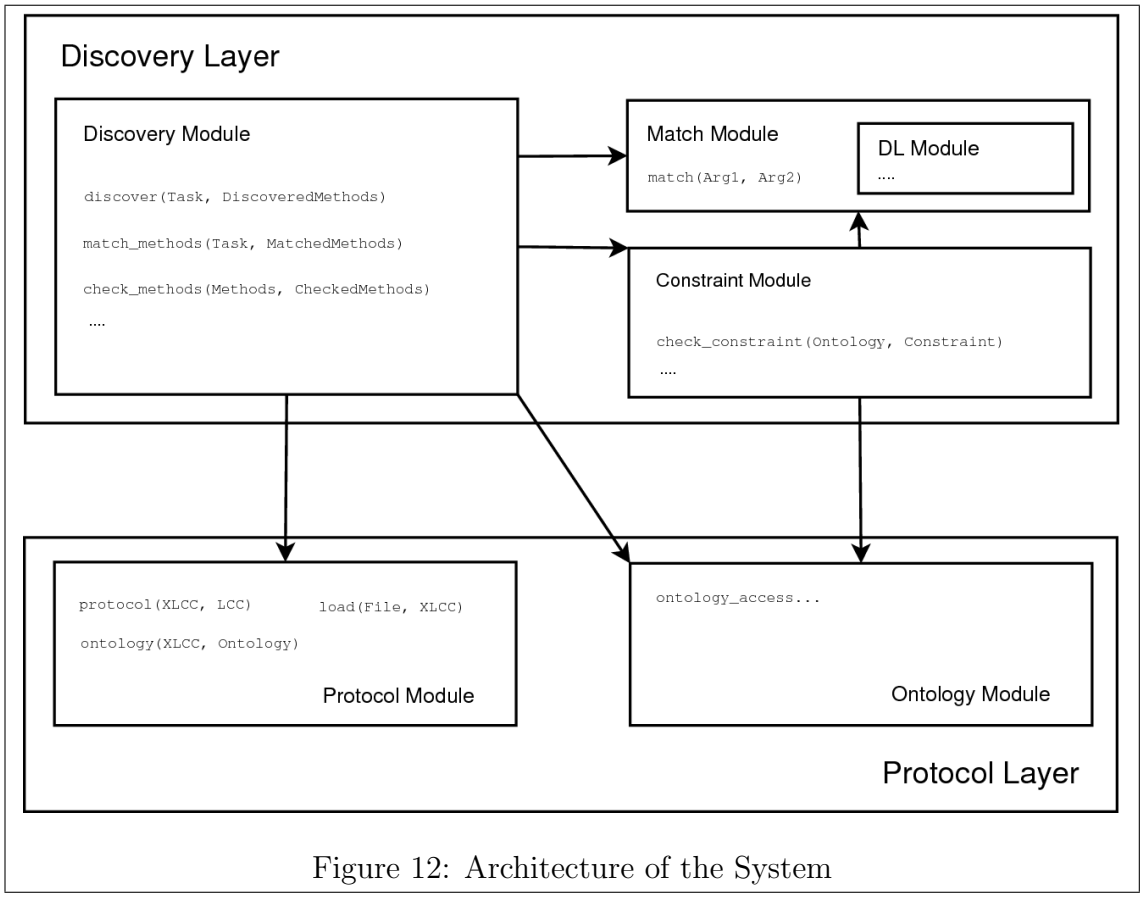

Figure 12: Architecture of the System

The Protocol Module loads XLCC files and extracts protocol information into a datastructure dubbed the LCC Ontology (see section 8.1). The module also takes care of conversion from XLCC to LCC (e.g. necessary for running XLCC protocols). The Ontology Module, also part of the Protocol Layer, provides access to the ontology datastructure, hiding its internal representation away.

The more interesting Discovery Layer is discussed next.

[18]Besides the modules shown in the architecture, a Misc Module has also been implemented to cater for miscellaneous tasks such as IO.

7.1 The Discovery Layer

The Discovery Layer implements discovery of LCC services. Its main entry point is `discover(Task, DiscoveredMethods)`. The discovery process of section 1.1 is presented in detail below [FGY07]:

- Find a suitable method description with respect to a task description
- Check support of method constraints
- Find providers for the subservices in the LCC protocol which the method belongs to
- Providers accept/reject subservice descriptions (probably role types)
- Providers check constraint support for the given role

The `discover`-entry will first match task descriptions against a set of method descriptions loaded from a local file registry of protocols (using the `load`-entry of the Protocol Module). Matched methods will be checked for support of their constraints. For the check of supported constraints, a Constraint Module is used. This follows the OK architecture of a constraint manager [DKK+][19]. Matched and checked methods are returned to the user. Thus, steps three, four, and five are *not* implemented.

Note that the Discovery Layer has a local design. The intention is that the Discovery Layer should be part of a larger LCC system which also includes an LCC intepreter and the Protocol Layer. Each peer who wishes to take part in LCC interactions would adopt an LCC system. This corresponds to the vision of the OK project [DKK+]. However, discovery is more likely to occur in a distributed manner (perhaps centralised) in real settings. With an interface also supporting only matching (and not checking) of methods, the Discovery Module could be extended to also work in a distributed setting. The matching-only entry would be used for discovery requests from other peers. The Discovery Module would need an extension to send discovery requests to other peers (and check results). An obvious suggestion for the discovery communication between peers would be to use LCC protocols. Notice that a means to specify a search depth would be necessary.

7.1.1 Matching of Tasks and Methods

The task description is obviously central for matching. This description consists of term sets in [SvHK+]. It is not revealed what these term sets represent. From the interface in [DKK+] it seems that the reason is that the discovery part of OK is not fully developed; [DKK+] states that "For now, the description is a set of keywords...". In this project a task description respects the structure of method descriptions. This enables a matching algorithm resembling the one in [PKPS02].

The matching algorithm used for OWL-S[20] descriptions in [PKPS02] uses only the parameter types of the service descriptions (and not service classification). It is checked whether the input types of the requested service matches the service description input types and whether the output types of the service description matches the requested service output types [PKPS02] section 3.1. Notice the asymmetry. 'Matches' is divided into several classes (in decreasing

[19]The constraint manager in [DKK+] actually handles the process of satisfying constraints during protocol interpretation. Nevertheless, the two are closely related.

[20]At that time DAML-S.

match-order): equals, superclass of, subsumes, subsumed by. Degrees of match can thus be determined. Subsumption queries are carried out using DL reasoning on the given concepts and some accompanying concept ontologies.

In this project, only one match-class will be considered: subsumes. When elaborating on this match-type it is interesting to note the asymmetry of the algorithm in [PKPS02]. Task inputs should subsume method inputs and method outputs should subsume task outputs. The idea is obvious: the service should be able to work with task inputs more specific than its advertised ones; also a method with outputs more specific than task outputs should not require any more work. This reasoning is not clear-cut, though. For instance, it can be imagined that a user is not interested in more specific outputs. If the user requests a shop list and the service output is a webshop list, the user may have lost results because of the restriction. This is most likely one of the reasons to include the opposite subsumption as a (lower-rated) match-type in [PKPS02].

In section 6.3.5 it was decided not to distinguish between inputs and outputs. The chosen subsumes match-type must therefore apply in the same way to all parameters. A match will occur when task parameters are *subsumed by* method parameters. Furthermore, the match algorithm in this project will make use of method types. A match between a task and a method also requires the task type to be *subsumed by* the method type. A structure with a type and some arguments also with types will be referred to as a *type structure*.

Note that the algorithm described above requires type inference capable of subsumption queries (implemented by the DL Module) and use of type ontologies. Type ontologies will consist of hierarchical assertions like TBox statements in DL (see section 8.2.2 for more on this).

We have now come a little closer to a semantics, i.e. interpretation, of the type annotations of XLCC protocols. With the choice of the algorithm above, XLCC types seem to correspond to the ranges of OWL-S type properties. The exact semantics, however, is still dependent on the implementation and is deferred to section 8 where subsumption support and type ontologies are also discussed.

7.1.2 Matching of Constraints and Constraint Solvers

Constraints are also type structures like methods, i.e. they also have a type and arguments with types. Constraint solvers are expected to have the same structure. Therefore, the same algorithm for matching methods and tasks can be used for constraints and constraint solvers. A constraint is supported if it is *subsumed by* some constraint solver in the solver registry.

It was discussed in section 6.1.1 that MAS Services and thereby LCC protocols could be composed into meta-compositions through method resolvement of constraints. A couple of issues were also discussed. In this project the Constraint Module will only consult a local file registry of constraint solvers. There exists no restriction on whether these are local solvers or in fact LCC methods. Recall that the process of satisfying constraints is handled by the LCC interpreter (perhaps using a constraint manager) which is not dealt with in this project. If LCC methods were allowed as constraint solvers, one might suggest that the Constraint Module should not only consult a local registry, but if search in this failed, try to *discover* an LCC method to solve the constraint. The scenario now becomes rather complex and reminds

of on-the-fly planning of composed services. Search depth is just one issue of such a planning scheme.

7.1.3 Advanced Matching Algorithms

The inclusion of method and constraint types (besides their parameter types) opens up for more advanced matching algorithms. Especially context matching (a.k.a. semantic matching) is an obvious extension. Semantic matching basically uses context to aid type inference. For instance, the arguments of constraints alone hold less information than the arguments in combination with their constraint type [FGY07].

Advanced matching algorithms are well known in the SW community because of the need to align or map ontology concepts. Mapping techniques include subsumption matching (used here), semantic matching (a.k.a. structural matching), string based matching (e.g. string distance), language based matching (e.g. WordNet), and use of upper ontologies[21] [SE04], [RB01], [KS05].

[21] Would, in our case, amount to external type ontologies, cf. section 8.2.2.

8 Implementation

The logic programming (LP) language, Prolog[22], has been chosen for implementation. The reasons for this choice was first and foremost that a relatively simple Prolog interpreter already exists for LCC protocols, see appendix D. Secondly, the relational nature of LCC and its close relationship to logic programming in terms of, e.g. constraints, makes Prolog an obvious choice. Finally, as partly revealed by the Discovery Layer design, cf. section 7.1, DL has been chosen for type inference. Prolog has been shown to support at least some DL inference, [GHVD03], and a DL reasoner implemented in Prolog is available from a previous Master's Thesis [Her06].

Space does not allow a description of the entire system. Instead, focus will be on the extracted LCC Ontology and the Match Module.

8.1 The LCC Ontology

The LCC Ontology holds the information which is extracted from XLCC protocols, cf. section 7. The reasoning that can be done (such as matching) is thus dependent on this datastructure and its format. It has been implemented as a compound Prolog term (see figure 13). The two datastructures, `method` and `constraint`, both respect the type structure mentioned in section 7.1.1 and 7.1.2. Another option for the ontology would have been an DL instance. Not all DL

```
Ontology   ::= ontology(methods([Method,...]))
Method     ::= method(protocol(XLCC), name(Name), type(Type), args([Arg,...]),
                      submethods([Submethod,...]), constraints([Constraint,...]))
Arg        ::= arg(name(Name), type(Type))
Submethod  ::= submethod(name(Name), type(Type), args([Arg,...]))
Constraint ::= constraint(name(Name), type(Type), args([Arg,...]))
Name       ::= PrologVariable | PrologTerm
Type       ::= PrologVariable | PrologConstant
```

Figure 13: Structure of the LCC Ontology

is expressible in Prolog so the DL instance would need some encoding (for instance an OWL XML string). The main advantage would be to have a format which is a standard and can be used by many other SW applications. The reasons *not* to choose such a format are:

- The ontology is used by numerous parts of the system not concerned directly with DL. Using a Prolog datastructure avoids repeated conversion between a DL format and Prolog terms.
- As mentioned in section 8.2.1, the DL reasoner used does not support DL individuals. Each instance of an LCC protocol would amount to an LCC Ontology individual, but this cannot be expressed in the DL format provided by the DL reasoner. For scope reasons, extensions to the DL reasoner format or creation of a new format has not been considered.

[22]In particular, the SICStus system, [sic].

A conversion tool between the LCC Ontology and a DL description would enable SW-supported descriptions of LCC protocols and would therefore be attractive.

8.2 The Match Module

The match algorithm of the Match Module was described in terms of method and task matching in section 7.1.1. The algorithm matches *type structures* and therefore also applies to constraints and constraint solvers, cf. section 7.1.2. A requirement of this algorithm was support of type inference. Types are not a standard part of Prolog so reasoning about them needs consideration. Even though typing is not part of Prolog, type theory of Prolog and Logic Programming in general has a considerable history (compared to the history of LP and Prolog) [YS91]. Most type systems implemented for Prolog, like [HL94] and [SW92], use the notion of modules to reason about datastructure types and changes Prolog quite fundamentally [GH00] section 2. The requirement of this project is more restricted. Type inference is only required for type objects not part of the implemented system itself. Furthermore, subsumption queries and use of ontologies were mentioned as requirements for the proposed algorithm in section 7.1.1. Finally, we would also like to keep the option open for advanced matching of type structures, cf. section 7.1.3. These needs correspond very well with DL reasoning. What would be appropriate is a plugin match module which matches type structures using DL reasoning techniques (and perhaps mapping techniques as well in the future). This module corresponds to the Match Module of the present project. The module uses DL reasoning (through use of the DL Module).

The extended syntax of LCC proposed in section 6.4 is only a syntax, it mentions nothing about how types should be interpreted, i.e. their semantics. Specifically, a DL interpretation is supported, but not enforced. At this implementation stage we must, however, now decide on the semantics of XLCC protocol types:

$$XLCC\ types\ are\ to\ be\ interpreted\ as\ DL\ concept\ references \tag{5}$$

It is important to note that DL concept references are equated to atomic DL concepts in this project[23]. This scheme follows OWL-S' parameterType property whose range is anyURI. anyURI is a 'long' reference to a DL concept that can be reasoned about as a regular DL concept. The 'long' name feature is introduced to avoid ambiguity between concepts. The types used here support 'long' names, but they are not enforced. The issue of type ambiguity is not considered any further.

Types in protocols are equated to atomic DL concepts which are not necessarily primitive concepts (see [NBB+03]). Complex expressions can therefore be asserted about XLCC types in the accompanying type ontologies. An example is given in figure 14.

The Match Module could, strictly speaking, be implemented in another language than Prolog. However, a DL reasoner in Prolog is available ([Her06]) and it keeps things simpler to implement the module in the same language as the rest of the system. This brings us to the relationship between Prolog and DL. A complete discussion is out of scope, but the relationship should be discussed with respect to this project and the theoretical and practical limitations of

[23]Complex DL expressions would require a choice of DL encoding breaking the flexibility of the XLCC type syntax.

XLCC protocol:
{
buy -- $a(client(ProductCode, Receipt), C)$ ::
...
}

Type ontology:
$sell \equiv trade \sqcap give$
$buy \equiv trade \sqcap \neg sell$

Figure 14: Example of XLCC with Type Ontology

the DL reasoner used should be acknowledged.

8.2.1 Reasoning about DL in LP

The intersection between DL and LP has received great attention recently because of the Semantic Web. As mentioned in section 1.3, the underlying knowledge representation of the Semantic Web is DL. However, it is recognised that some knowledge is better represented using rules. Rules can be seen as LP clauses and LP theory therefore applies to theory on rules. The need for both DL and rules and thereby LP theory is thus apparent. [GHVD03] describes how a subpart of DL and LP can be translated forth and back. This translation enables reasoning about DL using LP inference and of LP using DL inference. The translatable subpart is denoted Description Logic Programs (DLP) and is only a subpart of the two formalisms. In particular, this restricts what can be translated from DL to LP. The restrictions are listed in section 4.1 of [GHVD03] in terms of requirements for translated LP rules:

- No use of equality (no representation of partial-functionality of properties and no representation of maximum cardinality).
- All variables must be universally quantified (no assertions of possibly non-existent individuals).
- Existentials may not occur in head (no representation of minimum cardinality).
- Negation may not occur inside head or body.

The above restrictions apply when DL is translated into LP. The reason to translate DL to LP is, as mentioned above, either to mix DL and rules or to use efficient LP-reasoning on DL. However, type hierarchies are not expected to include rules [PKPS02]; neither is efficiency an issue at this phase in development.

The DL reasoner of Herchenröder, [Her06], uses its own DL-format and a tableaux algorithm (as opposed to Prolog inference) to carry out DL reasoning on assertions in this DL-format. DL representation and reasoning in this project is thus limited by the format of this reasoner and its reasoning capabilities.

Limitations of the DL Reasoner The limitations of the reasoner used in this project are well defined as it is stated in [Her06] that the implemented reasoner complies to the $\mathcal{ALC}$-DL. A thorough examination of the different DL-types and their capabilities is out of scope[24], but some important limitations of the basic $\mathcal{ALC}$-DL are given below:

- No cardinality constraints
- No support of role subsumption (role hierarchies)
- No allowance of datatypes

In addition, the reasoner does not support use of individuals [Her06] p. 14.

The above limitations are in terms of the reasoning capabilities of the DL reasoner. It turns out that the format used by the reasoner also poses some restrictions. The most important restriction is that the format only accepts equivalence statements about concepts. This seems in contrast with the envisioned type hierarchies. Thomas Herchenröder was kind enough to refer me to [NBB+03]:

> "A TBox with inclusions (subsumptions) would not be "definitorial" (p.63). All statements of the ontology should be definitorial, i.e. expressed by equivalences (necessary and sufficient)." email from Herchenröder

It is recommended that expression of subsumption is made through equivalence normalisation, [NBB+03] section 2.2.2.5:

$a \sqsubseteq b$ should be normalised to:

$a \equiv b \sqcap a\_restriction$
$b \equiv a \sqcup b\_rest$

8.2.2 Type Ontologies

The type ontologies used for matching of tasks - methods and constraints - resolvents are specified in the format used by the DL reasoner. Each place in the code where DL statements are added to the TBox[25] of the DL reasoner essentially corresponds to a type ontology. It has been decided to include two type ontologies: one maintained by the Discovery Module and another embedded in protocols. The Discovery Module type ontology is loaded from a file of ontology statements during setup of the Discovery Module. The protocol type ontology is loaded during use of a given protocol and is concretised as ontology statements in the shared knowledge component of LCC protocols, the CKB (Common Knowledge Base).

