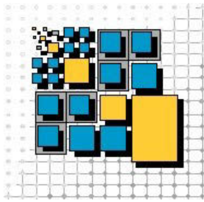

Otto-Friedrich-University Bamberg

Distributed and Mobile Systems Group

Diploma Thesis

in the course of studies Information Systems

at the faculty of Information Systems and Applied Computer Science

Topic:

A Proposal for a Web Services Agreement Negotiation Protocol Framework

Author:

Sebastian Hudert

Advisor:
Prof. Guido Wirtz

Submission Date:
30.09.2006

Contents

1 Introduction **1**

2 Service Level Agreements **5**

2.1 Service Level Agreements 5

2.2 Application in Business Integration Scenarios 5

2.3 Web Services Agreement 6

2.4 Reference Scenarios 9

2.4.1 Science Grids 9

2.4.2 The Amazon Web Services Platform 10

2.4.3 Conclusion 12

3 Negotiations and Negotiation Protocols **13**

3.1 Negotiations in Research 13

3.2 Negotiation Properties 14

3.3 Negotiation Protocols 16

3.4 Auctions versus Negotiations 17

3.5 Sample Negotiation Protocols 17

3.5.1 The English Auction 18

3.5.2 The Dutch Auction 19

3.5.3 The Sealed-Bid Auctions 19

3.5.4 Multi-Attribute Auctions 19

3.5.5 Combinatorial Auctions 20

3.5.6 One-on-One Bargaining 20

4 Data Model **22**

4.1 Taxonomies for Electronic Negotiation 22

4.1.1 Software Frameworks for Advanced Procurement Auction Markets . 22

4.1.2 Classification scheme for negotiation in electronic commerce 23

4.1.3 Parameterization of the Auction Design Space 24

4.1.4 Software Framework for Automated Negotiation 25
4.1.5 Montreal Taxonomy for Electronic Negotiations 27
4.1.6 Summary and Comparison 29
4.2 Negotiation Attributes 30
4.2.1 General Protocol Data 30
4.2.2 Attribute Categories 31
4.2.3 Process Information Category 33
4.2.4 Negotiation Context Category 35
4.2.5 Negotiated Issues Category 36
4.2.6 Offer Submission Category 38
4.2.7 Offer Allocation Category 39
4.2.8 Information Processing Category 39
4.2.9 Process Information Category 40
4.2.10 Negotiation Context Category 42
4.2.11 Negotiated Issues Category 42
4.2.12 Offer Submission Category 43
4.2.13 Offer Allocation Category 43
4.2.14 Information Processing Category 45
4.3 Data Model of Metainformation 46
4.3.1 Used Technology 46
4.3.2 Derivation of Data Entities and Relationships 47
4.3.3 Entity Relationship Model 52
4.4 Extensible Markup Language Documents 52

5 Exchange Protocol **64**
5.1 Negotiation Types and Instances 64
5.2 Abstract Exchange Process 66
5.3 Architecture of Negotiation Objects 68
5.4 Involved Roles and Methods 72

5.4.1 Role: *Negotiation Coordinator* 72
5.4.2 Role: *Negotiation Participant* 74
5.5 Illustration of Sample Protocol Components 75
5.5.1 Request for Negotiation Data 75
5.5.2 Proposal of Negotiation Data 79
5.5.3 Mediated Configuration 80
5.5.4 Update Process of Negotiation Instances 82
5.6 Look-up Server Infrastructure 83

6 Negotiation Protocol 85
6.1 Abstract Negotiation Process 85
6.2 Involved Roles and Methods 86
6.2.1 Role: *Negotiation Participant* 86
6.2.2 Role: *Informationservice* 87
6.3 Illustration of Sample Protocol Components 88
6.3.1 1:n Negotiations 88
6.3.2 1:1 Negotiations 90
6.3.3 Use of the Information Service 91

7 Web Service Description Documents 94
7.1 Fault Messages 94
7.2 Negotiation Coordinator 95
7.3 Participant 99
7.4 Information Service 103

8 Example Negotiations 106
8.1 Used Constraint Language 106
8.2 Auction 107
8.3 Bargaining 112

9 Conclusion and Prospects 115

Bibliography **118**

A Web Services Description Language 2.0 **123**

B Erklärung **137**

List of Figures

1 Web Services Agreement Structure 8

2 Agreement Creation and Monitoring Process 21

3 Montreal Taxonomy Structure 28

4 Relationship between *Domain* and *Values* parameter 37

5 Entity-Relationship Schema of Negotiation Types 52

6 Process of requesting all negotiation types available 76

7 Process of requesting all instantiated negotiations 77

8 Process of requesting available negotiation types and instantiating a negotiation 78

9 Process of requesting current negotiations and joining of the respective agent 79

10 Process of proposing a Negotiation Instance and joining of the other agent 80

11 Mediated Exchange Process 81

12 Process of updating a Negotiation Instance and redistributing it 83

13 Auction Process 89

14 One-on-One Bargaining Process 91

15 Auction Process with usage of the *Information Service* 93

Acknowledgement

This work was done in cooperation with Dr. Heiko Ludwig, IBM TJ Watson Research Center, New York. He made a major contribution to this thesis by assisting the author in assessing and formalising concepts as well as in delimiting the scope of this work.

Abbreviations

B2B	Business-to-Business
B2C	Business-to-Consumer
CDA	Continuous Double Auction
EAI	Enterprise Application Integration
e-commerce	Electronic Commerce
EPR	Endpoint Reference
ERM	Entity Relationship Model
ERP	Enterprise Resource Planning
FIPA	Foundation for Intelligent Physical Agents
GGF	Global Grid Forum
HP	Hewlett Packard
ID	identifier
Jess	Java Expert System Shell
LEAD	Linked Environment for Atmospheric Discovery
MAS	Multi-Agent System
NEES	Network for Earthquake Engineering and Simulation
QoS	Quality of Service
SLA	Service Level Agreement
SOAP	Simple Object Access Protocol
TTP	Trusted Third Party
UDDI	Universal Description Discovery and Integration
UML	Unified Modelling Language
WS	Web Service
WS-Agreement	Web Services Agreement
WSDL	Web Services Description Language
WSRF	Web Services Resource Framework
XML	Extensible Markup Language

1 Introduction

During the last decades emerging information and communication technologies fundamentally changed the way business is done. Not only did such techonologies provide companies with more and more complex information systems aiding internal processes like logistics or manufacturing, but also with new possibilities to conduct business transactions between different enterprises (Business-to-Business (B2B)). Most of the paperwork formally used in business life is replaced by data files handled by computers, like orders, bills etc. After the development of fundamental ciphering algorithms even financial transactions can be conducted in a very secure manner. Almost every business transaction concerning information is conducted electronically by now. The new information and communication technologies provide means for very elaborated supply-chain managment and administration of business relations. The effort needed to process orders or bills between already cooperating business partners is reduced to a minimum. Also the creation of new business relationships can be supported by providing information about the potential business partner on business portal web-sites or by easing the information exchange with email or video conference systems.

In Business-to-Consumer relationhips (B2C) the increased availability of internet access led to numerous company web-sites, offering a wide range of different services and products. Some companies only provide some information about their offered products while others also allow to purchase these by providing some kind of online-store functionality. Even very complex internet portals emerged offering all kinds of services in addition to online shopping, like registering users for receiving newsletters, incorporating user requests into new development cycles etc. Examples for such companies offering portal web-sites would be BMW [1] or Hewlett-Packard (HP) [2] among many others. Both offer a multitude of services via their portals, like online-shops, comprehensive information for enterprise or private business partners and newsletters one can register for.

The most recent step in that development is marked by a new paradigm concerning how distributed systems are to be created: service oriented computing.

Initially message passing systems were employed, focusing on messages and message formats to be transmitted from one system, respectively one company, to another one. The next remarkable step in this development was object-oriented computing which considers computational entities, called objects, that consist of some data and some methods that offer functionality to the other entities. The focus shifted from the messages passed to method invocations not caring about the underlying messages any more. The next big step was the already mentioned service oriented computing which presents a slightly different point of view: everything is a service. All functionality is provided as a service. Data is stored on some device which again offers the data access methods as a service. In contrast to object-oriented computing service orientation focuses on actually deployed services that can be combined to complex applications. Objects represent a computational concept specifying the way software systems are constructed, while not focussing on where these objects are available and how to interact them. This concept presents a new point of view on distributed systems, based on actually running distributedly available software entities: the services.