The use of protocol type ontologies fits well with the view that LCC protocols should present closed, terminological agreed models [RWB+06]. A drawback is the introduction of a new standard of ontology statements in LCC protocols. Furthermore, it complicates things a bit. We do not wish to mix type ontologies from different protocols and so it is necessary to *unload* the ontologies after use. At the same time it must be ensured that ontology statements which are part of other type ontologies (in our case the Discovery Module type ontology) are

[24] See for instance [NBB+03].

[25] TBox statements are ontology statements about concepts as opposed to individual statements, see [NBB+03] for more details.

not unloaded. The solution is to let the DL Module record whether some statement is already present in the aggregate ontology. If it is, it is not added and this is reported back. When unloading protocol statements only statements which were reported added during loading are removed from the aggregate ontology.

8.2.3 Typing of Compound Terms

Constraints consist of compound terms. Their arguments are also terms corresponding to the syntax definition in section 4.1. This raises the question of whether constraint arguments can also be compound terms. The answer is that nothing restricts them and examples are numerous. A well-known example is use of the equality constraint in combination with lists (e.g. section 9.2.2, figure 15). The lists are compound terms.

How are compound terms supposed to be typed? Constraints can be viewed as compound terms. These are typed by typing them as a whole and typing their arguments. The proposed XLCC syntax, however, only types arguments at a first level. Only support for one level typing of arguments has been implemented, following the syntax extension. Whether this is satisfactory can probably only be known after use of the scheme. With respect to the list example, it could be argued that the hierarchical structure of DL can compensate for the flat structure of first level typing. For instance, a type IntegerList would be subsumed by List and the level of List is in this way implicitly a part of the type. A concern could be the number of types which is probably the reason why many programming languages support composed types.

Having claimed that constraints are typed in a straightforward two level system, 'constraint modifiers' do need special treatment. Examples of constraint modifiers, which are part of the LCC language, are: `not` and `and`. `and`s are expanded into a set of constraints. `not`-modifiers are simply removed because they are not needed for the type inference.

8.3 The DL Module

The DL Module is introduced as a layer between the Match Module and the DL-reasoner. The DL Module handles assertations to and retractions from the TBox of the DL-reasoner (see section 8.2.2) as this had no support for such TBox handling. Furthermore, the DL Module handles missing annotation. When a type is missing in an XLCC protocol, it is considered a wild-card or $\top$ in DL-terminology. The underscore wild-cards in the protocols are handled by the DL Module without invocation of the DL-reasoner according to the following two rules:

- A wild-card subsumes everything
- Nothing but a wild-card can subsume a wild-card

An important consequence of the above handling of missing annotation is that constraints in protocols which are not annotated are very hard to match by constraint solvers. As solvers need to subsume constraints, a constraint solver needs to have a wild-card as type to match an unannotated constraint. This analysis suggests that either: constraints in xprotocols should all be typed or the Constraint Module should be extended with better handling of unmatched constraints (or perhaps more specifically, with better handling of wild-card typed constraints).

8.4 Modularisation in SICStus Prolog

The design is built on a modularised architecture. Prolog, in general, only has limitied support of modules. SICStus in particular has a rather simple support of modules. When building larger systems, especially name space problems seem to occur. Usually, the different modules developed are imported into a common environment (actually a module). While the different modules' private predicates may overlap, the public predicates must be unique. What would be more convenient is a module scheme closer to that of Java packages. Here, a package can either be imported and package prefixes can be left out; *or* if overlapping of names is an issue, package prefixes can be used.

The module system of SICStus Prolog has been slightly abused. A public interface has been defined for each module, in conformity with the architecture in section 7, and no predicate overlapping between these interfaces exists. However, they are referenced using long references including their module origin. Long referencing is only meant to be used for access to private predicates of modules, but have been used for public predicates to emphasise their origin and overall module connectivity.

9 Evaluation

The extension of LCC (sections 6 through 8) will now be evaluated. The evaluation is divided into a theoretical part and a practical part. The theoretical part evaluates the syntactical extension proposed in section 6. The practical part evaluates the chosen semantics of the syntactical extension and furthermore the discovery system developed (sections 7 and 8). The syntactical extension and the semantics of it will be evaluated by comparing both to OWL-S and OWL. The discovery system will be evaluated by usual software testing approaches.

9.1 Extension of LCC with a Service Notion

The key questions to ask in terms of evaluation of the LCC extension is whether the extension complies with an SOA and whether it supports the service challenges from section 1.1.

A Service Oriented Architecture requires that procedures are wrapped up in descriptions which include interfaces (cf. section 1.1). The LCC extension proposed in section 6 does not specifically add descriptions to LCC. Instead it adds annotations which enable an extraction of service descriptions. These LCC service descriptions include interface information (cf. section 6.2.1) and in this sense extracted LCC service descriptions comply with SOA requirements.

Whether the extension supports the SOA challenges will be evaluated next.

9.1.1 Extended LCC and Discovery

The OWL-S service description will be used to evaluate the extended LCC's discovery capabilities. In terms of discovery, the ServiceProfile is the important part of OWL-S (cf. section 6.3). An evaluation criterion is therefore: does extended LCC cover the whole service profile? The LCC extension amounts to typing of services, service parameters and service constraints. It is clear that this does not cover all of the OWL-S service profile. As mentioned in section 6.3, preconditions, effects, and results have all been left out. Arguments were given in section 6.3 why these parts are problematic with respect to LCC (and with respect to services in general). Service typing in LCC does also not cover all of the service profile's service classification features.

Besides the coverage, an important question is whether LCC types are as expressive as OWL-S' DL formalism. The answer is yes. The syntax extension proposed in section 6.4 does not enforce any format, i.e. semantics. This means that, for instance, the format of an OWL-S DL parameter can be encoded as a string and used for LCC parameter types. A decoding of the text into the OWL-S format and DL reasoning of the extracted DL would then take place when applications (e.g. discovery) were to make use of LCC annotations. This is very much like the approach proposed for OWL-S with respect to functions and logical formulae, see section 3.3.1. The expressiveness in terms of extraction and reasoning implemented in the discovery system is discussed in section 9.2.1.

9.1.2 Extended LCC and Composition

LCC protocols (both ordinary and extended) provide a first level of composition of services backed up by agents with MAS proxies. This first-level composition is rather powerful and

meets many of the composition challenge issues identified in section 1.1, see section 4.4.

Composition of MAS Services (i.e. LCC services) can be imagined through the use of constraints. The proposed extension supports this use by a typing of constraints which corresponds to that of methods. However, several issues are attached to this approach, cf. section 6.1.1.

9.2 Discovery System for LCC Services

The practical evaluation amounts to an evaluation of the chosen semantics for XLCC types and an evaluation of the discovery system developed.

9.2.1 Semantics of XLCC Types

The evaluation of the semantics chosen in section 8.2 consists of looking at expressiveness and reasoning capabilities compared with OWL-S.

As explained in section 8.2, XLCC types are interpreted as references to atomic DL concepts. OWL-S service classification and parameters are in contrast DL individuals. The only property of OWL-S parameters used for matching in [PKPS02] are their parameterType property, cf. section 7.1.1. This property points to a DL concept denoting the type of the parameter. It is this concept which LCC parameter types correspond to. They are both concept references and so exhibit same expressiveness. Because LCC method types are a great simplification of OWL-S service classification, the expressive power of the two is hard to compare. However, each service class of OWL-S is also a reference to some OWL concept. In this sense, LCC service types are as expressive as OWL-S service classifications.

The expressiveness of type ontologies in the implemented system and OWL ontologies is different though. With this difference follows a difference in reasoning power as well. In section 8.2.1 it was mentioned that the reasoner used in this project supports the Description Logics subclass: $\mathcal{ALC}$. OWL-DL, on the other hand, represents $\mathcal{SHOIN}(\mathbf{D})$ which is somewhat more expressive than $\mathcal{ALC}$ [Her06] p. 10. It should be emphasised that the expressiveness of the type ontologies and the reasoning power of the implemented discovery system only depends on the DL reasoner used. It would be fairly easy to adopt, for instance, the DL-reasoner presented in [Mei04] which is also implemented in Prolog, but supports the more powerful DL subclass: $\mathcal{ALCN}$.

9.2.2 Software Test of the Discovery System

The practical evaluation of the discovery system developed should consist of a regular software test. Preferably, a great number of real world examples would be tested using the system. The problem is, however, that: 1) very few Web Service discovery scenarios exist [MSvS03]; 2) few LCC interaction scenarios exist. Instead, a black box test of an expanded version of the trade protocol in section 6.4 (figure 11) will be carried out. As a consequence of the sparse black box testing, a white box test will also be carried out to document the system's behaviour and support its correctness. Software testing and in particular black box and white box testing is explained in [KFN93].

Black Box Test A black box test treats the software system as a black box, hence the name. Focus is put on a thorough examination of the output as well as the input which should amount to substantial, realistic examples. The input in our case will be the xprotocol in figure 15. The protocol is an extended version of our running example (figure 11). The buyer now takes on another role to lookup shops which trade the wanted product. The locator role has a recursive subrole which shows how LCC copes with recursion. The locate method can be seen as another independent service in the protocol besides the buy method. The two methods thus show the implemented system's conservation of symmetry in LCC protocols. Finally, the example makes use of lists which shows the system's handling of compound types, cf. section 8.2.3. A test for discovering the buy method will be carried out using a Prolog testing file,

$$
\begin{array}{l}
\{ \\
\quad buy \texttt{ -- } a(buyer(product \texttt{ -- } ProductCode, receipt \texttt{ -- } Receipt), B) :: \\
\qquad a(locator(ProductCode, Shops), B)\ then \\
\qquad buy(ProductCode, CreditCard) \Rightarrow a(shop, S) \leftarrow \\
\qquad\qquad\qquad\qquad getcc \texttt{ -- } cc(cc \texttt{ -- } CreditCard) \wedge \\
\qquad\qquad\qquad\qquad selector \texttt{ -- } favourite(shoplist \texttt{ -- } Shops, shop \texttt{ -- } S)\ then \\
\qquad receipt(Receipt) \Leftarrow a(shop, S). \\
\quad , \\
\quad a(shop, S) :: \\
\qquad buy(ProductCode, CreditCard) \Leftarrow a(buyer(ProductCode, \_), B)\ then \\
\qquad receipt(Receipt) \Rightarrow a(buyer(ProductCode, \_), B) \leftarrow \\
\qquad\qquad\qquad\qquad enough\_credit(ProductCode, CreditCard) \wedge \\
\qquad\qquad\qquad\qquad complete\_order(ProductCode, CreditCard, Receipt). \\
\quad , \\
\quad a(shop\_store, C) :: \\
\qquad inStore(ProductCode) \Leftarrow a(locator\_rec(ProductCode, \_, \_), B)\ then \\
\qquad \left(\begin{array}{l} confirm \Rightarrow a(locator\_rec(ProductCode, \_, \_), B) \leftarrow storage(ProductCode)\ or \\ reject \Rightarrow a(locator\_rec(ProductCode, \_, \_), B) \leftarrow not(storage(ProductCode)) \end{array} \right). \\
\quad , \\
\quad locate \texttt{ -- } a(locator(product \texttt{ -- } ProductCode, shoplist \texttt{ -- } Shops), B) :: \\
\qquad a(locator\_rec(ProductCode, Candidates, Shops), B) \leftarrow \\
\qquad\qquad\qquad\qquad registry \texttt{ -- } registry(shoplist \texttt{ -- } Candidates). \\
\quad , \\
\quad a(locator\_rec(ProductCode, Candidates, Shops), B) :: \\
\qquad null \leftarrow equal \texttt{ -- } Candidates = [] \wedge equal \texttt{ -- } Shops = []\ or \\
\qquad \left(\begin{array}{l} a(locator\_rec(ProductCode, Cs, Ss), B) \leftarrow equal \texttt{ -- } Candidates = [C|Cs]\ then \\ inStore(ProductCode) \Rightarrow a(shop\_store, C)\ then \\ \left(\begin{array}{l} equal \texttt{ -- } Shops = [C|Ss] \leftarrow confirm \Leftarrow a(shop, C)\ or \\ equal \texttt{ -- } Shops = Ss \leftarrow reject \Leftarrow a(shop, C) \end{array} \right) \end{array} \right). \\
\}
\end{array}
$$

Figure 15: Extensive Example of XLCC Protocol

`test.pl` (see A.1). Besides use of this testing file, the system needs to be setup. The setup used is described below:

- A protocol file, `buyregistry_buylocate.inst`, corresponding to the black box protocol in figure 15 is created and placed in the same directory as the system.

- `registrydisc.txt` (registry of protocols in the Discovery Module) will contain one entry: `buyregistry_buylocate.`
- `typeontdisc_blackbox.txt` (type ontology used by the Discovery Module) contains: `ont(equiv(shoplist, and(list, shoprestriction))).`
- `registrycons_blackbox.txt` (registry of supported constraint solvers) contains the entries:
  `registry(type(registry), args([arg(type(shoplist))])).`,
  `getcc(type(getcc), args([arg(type(cc))])).`,
  `selector(type(selector), args([arg(type(list)), arg(type(shop))])).`, and
  `equal(type(equal), args([arg(type(_)), arg(type(_))])).`
- The testing file will use the task below for invocation of the `discover` entry in the Discovery Module:

  `task(type(buy), args([arg(type(product)), arg(type(receipt))])).`

We expect the system to find the *buy* method because the task's type is the same as the method's type and their argument types are equal as well. We expect the constraints checked to amount to the constraints of the *buyer* role and its subroles, *locator* and *locator_rec*. The *favourite* constraint should pass the support check using the *selector* constraint solver. Even though their first arguments differ in type, the Discovery Module's type ontology states that the solver's argument type subsumes the constraint argument type. The other constraints are also supported and so we would expect the system to report the *buy* method as discovered. The expected results are documented in appendix A.5.14 where the matching of task and methods and of constraints and solvers can be followed as system printouts.

White Box Test A white box test (a.k.a. glass box test) makes, as opposed to black box testing, use of knowledge about the system interior. With this knowledge it is possible to setup equivalence classes and boundaries for inputs and outputs.

A white box test schedule is shown in table 9.2.2. Abbreviations are explained in the accompanying nomenclature table (table 9.2.2).

Test	Setup				Expected Output	Results
	task	registrydisc.txt	registrycons.txt	typeontdisc.txt		
Testing match of clauses. Boundaries: none, one, more (two) matches.						
T_Clause_0	buy-1-arg	buyregistry_buylocate.	EMPTY	EMPTY	No match. No methods with one argument.	No match. Printout in A.5.1.
T_Clause_1	buy-2-args	buyregistry_buylocate.	EMPTY	EMPTY	One match. Matches precisely buy-method.	One match. Printout in A.5.2.
T_Clause_2	buy-0-args	buyregistry_buylocate.	EMPTY	EMPTY	Two matches. Matched by shop and shop_store.	Two matches. Printout in A.5.3.
Testing match of protocols. Boundaries: none, one, more (two) matches.						
T_Protocol_0	buy-1-arg	buyregistry_buylocate. buy_none.	EMPTY	EMPTY	No match. No methods with one argument.	No match. Printout in A.5.4.
T_Protocol_1	locate_rec	buyregistry_buylocate. buy_none.	EMPTY	EMPTY	One match. One method with three arguments.	One match. Printout in A.5.5.
T_Protocol_2	buy-0-args	buyregistry_buylocate. buy_none.	EMPTY	EMPTY	Three matches - two in first protocol, one in second. Match like T-Clause-2 and with shop in buy_none.	Three matches. Printout in A.5.6.
Testing DL for subsumption in terms of equivalence, subsumption, and variables. Testing list subsumption with classes: equal lists, lists of different size (subset), and lists in different order (subsumes_set).						
T_DL_equiv	find-shoplist	buyregistry_buylocate.	EMPTY	typedisc_locate	One match. Match find-task with locate-method.	One match. Printout in A.5.7.
T_DL_subsum	find-webshoplist	buyregistry_buylocate.	EMPTY	typedisc_locate	One match. Match find-task with locate-method. Task argument is subsumed by method argument.	One match. Printout in A.5.8.
T_DL_var	buy-var & buy-0-args	buyregistry_buylocate.	EMPTY	EMPTY	[Task var - method const = var/const etc.] No matches for buy-var, showing negative var/const (buyer-method) and positive var/var (shop-method). One match for buy-0-args showing positive const/var.	No match, printout in A.5.9 and two matches, printout in A.5.3.
T_DL_lists	See T_Clause_1				One match. Match with equivalent argument lists of length two.	One match. Printout in A.5.2.
T_DL_sublist	See T_Clause_0				No match. Task arguments constitute only a sublist of the method arguments.	No match. Printout in A.5.1.
T_DL_subset	buy-wrong-order	buyregistry_buylocate.	EMPTY	typedisc_locate	No match. Arguments have wrong order in task.	No match. Printout in A.5.10.
Testing DL type ontologies: success with Discovery Module type ontology; success with protocol type ontology; no overlap between protocol ontologies, i.e. a negative test with necessary DL in first protocol.						
T_Ont_disc-ont	See T_DL_equiv				One match. Match find-task with locate-method.	One match. Printout in A.5.7.
T_Ont_prot-pos	find-shoplist	buy_none. buyregistry_ontfind.	EMPTY	EMPTY	One locate-method match (two matches in all). Match find-task with locate-method.	One locate-method match (two matches in all). Printout in A.5.11.
T_Ont_prot-neg	find-shoplist	buy_ontfind. buyregistry_buylocate.	EMPTY	EMPTY	No locate-method match (one other match). Ontology assertion in first protocol is not available in second.	No locate-method match (one other match). Printout in A.5.12.
Testing constraint check. DL matching has been checked with methods-tasks (Match Module used for both). Need to test deep constraints, non-looping and backtrack over constraint solvers.						
T_Cons	buy-2-args	buy_rec.	supercons-1-2-3	EMPTY	One discovery. Constraints of both roles (deep) are supported by three different solvers (backtrack) exactly once (non-looping).	One discovery. Printout in A.5.13.

Table 1: White Box Test Schedule

Abbreviation	Expansion
Tasks	
buy-0-args	`task(task(type(buy), args([]))).`
buy-1-arg	`task(task(type(buy), args([arg(type(product))]))).`
buy-2-args	`task(task(type(buy), args([arg(type(product)), arg(type(receipt))]))).`
buy-wrong-order	`task(task(type(buy), args([arg(type(receipt)), arg(type(product))]))).`
buy-var	`task(task(type(buy), args([arg(type(_)), arg(type(receipt))]))).`
find-shoplist	`task(task(type(find), args([arg(type(product)), arg(type(shoplist))]))).`
find-webshoplist	`task(task(type(find), args([arg(type(product)), arg(type(webshoplist))]))).`
locate_rec	`task(task(type(locate_rec), args([arg(type(product)), arg(type(shoplist)), arg(type(shoplist))]))).`
registrydisc.txt - refer to protocol files listed in registry	
buy_none	LCC syntax example with extra client parameter (Receipt). See A.2.1.
buy_ontfind	Like buy_none, but with DL assertion that find equals locate. See A.2.2.
buy_rec	Like buy_none example with recursive role shift between the two roles. See A.2.3.
buyregistry_buylocate	Black box example with only buyer and locator role annotated. See A.2.4.
buyregistry_ontfind	Like buyregistry_buylocate, but with DL assertion that find equals locate. See A.2.5.
registrycons.txt - refer to constraint registry files used instead of registrycons.txt	
supercons-1-2-3	Registry with constraint solvers matching all constraints with one, two, or three arguments. See A.3.1.
typeontdisc.txt - refer to type ontology files used instead of typeontdisc.txt	
typeontdisc_locate	Has assertions: equivalence of find and locate; subsumption of shoplist by list. See A.4.1.

Table 2: White Box Test Schedule - Nomenclature Table

The functionality test of the discovery system developed is divided into three classes:

- Test of method matching
- Test of constraint matching
- Test of DL

Method testing amounts to testing clause-boundaries (i.e. can zero, one, or more clauses be matched) and protocol-boundaries (i.e. can zero, one, or more protocols be matched). Constraint testing amounts to test of non-looping and deep checks and evidence of backtracking over constraint solvers. DL testing is further divided into:

- DL concept test
- Argument type list test
- Type ontology test

DL concept test consists of testing subsumption of equivalent concepts, subsumed concepts, and variable concepts. Argument type list test amounts to testing that subsumed lists match, but that neither sublists nor subsets are subsumed. Finally, the type ontology test should test use of the type ontology maintained by the Discovery Module and type ontologies part of protocols. It should be tested that protocol ontologies do not overlap.

All the tests mentioned above are described with a concrete setup, expected results, and actual system results in table 9.2.2. As shown in the table all tests have been passed succesfully.

10 Conclusion

The theoretical goal of this thesis was to compare the Semantic Web Service specification language, OWL-S, with the MAS protocol language, LCC. The practical goal was to extend LCC with a notion of services and test this notion through discovery of LCC services.

Sections 3 through 5 analysed the relationship between OWL-S and LCC and argued for the correctness of the project's hypothesis. Sections 6 through 9 attempted a service extension of LCC and evaluated this extension.

10.1 A Comparison of LCC and OWL-S

The hypothesis of the project, that LCC corresponds to the OWL-S ServiceModel, was supported by the comparison in section 5. The essence of the argument is that LCC has no notion of services (cf. section 4.4) and is only concerned with composition and that the ServiceModel is exactly the part of the OWL-S ontology which is occupied with composition (cf. section 3.4).

Besides substantiating the hypothesis of the project, the comparison noted that both LCC and the service model can be traced back to a foundation in process calculus. An important difference in their views was generalised to the orchestration versus choreography discussion. LCC has a choreographical view of interactions whereas OWL-S provides an orchestrational view. There is no doubt that the relationship between these two views will continue to be an important subject in the near future of service research. One interesting option is to investigate the translation from one view to the other (as done in [RWB+06] section 7, [LCBR04]).

With respect to the service challenges of discovery and composition, LCC is inferior to OWL-S in that LCC has no solution to discovery due to its lack of a service notion. In the challenge of composition LCC seems, on the other hand, superior mainly because it provides a more advanced communication scheme which is executable, has a well defined semantics, and features, among other things, important glue-logic, sections 5.3 and 5.4.

10.2 Extension of LCC

In section 6 an extension of LCC with a notion of services was analysed. The idea behind the extension is that LCC roles are viewed both as methods and as subservices used by methods. Method descriptions correspond to service interfaces and thus methods correspond in a way to LCC services (i.e. MAS Services). An annotation of methods which enables extraction of a service interface was developed in section 6.3. The annotation is designed to support discovery of LCC services, but some important differences to the OWL-S service profile exist:

- The comprehensive service classification system of OWL-S is greatly simplified to a single method type. This was chosen due to scope considerations, but also because OWL-S' classification seems too extensive, cf. sections 6.3 and 7.1.1.
- Preconditions, effects, and results are not included in the extended LCC because no satisfying counterparts were identified in LCC protocols, cf. sections 6.3.2 through 6.3.4.
- Constraints of a method/service are included in LCC service interfaces, but not in OWL-S interfaces. Constraints denote what capabilities the requester should possess before

invoking a service, rather than what the required conditions are, in terms of a world state, before successful invocation of a service can take place.

- Parameters are not divided into inputs and outputs because LCC is relational in nature, cf. section 6.3.5.

The annotation therefore came to consist of: method types, parameter types of methods and constraint types and parameter types of constraints. Section 9.1.1 argued that the syntactatical extension of LCC is just as expressive as OWL-S service descriptions, though not as comprehensive. The equal expressiveness opens up for the possibility of translation between LCC interfaces and OWL-S service profiles, see section 8.1.

10.2.1 Semantics for LCC Types and a Discovery System

Semantics was defined for LCC types in section 8.2 (definition 5). The semantics of LCC types is that types should be interpreted as DL concept references which equals the interpretation of type properties used in OWL-S. LCC types and OWL-S type properties are therefore also equal in expressiveness in terms of their semantics. As explained in section 9.2.1, the DL reasoner used in this project for type ontologies and type inference restricts ontology expressiveness and reasoning to a level inferior to that of OWL-S. Future work could consist of implementing another Match Module using a stronger DL-reasoner or maybe even introduce mapping techniques, see section 9.2.1 and 7.1.3.

The analysis in section 8.2.3 of compound arguments, which have no correspondence in OWL-S, showed that the proposed syntax and semantics may turn out to be inadequate for typing of compound terms. An example of such compound terms are lists in section 9.2.2. Deeper analysis and possible extensions to cater for compound types is a possible option for future work.

Using the LCC extension and its accompanying semantics, section 9.2.2 shows that a local system capable of discovering LCC services has been implemented successfully. As noted in section 7.1, an obvious extension to the system would be distributed discovery. Another important issue which has not been dealt with is the discovery of peers for the found services. Also, it remains to examine the possibility of composition of LCC services in greater detail, see section 6.1.1 and 7.1.2.

10.3 Bringing MASs and the Semantic Web together

The documentation of the project's hypothesis has brought LCC and OWL-S closer together in terms of a better understanding of the relationship between the two. The comparison led to an extension of LCC which in many aspects comply to the service view OWL-S, and the Semantic Web in general, has adopted. The developed discovery system brings MASs closer to deployment on the Semantic Web in that it provides a way to discover MAS Services. However, important differences still exist as explained above and it should be noted that the grounding of LCC services (including an LCC interpreter) and the grounding of Semantic Web Services are fundamentally different.

References

[AAA+06] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Canyang Kevin Liu, Dieter König, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0. Internet, May 2006. Draft.

[AvH04] Grigoris Antoniou and Frank van Harmelen. *A Sematic Web Primer*. The MIT Press - Cambridge, Massachusetts - London, England, 2004.

[BGG+05] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration: A synergic approach for system design. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3826, pages 228–240. Service-Oriented Computing, ICSOC 2005 - Third International Conference, Proceedings, Springer Verlag, 2005.

[BLFIM98] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. Uniform resource identifiers (uri): Generic syntax. Internet, August 1998. RFC (rfc2396) document.

[BMN+04] David Booth, Hugo Haasand Francis McCabe, Eric Newcomer, Iona, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. Technical report, W3C, 2004.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. Internet, 2001.

[DKK+] David Dupplaw, Uladzimir Kharkevich, Spyros Kotoulas, Adri´an Perreau de Pinninck, Ronny Siebes, and Chris Walton. Architecting open knowledge. Technical report, Open-Knowledge Project.

[eea04] David Martin (editor) et al. Owl-s: Semantic markup for web services. Technical report, W3C and DAML, 2004.

[FGY07] F. McNeill F. Giunchiglia and M. Yatskevich. Web service composition via semantic matching of interaction specifications. Open Knowledge Project Paper, Submitted to WWW'07 but appearently not accepted., May 2007.

[fip97] Foundation for intelligent physical agents - fipa 97 specification - part 1 - agent management. Internet, 1997. Obsolete specification.

[GBR05] Dominic Greenwood, Paul Buhler, and Alois Reitbauer. Web service discovery and composition using the web service integration gateway. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE05)*, 2005.

[GH00] Daniel Cabeza Gras and Manuel V. Hermenegildo. A new module system for prolog. *Computational Logic*, pages 131–148, 2000.

[GHVD03] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 48–57, New York, NY, USA, 2003. ACM Press.

[Her06] Thomas Herchenröder. Lightweight semantic web oriented reasoning in prolog: Tableaux inference for description logics. Master of science - artificial intelligence, School of Informatics, University of Edinburgh, 2006.

[HL94] P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, 1994.

[HL01] T. Berners-Lee J. Hendler and O. Lassila. The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific Am*, 284(5):28–37, May 2001.

[JCI02] Lakhmi C. Jain, Zhengxin Chen, and Nikhil Ichalkaranje. *Intelligent Agents and Their Applications*. Physica-Verlag - A Springer-Verlag Company, 2002.

[KBR+04] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web services choreography description language version 1.0. Internet, December 2004. Draft.

[KFN93] Cem Kaner, Jack Falk, and Hun Quoc Nguyen. *Testing Computer Software*. Van Nostrand Reinhold - International Thomson Publishing, 1993.

[KS05] Yannis Kalfoglou and Marco Schorlemmer. Ontology mapping: The state of the art. In Y. Kalfoglou, M. Schorlemmer, A. Sheth, S. Staab, and M. Uschold, editors, *Semantic Interoperability and Integration*, number 04391 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http://drops.dagstuhl.de/opus/volltexte/2005/40> [date of citation: 2005-01-01].

[LCBR04] G. Li, J. Chen-Burger, and D. Robertson. Mapping a business process model to a semantic web services model. In *In Proceedings of the IEEE International Conference on Web Services*, San Diego, 2004.

[LM07] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, Volume 70(Issue 1):96–118, January 2007.

[Mar05] David Martin. Owl-s 1.1 release. Internet, February 2005.

[mas] Multi-agent semantic web systems 2006-07 - assignment 2 for msc students. A course project description for a course taken at University of Edinburgh Spring semester 2007, Multi-agent Semantic Web Systems 2006-07.

[Mei04] A. Meissner. An automated deduction system for description logic with alcn language. *Studia z Automatyki i Informatyki*, 28-29:91–110, 2004.

[Mil91] Robin Milner. The polyadic pi-calculus: A tutorial. Technical report, University of Edinburgh, Laboratory for Foundations of Computer Science, Computer Science Department, University of Ediburgh, The King's Buildings, Edinburgh EH9 3JZ, UK, 1991.

[Mil93] Robin Milner. Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1):78–89, January 1993.

[MMM04] Frank Manola, Eric Miller, and Brian McBride. Rdf primer. Internet, February 2004.

[MSvS03] D. Richards M. Sabou and S. van Splunter. An experience report on using daml-s. In *In Proceedings of the WWW03 Workshop on EServices and the Semantic Web*, Budapest, Hungary, May 2003.

[MvHe04] Deborah L. McGuinness and Frank van Harmelen (editors). Owl web ontology language overview. Technical report, W3C, February 2004.

[NBB+03] D. Nardi, R. J. Brachman, F. Baader, W. Nutt, F. M. Donini, U. Sattler, D. Calvanese, R. Mölitor, G. De Giacomo, R. Küsters, F. Wolter, D. L. McGuinness, P. F. Patel-Schneider, R. Möller, V. Haarslev, I. Horrocks, A. Borgida, C. Welty, A. Rector, E. Franconi, M. Lenzerini, and R. Rosat. *The Description Logic Handbook: Theory, Implementation, Applications.* Cambridge University Press, Cambridge, UK, . ISBN 0-521-78176-0, 2003.

[NFH05] Barry Norton, Simon Foster, and Andrew Hughes. A compositional operational semantics for owl-s. Internet, 2005. 2nd Workshop on Web Services and Formal Methods (WS-FM 2005).

[NFPH5] Barry Norton, Simon Foster, Carlos Pedrinaci, and Andrew Hughes. Cashew project. Internet, 2005-.

[ope] Openknowledge project.

[owla] The owl-s editor. Internet. A project developing an OWL-S Editor for use as plugin for Protégé.

[owlb] Owl-s release status [for version 1.1]. Internet.

[owl04] The owl-s ontology. Internet, 2004. The URL refers to an entry site for the different OWL-S related ontologies including the top-ontology.

[PKPS02] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic matching of web services capabilities. In *The Semantic Web - ISWC 2002: First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002. Proceedings*, volume Volume Volume 2342/2002 of *Lecture Notes in Computer Science*, page 333. Springer Berlin / Heidelberg, 2002.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal: Very Large Data Bases*, 10(4):334–350, ???? 2001.

[RKM04] Jinghai Rao, Peep Küngas, and Mihhail Matskin. Logic-based web services composition: From service description to process model. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 446, Washington, DC, USA, 2004. IEEE Computer Society.

[Rob04a] David Robertson. A lightweight method for coordination of agent oriented web services. In *AAAI Spring Symposium on Semantic Web Services*, 2004.

[Rob04b] David Robertson. Multi-agent coordination as distributed logic programming. In *Logic Programming*, volume 3132/2004 of *Lecture Notes in Computer Science*, pages 416–430. Springer Berlin / Heidelberg, 2004.

[RWB+06] David Robertson, Chris Walton, Adam Barker, Paolo Besana, Yun-Heh Chen-Burger, Fadzil Hassan, David Lambert, Guo Li, Jarred McGinnis, Nardine Osman, Alan Bundy, Fiona McNeill, Frank van Harmelen, Carles Sierra, and Fausto Giunchiglia. Models of interaction as a grounding for peer to peer knowledge sharing. In Elizabeth Chang, Tharam S. Dillon, Robert Meersman, and Katia Sycara, editors, *Advances in Web Semantics*, volume 1 of *LNCS-IFIP (International Federation for Information Processing)*. 2006.

[SDF06] Omair M. Shafiq, Ying Ding, and Dieter Fensel. Bridging multi agent systems and web services: towards interoperability between software agents and semantic web services. In *EDOC '06: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, pages 85–96, Washington, DC, USA, 2006. IEEE Computer Society.

[SE04] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. Technical Report Technical Report DIT-04-087, Informatica e Telecomunicazioni, University of Trento, 2004.

[SH05] Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing - Semantics, Processes, Agents*. Wiley, 2005.

[sic] Sicstus prolog 4.0.0 manual. Internet.

[SPW+04] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. Htn planning for web service composition using shop2, 2004. Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, and Dana Nau. HTN planning for web service composition using SHOP2. Web Semantics Journal, 2004. To appear.

[SvHK+] Ronny Siebes, Frank van Harmelen, Spyros Kotoulas, Dave Dupplaw, Dietlind Geldoff, Fausto Giunchiglia, Maurizio Marchese, Fiona McNeill, Andrian Perreau de Pinninck, Dave Robertson, Marta Sabou, Carles Sierra, Lucia Specia, Austin Tate, and Mikalai Yatskevich. The functional description of the open-knowledge system. Technical report, Open-Knowledge Project.

[SW92] Donald Sannella and Lincoln Wallen. A calculus for the construction of modular prolog programs. *Journal of Logic Programming*, 12:147–177, 1992.

[udd] Universal description, discovery and integration. Internet. OASIS supported standard.

[w3c] World wide web consortium. Internet.

[Wal07] Christopher Walton. *Agency and the Sematic Web*. Oxford University Press, 2007.

[Woo02] Michael Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, 2002.

[YS91] Eyal Yardeni and Ehud Shapiro. A type system for logic programs. *The Journal of Logic Programming*, 10(2):125–153, February 1991.

A Tests

This appendix includes all test results from the evaluation section, 9. All tests have been run with the test file in A.1.

A.1 Test File - test.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% test.pl                                                                    %
% Testing file                                                               %
%                                                                           %
%                                                                           %
% 5th August 2007  v1.0                                                     %
% Mikkel Boje                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The file provides a way to test the discovery system.

:- ensure_loaded(discovery),
   ensure_loaded(basic),
   ensure_loaded(misc).


% Tasks for different tests. Refer to program documentation.
% buy-0-args
%task(task(type(buy), args([]))).

% buy-1-arg
%task(task(type(buy), args([arg(type(product))]))).