[1] http://www.bmwusa.com/
[2] http://www.hp.com/

This new paradigm led to a whole new understanding on how to implement cooperating information systems within and across company boundaries. Individual software systems weren't coupled as tightly as before, because they did not focus on special message formats any more but just offered some service in a generalised agreed-upon method using common languages. These commonly used languages and message formats mark the most critical aspect of this new way to create distributed systems. They have to exhibit sufficient expressiveness to enable the definition of various services, related message formats and corresponding data structures. On the other hand they should be easily understandable and applicable in order to reach a high acceptance in the software designer community. Another, quite important aspect is that mostly the different companies yet use older software systems internally as well as within B2B transaction. In order to support such systems the new languages have to be able to wrap older programming techniques or even paradigms in a transparent way. This guarantees a seemless changeover from older computing paradigms to new service-oriented systems.

The web services related specifications, like WSDL [1] , SOAP [2] or UDDI [3] provide software designers with instruments to achieve these desired aspects. These standards define means to describe services, message formats or to register and look-up services in distributed environments.

This service orientation led to a completely new way of how enterprises cooperate. Not only did they exchange information concerning business transactions or supply-chain managment but also formerly internally used information services can be offered to other companies. Such services, like accounting, stock management etc., are then charged for their usage respectively. This crucially changes the competition between such business partners as well as the requirements on the offered, formerly internal services. In the past, such business services were considered to only be accessible within a company. Now all enterprises can offer services incorporated into their internal information systems to all other market participants. Like producing goods, this allows companies to specialise in certain services and therefore profiting of economies of scale. The offered services develop from elements of the business administration to components to be sold to potential customers. This leads to situations in which managers have to decide whether to keep using the in-house service or to purchase services over the market which could be reasonable due to cost reasons. Consequently a huge amount of out-sourcing projects emerged where managers decided to use external, perhaps much more specialised services instead of the in-house pendants because of efficiency and cost reasons.

However, each external service used produces high risks for those services cannot be controlled by the invoking company. This is exceptionally important if the service is a crucial one for the purchasing company's core business. This company critically depends on the external service provider. In order to minimize this risk usually only non-critical services are purchased externally. Another approch was taken when service level guarantees were introduced. Such guarantees represent an approach to secure service consumer and provider relationships by expressing desired service properties which normally include attributes like availability or response time, that have to be met by the involved parties. In addition to such guarantees penalties for violations of such are specified. Service level guarantees provide the service consumer with reliable service execution and therefore with more predictability of the results. Furthermore these guarantees ease the capacity management for the service provider by providing information about individual service invocations. Hence service level guarantees or agreements (SLA) represent contracts be-

tween a service provider and consumer concerning some provided services in order to reduce the risk associated with the usage of these services.

The Global Grid Forum (GGF) [3] proposed a specification defining the structure of such SLAs within web services environments, the Web Services Agreement specification [4] (WS-Agreement). It will be presented in further detail in the next section 2.3.

At the moment the WS-Agreement specification defines a very simple protocol for how an agreement is reached. Either the service provider or the consumer proposes an agreement document to the other one which in turn accepts or rejects it.

As already known from economics or game theory research [5, 6, 7], there exist a lot of different negotiation protocols used when two or more parties try to reach an agreement from an initial conflict situation. Such protocols aim to generate the best possible agreement in terms of overall utility or efficiency etc. As can clearly be seen the agreement negotiation protocol offered, a simple request-response protocol, can barely achieve such goals. Also the offered protocol is highly unflexible. There is no way of generating counter-offers or allowing multiple participants in a negotiation about a service. This ultimately leads to agreements that have high potential for improvement, or to no agreements at all where one would have been theoretically possible with a different protocol.

The objective of this thesis is to address the issue of incorporating multiple negotiation protocols in the WS-Agreement specification. Enabling automated negotiations on such WS-Agreements is especially important for it allows for a more flexible creation of SLAs and will result in better agreements for both involved parties than a simple request-response protocol, as defined in the current specification draft, would.

Sections 2 and 3 provide an survey of SLA concepts and negotiation protocols. In section 4 the required attributes needed to define the different negotiation protocols will be defined using negotiation taxonomies originating in economic and computer science research efforts.

The enormous amount of services offered over the internet along with the huge amount of company information systems using and offering these causes the need for automated negotiations among software agents; in other words software programs that act and react autonomically according to some specified goals [8, 9]. This is the only way these countless negotiations can be conducted in an efficient way. Predictions for the future state, that the available amount of information technology specialists that would be needed to administrate the evolving amount of information systems is by far not sufficient. This again leads to the need of software systems that are able to behave in a way that a high amount of administrative tasks can be conducted without human interaction (self-managing and self-adapting systems). This concept marks the core assumption of the new challenging paradigm in system administration called Autonomic Computing [4]. This approach tries to develop software systems able to manage themselves without or with as little human help as possible. This thesis provides such systems with a means to automatically negotiate SLAs by defining the already mentioned negotiation attributes in a machine-processable manner.

After defining a common language for negotiation protocols in terms of the derived at-

[3] http://www.gridforum.org/

[4] http://www-03.ibm.com/autonomic/

tributes an exchange protocol is derived in section 5. This protocol is used to exchange this negotiation information to all prospective negotiation participants. Finally a generic negotiation protocol is presented in section 6 to enable the execution of the negotiation protocols defined with the means of the formerly presented datamodel.

This approach provides software designers with the required method declarations, message formats and data structures to implement software agents that are able to negotiate SLAs concerning some offered service(s). Negotiations as used in SLA environments normally involve two negotiating parties (service provider and consumer). Negotiations can therefore be conducted between only two agents, in case both sides consist only of one participant or of multiple agents in case of more participants per side. This thesis will only consider 1:1 or 1:n negotiations, that is at least one side only consists of one participants.

Accordingly a datastructure and respective protocols for only 1:1 and 1:n SLA negotiations in web services environments will be presented.

Note: n:m negotiations are not treated due to several reasons. Every n:m negotiation, e. g. a market or a continuous double auction (CDA), needs some kind of centralised instance handling the matching of offers while all participants on each side will regularly leave and rejoin the market. SLA negotiations however are commonly conducted between two agents negotiating over some SLA or between one agent offering/requesting a SLA to/from a set of other agents; both scenarios do not incorporate some central market instance. Also markets inherently exhibit a indefinite duration for the negotiation process which is semantically hard to express. Since this thesis focuses on such bilateral or multilateral service negotiations conducted without requiring some hierarchical ordering of services, as would be needed in a marketplace, n:m negotiations will not be considered. Only negotiations between agents directly or indirectly involved in the resulting service invocations will be involved in the negotiations as proposed by this framework. The *Negotiation Coordinator* rule introduced later represents a promising concept for such a centralised market instance, however. The incorporation of n:m negotiations into this work, perhaps employing this possibility, will be subject of future work.

This work will conclude by presenting the respective interface descriptions expressed as Web Service Description Language (WSDL) documents in section 7 and by describing two sample negotiation protocols with the means of this framework in section 8.

In the following the concepts agent and service will be used regularly. Service defines a set of methods/functionalites exposed to other software systems, while an agent denotes a software agent implementing the respective service. Both concepts will be used interchangeably in this thesis, because of their close relation. The context these terms are used in will unambiguously state whether the software agent or the service it exposes is meant.

2 Service Level Agreements and Web Services Agreement

This section will give an overview of the concept mainly used in computer science to define, structure and monitor quality-of-service (QoS) aspects: a SLA. Subsequently the implementation of SLAs for the Web Services environment, the WS-Agreement standard [4] is presented. This section will conclude by describing some reference scenarios the WS-Agreement specification and consequently this WS-Agreement Negotiation Protocol Framework, are supposed to support.

2.1 Service Level Agreements

In general a SLA represents an agreement between a service provider and a service consumer concerning the quality of the provided service. A SLA therefore comprises the involved parties, e. g. service provider and consumer, the concerned service and the QoS guarantees to be ensured by the service provider. Normally a SLA also defines methods to monitor the QoS aspects, in order to detect violations, and in case a violation occurs, penalties for the violating party (most commonly the service provider).

After regular application of SLAs in computer networks research recently much effort is done to adopt such concepts and methodologies, used in the lower level network services, to provide QoS guarantees for more abstract and aggregated high-level service. This eventually aims on high level business services provided and used by real-life business partners.

The remainder of this thesis will refer to the following definition of a SLA:

A Service Level Agreement represents an (electronic) contract between a service provider and a service consumer concerning one particular or a bundle of services. This contract defines the set of QoS guarantees given by the service provider for this concrete service instance(s). Optionally the SLA also defines the used quality metrics for the different concerned attributes of the service(s), the way they are assessed, and therefore monitored, as well as the penalties resulting from a violation of the SLA.

2.2 Application in Business Integration Scenarios

In the last couple of years a new vision of business interaction has emerged, where arbitrary "'service consumers and service providers can locate each other over the Internet, negotiate terms and conditions of business electronically, connect with each other dynamically, transact business and tear down their relationship when it is no longer needed"' [10]. In the depicted scenarios organizations can offer and buy electronic services of any kind and complexity and integrate them in their application architecture [11].