% buy-2-args
task(task(type(buy), args([arg(type(product)), arg(type(receipt))]))).

% buy-wrong-order
%task(task(type(buy), args([arg(type(receipt)), arg(type(product))]))).

% buy-var
%task(task(type(_), args([arg(type(_)), arg(type(receipt))]))).

% find-shoplist
%task(task(type(find), args([arg(type(product)), arg(type(shoplist))]))).

% find-webshoplist
%task(task(type(find), args([arg(type(product)), arg(type(webshoplist))]))).

% locate_rec
%task(task(type(locate_rec),
%    args([arg(type(product)), arg(type(shoplist)), arg(type(shoplist))]))).


% Tests discovery of the task described above.
test_disc :-
   task(Task),
   setup_discovery,
   discover(Task, Result),
   close_discovery,
   print('\n\n'),
   print('*** Test: test_disc ***\n'),
   print('Discovered '), length(Result, Len),
   print(Len), print(' method(s): \n'),
   misc:print_methods(name, Result), print('\n'),
   print('\n\n').

% Discovers the task above and simulates the protocol belonging to the
% first method found.
simul_disc :-
   task(Task),
   setup_discovery,
   discover(Task, [M|Ms]),
   close_discovery,
   print('\n\n'),
   print('*** Test: test_disc ***\n'),
   print('Discovered the following methods: \n'),
   misc:print_methods(name, [M|Ms]), print('\n'),
   print('Simulating first protocol: '),
   ontology:method_protocol(M, XProtocol),
   protocol:protocol(XProtocol, Protocol), print(Protocol),
   print('\n'),
   agents_icrf(Protocol, Agents),
   simulate([], Agents, Protocol),
   print('\n\n').

% Extracts a list of agents (roles heads) from a given (non-typed)
% protocol.
agents(def(_, [], _), []).
agents(def(_, [Agent ::= _|Clauses], _), [Agent|Agents]) :-
   agents(def(_, Clauses, _), Agents).

% Hardcoded list of roles and provider identifiers for the icrf
% protocol.
agents_icrf(_, Agents) :-
   Agents = [a(client,c1), a(client,c2), a(broker,b1),
```

```
        a(supplier,s1), a(supplier,s2)].
```

```
% ------------------------------------------------------------------- %
% Copied from simulator.pl - part of LCC interpreter package developed
% at Edinburgh University.
simulate(Ms, Agents, Prot) :-
    sim_step(Ms, Agents, Prot, NewMs, EProt), !,
    portray_clause(NewMs),
    simulate(NewMs, Agents, EProt).
simulate(Ms, Agents, Prot) :-
    \+ sim_step(Ms, Agents, Prot, _, _).

sim_step(Ms, Agents, Prot, NewMs, EProt) :-
    member(Agent, Agents),
    expansion(Agent, Ms, [], Prot, RestMs, OMessages, EProt),
    \+ Prot = EProt,
    append_messages(OMessages, RestMs, NewMs).

append_messages([m(Af,M => At)|T], List, [m(At,M <= Af)|R]) :-
    append_messages(T, List, R).
append_messages([], List, List).
% ------------------------------------------------------------------- %
```

A.2 Protocols

XLCC and LCC protocols used in the tests.

A.2.1 buy_none.inst

```
a(client(ProductCode, Receipt), C) ::=
  buy(ProductCode, CreditCard) => a(shop, S) <-- cc(CreditCard) and lookup(S) then
  receipt(Receipt) <= a(shop, S).

a(shop, S) ::=
  buy(ProductCode, CreditCard) <= a(client(_), C) then
  receipt(Receipt) => a(client(_), C) <--
   enough_credit(CreditCard, ProductCode) and
   complete_order(ProductCode, CreditCard, Receipt).
```

A.2.2 buy_ontfind.inst

```
a(client(ProductCode, Receipt), C) ::=
  buy(ProductCode, CreditCard) => a(shop, S) <-- cc(CreditCard) and lookup(S) then
  receipt(Receipt) <= a(shop, S).

a(shop, S) ::=
  buy(ProductCode, CreditCard) <= a(client(_), C) then
  receipt(Receipt) => a(client(_), C) <--
   enough_credit(CreditCard, ProductCode) and
   complete_order(ProductCode, CreditCard, Receipt).

known(_, ont(equiv(find, locate))).
```

A.2.3 buy_rec.inst

```
a(client(ProductCode, Receipt), C) ::=
  buy(ProductCode, CreditCard) => a(shop, S) <-- cc(CreditCard) and lookup(S) then
  receipt(Receipt) <= a(shop, S) then
  a(shop, S).

a(shop, S) ::=
  buy(ProductCode, CreditCard) <= a(client(_), C) then
  receipt(Receipt) => a(client(_), C) <--
   enough_credit(CreditCard, ProductCode) and
   complete_order(ProductCode, CreditCard, Receipt) then
  a(client(_), C).
```

A.2.4 buyregistry_buylocate.inst

```
buy -- a(buyer(product -- ProductCode, receipt -- Receipt), B) ::=
  a(locator(ProductCode, Shops), B) then
  buy(ProductCode, CreditCard) => a(shop, S) <--
   getcc -- cc(cc -- CreditCard) and
   selector -- favourite(shoplist -- Shops, shop -- S)  then
  receipt(Receipt) <= a(shop, S).

a(shop, S) ::=
  buy(ProductCode, CreditCard) <= a(buyer(ProductCode, _), B)  then
  receipt(Receipt) => a(buyer(ProductCode, _), B) <--
   enough_credit(ProductCode, CreditCard) and
   complete_order(ProductCode, CreditCard, Receipt).
```

```
a(shop_store, C) ::=
  inStore(ProductCode) <= a(locator_rec(ProductCode, _, _), B)  then
  confirm => a(locator_rec(ProductCode, _, _), B) <-- storage(ProductCode)  or
  reject => a(locator_rec(ProductCode, _, _), B) <-- not(storage(ProductCode)).

locate -- a(locator(product -- ProductCode, shoplist -- Shops), B) ::=
  a(locator_rec(ProductCode, Candidates, Shops), B) <--
   registry -- registry(shoplist -- Candidates).

a(locator_rec(ProductCode, Candidates, Shops), B) ::=
  null <-- equal -- Candidates = [] and equal -- Shops = []  or
  a(locator_rec(ProductCode, Cs, Ss), B) <-- equal -- Candidates = [C|Cs] then
  inStore(ProductCode) => a(shop_store, C) then
  equal -- Shops = [C|Ss] <-- confirm <= a(shop, C) or
  equal -- Shops = Ss <-- reject <= a(shop, C).
```

A.2.5 buyregistry_ontfind.inst

```
buy -- a(buyer(product -- ProductCode, receipt -- Receipt), B) ::=
  a(locator(ProductCode, Shops), B) then
  buy(ProductCode, CreditCard) => a(shop, S) <--
   getcc -- cc(cc -- CreditCard) and
   selector -- favourite(shoplist -- Shops, shop -- S)  then
  receipt(Receipt) <= a(shop, S).

a(shop, S) ::=
  buy(ProductCode, CreditCard) <= a(buyer(ProductCode, _), B)  then
  receipt(Receipt) => a(buyer(ProductCode, _), B) <--
   enough_credit(ProductCode, CreditCard) and
   complete_order(ProductCode, CreditCard, Receipt).

a(shop_store, C) ::=
  inStore(ProductCode) <= a(locator_rec(ProductCode, _, _), B)  then
  confirm => a(locator_rec(ProductCode, _, _), B) <-- storage(ProductCode)  or
  reject => a(locator_rec(ProductCode, _, _), B) <-- not(storage(ProductCode)).

locate -- a(locator(product -- ProductCode, shoplist -- Shops), B) ::=
  a(locator_rec(ProductCode, Candidates, Shops), B) <--
   registry -- registry(shoplist -- Candidates).

a(locator_rec(ProductCode, Candidates, Shops), B) ::=
  null <-- Candidates = [] and Shops = []  or
  a(locator_rec(ProductCode, Cs, Ss), B) <-- Candidates = [C|Cs] then
  inStore(ProductCode) => a(shop_store, C) then
  Shops = [C|Ss] <-- confirm <= a(shop, C) or
  Shops = Ss <-- reject <= a(shop, C).


known(_, ont(equiv(find, locate))).
```

A.3 Constraint Registry Files

Files used instead of the empty default `registrycons.txt` file.

A.3.1 registrycons_supercons123.txt

```
supercons1(type(_), args([arg(type(_))])).
supercons2(type(_), args([arg(type(_)), arg(type(_))])).
supercons3(type(_), args([arg(type(_)), arg(type(_)), arg(type(_))])).
```

A.3.2 registrycons_blackbox.txt

```
registry(type(registry), args([arg(type(shoplist))])).
getcc(type(getcc), args([arg(type(cc))])).
selector(type(selector), args([arg(type(list)), arg(type(shop))])).
equal(type(equal), args([arg(type(_)), arg(type(_))])).
```

A.4 Discovery Type Ontology Files

Files used as type ontologies for the Discovery Module instead of the empty default `typeontdisc.txt`.

A.4.1 typeontdisc_locate.txt

```
ont(equiv(find, locate)).
ont(equiv(webshoplist, and(shoplist, shoprestriction))).
```

A.4.2 typeontdisc_blackbox.txt

```
ont(equiv(shoplist, and(list, shoprestriction))).
```

A.5 Test Results

These are the results from the different tests in terms of printouts from the system.

A.5.1 Clause Test - No Matches

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 --------------------------
 --- Matching of methods ---
 --------------------------

Match method: [buyer]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...
  buy subsumes buy

Match method: [shop]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...
  _17302 subsumes buy

Match method: [shop_store]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...
  _17212 subsumes buy

Match method: [locator]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...

Match method: [locator_rec]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...
  _17083 subsumes buy

0 methods matched:
no
| ?-
```

A.5.2 Clause Test - One Match

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 --------------------------
 --- Matching of methods ---
 --------------------------

Match method: [buyer]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...
  buy subsumes buy
   [product,receipt] subsume [product,receipt]
match(ES) Success!

Match method: [shop]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _16622 subsumes buy

Match method: [shop_store]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _16532 subsumes buy

Match method: [locator]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...

Match method: [locator_rec]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _16403 subsumes buy

1 methods matched:
[buyer]
```

```
 ------------------------------------
 --- Checking constraints support ---
 ------------------------------------

Check constraints support of method: buyer
Constraints to check: [favourite, cc, =, =, =, =, =, registry]


*** Test: test_disc ***
Discovered 0 method(s):
no
| ?-
```

A.5.3 Clause Test - Two Matches

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 --------------------------
 --- Matching of methods ---
 --------------------------

Match method: [buyer]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...
  buy subsumes buy

Match method: [shop]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...
  _16610 subsumes buy
    [] subsume []
match(ES) Success!

Match method: [shop_store]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...
  _16520 subsumes buy
    [] subsume []
match(ES) Success!

Match method: [locator]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...

Match method: [locator_rec]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...
  _16391 subsumes buy

2 methods matched:
[shop, shop_store]


 ------------------------------------
 --- Checking constraints support ---
 ------------------------------------

Check constraints support of method: shop
Constraints to check: [complete_order, enough_credit]

Check constraints support of method: shop_store
Constraints to check: [storage, storage]


*** Test: test_disc ***
Discovered 0 method(s):
no
| ?-
```

A.5.4 Protocol Test - No Matches

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 --------------------------
```

```
 --- Matching of methods ---
 ---------------------------

Match method: [buyer]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...
  buy subsumes buy

Match method: [shop]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...
  _22497 subsumes buy

Match method: [shop_store]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...
  _22407 subsumes buy

Match method: [locator]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...

Match method: [locator_rec]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...
  _22278 subsumes buy

Match method: [client]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...
  _22074 subsumes buy

Match method: [shop]
        with: task(type(buy),args([arg(type(product))]))
match(ES) - trying to perform match...
  _21903 subsumes buy

0 methods matched:
no
| ?-
```

A.5.5 Protocol Test - One Match

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 ---------------------------
 --- Matching of methods ---
 ---------------------------

Match method: [buyer]
        with: task(type(locate_rec),args([arg(type(product)),arg(type(shoplist)),arg(type(shoplist))]))
match(ES) - trying to perform match...

Match method: [shop]
        with: task(type(locate_rec),args([arg(type(product)),arg(type(shoplist)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21823 subsumes locate_rec

Match method: [shop_store]
        with: task(type(locate_rec),args([arg(type(product)),arg(type(shoplist)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21733 subsumes locate_rec

Match method: [locator]
        with: task(type(locate_rec),args([arg(type(product)),arg(type(shoplist)),arg(type(shoplist))]))
match(ES) - trying to perform match...

Match method: [locator_rec]
        with: task(type(locate_rec),args([arg(type(product)),arg(type(shoplist)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21604 subsumes locate_rec
   [_21593,_21584,_21575] subsume [product,shoplist,shoplist]
match(ES) Success!

Match method: [client]
        with: task(type(locate_rec),args([arg(type(product)),arg(type(shoplist)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21400 subsumes locate_rec

Match method: [shop]
        with: task(type(locate_rec),args([arg(type(product)),arg(type(shoplist)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21229 subsumes locate_rec
```

```
1 methods matched:
[locator_rec]


 ----------------------------------
 --- Checking constraints support ---
 ----------------------------------

Check constraints support of method: locator_rec
Constraints to check: [=, =, =, =, =]


*** Test: test_disc ***
Discovered 0 method(s):
no
| ?-
```

A.5.6 Protocol Test - Three Matches

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 --------------------------
 --- Matching of methods ---
 --------------------------

Match method: [buyer]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...
  buy subsumes buy

Match method: [shop]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...
  _21805 subsumes buy
    [] subsume []
match(ES) Success!

Match method: [shop_store]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...
  _21715 subsumes buy
    [] subsume []
match(ES) Success!

Match method: [locator]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...

Match method: [locator_rec]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...
  _21586 subsumes buy

Match method: [client]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...
  _21382 subsumes buy

Match method: [shop]
        with: task(type(buy),args([]))
match(ES) - trying to perform match...
  _21211 subsumes buy
    [] subsume []
match(ES) Success!

3 methods matched:
[shop, shop_store, shop]


 ----------------------------------
 --- Checking constraints support ---
 ----------------------------------

Check constraints support of method: shop
Constraints to check: [complete_order, enough_credit]

Check constraints support of method: shop_store
Constraints to check: [storage, storage]

Check constraints support of method: shop
Constraints to check: [complete_order, enough_credit]


*** Test: test_disc ***
```

```
Discovered 0 method(s):
no
| ?-
```

A.5.7 DL Equivalence Test - One Match

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc_locate.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 ---------------------------
 --- Matching of methods ---
 ---------------------------

Match method: [buyer]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...

Match method: [shop]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _17088 subsumes find

Match method: [shop_store]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _16998 subsumes find

Match method: [locator]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  locate subsumes find
   [product,shoplist] subsume [product,shoplist]
match(ES) Success!

Match method: [locator_rec]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _16869 subsumes find

1 methods matched:
[locator]


 ------------------------------------
 --- Checking constraints support ---
 ------------------------------------

Check constraints support of method: locator
Constraints to check: [registry, =, =, =, =, =]


*** Test: test_disc ***
Discovered 0 method(s):
no
| ?-
```

A.5.8 DL Subsumption Test - One Match

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc_locate.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 ---------------------------
 --- Matching of methods ---
 ---------------------------

Match method: [buyer]
        with: task(type(find),args([arg(type(product)),arg(type(webshoplist))]))
match(ES) - trying to perform match...

Match method: [shop]
        with: task(type(find),args([arg(type(product)),arg(type(webshoplist))]))
match(ES) - trying to perform match...
  _17088 subsumes find

Match method: [shop_store]
        with: task(type(find),args([arg(type(product)),arg(type(webshoplist))]))
```

```
match(ES) - trying to perform match...
  _16998 subsumes find

Match method: [locator]
        with: task(type(find),args([arg(type(product)),arg(type(webshoplist))]))
match(ES) - trying to perform match...
  locate subsumes find
   [product,shoplist] subsume [product,webshoplist]
match(ES) Success!

Match method: [locator_rec]
        with: task(type(find),args([arg(type(product)),arg(type(webshoplist))]))
match(ES) - trying to perform match...
  _16869 subsumes find

1 methods matched:
[locator]


 -----------------------------------
 --- Checking constraints support ---
 -----------------------------------

Check constraints support of method: locator
Constraints to check: [registry, =, =, =, =, =]


*** Test: test_disc ***
Discovered 0 method(s):
no
| ?-
```

A.5.9 DL Variable Test

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 --------------------------
 --- Matching of methods ---
 --------------------------

Match method: [buyer]
        with: task(type(_592),args([arg(type(_584)),arg(type(receipt))]))
match(ES) - trying to perform match...

Match method: [shop]
        with: task(type(_592),args([arg(type(_584)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _17308 subsumes _592

Match method: [shop_store]
        with: task(type(_592),args([arg(type(_584)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _17218 subsumes _592

Match method: [locator]
        with: task(type(_592),args([arg(type(_584)),arg(type(receipt))]))
match(ES) - trying to perform match...

Match method: [locator_rec]
        with: task(type(_592),args([arg(type(_584)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _17089 subsumes _592

0 methods matched:
no
| ?-
```

A.5.10 DL Argument Order Test - No Matches

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc_locate.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 --------------------------
 --- Matching of methods ---
 --------------------------
```

```
Match method: [buyer]
        with: task(type(buy),args([arg(type(receipt)),arg(type(product))]))
match(ES) - trying to perform match...
  buy subsumes buy

Match method: [shop]
        with: task(type(buy),args([arg(type(receipt)),arg(type(product))]))
match(ES) - trying to perform match...
  _17088 subsumes buy

Match method: [shop_store]
        with: task(type(buy),args([arg(type(receipt)),arg(type(product))]))
match(ES) - trying to perform match...
  _16998 subsumes buy

Match method: [locator]
        with: task(type(buy),args([arg(type(receipt)),arg(type(product))]))
match(ES) - trying to perform match...