This includes outsourcing business services dynamically, if the externally provided services are of better quality or lower costs than the in-house pendants, as well as providing complex enterprise resource planning (ERP) application services to other organizations. This scenario of highly dynamic composition of services within and across business boundaries

clearly needs underlying techonology enabling runtime discovery and binding of services. The Web Services [12] technology alongside related standards like WSDL [1] or Universal Description, Discovery and Integration (UDDI) [3] provides such an integration platform [11]. With the Web Services framework services can be offered to consumers over commonly used internet protocols using Simple Object Access Protocol (SOAP) messages as communication mechanism [2]. The WSDL standard facilitates the accurate description of a Web Service in terms of its interfaces, acceptable messages and data types used. UDDI describes directory services used for finding and dynamically binding to the appropriate Web Services at run-time. These advantages have caused the Web Services framework to become the major Enterprise Application Integration (EAI) technology.

In order to create reliable relationships between the different involved parties in a business transaction they have to create agreements regarding the quality of the involved services. SLAs are needed to define the expected performance of the used web services and the corresponding metrics like average response time, supported throughput, service availability etc [13]. Integration of different services, and therefore SLAs, is not only needed in scenarios where different organizations buy electronic services from each other, but also when different organizational units within one company interact.

To augment the common Web Service framework with SLAs new technologies are needed. The WS-Agreement standard presented in the next subsection represents an attempt to "‘provide standard means to establish and monitor agreements on services independent of a particular application domain"’ [14].

Even in a very simple business scenario, there are countless different possible agreements the service providers and consumers could reach. Thus the way such an agreement is derived should not be restricted to one particular process. Unfortunately the WS-Agreement standard only defines one way to generate an agreement between two parties: the agreement initiator proposes such an agreement and the agreement responder accepts or rejects this offer [4]. The main goal of this thesis is therefore to augment the WS-Agreement standard with the possibility to conduct arbitrary negotiations in order to reach an agreement. For this purpose a data structure used to describe negotiations, a protocol to exchange this data and a protocol to conduct the negotiation itself is introduced.

2.3 Web Services Agreement

The WS-Agreement is a standardization effort conducted in the GGF in order to facilitate creation and monitoring agreements between a service provider and consumer. The specification draft [4] defines an Extensible Markup Language (XML) [15] representation of agreements and agreement templates, a simple agreement establishment protocol (which is to be extended by this thesis) as well as corresponding interfaces for creating an agreement at binding-time and monitoring it at runtime [14]. Runtime monitoring of agreement states utilizes Web Services Resource Framework standards (WSRF) [16].

Roles in WS-Agreement: As depicted before WS-Agreement defines two roles in creating agreements: *agreement initiator* and *agreement responder*.

These two roles are completely independent from service provider and consumer. Service

providers can act as agreement reponders, which would be the intuitive configuration because a service provider would be coupled to the service a lot closer. It therefore would be able to enforce agreements much easier than the service consumer would. However, this specification explicitly allows the opposite configuration, too.

Since the distinction between service provider and consumer is not significant in the context of WS-Agreement, in the following only *initiator* and *responder* will be distinguished.

Conceptual Architecture and Interaction Protocol: WS-Agreement depicts a layered service model consisting of two layers: the *service layer* and on top of that the *agreement layer*.

The *service layer* represents the domain- and application specific part of the proposed architecture. It contains the actual services the agreements are created for. Since this layer is domain-specific the WS-Agreement standard doesn't restrict this layer at all to any particular technology (e.g. Web Services).

"'The *agreement layer* provides a Web service-based interface that can be used to create, represent and monitor agreements"' [4]. The agreement responder exposes two different interfaces: an *agreement factory* and an *agreement* interface. The *agreement factory* enables creating an agreement. In order to facilitate agreement creation it can also provide agreement templates, from which concrete agreements can be derived. An agreement template represents an agreement with fields to be filled in. The derived agreement cannot only bind to existing services but also services can be created per agreement (potentially by the agreement layer) [4].

In order to create an agreement the agreement initiator proposes an agreement, optionally derived from an agreement template. The agreement responder then checks the offered agreement in a two-staged manner: first the offered agreement is syntactically checked, which means its structure must accord to the one defined in the WS-Agreement standard and, if it was created from an agreement template, it must not violate any constraints given in the template. After that the agreement responder decides to accept or reject the offer according to its resource situation.

In order to facilitate runtime monitoring of the agreement state the *agreement factory* also returns an Endpoint Reference (EPR) to an agreement service (*agreement* interface). This interface exposes operations as defined within the WSRF [16], allowing runtime inspections of the defined agreement's state. For further information on the possible runtime states of agreements or agreement terms, the acceptance process or the interfaces for agreement creation and monitoring see the specification draft document [4].

Structure of Agreements and Agreement Templates: Agreements and agreement templates are both defined using the XML language. The high-level elements are illustrated in the following picture.

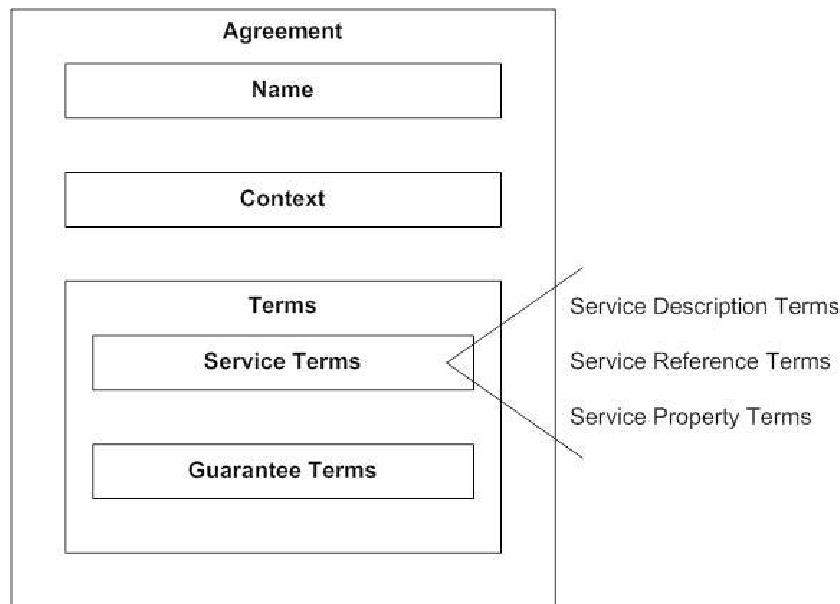

Figure 1: Web Services Agreement Structure

First of all every agreement and agreement template is identified by an agreement (template) id attribute. This id has to be unique between the agreement initiator and responder. The main body of any agreement consists of an optional name element used for human understandability, a context section and the agreement terms. Agreement templates additionally contain a constraint section [4] which will be described later.

The context section contains information about initiator and responder of the agreement along with other metatdata concerning the agreement as a whole, like the id of the service provider, the duration of the agreement or the id of the template it was derived from.

The agreement terms represent the main part of an agreement. They denote the obligations of the involved parties resulting from the agreement. The WS-Agreement defines two types of terms: *service terms* and *guarantee terms*.

Service terms "'provide information needed to instantiate or otherwise identify a service to which this agreement pertains and to which guarantee terms can apply"' [4]. The *service terms* are further subcategorised as *service description*, *service reference* and *service property terms*.

Service description terms are the most essential parts of an agreement. They define the functionality of the service - existing or not - for which the agreement is created.

Service reference terms point to an existing service (e. g. by providing an EPR) to which the agreement relates.

Service property terms "'define measurable and exposed properties associated with a service"' [4]. The measurable aspects of a service are described as a set of variables. Each variable relates to an attribute of the service (therefore to a field in the service terms) and is associated with a metric to enable evaluation of this variable.

In order to define assurances on service quality for the described services additional *guarantee terms* can be specified. *Guarantee terms* therefore represent the service levels both parties are agreeing on.

Each guarantee term specifies the obligated party of the guarantee which is needed for enabling consumer-side guarantees. In addition to listing the services the guarantee applies to it can also define an optional qualifying condition which must be met for the guarantee to be enforced [4].

Qualifying conditions are assertions over service attributes and/or external factors such as date or time. The actual guarantee is described in the service level objective element. Also each guarantee is accompanied with some business values. Business values represent different value aspects of an agreement. This construct can be used to represent the importance of an agreement, the associated penalties and rewards for compliance to or violation of an agreement or preferences on different service configurations.

All elements described up until now are present for agreements as well as for agreement templates. However, agreement templates can be supplemented with an optional section called agreement *creation constraints*. This element specifies "'constraints on possible values of terms for creating an agreement"' [4].

2.4 Reference Scenarios

In this subsection two possible usage scenarios of WS-Agreements and respective WS-Agreement negotiations will be sketched shortly. To show the variety of possible application domains two scenarios were chosen that differ quite fundamentally in terms of overall goals or architecture of involved services. However, both represent actually in-use information systems heavily relying on SLAs, expressed for example as WS-Agreements, which would benefit from automated agreement negotiations crucially.