Match method: [locator_rec]
        with: task(type(buy),args([arg(type(receipt)),arg(type(product))]))
match(ES) - trying to perform match...
  _16869 subsumes buy

0 methods matched:
no
| ?-
```

A.5.11 DL Protocol Ontology Test - One Match

One match refers to the match of the locate method.

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 --------------------------
 --- Matching of methods ---
 --------------------------

Match method: [buyer]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...

Match method: [shop]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _22098 subsumes find

Match method: [shop_store]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _22008 subsumes find

Match method: [locator]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  locate subsumes find
   [product,shoplist] subsume [product,shoplist]
match(ES) Success!

Match method: [locator_rec]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21879 subsumes find

Match method: [client]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21675 subsumes find
   [_21664,_21655] subsume [product,shoplist]
match(ES) Success!

Match method: [shop]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21504 subsumes find

2 methods matched:
[locator, client]


 ----------------------------------
 --- Checking constraints support ---
 ----------------------------------
```

```
Check constraints support of method: locator
Constraints to check: [registry, =, =, =, =, =]

Check constraints support of method: client
Constraints to check: [lookup, cc]


*** Test: test_disc ***
Discovered 0 method(s):
no
| ?-
```

A.5.12 DL Protocol Ontology Test - No Matches

No matches refer to the locate method.

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc.txt
Setting up Constraint Module
Using constraint registry file: registrycons.txt

Discover
registry consulted


 --------------------------
 --- Matching of methods ---
 --------------------------

Match method: [client]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _22501 subsumes find
   [_22490,_22481] subsume [product,shoplist]
match(ES) Success!

Match method: [shop]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _22320 subsumes find

Match method: [buyer]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...

Match method: [shop]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21812 subsumes find

Match method: [shop_store]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21722 subsumes find

Match method: [locator]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...

Match method: [locator_rec]
        with: task(type(find),args([arg(type(product)),arg(type(shoplist))]))
match(ES) - trying to perform match...
  _21593 subsumes find

1 methods matched:
[client]


 ------------------------------------
 --- Checking constraints support ---
 ------------------------------------

Check constraints support of method: client
Constraints to check: [lookup, cc]


*** Test: test_disc ***
Discovered 0 method(s):
no
| ?-
```

A.5.13 Constraint Test - One Match

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc.txt
Setting up Constraint Module
Using constraint registry file: registrycons_supercons123.txt
```

```
Discover
registry consulted


 ---------------------------
 --- Matching of methods ---
 ---------------------------

Match method: [client]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _8822 subsumes buy
   [_8811,_8802] subsume [product,receipt]
match(ES) Success!

Match method: [shop]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _8631 subsumes buy

1 methods matched:
[client]


 -----------------------------------
 --- Checking constraints support ---
 -----------------------------------

Check constraints support of method: client
Constraints to check: [lookup, cc, complete_order, enough_credit]

Match solver: supercons3(type(_23750),args([arg(type(_23742)),arg(type(_23736)),arg(type(_23730))]))
        with: [lookup]
match(ES) - trying to perform match...
  _23750 subsumes _8782

Match solver: supercons2(type(_23723),args([arg(type(_23715)),arg(type(_23709))]))
        with: [lookup]
match(ES) - trying to perform match...
  _23723 subsumes _8782

Match solver: supercons1(type(_23702),args([arg(type(_23694))]))
        with: [lookup]
match(ES) - trying to perform match...
  _23702 subsumes _8782
   [_23694] subsume [_8771]
match(ES) Success!

Match solver: supercons3(type(_25912),args([arg(type(_25904)),arg(type(_25898)),arg(type(_25892))]))
        with: [cc]
match(ES) - trying to perform match...
  _25912 subsumes _8761

Match solver: supercons2(type(_25885),args([arg(type(_25877)),arg(type(_25871))]))
        with: [cc]
match(ES) - trying to perform match...
  _25885 subsumes _8761

Match solver: supercons1(type(_25864),args([arg(type(_25856))]))
        with: [cc]
match(ES) - trying to perform match...
  _25864 subsumes _8761
   [_25856] subsume [_8750]
match(ES) Success!

Match solver: supercons3(type(_28074),args([arg(type(_28066)),arg(type(_28060)),arg(type(_28054))]))
        with: [complete_order]
match(ES) - trying to perform match...
  _28074 subsumes _18582
   [_28066,_28060,_28054] subsume [_18785,_18867,_18949]
match(ES) Success!

Match solver: supercons3(type(_30518),args([arg(type(_30510)),arg(type(_30504)),arg(type(_30498))]))
        with: [enough_credit]
match(ES) - trying to perform match...
  _30518 subsumes _18141

Match solver: supercons2(type(_30491),args([arg(type(_30483)),arg(type(_30477))]))
        with: [enough_credit]
match(ES) - trying to perform match...
  _30491 subsumes _18141
   [_30483,_30477] subsume [_18338,_18420]
match(ES) Success!


*** Test: test_disc ***
Discovered 1 method(s):
[client]


yes
| ?-
```

A.5.14 Black Box Test Results

```
| ?- test_disc.
Setting up Discovery Module
Using discovery registry file:  registrydisc.txt
       and type ontology file:  typeontdisc_blackbox.txt
Setting up Constraint Module
Using constraint registry file: registrycons_blackbox.txt

Discover
registry consulted


 --------------------------
 --- Matching of methods ---
 --------------------------

Match method: [buyer]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...
  buy subsumes buy
   [product,receipt] subsume [product,receipt]
match(ES) Success!

Match method: [shop]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _18061 subsumes buy

Match method: [shop_store]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _17971 subsumes buy

Match method: [locator]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...

Match method: [locator_rec]
        with: task(type(buy),args([arg(type(product)),arg(type(receipt))]))
match(ES) - trying to perform match...
  _17842 subsumes buy

1 methods matched:
[buyer]


 ----------------------------------
 --- Checking constraints support ---
 ----------------------------------

Check constraints support of method: buyer
Constraints to check: [favourite, cc, =, =, =, =, =, registry]

Match solver: equal(type(equal),args([arg(type(_47281)),arg(type(_47275))]))
        with: [favourite]
match(ES) - trying to perform match...

Match solver: selector(type(selector),args([arg(type(list)),arg(type(shop))]))
        with: [favourite]
match(ES) - trying to perform match...
  selector subsumes selector
   [list,shop] subsume [shoplist,shop]
match(ES) Success!

Match solver: equal(type(equal),args([arg(type(_49740)),arg(type(_49734))]))
        with: [cc]
match(ES) - trying to perform match...

Match solver: selector(type(selector),args([arg(type(list)),arg(type(shop))]))
        with: [cc]
match(ES) - trying to perform match...

Match solver: getcc(type(getcc),args([arg(type(cc))]))
        with: [cc]
match(ES) - trying to perform match...
  getcc subsumes getcc
   [cc] subsume [cc]
match(ES) Success!

Match solver: equal(type(equal),args([arg(type(_52009)),arg(type(_52003))]))
        with: [=]
match(ES) - trying to perform match...
  equal subsumes equal
   [_52009,_52003] subsume [_37111,_37193]
match(ES) Success!

Match solver: equal(type(equal),args([arg(type(_54355)),arg(type(_54349))]))
        with: [=]
match(ES) - trying to perform match...
  equal subsumes equal
   [_54355,_54349] subsume [_36700,_36782]
match(ES) Success!
```

```
| ?-

Match solver: equal(type(equal),args([arg(type(_56701)),arg(type(_56695))]))
     with: [=]
match(ES) - trying to perform match...
  equal subsumes equal
    [_56701,_56695] subsume [_36228,_36310]
match(ES) Success!

Match solver: equal(type(equal),args([arg(type(_59047)),arg(type(_59041))]))
     with: [=]
match(ES) - trying to perform match...
  equal subsumes equal
    [_59047,_59041] subsume [_35676,_35758]
match(ES) Success!

Match solver: equal(type(equal),args([arg(type(_61393)),arg(type(_61387))]))
     with: [=]
match(ES) - trying to perform match...
  equal subsumes equal
    [_61393,_61387] subsume [_35284,_35366]
match(ES) Success!

Match solver: equal(type(equal),args([arg(type(_63739)),arg(type(_63733))]))
     with: [registry]
match(ES) - trying to perform match...

Match solver: selector(type(selector),args([arg(type(list)),arg(type(shop))]))
     with: [registry]
match(ES) - trying to perform match...

Match solver: getcc(type(getcc),args([arg(type(cc))]))
     with: [registry]
match(ES) - trying to perform match...

Match solver: registry(type(registry),args([arg(type(shoplist))]))
     with: [registry]
match(ES) - trying to perform match...
  registry subsumes registry
    [shoplist] subsume [shoplist]
match(ES) Success!


*** Test: test_disc ***
Discovered 1 method(s):
[buyer]


yes
```

B Modules

Modules developed by the author.

B.1 Protocol Module - protocol.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% protocol.pl                                                              %
% Protocol Module                                                          %
%                                                                          %
%                                                                          %
% 26th July 2007  v1.0                                                     %
% Mikkel Boje                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Implements the Protocol Module. The module handles loading of
% protocol files and conversion between protocol formats, i.e. pure
% LCC, extended LCC and the LCC Ontology.

% The LCC Ontology datastructure is discussed in the program
% documentation. It has the following shape:
% ontology(
%   methods([method(name(Name), type(Type),
%                   args([arg(name(Name), type(Type)),...]),
%                   submethods([submethod(name(Name)),...]),
%                   constraints([constraint(name(Name), type(Type),
%                                           args([arg(name(Name),type(Type)),
%                                                 ...])),
%                                ...]),
%                   protocol(XProtocol)),
%            ...]))
%

% Nomenclature conventions
% 1. Extended w.r.t. syntax will be shortened with an x, for instance
%    ExtendedProtocol to XProtocol.




% --- Head --- %
% Module declaration
:- module(protocol, [load/2, protocol/2, ontology/2]).

% Ensure that libraries used are loaded
:- ensure_loaded(loader).
:- ensure_loaded(library(terms)).

% Declare the operator of the extended LCC syntax
:- op(750, xfx, '--').
% --- Head --- %




%%%%%%%%%%%%%%
% Public     %
%%%%%%%%%%%%%%

% Loads an (possibly) extended LCC protocol from a file with name,
% FileName. Uses loading capabilities of basic LCC interpreter.
load(FileName, XProtocol) :-
   load_institution(FileName, XProtocol).


% Extracts a standard LCC protocol from an annotated one; i.e. it
% removes any annotations.
protocol(def(Exp, XProtocol, CKB), def(Exp, Protocol, CKB)) :-
   xlcc2lcc(XProtocol, Protocol).


% Extracts an LCC ontology from a given annotated protocol.
ontology(XProtocol, Ontology) :-
   XProtocol = def(_, XClauses, _),
   xlcc2ont(XClauses, XProtocol, Ontology).




%%%%%%%%%%%%%%
% Private    %
%%%%%%%%%%%%%%

%%%%%%%%%%%%%%
% xlcc2lcc   %
%%%%%%%%%%%%%%
% Removes any annotations
xlcc2lcc([], []).
xlcc2lcc([XClause|XClauses], [Clause|Clauses]) :-
   xlcc2lcc_clause(XClause, Clause),
```

```
   xlcc2lcc(XClauses, Clauses).

xlcc2lcc_clause(XRole ::= XBody, Role ::= Body) :-
   xlcc2lcc_role(XRole, Role),
   xlcc2lcc_body(XBody, Body).

% Roles without annotation need to have their arguments 'cleaned'.
xlcc2lcc_role(a(XRoleType, Id), a(RoleType, Id)) :-
   XRoleType =.. [F|XArgs],
   xlcc2lcc_args(XArgs, Args),
   RoleType =.. [F|Args].
% Roles with annotation need their annotation removed and their
% arguments 'cleaned'.
xlcc2lcc_role(_ -- XRole, Role) :-
   xlcc2lcc_role(XRole, Role).


% Parsing of body follows LCC syntax, witnessed by close resemblance
% to the LCC interpreter.
% Annotation only exists around constraints.

% Parts without constraints:
xlcc2lcc_body(XPartA or XPartB, PartA or PartB) :-
   xlcc2lcc_body(XPartA, PartA), xlcc2lcc_body(XPartB, PartB).
xlcc2lcc_body(XPartA then XPartB, PartA then PartB) :-
   xlcc2lcc_body(XPartA, PartA), xlcc2lcc_body(XPartB, PartB).
xlcc2lcc_body(XPartA par XPartB, PartA par PartB) :-
   xlcc2lcc_body(XPartA, PartA), xlcc2lcc_body(XPartB, PartB).
xlcc2lcc_body(M <= A, M <= A).
xlcc2lcc_body(M => A, M => A).
xlcc2lcc_body(null, null).
xlcc2lcc_body(a(Agent, Id), a(Agent, Id)).
% Parts that need constraint 'cleaning':
xlcc2lcc_body(XC <-- M <= A, C <-- M <= A) :-
   xlcc2lcc_constraint(XC, C).
xlcc2lcc_body(M => A <-- XC, M => A <-- C) :-
   xlcc2lcc_constraint(XC, C).
xlcc2lcc_body(a(Agent, Id) <-- XC, a(Agent, Id) <-- C) :-
   xlcc2lcc_constraint(XC, C).
xlcc2lcc_body(null <-- XC, null <-- C) :-
   xlcc2lcc_constraint(XC, C).


% 'Cleaning' of constraints is similar to that of roles.
xlcc2lcc_constraint(XC, C) :-
   XC =.. [F|XArgs], F \== --,
   xlcc2lcc_args(XArgs, Args),
   C =.. [F|Args], !.
xlcc2lcc_constraint(_ -- XC, C) :-
   xlcc2lcc_constraint(XC, C).

xlcc2lcc_args([], []).
xlcc2lcc_args([Arg|Args], [XArg|XArgs]) :-
   xlcc2lcc_arg(Arg, XArg),
   xlcc2lcc_args(Args, XArgs).
% Combination of order, subsumes_chk and cut, !, secures that variables
% are not unified with compound '--' terms.
xlcc2lcc_arg(Arg, Arg) :-
   \+ subsumes_chk(--(_, _), Arg), !.
xlcc2lcc_arg(_ -- Arg, Arg).



%%%%%%%%%%%%%%
% xlcc2ont   %
%%%%%%%%%%%%%%
% Extracts an lcc ontology from an xlcc protocol.
% No-type is represented by an anonymous variabe: _

% The code uses the notions of XTerm and OntTerm. An XTerm is an
% possibly annotated term. OntTerm is an ontology term which has the
% following shape: functor(name(Name), type(Type),...). It is worth
% noting that both method, constraint and arg from the ontology
% datastructure follows this format.

% The predicate works by parsing xlcc clauses top down.
xlcc2ont([], _, ontology(methods([]))).
xlcc2ont([Clause|Clauses], XProtocol, ontology(methods([Method|Methods]))) :-
   xlcc2ont_clause(Clause, XProtocol, Method),
   xlcc2ont(Clauses, XProtocol, ontology(methods(Methods))).

xlcc2ont_clause(Role ::= Body, XProtocol, FinishedMethod) :-
   Method =.. [method, name(_), type(_), args(_), submethods([]),
               constraints([]), protocol(XProtocol)],
   xlcc2ont_role(Role, Method),
   xlcc2ont_body(Body, Method, FinishedMethod).

% Regards Method as an ontterm. The ontterm is built from the
% xterm, Type -- Agent. The ontterm's functor is already given as
% method.
xlcc2ont_role(a(Agent, _), Method) :-
   xlcc2ont_ins_xterm(Agent, Method), !.
xlcc2ont_role(Type -- a(Agent, _), Method) :-
   xlcc2ont_ins_xterm(Type -- Agent, Method).
```

```
% Parsing of body follows LCC syntax; witnessed by close resemblance
% to the LCC interpreter.
% Submethods and constraints need to be inserted into the ontology.

% Parts without insertions.
xlcc2ont_body(A or B, Method, FinishedMethod) :-
   xlcc2ont_body(A, Method, SoFar), xlcc2ont_body(B, SoFar, FinishedMethod).
xlcc2ont_body(A then B, Method, FinishedMethod) :-
   xlcc2ont_body(A, Method, SoFar), xlcc2ont_body(B, SoFar, FinishedMethod).
xlcc2ont_body(A par B, Method, FinishedMethod) :-
   xlcc2ont_body(A, Method, SoFar), xlcc2ont_body(B, SoFar, FinishedMethod).
xlcc2ont_body(_ <= _, Method, Method).
xlcc2ont_body(_ => _, Method, Method).
xlcc2ont_body(null, Method, Method).
% Parts with insertions:
xlcc2ont_body(C <-- _ <= _, Method, FinishedMethod) :-
   xlcc2ont_ins_constraint(C, Method, FinishedMethod).
xlcc2ont_body(_ => _ <-- C, Method, FinishedMethod) :-
   xlcc2ont_ins_constraint(C, Method, FinishedMethod).
xlcc2ont_body(Submethod <-- C, Method, FinishedMethod) :-
   xlcc2ont_ins_submethod(Submethod, Method, SoFar),
   xlcc2ont_ins_constraint(C, SoFar, FinishedMethod).
xlcc2ont_body(Submethod, Method, FinishedMethod) :-
    xlcc2ont_ins_submethod(Submethod, Method, FinishedMethod).
xlcc2ont_body(null <-- C, Method, FinishedMethod) :-
   xlcc2ont_ins_constraint(C, Method, FinishedMethod).