2.4.1 Science Grids

The first application domain for WS-Agreements and respective negotiations to be presented is the emerging concept of large scale grid systems for scientific simulations. A grid in general is a network of (possibly heterogenous) compute and data resources located in various different administrative domains, such as universities, private service providers or laboratories. These resources are supplemented with a layer of grid services providing uniform access to the logical resources as well as authentication and authorization tasks and possibilities to create, run and monitor experiments and workflows on the grid, employing the distributed logical resources respectively.

Grid applications regularly address two main goals: incorporate heterogenous resources, such as sophisticated services provided by some nodes within the network, into more comprehensive applications and combine computational power and data storage space of the different servers in the network to achieve extremely high performant systems that would not be possible to implement on just one dedicated server.

These benefits of grid systems along with their possibility to connect distant servers

from all over the world lead to their application in distributed scientific collaborations. Especially research communities employing massive computational power for simulations of for example the climate or earthquakes massively use such grid applications. The Network for Earthquake Engineering and Simulation project (NEES) [5] for example was funded to allow comprehensive earthquake and tsunami research projects by combining fifteen advanced experimental facilities at fifteen different leading universities involved in such research areas. The related project implementing the needed IT infrastructure, a grid-based system of distributed servers, was called NEESgrid. The developed grid system was implemented to support a variety of use cases by incorporating a wide range of different system components including monitoring and control systems, data storage facilities and collaborative work infrastructure.

Analogously the Linked Environment for Atmospheric Discovery portal (LEAD) [6] provides a grid application for weather forecast simulations especially used for tornado predictions. This portal provides a multitude of services assisting researchers in creating weather simulations and real-time predictions of tornadoes and other massively destructive storms. In addition this portal focuses on educational tasks by presenting educational data to a large community of users. This way the initiators try to rise awareness and knowledge on weather phenomena like tornadoes and hurricanes, which ultimately should help society in general to face such devastating storms.

Both scientific grid applications presented is constructed by combining a set of low-level grid resources, such as highly specialised applications or measurement data, with some infrastructure of administrative grid service realizing transparent and secure access to the low-level resources. Each experiment or simulation is specified as a concrete workflow defining the orchestration of involved grid and resource services. Such workflows include how data is transfered from one service to another, when each service is called and how the results are presented and stored. This combination of services to be invoked crucially relies on SLAs between the coordinating and the individual running processes. In such massively distributed environments dealing with very specialised systems and huge amount of processed data SLAs are critically important to ensure a smooth execution of the respective workflows.

The respective SLAs are regularly created manually by the user when creating the workflow. Automated negotiations as enabled with this framework could reduce the necessary user interaction with the system to a minimum. The different workflows submitted at a time in such grid networks can possibly be scheduled in a much more efficient way by applying automated negotiations than by conducting such scheduling tasks manually. Different protocols would also ease the orchestration tasks within a workflow . For example in case of multiple possible servers for the next task to be conducted an auction protocol could easily find the optimal service provider for the respective task.

2.4.2 The Amazon Web Services Platform

Another scenario, suiting SLAs and SLA negotiations is a system of middle-tier web services regularly offering their services to customers for a period of time, not for a single invocation. This class of services does not focus on the individual invocations by discrete

[5] http://it.nees.org/
[6] http://lead.ou.edu/

customers. The services are rather made available to some customers for a distinct amount of time during which they can invoke the service as many times as they want. A good example for a set of such services is the Amazon.com Web Services platform [7].

Amazon.com is not only one of the biggest online retailers in the world, but also a comprehensive technology platform, on which more than 1 million active retail partners do business. In the beginning Amazons architecture was quite simple and only embraced a monolithic application on an application server connected to a database. Later the company made a huge effort to convert the monolithic Amazon.com application into a system of independent web services. This convertion did not only allow for independent development and maintainance of the individual services, and therefore increased scalability and manageability, but also opened the doors for a whole new level of cooperation with business partners in the newly created service oriented architecture [17].

Today Amazons portal itself is constructed of a multitude of different web services using WSDL descriptions [1] with optimised transport and marshalling technologies for efficient resource use. For cooperation with retail partners wanting to advertise their products on Amazon.com other services are provided using web service or feed-processing interfaces. This way the business partners can leverage already in use Amazon services like shopping cart management or recommendation systems to efficiently promote their products on a world-wide available platform without having to implement all these services themselves.

Amazon also offers a large collection of e-commerce platform services to enterprise business partners which in turn can be used to build fully independent e-commerce websites with personalised branding. This way other organizations can promote their products on their own individual websites, using the expertise of Amazon.com whenever wished. For example a company could only use Amazon's shopping cart related services while other ones would implement own shopping carts but wish to use the similarity search or recommendation systems of Amazon.com. This way a multitude of services for the retail customers can be developed (even by other companies than Amazon.com itself) using the complete range of Amazon functions.

Such applications rely on the web services offered by Amazon.com and therefore crucially depend on their availability and robustness to work properly. Especially for long-time relationships, like with enterprise retail partners or smaller development companies providing software solutions based on Amazon Web Services, QoS guarantees would help making the very dynamic connections between the business partners more predictable. Even though for long running business contracts human negotiation would do in most cases those agreements will mostly denote only frame contracts. This is because of the unpredictability of the consumers' behaviour using the services.

In order to manage the resource allocation for concrete service invocations human negotiation would not be appropriate. The sheer volume of negotiations to be conducted would simply exceed the human capability of conducting negotiations.

Hence it is necessary, for negotiating certain service guarantees (even within given frame contracts) to automatically negotiate over resource needs of incoming requests. This way Amazon services would be able to provision more service resources if needed or to intelligently queue service invocations in order to minimize the violations of the QoS contracts.

[7] http://www.amazon.com/b/ref=smm_sn_aws/102-4173233-1544160?ie=UTF8&node=3435361

WS-Agreement can help in defining alternative resource configurations needed for service invocations with associated business values like penalty or importance. Enabling automated negotiations over QoS aspects with the associated business values would enable a more efficient provisioning of service resources by increasing predicability of resource needs.

2.4.3 Conclusion

Both scenarios sketched in this section show application domains for SLAs in assuring QoS aspects for Web Service communications. However, currently the WS-Agreement standard defines only one way to establish an agreement between an agreement initiator and a responder. As shown in the reference scenarios, however there are numerous situations and application domains to which SLAs according to the WS-Agreement specification could be applied. As different these situations are, as different are the individual demands made on the agreement creation process. One could wish to negotiate the different service terms in different rounds or to negotiate with more than only two involved agents, as with auctions.

Unfortunately all those negotiations would not be possible with the current specification draft. In order to enable the WS-Agreement standard to be applied to a much more comprehensive set of situations, this thesis will present a negotiation protocol framework applicable for negotiating agreements according to the WS-Agreement specification. After deriving a data model of the negotiation metadata, needed for prospective negotiators to participant in the respective automated negotiation, an exchange protocol for this information will be defined. Finally a generic negotiation protocol used to actually conduct negotiations that have been described with the introduced data model will be presented.

3 Negotiations and Negotiation Protocols

Before deriving the negotiation data model this section will give a short overview of negotiation protocols in research and practice. First negotiations and their properties as discussed in research are presented. After defining negotiation protocols and distinguishing two major concepts in this research field, some sample protocols will be given to illustrate the introduced concepts.

3.1 Negotiations in Research

In general negotiations constitute the process of two or more parties communicating in order to proceed from some conflict situation to an agreement. Hence negotiations are carried out in situations where the involved parties cannot reach a consensus unilaterally and have to coordinate with each other. Like indicated by this very abstract definition of the negotiation concept, it covers a multitude of different negotiation types conducted in various situations differing in the involved parties, the means of communication used and the social and cultural circumstances.

Since the contexts in which negotiations take place are numerous there are many different definitions of negotiation processes describing the concept from different perspectives. Pruitt for example [18] describes negotiations as "'a form of decision making in which two or more parties talk with one another in an effort to resolve their opposing interests"' ([18], page xi). "'The parties first verbalize contradictory demands and then move toward agreement by a process of concession making or search for new alternatives"' ([18], page 1). Gulliver characterizes a negotiation as "'one kind of problem solving process - one in which people attempt to reach a joint decision on matters of common concern in situations where they are in disagreement and conflict"' ([5], page xiii).

Both approaches focus on the involved parties and their opposing interests. Since both definitions represent sociological and psychological approaches they don't present concrete concepts used in negotiations to reach an agreement, which would be necessary for adopting the negotiation concept in a fully automated scenario. Mertens gives another definition of negotiation already related to electronic negotiations: "'Negotiation is a process of social interaction and communication about distribution and redistribution of power, resources and commitments"' [19]. This definition explicitly references the exchanged resources or commitments, which makes it more applicable in e-commerce or SLA scenarios.

Bichler et al. describe negotiations as "'iterative communication and decision making process[es] between two or more agents (parties or representatives) who: 1) cannot achieve their objectives through unilateral actions; 2) exchange information comprising offers, counter-offers and arguments; 3) deal with interdependent tasks; and 4) search for a consensus which is a compromise decision"' [20]. This definition explicitly describes the information exchanged in the communication process as offers and counter-offers. Additionally the parties involved in a negotiation are further specified to be the parties, affected by the consensus to be reached, or alternatively agents acting on their behalf. Bichler et al. also introduce the concepts of negotiation arena as "'place where the negotiators communicate"', decision making rules as rules "'used to determine, analyse and select decision alternatives..."' and rules of communication as rules to "'determine the way

offers and messages (...) are exchanged."' [20]. These concepts allow to concretize the abstract concept of negotiation. They facilitate the specification of a concrete negotiation process in terms of its specific rules, its participants and context.

The remainder of this thesis will refer to this definition of a negotiation process, because of the possibility of describing a negotiation in a very comprehensive way. The different sets of rules allow to unambiguously define a concrete negotition as precisely as needed to conduct electronic negotiations with software agents which is the case in the intended SLA scenarios.

As already scetched various research disciplines have investigated the different aspects to negotiations from different perspectives. The outcomes of the different approaches were metrics, descriptive and prescriptive models, rational strategies in negotiations, ways to predict outcomes and their stability as well as tools and technologies for assisting the negotiating parties. Economic research focuses on formal models of negotiations, stability and equilibrium concepts and rational strategies (compare [21, 22, 23]). Such concepts represent negotiation attributes that can be used to compare different negotiation protocols in terms of these parameters. This has led to a research discipline called Mechanism Design which is concerend with designing "'systems so that certain systemwide properties (for example, efficiency, stability and fairness) emerge"' [24]. Political and Law sciences try to give advices on how to behave in negotiations and therefore present heuristics, prescriptive and descriptive models on negotiation processes [25, 26]. Psychological and Sociological research is concentrating on the motives, interests and behavior of the negotiators as well as their coordination. [18, 27]. Hence these disciplines analyse interpersonal communications and provide heuristics and qualitative models regarding negotiations. From this short (and by far not complete) list of involved research disciplines and their perspectives one can easily see that the field of negotiation research is a very interdisciplinary one. Only such an approach of integrating different research areas promises to enable comprehensive understanding and correct design of negotiation processes. Even though this thesis will not cover points like the agents' strategies, equilibria or stability, a short list of desirable negotiation properties will be given in the next subsection. This allows to assess some negotiation protocols presented later in terms of their applicability in the desired scenarios.

3.2 Negotiation Properties

The negotiation properties given in this subsection can only indirectly be influenced by the negotiation designer. They represent criteria used for the assessment of different negotiation protocols, after they have been designed. This way newly created negotiation protocols can not only be evaluated in terms of certain properties, but they can also be compared to already existing ones.

The properties presented here mostly originate in Mechanism Design which is concerned with designing mechanisms or protocols resulting in some desirable characteristics. Since there do exist a lot of possible criteria for negotiation protocols within economic research only some selected ones will be presented in the following list:

- Pareto Efficiency
  Pareto Efficiency is a property of a certain solution or result of a negotiation. A

solution is pareto-efficient, if there is no other possible solution which is better for one agent without being worse for another agent [28]. That means, a pareto-efficient solution of a negotiation can only be modified in order to improve the results for one agent by reducing the resulting benefits for another one [29]. There is no possible way of modifying the solution or agreement in a way that all involved agents get the same utility out of the negotiation as before but at least one party is better off. This very common criterion is often used to assess protocols in economic theory. Negotiation protocols that lead to pareto-efficient solutions should always be preferred over those that do not because they exploit all utility or payoff possible. Protocols that are not pareto-efficient still leave room for improvement for some agents without having to reduce the resulting utility for all others.

- Welfare Maximization
  Negotiation processes that lead to solutions which maximize the sum of utilities of all participants are defined to maximize social welfare of the involved agents [28]. Since each agent defines an internal utility function defining the gained uitility for the reached agreement the welfare maximization (or allocative efficiency [30]) can be seen as a distributed optimization problem. Another property related to welfare maximization is payoff maximization. This criterion defines whether a negotiation protocol maximizes the individual utility of a certain participant. Depending on the scenario, welfare or payoff maximizing properties can be desirable.

- (Speed of) Convergence
  The convergence criterion defines whether the negotiation protocol converges to an agreement at all [28]. Another related property is the speed of convergence which relates to the speed with which a negotiation protocol produces an agreement [30]. Speed, as used with negotiation protocols, can be defined in terms of time intervalls or of necessary steps from the start to the reached agreement. Clearly convergence is to be aimed at by every negotiation protocol because it defines whether such a protocol reaches a compromise at all. Different converging protocols can differ in their speed of convergence, however. All other properties being equal protocols with higher speed of convergence should be preferred.

- Computational Complexity
  Computational complexity refers to the complexity of the negotiation algorithm itself. Although the software agents conducting negotiations will have increasing resources at their disposals as the employed machines get faster, due to technological progress, algorithms with low complexity are to be preferred over those exposing high complexity if possible. Often this property refers to the offer matching algorithm and not the complexity of taking part in a negotiation. Matching algorithms can inherently be very complex as can be seen with multidimensional auctions which can easily represent NP-hard problems [30].

- Distribution of Computation
  The property is primarily concerned with technical issues of electronic negotiations. Distribution of computation defines the degree of de-centralization of computation tasks in a negotiation. Negotiation protocols distributing the computation over all participating agents should be preferred over those that concentrate the computational effort on one centralised server [29]. Such systems avoid crucial issues in distributed systems like performance bottlenecks and single points of failure.

- Communication Efficiency
  Since a negotiation represents a highly communicative process with lots of messages to be transmitted mechanisms that handle communication efficiently should be preferred over those that don't [29]. Communication efficiency can be reached by reducing the number of transmitted messages to a minimum as well as avoiding redundancy concerning the messages' content. This way the traffic of a negotiation protocol can be kept at a minimum.

3.3 Negotiation Protocols

As described in the previous section, there exist numerous approaches on investigating negotiations, all with different perspectives and focuses. In order to conduct a concrete negotiation the above mentioned definitions have to be made more concrete. Rules have to be derived that unambiguously describe what has to be done by which negotiation participant in a specific situation or state. This leads to the definition of negotiation protocols.

A protocol in general explicitly describes the rules of an interaction between two or more parties. This involves the definition of states, the individual participants can be at, the possible actions of the participants dependent of the current state and optionally particular stimuli like received messages for example. Another facet of protocols is to define the messages and according message formats used in the interactions. In order to model reactive behavior the sending and receiving of messages can be described by rules of state transitions showing what actions are conducted (for example internal tasks that are executed or messages that are sent) when a certain message is received. This allows to model possible, or compulsory, reactions to specific messages. A negotiation protocol consequently describes the interaction of the negotiating agents. In relation to the definition of a negotiation process by Bichler mentioned above, a negotiation protocol includes rules about the negotiation arena and the chronology of permissible decision-making and communication activities [20]. Negotiation protocols therefore mark the application of the protocol concept within (electronic) negotiation

Such an attempt to structure processes and define the permissible behavior of the participants plays a key role in electronic interactions. Due to the deterministic nature of software agents every electronically conducted interaction has to be exhaustively described by deterministic rules.

As indicated by the multitude of negotiation situations there exists a large number of different negotiation protocols. These include traditional negotiations like the English auction [6] that relate to real-world market traditions as well as artificial negotiation protocols that emerged recently in research about automated negotiations and e-markets, like combinatorial auctions [31] or multi-attribute auctions [32, 33].

In the next subsection the concepts of negotiation and auction in general are discussed, before this section will conclude by describing some sample negotiation protocols to illustrate different possible ways to conduct a negotiation.

3.4 Auctions versus Negotiations

Recently online auctions have gained a lot of attention, leading to the assumption auctions are the only class of negotiation protocols there is. The assumption underlying this thesis is, that auctions are just one possible group of negotiation protocols and not the only one. This section will briefly compare these two concepts and justify the assumption made before.

Bichler et al. argue that "‘traditional auctions are resource allocation mechanisms based on competitive biddig over a single issue (i. e. price) of a single, well-defined object"’ [20]. Auction rules "‘specify how the winner is determined and how much he has to pay"’ [7] and therefore govern the auction as a whole. Auction participants post bids indicating their willingness to pay and on termination of an auction a clear is created which assigns the negotiated object to the winning participant, following a "‘set of rules determining resource allocation and prices on the basis of the bids from the market agents"’ [6], above stated as auction rules. Thus, the main goal of an auction is the establishment of a certain value through a bidding process. During this process the value of an object of initially unknown value is determined by the bids received and finally the clear reached [34].