% ********************************************************************
% * Notice that the 'built-in' constraints of 'assert' and 'retract'  *
% * are not handled!                                                  *
% ********************************************************************
% 'not' constraints are handled by simply removing 'not'. The reasoning
% is that the interpreter will take care of the 'not'-part and the
% constraint manager should be capable of executing the constraint.
xlcc2ont_ins_constraint(not(Constraint), Method, FinishedMethod) :-
   xlcc2ont_ins_constraint(Constraint, Method, FinishedMethod).
% 'and'-constraints are expanded. I.e. peer should be capable of all
% 'and-built-in'-constraints.
xlcc2ont_ins_constraint(C1 and C2, Method, FinishedMethod) :-
   xlcc2ont_ins_constraint(C1, Method, SoFar),
   xlcc2ont_ins_constraint(C2, SoFar, FinishedMethod).
% Uses xlcc2ont_ins_xterm by regarding constraints as xterms and
% converts them to ontterms with 'constraint' as functor.
xlcc2ont_ins_constraint(Constraint, Method, FinishedMethod) :-
   Constraint =.. [F|_], F \== and,
   F \== not, % 'not' - handling
   OntTerm =.. [constraint, name(_), type(_), args(_)],
   xlcc2ont_ins_xterm(Constraint, OntTerm),
   Method =.. [method, MName, MType, MArgs, MSubmethods,
          constraints(Constraints), protocol(XProtocol)],
   FinishedMethod =.. [method, MName, MType, MArgs, MSubmethods,
             constraints([OntTerm|Constraints]),
             protocol(XProtocol)].

xlcc2ont_ins_submethod(a(Role, _), Method, FinishedMethod) :-
   Role =.. [Name|_],
   Method =.. [method, MName, MType, MArgs, submethods(Submethods),
          MConstraints, protocol(XProtocol)],
   FinishedMethod =.. [method, MName, MType, MArgs,
             submethods([submethod(name(Name))|Submethods]),
             MConstraints, protocol(XProtocol)].

% General predicate which converts an xterm into an ontology term:
% [XType --] XFunctor(XArgs)     converted into:
% OntFunctor(name(XFunctor), type(XType), args(OntArgs)).
% - OntFunctor is already given to the OntTerm before using predicate.
% - XType is either '_' or the type annotated on the xterm.
% - OntArgs are XArgs converted, cf. xlcc2ont_args.
xlcc2ont_ins_xterm(Term, OntTerm) :-
   Term =.. [F|_], F \== --,
   xlcc2ont_ins_xterm(_ -- Term, OntTerm), !.
xlcc2ont_ins_xterm(Type -- Term, OntTerm) :-
   Term =.. [Name|Args],
   xlcc2ont_args(Args, OntArgs),
   OntTerm =.. [_, name(Name), type(Type), args(OntArgs)|_].

xlcc2ont_args([], []).
xlcc2ont_args([Arg|Args], [OntArg|OntArgs]) :-
   xlcc2ont_arg(Arg, OntArg),
   xlcc2ont_args(Args, OntArgs).
% Combination of order, subsumes_chk and cut, !, secures that variables
% are not unified with compound '--' terms.
xlcc2ont_arg(Arg, OntArg) :-
   \+ subsumes_chk(--(_, _), Arg),
   xlcc2ont_arg(_ -- Arg, OntArg), !.
xlcc2ont_arg(Type -- Arg, OntArg) :-
   OntArg = arg(name(Arg), type(Type)).
% Deep constraints are not handled: name(Arg) above.
% Alternative: Arg =.. [Name|_], name(Name)
```

```
%%%%%%%%%%%%%
% xlcc2xlist %
%%%%%%%%%%%%%
% *** Unused parser. Parsing xlcc code into an intermediate list
% format. Advantage of use is expected to be a single parser of xlcc
% code. However, it seems that knowledge of lcc code structure is
% still necessary for use of intermediate format. ***

% Flattens (x)protocols by flattening the following operators:
% ::=, --, or, then, par, =>, <=, <--.
% The datastructure is
% [clause,...],
% Clauses are lists of flattened protocol elements from the operators
% above. The operators above are flattened in first level and second
% level of arguments of role types and constraints. E.g.:
% [constrainttype, --, constraint([type1, --, arg1, type2, --, arg2, arg3])]
xlcc2xlist([], []).
xlcc2xlist([XClause|XClauses], [ClauseLs|ClausesLs]) :-
   xlcc2xlist_clause(XClause, ClauseLs),
   xlcc2xlist(XClauses, ClausesLs).
xlcc2xlist_clause(::=(Role, Body)) -->
   xlcc2xlist_role(Role), [::=],
   xlcc2xlist_body(Body).
xlcc2xlist_role(Role) -->
   {\+ subsumes_chk(--(_, _), Role), Role = a(Agent, Id),
    phrase(xlcc2xlist_term(Agent), [ParsedAgent|_])},
   [a(ParsedAgent, Id)], !.
xlcc2xlist_role(Type -- a(Agent, Id)) -->
   [Type, --], {phrase(xlcc2xlist_term(Agent), [ParsedAgent|_])},
   [a(ParsedAgent, Id)].

% Parsing of body follows LCC syntax, witnessed by close resemblance
% to the LCC interpreter.
xlcc2xlist_body(or(A, B)) -->              % or
   xlcc2xlist_body(A), [or], xlcc2xlist_body(B).
xlcc2xlist_body(then(A, B)) -->            % then
   xlcc2xlist_body(A), [then], xlcc2xlist_body(B).
xlcc2xlist_body(par(A, B)) -->             % par
   xlcc2xlist_body(A), [par], xlcc2xlist_body(B).
xlcc2xlist_body(M => A <-- C) -->          % => <--
   [M, =>, A, <--], xlcc2xlist_constraint(C), !.
xlcc2xlist_body(C <-- M <= A) -->          % <-- <=
   xlcc2xlist_constraint(C), [<--, M, <=, A], !.
xlcc2xlist_body(M => A) --> [M, =>, A].    % =>
xlcc2xlist_body(M <= A) --> [M, <=, A].    % <=
xlcc2xlist_body(null <-- C) -->            % null <--
   [null, <--], xlcc2xlist_constraint(C).
xlcc2xlist_body(a(Agent, Id) <-- C) -->    % role <--
   [a(Agent, Id), <--], xlcc2xlist_constraint(C), !.
xlcc2xlist_body(a(Agent, Id)) -->          % role
   [a(Agent, Id)].
xlcc2xlist_constraint(C) -->
   {\+ subsumes_chk(--(_, _), C)}, xlcc2xlist_term(C), !.
xlcc2xlist_constraint(Type -- C) -->
   [Type, --], xlcc2xlist_term(C).
xlcc2xlist_term(T) -->
   {\+ subsumes_chk(--(_, _), T), T =.. [Func|Args],
    phrase(xlcc2xlist_args(Args), Parsed),
    T2 =.. [Func|[Parsed]]}, [T2], !.
xlcc2xlist_term(Type -- T) -->
   {T =.. [Func|Args], xlcc2xlist_args(Args, Parsed),
    T2 =.. [Func|[Parsed]]}, [Type, --, T2].

xlcc2xlist_args([]) --> [].
% Combination of order, subsumes_chk and cut, !, secures that variables
% are not unified with compound '--' terms.
xlcc2xlist_args([Arg|Rest]) -->
   {\+ subsumes_chk(--(_, _), Arg)}, [Arg],
   xlcc2xlist_args(Rest), !.
xlcc2xlist_args([Type -- Arg|Rest]) --> [Type, --, Arg],
   xlcc2xlist_args(Rest).
```

B.2 Ontology Module - ontology.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ontology.pl                                                                 %
% Ontology Module                                                             %
%                                                                             %
%                                                                             %
% 31th July 2007  v1.0                                                        %
% Mikkel Boje                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Implements the Ontology Module, part of the Protocol Layer. The
% module provides access to LCC ontologies, thereby hiding away the
% internals of the datastructure. In this way it is dependent and
% closely related to the Protocol Module which implements construction
% of ontologies by parsing XLCC code.
```

```
% --- Head --- %
% Module declaration
:- module(ontology, [ontology_protocol/2, methods/2, method/2,
           method/3,
           method_protocol/2, method_ontology/2,
           method_name/2, method_type/2, method_args/2,
           method_submethods/3, method_constraints/3,
           constraint_name/2, constraint_type/2,
           submethod_name/2, submethod_method/3,
           submethods_methods/3,
           arg_name/2, arg_type/2]).

% Ensure that libraries used are loaded
:- ensure_loaded(library(lists)).
:- ensure_loaded(protocol).
% --- Head --- %




%%%%%%%%%%%%%%
% Public     %
%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%
% Access predicates %
%%%%%%%%%%%%%%%%%%%%%
% Ontology datastructure access predicates

% Accesses the protocol of an ontology
ontology_protocol(Ontology, Protocol) :-
   methods(Ontology, [M|_]),
   method_protocol(M, Protocol).

% Accesses the methods of an ontology
methods(Ontology, Methods) :-
   ontology_part(Ontology, methods, Methods).

% Provides member functionality over the methods of an ontology.
method(Ontology, Method) :-
   methods(Ontology, Methods),
   member(Method, Methods).

% Accesses a method with name, Name, in ontology, Ontology.
method(Ontology, Name, Method) :-
   methods(Ontology, Methods),
   member(Method, Methods),
   method_name(Method, Name).


%%%%%%%%%%%%%%%%%%%%%
% Method parts      %
%%%%%%%%%%%%%%%%%%%%%
% Accesses the protocol which method, Method, belongs to.
method_protocol(Method, Protocol) :-
   ontology_part(Method, protocol, Protocol).

% Accesses the ontology which method, Method, belongs to.
method_ontology(Method, Ontology) :-
   ontology_part(Method, protocol, Protocol),
   protocol:ontology(Protocol, Ontology).

% Accesses the name of method, Method.
method_name(Method, Name) :-
   ontology_part(Method, name, Name).

% Accesses the type of method, Method.
method_type(Method, Type) :-
   ontology_part(Method, type, Type).

% Accesses the arguments of method, Method.
method_args(Method, Args) :-
   ontology_part(Method, args, Args).

% Accesses the submethods of method, Method.
method_submethods(shallow, Method, Submethods) :-
   ontology_part(Method, submethods, Submethods).

% Accesses the submethods of method, Method.
% The predicate should not backtrack and find incomplete submethod
% lists, hence the cut at the end.
method_submethods(deep, Method, DeepSubmethods) :-
   method_submethods(shallow, Method, ShallowSubmethods),
   method_ontology(Method, Ontology),
   method_submethods(Ontology, ShallowSubmethods, [],
           DeepSubmethods), !.

% Accesses the constraints of method, Method.
method_constraints(shallow, Method, Constraints) :-
   ontology_part(Method, constraints, Constraints).
```

```
% Accesses the constraints of method, Method.
method_constraints(deep, Method, Constraints) :-
   method_submethods(deep, Method, Submethods),
   % Remove any recursive appearances
   method_name(Method, Name),
   delete_submethod(Name, Submethods, NewSubmethods),
% D     print('method_constraints, submethods found:'),
% D   print(NewSubmethods), print('\n'),
   method_ontology(Method, Ontology),
   submethods_methods(Ontology, NewSubmethods, Ms),
   constraints([Method|Ms], Constraints).


%%%%%%%%%%%%%%%%%%%%%
% Constraint parts  %
%%%%%%%%%%%%%%%%%%%%%
% Accesses the name of constraint, Constraint.
constraint_name(Constraint, Name) :-
   ontology_part(Constraint, name, Name).

% Accesses the type of constraint, Constraint.
constraint_type(Constraint, Type) :-
   ontology_part(Constraint, type, Type).

% Accesses the arguments of constraint, Constraint.
constraint_args(Constraint, Args) :-
   ontology_part(Constraint, args, Args).


%%%%%%%%%%%%%%%%%%%%%
% Argument parts    %
%%%%%%%%%%%%%%%%%%%%%
% Accesses the name of argument, Arg.
arg_name(Arg, Name) :-
   ontology_part(Arg, name, Name).

% Accesses the type of argument, Arg.
arg_type(Arg, Type) :-
   ontology_part(Arg, type, Type).


%%%%%%%%%%%%%%%%%%%%%
% Submethod parts   %
%%%%%%%%%%%%%%%%%%%%%
% Accesses the name of submethod, Submethod.
submethod_name(Submethod, Name) :-
   ontology_part(Submethod, name, Name).

% Finds the method which submethod, Submethod, represents in ontology,
% Ontology.
submethod_method(Ontology, Submethod, Method) :-
   submethod_name(Submethod, Name),
   method(Ontology, Name, Method).

% Converts a list of submethods into a list of methods
submethods_methods(_, [], []).
submethods_methods(Ontology, [Sub|Submethods], [Method|Methods]) :-
   submethod_method(Ontology, Sub, Method),
   submethods_methods(Ontology, Submethods, Methods).




%%%%%%%%%%%%%%%
% Private    %
%%%%%%%%%%%%%%%

% Extracts the single argument from argument, Name, of term, Term.
ontology_part(Term, Name, Val) :-
   Term =.. [_|Args],
   member(T, Args),
   T =.. [Name, Val].

% Deletes all occurrences of submethod with name, Name, in a list of
% submethods. Backtracking will not allow deletion of other elements.
delete_submethod(_, [], []).
delete_submethod(Name, [Submethod|Submethods], DelSubmethods) :-
   submethod_name(Submethod, Name),
   delete_submethod(Name, Submethods, DelSubmethods).
delete_submethod(Name, [Submethod|Submethods], [Submethod|DelSubmethods]) :-
   \+ submethod_name(Submethod, Name),
   delete_submethod(Name, Submethods, DelSubmethods).

% Builts on invariant: All submethods in Sofar have been expanded.
% Therefore, they need not be expanded or added to Sofar.
% If a submethod is not in Sofar it is expanded and added to Sofar.
method_submethods(_, [], Found, Found).
method_submethods(Ontology, [Submethod|Submethods], Sofar, Found) :-
   \+ member(Submethod, Sofar),
   submethod_method(Ontology, Submethod, Method),
   method_submethods(shallow, Method, Subsubmethods),
   append(Subsubmethods, Submethods, NewSubmethods),
```

```
   method_submethods(Ontology, NewSubmethods, [Submethod|Sofar], Found).
method_submethods(Ontology, [_|Submethods], Sofar, Found) :-
   method_submethods(Ontology, Submethods, Sofar, Found).

% Extracts constraints from a list of methods.
constraints([], []).
constraints([Method|Methods], AllConstraints) :-
   method_constraints(shallow, Method, Constraints),
   constraints(Methods, RestConstraints),
   append(Constraints, RestConstraints, AllConstraints).
```

B.3 Discovery Module - discovery.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% discovery.pl                                                              %
% Discovery Module                                                          %
%                                                                           %
%                                                                           %
% 31th July 2007  v1.0                                                      %
% Mikkel Boje                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Implements the Discovery Module, part of the Discovery Layer. The
% module provides discovery of methods supporting some task (see below
% for task description). It uses a registry (see the
% load_registry-predicate) of methods (see the Protocol Module for a
% description of the method datastructure).

% The module's interface makes use of a special task description
% datastructure. This must correspond to the method datastructure and
% respect the typeterm datastructure (cf. Match Module):
% task(type(TaskType), args([arg(type(ArgType)),...]))




% --- Head --- %
% Module declaration
:- module(discovery, [discover/2, match_methods/2, check_methods/2,
            setup_discovery/0, close_discovery/0]).

% Declare the protocol registry dynamic.
:- dynamic registry/1.
% And the asserted type ontology statements.
:- dynamic asserted_type_ont/1.

% Ensure that libraries used are loaded
:- ensure_loaded(library(lists)).
:- ensure_loaded(protocol).
:- ensure_loaded(ontology).
:- ensure_loaded(dl).
:- ensure_loaded(misc).
:- ensure_loaded(match).
:- ensure_loaded(constraint).

% Protocol file names for registry. Protocols are loaded into registry
% when module is setup (using setup_discovery/0).
registry_file('registrydisc.txt').

% Type ontology statements belonging to the discovery module
type_ont_file('typeontdisc_blackbox.txt').
% --- Head --- %




%%%%%%%%%%%%%%%%%%
% Public         %
%%%%%%%%%%%%%%%%%%

% Setup of the discovery module comprises loading, i.e. asserting, the
% protocol registry and loading the discovery module's type ontology.
setup_discovery :-
   print('Setting up Discovery Module\n'),
   print('Using discovery registry file:  '), registry_file(RFile),
   print(RFile), print('\n       and type ontology file:  '),
   type_ont_file(TFile), print(TFile), print('\n'),
   load_registry,
   load_type_ont,
   constraint:setup_constraint.


% Closing the discovery module means retracting the protocol registry
% and the type ontology.
close_discovery :-
   retract_registry,
   retract_type_ont,
   constraint:close_constraint.
```

```
% Matches task with methods which constraints are supported by the
% peer.
% Discovery is divided into two parts: matching of task and method
% description; and checking of constraint compability of matched
% methods.
discover(Task, CheckedMethods) :-
   print('\nDiscover\n'),
   match_methods(Task, MatchedMethods),
   length(MatchedMethods, Len), print('\n'), print(Len),
   print(' methods matched: \n'),
   misc:print_methods(name, MatchedMethods),
   print('\n'),
   print('\n\n -----------------------------------\n'),
   print(' --- Checking constraints support ---\n'),
   print(' -----------------------------------\n'),
   check_methods(MatchedMethods, CheckedMethods), !.


% Matches a task description against the methods stored in the local
% registry.
match_methods(Task, MatchedMethods) :-
   registry(Methods),
   print('registry consulted\n'),
   print('\n\n --------------------------\n'),
   print(' --- Matching of methods ---\n'),
   print(' --------------------------\n'),
   match_methods(Task, Methods, MatchedMethods), !.