On the other hand, "‘traditional negotiations are based on bilateral, multilateral or multi-bilateral negotiation processes over a single or multiple issue/s of one or more well-, partly, or ill-defined objects and involve cooperation and/or competition among the negotiating agents"’ [20]. Negotiations can therefore also produce win-win situations. This can be defined by using utility as a measure of offers [34]. This concept allows for scenarios with multiple negotiated issues to produce simultaneous improvements for all negotiating parties (integrative negotiating [35]).

The emergence of new electronically conducted forms for decision making processes, like multidimensional auctions, more and more show that the traditional auctions are just a subclass of the negotiation protocols possible. Negotiation protocols do not only include single issue negotiation in some form of bidding process (i. e. auctions) but also protocols like bilateral bargaining or multi-bilateral negotiations, where the participants engage in multiple bilateral negotiations with many other parties [20]. Therefore for the rest of this thesis auctions are assumed to be a subclass of all possible negotiation protocols. The following section will therefore list some of the traditional auction protocols as examples for negotiation protocols, but also multidimensional negotiations or one-on-one bargaining will presented to give a more comprehensive overview on what protocols are possible to conduct negotiations.

3.5 Sample Negotiation Protocols

The following negotiation protocols illustrate the wide variety of possible negotiation processes. By far this list doesn't give an exhausting overview of all possible protocols, it rather presents some typical negotiation processes to prepare for the introduction of negotiation taxonomies in the next section.

First some of the classical auction protocols [6] are listed. The so-called classical auctions, or single-item auctions are auctions concerning only one item to be purchased or sold. In contrast, multi-dimensional auctions denote processes used to negotiate over multiple

units of a single object or multiple attributes of a given item. Some representatives of this classes of negotiation processes will subsequently be listed.

3.5.1 The English Auction

The English Auction, or ascending-bid auction, is by far the most common and thereby mostly known auction type. The defining attribute of an English Auction is that the one negotiatable attribute (mostly the price) is successively raised until only one bidder remains. As with all classical auctions the situation of one seller and many buyers is assumed (1-n configuration). Every auction form is applicable to the opposite situation, as with many sellers and one buyer, as well. In this case the 'main direction of the negotiation' process is inverted. In case of the English Auction if there would be many sellers and only one buyer the price would be lowered until only one seller would remain. For ease of description in the following the one-seller and many-buyer situation is assumed without loss of generality.

After the bidding process a committment is reached between the seller and the remaining bidder. The ascending of bids can be achieved by an auctioneer announcing prices consecutively or by the bidders calling the bids themselves. In terms of the negotiation definition provided above the correlation of bids, in this case the rule that any new bid has to succed the current highest bid, represent a distinct rule of negotiation. An essential feature of the English Auction as well as the Dutch Auction, presented next, is that at any point in time any bidder can detect the current best bid and process this information in order to place his next bid. Every bidder in an English Auction can see the bids posted by the other bidders. This distinguishes the English and Dutch Auctions, so-called open-cry auctions from the sealed-bid auctions, described later. There are different ways to finalize an English Auction: one possibility is to clear the auction whenever no bid was made in a specified time interval. Another way would be a deadline set in advance to the negotiation process. As denoted by the different ways to clear an English Auction there are numerous different variations of the English Auction, varying in the correlation of the bids, the clearing mechanisms or other rules of negotiation.

One very popular example of an English Auction is the auction protocol used on one of the most successful online auction platforms: Ebay.com [8]. For an offer to be accepted by Ebay.com it has to be higher in prices than the current highest bid, additionally a minimum increment of for example 50 cents is applied. The termination rules for a standard Ebay.com auction state, that the final clear will be produced when a certain deadline is reached which is known to all participants. In terms of information revelation Ebay.com provides several means of accessing the negotiation data like offer history for the given product or the most current bid with information on the price offered and the agent offering it. Some extensions made recently to the way Ebay.com is conducting auctions, like the "‘Buy it Now"’ Option introduce new possibilities for creating a final clear and thus terminate the auction.

[8] http://www.ebay.com/

3.5.2 The Dutch Auction

The Dutch Auction denotes another open-cry auction form. However the Dutch Auction represents the exact reverse of the English Auction. An auctioneer calls an initial high price and successively lowers the price until a bidder accepts the current price and therefore agrees to the commitment with the seller. Though the auction form is somewhat similar to an English Auction it differs in one particular fundamental aspect, regarding the information revealed in the process: the involved bidders cannot presume the valuation of the other bidders. This is because in a Dutch Auction only one bid is placed and with that bid the auction ends. Therefore the current announced price is not indicating any valuations of the other participants of the negotiation. Analogous to the English Auction there are many different variants of the Dutch Auction varying in some specific aspects of clearing etc. As it is becoming clear the traditional way to specify negotiation processes by defining the involved roles and communication models is by far not sufficient to conduct arbitrary negotiations electronically. In order to specify a negotiation in a electronically executable form some taxonomies will be presented in the next section.

3.5.3 The Sealed-Bid Auctions

In sealed-bid auctions each bidder posts secret bids which can only be read by the auctioneer or respectively the seller. After collecting the bids from the participating agents the bidder with the highest bid is awarded the item negotiated. In contrast to the English Auction no participant can assess the other bidders valuations and therefore evaluate its chances to win the auction. Furthermore every participant can place only one bid unlike in an English Auction, where one can post several times and thus adopt to the evolution of the highest bid price. There are two major alternative sealed-bid auctions: the first-price and the second-price sealed bid auction. In both sealed-bid auctions the bidder posting the highest bid receives the item and therefore wins the auction. In a first-price sealed-bid auction the winner has to pay the price it offered. In a second-price sealed-bid auction, also called Vickrey Auction, the winner only has to pay the price stated in the second-highest bid. Both sealed-bid auctions have similar economic properties, but the second one is of mainly academic nature and rarely used in practice.

3.5.4 Multi-Attribute Auctions

Multi-Attribute Auctions "‘automate(s) multilateral negotiations on multiple attributes"’ ([32], page 140). They generalize traditional auctions to support not only negotiations on one particular attribute, mostly the price, but also negotiations on multiple different attributes of some good to be purchased or sold. Such attributes could comprise date of delivery or guarantee aspects. Participants of such auctions place offers concerning all negotiated attributes within this auction, not only the price like in traditional auctions. Hence Multi-Attribute (reverse) Auctions "‘combine the advantages of auctions, such as high efficiency and speed of convergence, and permit negotiation on multiple attributes"’ ([32], page 139).

Multi-Attribute Auctions are used in scenarios where the negotiated goods expose more than one attribute that is subject to the negotiation, where there are several agents posting

offers to a single agent (which would be the seller in a forward and the buyer in a reverse auction) and this single agent has "‘certain preferences on these attributes"’ ([32], page 140). The simplest possible algorithm for such auctions would be that every offer has to represent an "‘improvement over the previous bid in at least one of the attributes and no worse in any of the attributes"’ ([32], page 140).

Teich et al. propose several auction algorithms for Multi-Attribute Auctions [33] including the very simple one just described which is called the *leap frog method* there. The *auction maker controlled bid mechanism* however also takes into account the preferences of the seller (or buyer in reverse auctions).

Multi-Attribute Auctions can be structured analogously to the classical auctions according to whether the offers are posted sealed or not. Hence there are multi-attribute equivalents to all classical auctions like Multi-Attribute English and Multi-Attribute Sealed-Bid Auctions etc.

3.5.5 Combinatorial Auctions

Another multi-dimensional auction type is the Combinatorial Auction. In Combinatorial Auctions bidders "‘place bids on combinations of items, called "‘packages"’ rather than just individual items"’ [31]. Each bid therefore represents an offer describing the desired bundle of items and the price one offers to pay.

This auction type is especially useful when complementaries are present among the items to be sold. Items are complementary if the utility of a set of items succeeds the sum of the individual utilities. For example, a person wants to buy a house without a garage for his car. There are also some garages available within the auction. In this case the potential buyer would probably pay more for the house and the one garage that is located right next to the house than he would pay for the house and an arbitrary garage together. This is because these two items are complementary, they are worth more if sold as a bundle, because of the fact that the garage is located directly next to the house and is therefore of much more use than a garage far away.

In order to be able to express such preferences in a more detailed manner than possible with classical auctions, where each item would be negotiated individually, Combinatorial Auctions allow to specify offers on bundles of items. This type of auctions therefore allows the participants to specify their preferences in much more detail than classical auctions would.

Combinatorial Auction algorithms however are much more complex than their classical equivalents as the matchmaking task can easily evolve to a complex optimization problem.