% Checks whether all constraints of each method in a method list are
% supported by the constraint manager. Constraints of a method amounts
% to constraints of the method and all its submethods and their
% submethods etc (i.e. deep).
check_methods([], []).
check_methods([Method|Methods], [Method|CheckedMethods]) :-
   print('\nCheck constraints support of method: '),
   ontology:method_name(Method, Name), print(Name), print('\n'),
   ontology:method_constraints(deep, Method, Constraints),
   ontology:method_ontology(Method, Ontology),
   print('Constraints to check: '),
   misc:print_constraints(name, Constraints), print('\n'),
   constraint:check_constraints(Constraints, Ontology),
   check_methods(Methods, CheckedMethods).
check_methods([_|Methods], CheckedMethods) :-
   check_methods(Methods, CheckedMethods).


%%%%%%%%%%%%%%
% Private    %
%%%%%%%%%%%%%%

% The core method matching algorithm consists of the Match Module's
% match predicate. Note that this uses subsumption and should be
% referred to as matchES (cf. Match Module). This means that the
% arguments should be Method, Task instead of Task, Method.
match_methods(_, [], []).
match_methods(Task, [Method|Methods],
              [Method|Matched]) :-
   print('\nMatch method: '), misc:print_methods(name, [Method]),
   print('\n        with: '), print(Task),
   print('\n'),
   ontology:method_protocol(Method, XProtocol),
   dl:extract_dl(XProtocol, DL),
   match:match(Method, Task, DL),
   match_methods(Task, Methods, Matched).
match_methods(Task, [_|Methods], Matched) :-
   match_methods(Task, Methods, Matched).


%%%%%%%%%%%%%%%%%%%%%%%%%
% Protocol Registry     %
%%%%%%%%%%%%%%%%%%%%%%%%%
% An important datastructure in the discovery module is its registry
% which holds information about known protocols and providers. The
% registry here simply consists of a list of methods. Discovering of
% provider peers is not addressed in this project. Because the module
% loads all methods of a procotol into the registry it makes most
% sense if provider peers are discovered separately after matching
% methods against tasks.

load_registry :-
   registry_file(FileName),
   misc:read_list(FileName, Files),
   load_registry_files(Files).

load_registry_files(Files) :- load_registry_files(Files, []).
load_registry_files([], Methods) :-
   asserta(registry(Methods)).
load_registry_files([File|Files], MethodsSofar) :-
   protocol:load(File, XProtocol),
   protocol:ontology(XProtocol, Ontology),
   ontology:methods(Ontology, Methods),
   append(Methods, MethodsSofar, AllMethods),
```

```
   load_registry_files(Files, AllMethods).

% Retracts the registry of protocols
retract_registry :-
   retract(registry(_)).



%%%%%%%%%%%%%%%%%%%%%%
% Type Ontology       %
%%%%%%%%%%%%%%%%%%%%%%
% Loading of type ontology. The statements are added to the TBox of
% the DL Module.
load_type_ont :-
   type_ont(TypeOnt),
   dl:tbox_add(TypeOnt, Asserted),
   asserta(asserted_type_ont(Asserted)).

% Retracts type ontology
retract_type_ont :-
   asserted_type_ont(Asserted),
   dl:tbox_remove(Asserted).

% Reads in a list of DL statements from an ontology file (cf. Module
% Head).
type_ont(TypeOnt) :-
   type_ont_file(FileName),
   misc:read_list(FileName, TypeOnt).
```

B.4 Constraint Module - constraint.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% constraint.pl                                                              %
% Constraint Module                                                          %
%                                                                           %
%                                                                           %
% 1st August 2007  v1.0                                                     %
% Mikkel Boje                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Implements the Constraint Module, part of the Discovery Layer. The
% module manages constraints. It keeps a registry of supported
% constraints, known as solvers. The module must be set up and closed
% (because of the registry). It provides check of constraints with
% respect to some ontology.

% The module uses the Match Module for matches between constraint
% solvers and constraints. Thus, constraint solvers respect the
% typeterm datastructure like constraints (cf. Match Module).

% Nomenclature conventions
% 1. Solvers are interfaces for procedures that can solve constraints.



% --- Head ---%
% Module declaration
:- module(constraint, [setup_constraint/0, close_constraint/0,
             check_constraint/2, check_constraints/2]).

% Declare the protocol registry dynamic.
:- dynamic registry/1.

% Ensure that libraries used are loaded
:- ensure_loaded(misc).
:- ensure_loaded(match).
:- ensure_loaded(ontology).
:- ensure_loaded(dl).

% Solver file name for registry. Solvers are loaded into registry when
% module is setup (using setup_constraint/0).
registry_file('registrycons_blackbox.txt').
% --- Head ---%



%%%%%%%%%%%%%
% Public     %
%%%%%%%%%%%%%

% By setting up the constraint module, the solver registry is loaded.
setup_constraint :-
   print('Setting up Constraint Module\n'),
   print('Using constraint registry file: '),
   registry_file(File), print(File), print('\n'),
   load_registry.


% Closing the constraint module means retracting the constraint
% registry.
```

```
close_constraint :-
   retract_registry.


% Succeeds if the constraint, Constraint, in LCC ontology, Ontology,
% is supported by the constraint manager. It is supported if a solver
% - from the solver registry - can be matched against the constraint.
check_constraint(Constraint, Ontology) :-
   registry(Solvers),
   ontology:ontology_protocol(Ontology, XProtocol),
   dl:extract_dl(XProtocol, DL),
   match_solver(Solvers, Constraint, DL), !.


% Checks a list of constraints.
check_constraints([], _).
check_constraints([Constraint|Constraints], Ontology) :-
   check_constraint(Constraint, Ontology),
   check_constraints(Constraints, Ontology).




%%%%%%%%%%%%%%
% Private    %
%%%%%%%%%%%%%%

% Constraint solvers and constraints are matched using the Match
% Module. Therefore as mentioned in header, a solver and a constraint
% are both regarded as typeterms (cf. Match Module). As match should
% be regarded as matchES (cf. Match Module), a Solver is the first
% argument.
match_solver([Solver|_], Constraint, DL) :-
   print('\nMatch solver: '), print(Solver),
   print('\n        with: '),
   misc:print_constraints(name, [Constraint]),
   print('\n'),
   match:match(Solver, Constraint, DL).
match_solver([_|Solvers], Constraint, DL) :-
   match_solver(Solvers, Constraint, DL).


% Loading of registry from a solver file (cf. Module Head).
load_registry :-
   registry_file(FileName),
   misc:read_list(FileName, Solvers),
   asserta(registry(Solvers)).

% Retracts registry
retract_registry :-
   retract(registry(_)).
```

B.5 Match Module - match.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% dl.pl                                                                 %
% DL Module                                                             %
%                                                                       %
%                                                                       %
% 31th July 2007  v1.0                                                  %
% Mikkel Boje                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Implementation of the DL Module, part of the Discovery Layer. The
% module is basically an interface to the DL-reasoner used by this
% project, 'tableaux' by Herchenroeder. It adds TBox management and
% handling of variables in a way reasonable to the present project.




% --- Head --- %
% Module declaration
:- module(dl, [extract_dl/2, tbox_add/2, tbox_remove/1, subsumes/2,
         subsumes_list/2, subsumes_set/2, subset/2]).

% Ensure that libraries used are loaded
:- ensure_loaded(library(lists)).
:- ensure_loaded(tableaux).
% --- Head --- %




%%%%%%%%%%%%%%%%%
% Public        %
%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%
```

```
% DL            %
%%%%%%%%%%%%%%%%%

% Extracts DL-statements from a protocol's CKB.
extract_dl(def(_, _, []), []).
extract_dl(def(_, _, [known(_, ont(Statement))|CKB]), [ont(Statement)|DL]) :-
   !, extract_dl(def(_, _, CKB), DL).
extract_dl(def(_, _, [_|CKB]), DL) :-
   extract_dl(def(_, _, CKB), DL).


% Asserts DL-statements if not already asserted. Records which
% statements have been asserted.
tbox_add([], []).
tbox_add([ont(Statement)|DL], [ont(Statement)|Asserted]) :-
   \+ ont(Statement),
   asserta(ont(Statement)),
   tbox_add(DL, Asserted).
tbox_add([Statement|DL], Asserted) :-
   ont(Statement),
   tbox_add(DL, Asserted).


% Retracts DL-statements.
tbox_remove([]).
tbox_remove([ont(Statement)|DL]) :-
   retract(ont(Statement)),
   tbox_remove(DL).



%%%%%%%%%%%%%%%%%
% Subsumption   %
%%%%%%%%%%%%%%%%%

% Succeeds if Concept1 subsumes Concept2.
% A variable has been decided to subsume everything.
% Notice that only a variable can subsume another variable.
subsumes(Concept1, _) :-
   var(Concept1).
subsumes(Concept1, Concept2) :-
   nonvar(Concept1), nonvar(Concept2),
   tableaux_proof(subsum(Concept2, Concept1)).
% ! Notice that the tableaux's subsum(C1, C2) means C1 is subsumed by
% C2. This is opposite to the present subsumes(C1, C2) where C1
% subsumes C2 (i.e. C2 is subsumed by C1).


% List1 subsumes List2 if all elements in List1 subsumes all elements
% in List2 and element pairs have same index.
subsumes_list([], []).
subsumes_list([El1|List1], [El2|List2]) :-
   subsumes(El1, El2),
   subsumes_list(List1, List2).


% Set1 subsumes Set2 if every element in Set2 is subsumed by some
% element in Set1 (removing the subsuming element) and the two
% sets are of the same size.
subsumes_set([], []).
subsumes_set(Set1, [El2|Set2]) :-
   member(El1, Set1),
   subsumes(El1, El2),
   select(El1, Set1, NewSet1),
   subsumes_set(NewSet1, Set2).


% Set1 is a subset of Set2 if every element in Set1 is subsumed by
% some element in Set2 (removing the subsuming element).
subset([], _).
subset([El1|Set1], Set2) :-
   member(El2, Set2),
   subsumes(El2, El1),
   select(El2, Set2, NewSet2),
   subset(Set1, NewSet2).
```

B.6 DL Module - dl.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% dl.pl                                                                  %
% DL Module                                                              %
%                                                                        %
%                                                                        %
% 31th July 2007  v1.0                                                   %
% Mikkel Boje                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Implementation of the DL Module, part of the Discovery Layer. The
% module is basically an interface to the DL-reasoner used by this
% project, 'tableaux' by Herchenroeder. It adds TBox management and
% handling of variables in a way reasonable to the present project.
```

```
% --- Head --- %
% Module declaration
:- module(dl, [extract_dl/2, tbox_add/2, tbox_remove/1, subsumes/2,
          subsumes_list/2, subsumes_set/2, subset/2]).

% Ensure that libraries used are loaded
:- ensure_loaded(library(lists)).
:- ensure_loaded(tableaux).
% --- Head --- %




%%%%%%%%%%%%%%%%%%
% Public         %
%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%
% DL             %
%%%%%%%%%%%%%%%%%%

% Extracts DL-statements from a protocol's CKB.
extract_dl(def(_, _, []), []).
extract_dl(def(_, _, [known(_, ont(Statement))|CKB]), [ont(Statement)|DL]) :-
   !, extract_dl(def(_, _, CKB), DL).
extract_dl(def(_, _, [_|CKB]), DL) :-
   extract_dl(def(_, _, CKB), DL).


% Asserts DL-statements if not already asserted. Records which
% statements have been asserted.
tbox_add([], []).
tbox_add([ont(Statement)|DL], [ont(Statement)|Asserted]) :-
   \+ ont(Statement),
   asserta(ont(Statement)),
   tbox_add(DL, Asserted).
tbox_add([Statement|DL], Asserted) :-
   ont(Statement),
   tbox_add(DL, Asserted).


% Retracts DL-statements.
tbox_remove([]).
tbox_remove([ont(Statement)|DL]) :-
   retract(ont(Statement)),
   tbox_remove(DL).



%%%%%%%%%%%%%%%%%%
% Subsumption    %
%%%%%%%%%%%%%%%%%%

% Succeeds if Concept1 subsumes Concept2.
% A variable has been decided to subsume everything.
% Notice that only a variable can subsume another variable.
subsumes(Concept1, _) :-
   var(Concept1).
subsumes(Concept1, Concept2) :-
   nonvar(Concept1), nonvar(Concept2),
   tableaux_proof(subsum(Concept2, Concept1)).
% ! Notice that the tableaux's subsum(C1, C2) means C1 is subsumed by
% C2. This is opposite to the present subsumes(C1, C2) where C1
% subsumes C2 (i.e. C2 is subsumed by C1).


% List1 subsumes List2 if all elements in List1 subsumes all elements
% in List2 and element pairs have same index.
subsumes_list([], []).
subsumes_list([El1|List1], [El2|List2]) :-
   subsumes(El1, El2),
   subsumes_list(List1, List2).


% Set1 subsumes Set2 if every element in Set2 is subsumed by some
% element in Set1 (removing the subsuming element) and the two
% sets are of the same size.
subsumes_set([], []).
subsumes_set(Set1, [El2|Set2]) :-
   member(El1, Set1),
   subsumes(El1, El2),
   select(El1, Set1, NewSet1),
   subsumes_set(NewSet1, Set2).


% Set1 is a subset of Set2 if every element in Set1 is subsumed by
% some element in Set2 (removing the subsuming element).
subset([], _).
subset([El1|Set1], Set2) :-
```

```
    member(El2, Set2),
    subsumes(El2, El1),
    select(El2, Set2, NewSet2),
    subset(Set1, NewSet2).
```

B.7 Misc Module - misc.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% misc.pl                                                                    %
% Misc Module                                                                %
%                                                                            %
%                                                                            %
% 2nd August 2007  v1.0                                                      %
% Mikkel Boje                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Implements the Misc Module, part of the Discovery Layer. The
% module implements functionality which is of miscellaneous nature
% like I/O.




% --- Head --- %
% Module declaration
:- module(misc, [read_list/2, print_methods/2, print_constraints/2]).

% Ensure that libraries used are loaded
:- ensure_loaded(ontology).
% --- Head --- %




%%%%%%%%%%%%%%%%%%%
% Public          %
%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%
% Print           %
%%%%%%%%%%%%%%%%%%%
% Pretty-prints a list of methods. The first argument controls what
% (how much) is printed.
print_methods(name, [Method|Methods]) :-
   print('['),
   ontology:method_name(Method, Name), print(Name),
   print_methods_aux(name, Methods).
print_methods(all, [Method|Methods]) :-
   print('['), print(Method),
   print_methods_aux(all, Methods).

% Pretty-prints a list of constraints. The first argument controls what
% (how much) is printed.
print_constraints(name, [Constraint|Constraints]) :-
   print('['),
   ontology:constraint_name(Constraint, Name), print(Name),
   print_constraints_aux(name, Constraints).
print_constraints(all, [Constraint|Constraints]) :-
   print('['), print(Constraint),
   print_constraints_aux(all, Constraints).


%%%%%%%%%%%%%%%%%%%
% I/O             %
%%%%%%%%%%%%%%%%%%%
% Reads in a dot-terminated list of terms from file with file name,
% FileName.
read_list(FileName, List) :-
   see(FileName),
   read_list_sub(List),
   seen.




%%%%%%%%%%%%%%%%%%%
% Private         %
%%%%%%%%%%%%%%%%%%%

% Pretty prints lists of methods.
print_methods_aux(_, []) :- print(']').
print_methods_aux(name, [Method|Methods]) :-
   ontology:method_name(Method, Name),
   print(', '), print(Name), print_methods_aux(name, Methods).
print_methods_aux(all, [Method|Methods]) :-
   print(', '), print(Method), print_methods_aux(all, Methods).

% Pretty prints lists of constraints.
print_constraints_aux(_, []) :- print(']').
```

```
print_constraints_aux(name, [Constraint|Constraints]) :-
    ontology:constraint_name(Constraint, Name),
    print(', '), print(Name), print_constraints_aux(name, Constraints).
print_constraints_aux(all, [Constraint|Constraints]) :-
    print(', '), print(Constraint), print_constraints_aux(all, Constraints).