3.5.6 One-on-One Bargaining

One-on-One Bargaining Situations "‘concern as few as two individuals who may try to reach an agreement on any of a range of transactions"’ ([32], page 85). In order to reach this agreement the negotiating agents exchange messages to proceed from the initial conflict situation to an offer acceptable by both of them. In [36] a set of possible message types

for One-on-One Bargaining situations is proposed. These messages include proposals, critiques, counter-proposals, and concepts providing additional informations accompanying the proposals like explanations and meta-information messages.

Proposals represent agreements offered by some agent, that are currently acceptable to this agent. The other agent can respond by sending a critique, which is a "'remark as to whether or not the proposal is accepted, or a comment on which parts of the proposal the agent likes"' [36], or else a Counter-proposal. Counter-proposals are proposals created as an answer to preceding proposals, presenting single complete solutions or partial solutions of the problem faced by the negotiating agents. Explanations and Meta-Information provide additional information about the submitted critiques used for the approach taken by Parsons et al. to augment One-on-One Bargaining with argumentations between agents [37].

This list of sample negotiation protocols already shows the huge variety of possible negotiation types. Each negotiation type differs not only in the respective negotiation process, but also in the resulting agreements. In order to support the huge amount of different scenarios in which WS-Agreement negotiations take place, each possibly demanding a different negotiation protocol, this framework will provide the means to define a multitude of negotiation protocols and conduct them consecutively. This provides a much more flexible approach than the current proposal within the WS-Agreement specification, resulting in superior agreements for the involved parties. The proposed framework should be based on the WS-Agreement specification in terms that the outcome of the negotiations to be defined is a valid WS-Agreement document. Also the negotiation protocols that can be specified with the means of this framework should integrate into the overall WS-Agreement protocol seamlessly. As depicted in the following diagram the negotiation protocols that can be defined using this framework will substitute the agreement creation protocol already defined in the WS-Agreement specification (a simple request-response protocol):

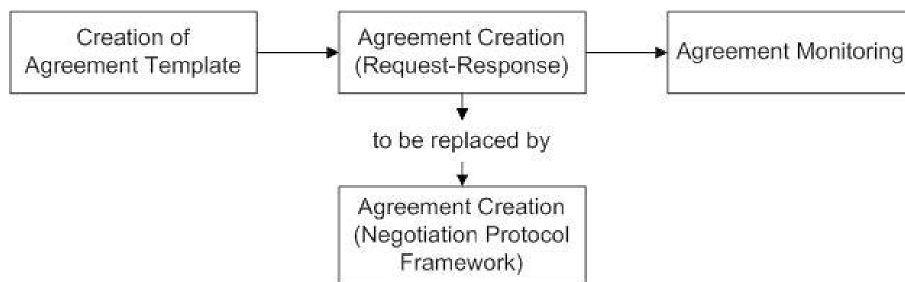

Figure 2: Agreement Creation and Monitoring Process

Furthermore this framework should employ Web Services standards in order to be incorporated into the WS-Agreement standard, which also utilises such technologies.

4 Data Model of Negotiation Metadata

In this section the data model for the negotiation metadata, needed to conduct a respective negotiation, will be presented. This model will allow to specify a multitude of negotiation protocols in a way that allows for application with software agents. It will contain all the information needed for such agents to take part in the particular negotiation. As a theoretical foundation taxonomies for electronic negotiation will be presented first. These already propose concepts to categorize and describe negotiation protocols using a set of attributes or rules. After that attributes and attribute groups will be identified based on these taxonomies, and complemented with associated domains and restrictions. These concepts will be used to define the data model later.

By using taxonomies as a foundation for the attribute identification a very broad range of negotiation protocols can be covered. That is because taxonomies already try to allow description of as much negotiation protocols as possible. Since the different taxonomies focus on very differentiated aspects or types of negotiation protocols it is not reasonable to use only one particular taxonomy as a theoretical foundation. This thesis follows the approach to incorporate the concepts of several taxonomies to one comprehensive set of negotiation protocol attributes. Indeed a lot of attributes will occur in more than one of the taxonomies but there are some concepts that are only presented in one or two that are crucial for an exhaustive description of negotiations. Integrating these different taxonomies allows a more comprehensive description of negotiation protocols by integrating the different foci and different concepts to one exhaustive framework of negotiation attributes.

After defining the attributes the actual data model is presented. It will be modelled as an Entity-Relationship Diagram [38] using the attributes and domains identified before. Finally the structure of corresponding XML documents containing all the negotiation metadata is presented as an XML schema document description. This eases the application in Web Services environments because of the widely spread use of the XML language especially for Web Services applications.

4.1 Taxonomies for Electronic Negotiation

In the following the negotiation taxonomies used for deriving the data model will be presented. For each taxonomy the presented concepts and corresponding foci are described to point out the differences and similarities between them.

4.1.1 Software Frameworks for Advanced Procurement Auction Markets

The first taxonomy used in this thesis has been presented by Martin Bichler and Jayant R. Kalgnanam in their paper "‘Software Frameworks for Advanced Procurement Auction Markets"’ [30]. As already indicated by the name of this paper it focuses on one particular class of negotiation protocols: auctions. Although this approach does not cover as many negotiation protocols as would be desirable it nevertheless presents some interesting aspects that can be applied either to auctions and negotiations in general.

The authors start off by shortly describing the bidding process in general. They identify four different phases: bid submission, bid evaluation, feedback to bidders and optionally reformulation of bids. A protocol description in a comprehensive sense, as intended to be derived in this thesis does not only describe the 'core' activities in such bidding processes like bid submission or feedback to bidders, but also how bids are matched and whether or when bids can be altered or when the auction ends. Although this schema of a bidding process is no more than one particular process for one particular class of negotiations it shows how different protocols can effect the different phases of a negotiation and what concepts need to be covered by a comprehensive negotiation data structure for it to support automated negotiations.

Based on this very generic bidding process Bichler and Kalagnanam propose a conceptual framework for multidimensional auctions. They view auctions and auction rules from a design point of view. Therefore they list and describe some auction properties concerning efficient resource allocation that represent desirable economic properties of an auction. Those properties include concepts like allocative efficiency, speed of convergence or payoff maximization. After that actual auction rules are presented that should lead to these auction properties [30].

Since this thesis aims at the comprehensive description of auctions and negotiations and not at designing negotiations according to certain properties, these negotiation properties are of minor interest. To give a short overview they have been depicted in the last section, though.

In contrast, the auction rules presented subsequently do represent a major contribution to this thesis. They are further categorized into auction rules in a narrow sense and the *auction environment*. The authors identify the auction's participants and negotiated issues as the auction environment and the *auction protocol*, *allocation and payment rules* as the auction rules in a narrow sense. Bichler and Kalagnanam unfortunately only list and shortly describe the attributes an auction has, categorized as just shown. This means that the precise domains and restrictions of these attributes are not identified. Nevertheless the listed attributes, that will be described in more detail later, contributed to the derived data model by presenting significant aspects of auctions and structuring them into semantically consistent groups.

4.1.2 Classification scheme for negotiation in electronic commerce

The next paper proposing a taxonomy for electronic negotiations was published by Alessio R. Lomuscio, Michael Wooldridge and Nicholas R. Jennings and is called "'A classification scheme for negotiation in electronic commerce"' [29]. They emphasize automated negotiations in electronic commerce (e-commerce) settings which is also hinted in the title. To set the context the authors therefore describe typical e-commerce scenarios and software systems already in use in this settings, like primitive shopping assistants or clients easing the product or merchant choice.

Even though these scenarios and software agents do not exactly fit the SLA settings, because of the focus on pre-negotiation phases and high human interaction rates, they already show some of the issues that arise when automated negotiations take place between software agents. Such problems are formalisation of negotiations or implementation of

negotiation strategies by agents themselves. The authors want "'to define (...) the negotiation space for electronic commerce"' [29]. One aspect, that makes this work very suitable as a basis for this thesis is that the authors do not only concentrate on auctions, but on negotiations in general which fits the approach of creating a data structure for multiple negotiation protocols.

Even though Lomuscio et al. focus on the negotiation protocol's parameters they also introduce some desirable negotiation properties like computational efficiency, pareto efficiency or symmetry. These features again show the focus on automated negotiations between software agents, because they concentrate much more on computational aspects as the ones presented by Bichler and Kalagnanam.

The actual negotiation space is represented as a list of attributes furthermore structured into six categories, namely *cardinality of the negotiation*, *agent characteristics*, *environment and goods characteristics*, *event paramenters*, *information parameters* and *allocation parameters*.

Lomuscio et al. list a lot of the attributes that Bichler and Kalagnanam also describe but obviously structure them in a different and more distinct way. What represented the *auction environment* in Bichler and Kalagnanams work is now called the *cardinality of negotiation*. *Agents characteristics* is a group of attributes that Bichler and Kalagnanam do not cover, related to roles, strategies and other attributes of software agents. Lomuscio et al. also go into the valuation of products or special events that can occur during a negotiation within the *environments and goods characteristics* and *event parameters* respectively.