% Reads a dot-terminated list of terms.
read_list_sub(FinalList) :- read_list_sub([], FinalList).
read_list_sub(List, FinalList) :-
    read(Element),
    \+ Element = end_of_file, !,
    read_list_sub([Element|List], FinalList).
read_list_sub(List, List).
```

C Tableaux Algorithm

Tableaux algorithm for reasoning with Description Logic in Prolog. Implemented by Thomas Herchenröder [Her06].

C.1 Tableaux Algorithm - tableaux.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Thomas Herschenroeder                                %
% search changed to searchDL for overloading matters. %
% remove_duplicates changed to remove_duplicatesDL    %
% for the same reasons (w.r.t. list library)          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% tableaux.pl -- tableaux reasoner for description logics
:- use_module(library(lists)).
:- op(100,fy,~).
:- dynamic ont/1.
:- dynamic id/1.


% Main Proof Goal
tableaux_proof(Exp) :-    % true/fals = satisfiable/unsatisfiable
        proof(Exp).


%construct_goal
proof(equiv(A,B)) :-
        \+ tabl(and(A,~B)),
        \+ tabl(and(B,~A)).
proof(subsum(A,B)) :-
        \+ tabl(and(A,~B)).
proof(disjoint(A,B)) :-
        \+ tabl(and(A,B)).
proof(unsat(A)) :- % unsatisfiable A
        \+ tabl(A).
proof(A) :-          % try to satisfy everything else
        tabl(A).


% main worker
tabl(Exp) :-
        expand_defs(Exp,Exp1),      % expand expression into most basic
        negnormform(Exp1,Exp2),     % NNF transformation
        setID(0),
        !,
        searchDL([[Exp2]],df,_).      % do the proof as an agenda searchDL
```

```
negnormform(~ ~X,X1) :-
        negnormform(X,X1).
negnormform(~forall(R,C),exist(R,C1)) :-
        negnormform(~C,C1).
negnormform(forall(R,C),forall(R,C1)) :-
        negnormform(C,C1).
negnormform(~exist(R,C),forall(R,C1)) :-
        negnormform(~C,C1).
negnormform(exist(R,C),exist(R,C1)) :-
        negnormform(C,C1).
negnormform(~and(A,B),or(A1,B1)) :-
        negnormform(~A,A1),
        negnormform(~B,B1).
negnormform(and(A,B),and(A1,B1)) :-
        negnormform(A,A1),
        negnormform(B,B1).
negnormform(~or(A,B),and(A1,B1)) :-
        negnormform(~A,A1),
        negnormform(~B,B1).
negnormform(or(A,B),or(A1,B1)) :-
        negnormform(A,A1),
        negnormform(B,B1).
negnormform(~X,~X) :-
        atom(X).
negnormform(X,X) :-
        atom(X).
expand_defs(forall(R,C),forall(R,C1)) :-
        expand_defs(C,C1).
expand_defs(exist(R,C),exist(R,C1)) :-
        expand_defs(C,C1).
expand_defs(and(A,B),and(A1,B1)) :-
        expand_defs(A,A1),
        expand_defs(B,B1).
expand_defs(or(A,B),or(A1,B1)) :-
        expand_defs(A,A1),
        expand_defs(B,B1).
expand_defs(~A,~A1) :-
        expand_defs(A,A1).
expand_defs(X,Y) :-
        atom(X),
        ont(equiv(X,X1)),
        expand_defs(X1,Y).
expand_defs(X,X) :-
        atom(X),
        \+ ont(equiv(X,_)).
```

```
% searchDL(+Goal,+Style,-ResultList) -- agenda style searchDL
% -- transforms Goal into a list of [clash]/[model] elements
searchDL([],_,[]).
searchDL(Reduced, _, Reduced) :-
        Reduced = [H|_],
        H = [clash],           % a clash leaf fails the proof
        !,
        fail.
searchDL([Node| T], Style, Reduced) :-
        process_node(Node,NewNodes),
        filter_nodes(NewNodes,NewNodes1),
        merge_agendas(NewNodes1, T, Style, New),
        searchDL(New, Style, Reduced).
filter_nodes(NewNodes,NewNodes1) :-
        (   setof(X,(member(X,NewNodes),X\=[model]),NewNodes1);
            NewNodes1 = []),
        !.
merge_agendas(A1, A2, df, New) :-
        append(A1, A2, New),!.
merge_agendas(A1, A2, bf, New) :-
        append(A2, A1, New),!.


% reduce a node of the proof tree
process_node([clash],[]) :- !.
process_node([model],[]) :- !.
process_node(ListOfDLExps,ResultListOfLists) :-
        transform_connect(ListOfDLExps,_,LoL3),
        expand_nodes(LoL3,ResultListOfLists).
expand_nodes([],[]).
expand_nodes([H|R],RLoL) :-
        expand_node(H,RLoL1),
        expand_nodes(R,RLoL2),
        append(RLoL1,RLoL2,RLoL).


% process a single node
expand_node([],[]).
expand_node([model],[[model]]) :-!.
expand_node([clash],[[clash]]) :-!.
expand_node(Node,[N1]) :-
        check_clash(Node,N1),!. % clash closes this branch
expand_node(Node,N1) :-
        expand_exist(Node,[],LoE),       % expand existential restrictions
        LoE \= [],                       % only continue with non-empty edges
        expand_forall(Node,LoE,LoE2), % try eliminate value restrictions
        extract_nodes(LoE2,N1).          % get the list of new fringe nodes
expand_node(Node,[N1]) :-           % model closes this branch
        expand_exist(Node,[],LoE),
        LoE = [],
        N1 = [model].


% extract list of nodes from list of edges
extract_nodes([],[]).
extract_nodes([edge(_,_,N)|R],[N|R1]) :-
        extract_nodes(R,R1).


% transform and/or connectives
transform_connect([],_,[[]]).
transform_connect([and(A,B)|R],N,R1) :-
        ( member(A,R) ->
            ( member(B,R) ->
                transform_connect(R,N,R1);
                transform_connect([B|R],N,R1));
            ( member(B,R) ->
                transform_connect([A|R],N,R1);
                transform_connect([A,B|R],N,R1))).
transform_connect([or(A,B)|R],N,LoL) :-
        ( \+ (member(A,R) ; member(B,R)) ->
            ( transform_connect([A|R],N,LoL);
              transform_connect([B|R],N,LoL));
            transform_connect(R,N,LoL)).
transform_connect([H|R],N,LoL) :-
        H \= and(_,_),
        H \= or(_,_),
        transform_connect(R,N,LL1),
        LL1 = [LL2],
        LoL = [[H|LL2]].


% transform forall/exist quantifiers
expand_exist([],L,L).
expand_exist([exist(R,C)|T],T1,L) :-
        (   (member(edge(R,_,X),T1), member(C,X) ) -> % existing R edge
            expand_exist(T,T1,L);
            getID(Id),
            expand_exist(T,[edge(R,Id,[C])|T1],L)
        ).
expand_exist([H|T],T1,L) :-
        H \= exist(_,_),
```

```
    expand_exist(T,T1,L).
expand_forall(_,[],[]).


% push concept into exist. node
expand_forall(Node,[edge(R,I,N)|RoE],[edge(R,I,N2)|R1]) :-
        setof(X,member(forall(R,X) , Node),C1),
            append(C1,N,N1),
            remove_duplicatesDL(N1,N2),
            expand_forall(Node,RoE,R1).
expand_forall(Node,[edge(R,I,N)|RoE],[edge(R,I,N)|LoE]) :-
        \+ member(forall(R,_),Node),
            expand_forall(Node,RoE,LoE).
check_clash(Exp,Exp1) :-
        member(A,Exp),
        member(~A,Exp),
        Exp1 = [clash].
remove_duplicatesDL([],[]).
remove_duplicatesDL([H|T],[H|R]) :-
        r_d(H,T,[],R1),
            remove_duplicatesDL(R1,R),!.
r_d(_,[],T,T).
r_d(E,[E|T],TT,T1) :- r_d(E,T,TT,T1).
r_d(E,[X|T],TT,T1) :- X \= E, r_d(E,T,[X|TT],T1).
getID(I):-
        id(I1),
        I is I1 + 1,
        retract(id(I1)),
        asserta(id(I)),!.
getID(I):-
        \+ id(_),
        I is 0,
        asserta(id(I)),!.
setID(X):-
        ( id(Y) ->
            retract(id(Y));
            true
        ),
        asserta(id(X)).
```

D Interpreter

Basic interpreter for LCC developed at the University of Edinburgh.

D.1 Simulator Part - simulator.pl

```
:- ensure_loaded(basic),
   ensure_loaded(loader).

portray(def(_,_,_)) :- print('Definition').

test :-
   sim([a(client,c1),a(client,c2),a(broker,b1),a(supplier,s1),a(supplier,s2)], icrf).


sim(Agents, InstitutionFile) :-
    load_institution(InstitutionFile, Prot),
    simulate([], Agents, Prot).

simulate(Ms, Agents, Prot) :-
    sim_step(Ms, Agents, Prot, NewMs, EProt), !,
    portray_clause(NewMs),
    simulate(NewMs, Agents, EProt).
simulate(Ms, Agents, Prot) :-
    \+ sim_step(Ms, Agents, Prot, _, _).

sim_step(Ms, Agents, Prot, NewMs, EProt) :-
    member(Agent, Agents),
    expansion(Agent, Ms, [], Prot, RestMs, OMessages, EProt),
    \+ Prot = EProt,
    append_messages(OMessages, RestMs, NewMs).

append_messages([m(Af,M => At)|T], List, [m(At,M <= Af)|R]) :-
    append_messages(T, List, R).
append_messages([], List, List).
```

D.2 Basic Interpreter - basic.pl

```
:- use_module(library(lists)),
   use_module(library(random)).

:- ensure_loaded(util).

:- op(900, xfx, '::='),
   op(900, xfx, '::'),
   op(900, xfx, '>>'),
   op(800, xfx, '=>'),
   op(800, xfx, '<='),
   op(830, xfx, '<--'),
   op(820, xfy, and),
   op(850, xfy, par),
   op(850, xfy, then),
   op(850, xfy, or).


expansion(Agent, Ms, Os, P, FinalMs, FinalOs, FinalP) :-
    expansion_step(Agent, Ms, Os, P, NewMs, NewOs, NewP),
    expansion(Agent, NewMs, NewOs ,NewP, FinalMs, FinalOs, FinalP).
expansion(Agent, Ms, Os, P, Ms, Os, P) :-
    \+ expansion_step(Agent, Ms, Os, P, _, _, _).

expansion_step(a(Role,Id), Ms, Os, P, NewMs, NewOs, NewP) :-
    protocol_select(agent, P, (a(ARole,Id) ::= Def), P1),
    expand_protocol((a(ARole,Id) ::= Def), Role, Id, Ms, Os, P1,
                    NewA, NewMs, NewOs, P2),
    protocol_add(agent, P2, NewA, NewP).
expansion_step(a(Role,Id), Ms, Os, P, NewMs, NewOs, NewP) :-
    \+ protocol_select(agent, P, (a(_,Id) ::= _), _),
    protocol_member(dialogue, P, Clause),
    Clause = (a(Role,Id) ::= Def),
    expand_protocol((a(Role,Id) ::= Def), Role, Id, Ms, Os, P,
                    NewA, NewMs, NewOs, P2),
    protocol_add(agent, P2, NewA, NewP).

expand_protocol(Role ::= Def, _, Id, Ms, Os, P, Role ::= E, Mf, Of, Pf) :-
    expand_protocol(Def, Role, Id, Ms, Os, P, E, Mf, Of, Pf).
expand_protocol(A or _, Role, Id, Ms, Os, P, E, Mf, Of, Pf) :-
    expand_protocol(A, Role, Id, Ms, Os, P, E, Mf, Of, Pf).
expand_protocol(_ or B, Role, Id, Ms, Os, P, E, Mf, Of, Pf) :-
    expand_protocol(B, Role, Id, Ms, Os, P, E, Mf, Of, Pf).
expand_protocol(A then B, Role, Id, Ms, Os, P, EA then B, Mf, Of, Pf) :-
    expand_protocol(A, Role, Id, Ms, Os, P, EA, Mf, Of, Pf).
expand_protocol(A then B, Role, Id, Ms, Os, P, A then EB, Mf, Of, Pf) :-
    closed(A),
    expand_protocol(B, Role, Id, Ms, Os, P, EB, Mf, Of, Pf).
expand_protocol(A par B, Role, Id, Ms, Os, P, EA par EB, Mf, Of, Pf) :-
    expand_protocol(A par B, Role, Id, Ms, Os, P, EA, Mn, On, Pn),
    expand_protocol(A par B, Role, Id, Mn, On, Pn, EB, Mf, Of, Pf).
expand_protocol(C <-- M <= A, Role, Id, Ms, Os, P, c(M <= A), Mf, Os, Pf) :-
    select(m(Role,M <= A), Ms, Mf),
    satisfied(Id, P, C, Pf).
```

```
expand_protocol(M => A <-- C, Role, Id, Ms, Os, P, c(M => A), Ms,
        [m(Role,M => A) | Os], Pf) :-
    satisfied(Id, P, C, Pf).
expand_protocol(M <= A, Role, _, Ms, Os, P, c(M <= A), Mf, Os, P) :-
    select(m(Role,M <= A), Ms, Mf).
expand_protocol(M => A, Role, _, Ms, Os, P, c(M => A), Ms,
        [m(Role,M => A) | Os], P).
expand_protocol(Role <-- C, _, Id, Ms, Os, P, Role ::= Def, Ms, Os, Pf) :-
    Role = a(_,_),
    satisfied(Id, P, C, Pf),
    protocol_member(dialogue, P, (Role ::= Def)).
expand_protocol(Role, _, _, Ms, Os, P, Role ::= Def, Ms, Os, P) :-
    Role = a(_,_),
    protocol_member(dialogue, P, (Role ::= Def)).
expand_protocol(null <-- C, _, Id, Ms, Os, P, c(null), Ms, Os, Pf) :-
    satisfied(Id, P, C, Pf).
expand_protocol(null, _, _, Ms, Os, P, c(null), Ms, Os, P).

closed(c(_)).
closed(A or _) :-
    closed(A).
closed(_ or B) :-
    closed(B).
closed(A then B) :-
    closed(A),
    closed(B).
closed(A par B) :-
    closed(A),
    closed(B).
closed(_ ::= Def) :-
    closed(Def).

satisfied(Id, P, A and B, Pf) :- !,
    satisfied(Id,  P, A, Pn),
    satisfied(Id,  Pn, B, Pf).
satisfied(Id, P, X, Pf) :-
    meta_pred(Id, X, P, Pf, Call), !,
    Call.
satisfied(Id, P, X, P) :-
    \+ meta_pred(Id, X, P, _, _),
    call_direct(X),
    X.
satisfied(Id,  P, X, P) :-
    protocol_member(common_knowledge, P, known(Id, X)).
satisfied(Id, P, X, Pf) :-
    protocol_member(common_knowledge, P, known(Id, X <-- C)),
    satisfied(Id, P, C, Pf).

call_direct(X) :-
    (predicate_property(X, built_in) ;
     predicate_property(X, interpreted) ;
     predicate_property(X, imported_from(_))), !.

meta_pred(Id, not(X), P, P, \+ satisfied(Id,P,X,_)).
meta_pred(Id, retract(X), P, Pf,
        protocol_remove(common_knowledge,P,known(Id,X),Pf)).
meta_pred(Id, assert(X), P, Pf,
        protocol_add(common_knowledge,P,known(Id,X),Pf)).

protocol_component(agents, def(Clauses, _, _), Clauses).
protocol_component(dialogue, def(_, Clauses, _), Clauses).
protocol_component(common_knowledge, def(_, _, Clauses), Clauses).

protocol_member(agent, def(Clauses,_,_), Clause) :-
    member(Clause, Clauses).
protocol_member(dialogue, def(_,Clauses,_), ClauseCopy) :-
    member(Clause, Clauses),
    copy_term(Clause, ClauseCopy).
protocol_member(common_knowledge, def(_,_,Clauses), ClauseCopy) :-
    member(Clause, Clauses),
    copy_term(Clause, ClauseCopy).

protocol_select(agent, def(Clauses,A,B), Clause, def(R,A,B)) :-
    select(Clause, Clauses, R).
protocol_select(dialogue, def(A,Clauses,B), ClauseCopy, def(A,R,B)) :-
    select(Clause, Clauses, R),
    copy_term(Clause, ClauseCopy).
protocol_select(common_knowledge, def(A,B,Clauses), ClauseCopy, def(A,B,R)) :-
    select(Clause, Clauses, R),
    copy_term(Clause, ClauseCopy).

protocol_remove(agent, def(Clauses,A,B), Clause, def(R,A,B)) :-
    select(Clause, Clauses, R).
protocol_remove(dialogue, def(A,Clauses,B), Clause, def(A,R,B)) :-
    select(Clause, Clauses, R).
protocol_remove(common_knowledge, def(A,B,Clauses), Clause, def(A,B,R)) :-
    select(Clause, Clauses, R).

protocol_add(agent, def(Clauses,A,B), X, def([X|Clauses],A,B)).
protocol_add(dialogue, def(A,Clauses,B), X, def(A,[X|Clauses],B)).
protocol_add(common_knowledge, def(A,B,Clauses), X, def(A,B,[X|Clauses])).
```

D.3 Loader for Protocol Files - loader.pl

```
:- ensure_loaded(basic).

load_institution(Institution, InstDef) :-
    concat(Institution,'.inst',File),
    see(File),
    read_institution(InstDef),
    seen.

%*********************************************************************

read_institution(InstDef) :-
    read_institution1(def([],[],[]), InstDef).

read_institution1(InstDef, FinalInstDef) :-
    read(Clause),
    \+ Clause = end_of_file, !,
    add_to_institution_def(Clause, InstDef, NewInstDef),
    read_institution1(NewInstDef, FinalInstDef).
read_institution1(InstDef, InstDef).

add_to_institution_def((Head ::= Body),
                       def(I,D,K),
                       def(I,D1,K)) :-
    append(D, [(Head ::= Body)], D1).
add_to_institution_def(known(Agent,Clause),
                       def(I,D,K),
                       def(I,D,K1)) :-
    append(K, [known(Agent,Clause)], K1).
```

D.4 Utilities - util.pl

```
:-use_module(library('linda/client')).
:-use_module(library(lists)).

find_server(Server, PID) :-
    see('~/agent/server.addr'),
    read(server(Server, PID)),
    seen.

add_message(Server, PID, From, To, Content):-
    linda_client(Server:PID),
    out(message(From,To,Content)),
    close_client.

read_message(Server, PID, From, To, Content) :-
    linda_client(Server:PID),
%    in(message(From,To,Content)),
    in_noblock(message(From,To,Content)),
    close_client.

await_message(_, Server, PID, From, To, Goal) :-
    read_message(Server, PID, From, To, Goal), !.
await_message(N, Server, PID, From, To, Goal) :-
    sleep(1),
    N > 0, N1 is N - 1, !,
    await_message(N1, Server, PID, From, To, Goal).

concat(Atom1, Atom2, Concatenated) :-
    name(Atom1, A1),
    name(Atom2, A2),
    append(A1, A2, AA),
    name(Concatenated, AA).

concat_list([X], X).
concat_list([H|T], Atom) :-
    \+ T = [],
    concat_list(T, AT),
    concat(H, AT, Atom).

assert_if_new(Fact) :-
    \+ Fact,
    assert(Fact), !.
assert_if_new(_).

assert_list([H|T]) :-
    assert(H),
    assert_list(T).
assert_list([]).

set_timeout(Secs:MSecs) :-
    integer(Secs),
    Secs >= 0,
    integer(MSecs),
    MSecs >= 0, MSecs < 1000,
    linda_timeout(_, Secs:MSecs).
```