At large the authors focus more on the computational aspects of negotiations, like agent characteristics, concrete events and quotes, than Bichler and Kalagnanam do and therefore present a more applicable classification scheme in SLA environments. The most affirmative aspect of this paper however is that the authors propose domains for some of the attributes given. This already delimits the negotiation space of electronic negotiations in a very concrete way and eases the definition of domains for the attributes in the presented data model. Hence this paper already sketches the attributes and domains needed to describe negotiations in general.

Note: The authors published this work twice. The first time, in 2001, this article was published within a book and the last section was used to categorise the subsequent papers with this taxonomy [39] and the second time, in 2003, the authors added an attribute for argumentation-based negotiation [40, 41] and applied their classification to some sample negotiation protocols in the last section instead of structuring other papers [29].

4.1.3 Parameterization of the Auction Design Space

Peter R. Wurman, Michael P. Wellman and William E. Walsh presented a set of auction parameters while developing an internet-based "'platform for price-based negotiation - the Michigan Internet AuctionBot"' [42]. This system was designed to serve as an auction server for humans as well as software agents and only focused on one negotiated issue: the price. Therefore unfortunately only one-dimensional auctions were supported.

The same authors extended this taxonomy to also cover multidimensional auctions in a

follow-up, much more comprehensive paper called "‘A Parameterization of the Auction Design Space"’ [43]. In this paper Wurman et al. do not only cover auction protocols and their parameters as a necessary byproduct of the development of an internet auction server as before, but focus especially on the different possible auction protocols and respective parameters. This approach allows for a much more comprehensive examination of the auction design space. The second improvement regarding this thesis is the already mentioned incorporation of multidimensional auctions. Since in a SLA environment the involved parties will mostly negotiate more than one desired aspect of a service, multidimensional auctions will be much more suitable than traditional auction protocols which allow to negotiate only one attribute (most commonly the price).

Wurman et al. present three different perspectives on an auction, based on the main tasks an auction protocol has to perform: *receive bids*, *clear* and *reveal intermediate information*. *Bid receiving* represents the process of agents "‘indicat[ing] their willingness to participate in exchanges"’ [43] according to certain conditions stated in the posted bids. This activity also incorporates the verification of bids, that is if or if not bids satisfy the set of auction rules. *Clearing* stands for the main purpose of any auction: "‘the determination of resource exchanges and corresponding payments between buyers and sellers"’ [43]. The third activity, *revealing intermediate information*, represents an optional auction activity, offering intermediate status information to the agents involved in the auction. This information typically contains to what conditions the auction would clear at the moment.

In addition to a very extensive formal model of bid semantics the authors present the set of auction parameters, able to cover a multitude of common auction protocols. According to the three dimensions presented earlier, Wurman et al. structure their auction parameters into three groups: *bidding rules*, *clearing policy* and *information revelation policy*.

According to the authors the "‘bidding rules determine under what conditions bids may be introduced, modified, or withdrawn, as a function of agent identity, current bid status, or even the entire auction history"’ [43]. This category extensively describes the bidding process by presenting auction attributes and respective domains. The *clearing policy* attributes focus on the matchmaking process of an auction. The authors focus on the concept of matching functions and describe this concept comprehensively in a seperate section. The *information revelation policy* describes the information about the current auction instance that can be accessed along with how to query it.

Finally the authors also classify some of the well known auction types with the means of their parameter set. This paper represents a fundamental taxonomy for auctions which is often cited by other authors in this research field. Although it again only focuses on auctions and not negotiations in general, like the first taxonomy presented, it defines some concepts for describing and structuring auctions that, because of their comprehensive nature, made a critical contribution to this thesis.

4.1.4 Software Framework for Automated Negotiation

The next taxonomy used in this thesis was proposed by Claudio Bartolini, Chris Preist and Nicholas R. Jennings together with their specification of a framework for automated negotiation in multi-agents systems (MAS). The paper is called "‘A Software Framework

for Automated Negotiation"' [44]. The authors focus on executable specifications for MAS. Thus they concentrate on message formats used and activities conducted by software agents which represents a very technical and practical approach to negotiation research. This contributes to this thesis as it assumes similar conditions as in automated SLA negotiations between software agents as to be supported by this framework. The messages and activity sequences identified are used as a foundation for the generic negotiation protocol presented later in this thesis.

Bartolini et al. set the context for their negotiation framework by describing the negotiation process and its architecture in an abstract way. They identify two distinct roles involved in any negotiation process: the *negotiation host* and the *negotiation participant*. The *negotiation host* represents the coordinator of the negotiation, it receives messages and forwards them to the appropriate agents. The common information of a negotiation, like current offers or involved agents is held in a logical centralized negotiation medium, called the *negotiation locale*. The *negotiation locale* represents a logical blackboard with read and write access rules for all involved agents to submit and read proposals.

The negotiation process itself is subdivided into three different phases: *admission*, *proposal submission* and *agreement formation phase*. During the *admission phase*, which is considered most comprehensively in this paper of all the taxonomies used, the agents request admission to the negotiation locale, and therefore the negotiation itself, and are informed of the respective rules. To start a negotiation, all parties involved must share a common negotiation template, specifying the different issues to be negotiated. Some of these different negotiated attributes will typically be subject of a constraint which enables to specify types and domains of negotiated attributes or to set some attributes of the negotiated product as fixed.

The negotiation process therefore stands for the process of moving "'from a negotiation template to an acceptable agreement"' [44]. This directly relates to WS-Agreement because it also uses agreement templates to derive a final agreement between the negotiating parties. During the *proposal submission* phase the agents exchange "'proposals representing the agreements currently acceptable to them"' [44]. This communication is always mediated by the *negotiation host*. All of the phases identified are conducted according to corresponding rules. Every negotiation locale is therefore complemented with a set of rules for each phase. An instantiation of a concrete negotiation protocol is thus acchieved by parameterization of a locale with a set of concrete rules. The types of rules applied are categorized to form the negotiation taxonomy presented.

Bartolini et al. identified six different rule categories: *Rules for admission of participants* govern the admission of agents to the negotiation locale. *Rules for proposal validity* enforce compliance of proposals to the negotiation templates. *Rules for protocol enforcement* are concerned with the agents' activities and their respective sequences permitted in a specific negotiation protocol. *Rules for updating status and informing participants* govern the information processing within a negotiation. Agreement formation is reached according to the *rules for agreement formation* and finally the termination of a negotiation is described with the *rules for lifecycle of negotiation*. The authors explicitly do not complement the different attributes of a negotiation protocol with domains. Instead they use Java Expert System Shell (Jess) assertions and rules to express the different aspects of a negotiation [45].

This allows for a very flexible definition of a negotiation protocol, because of the use

of an external rule language. The approach taken in this thesis tries to simplify the negotiation datastructure by applying domains and types for the identified attributes where possible. However some of the attributes identified in this thesis also allow the use of external languages in order to express assertions and constraints on other parts of the datastructure or on external systems as shown later. In order to enable exhaustive describtion of a multitude of negotiation protocols the use of such languages could not be avoided without loosing to much of the desired expressiveness.

Bartolini et al. continue the definition of their framework by describing the messages used in the different phases in terms of FIPA (Foundation for Intelligent Physical Agents) message formats [46] to specify the generic negotiation protocol. They conclude their paper by describing the implementation of their framework in terms of agents and their behavior, the definition of negotiation templates and proposals in the OWL-Lite language [47] and the classification of some well-known negotiation protocols with the framework.

The taxonomy presented by Bartolini et al. provides a perspective on automated negotiations, especially because of focusing on software agents and the message formats and activities involved, that helped structuring the negotiation data and defining the proposed generic negotiation protocol in this thesis.

Note: the paper summarized above represents the latest work on this framework done by Bartolini et al. Initially the concept was puplished as a HP research report [48] which included an extensive analysis of requirements and goals for the framework. Also different use cases were given to illustrate the scenarios the authors intended to support. In 2002 Bartolini et al. wrote another HP report focusing more on the abstract architecture of the negotiation process [49]. During 2002 this research group also published a consolidated version of their work at a workshop on agent-oriented computing [50]. The paper used as a basis for this summary finally extended the paper from 2002 by describing the negotiation process in more detail and incorporated the definition of negotiation templates and proposals as OWL-Lite descriptions which formerly have been formulated as Jess assertions as well [44].

4.1.5 Montreal Taxonomy for Electronic Negotiations

The last taxonomy used in this thesis was published by Michael Ströbel and Christof Weinhardt, and is called "‘The Montreal Taxonomy for Electronic Negotiations"’ [28]. This paper represents the most comprehensive categorization and description of electronic negotiations of all taxonomies. In contrast to Wurman et al. Ströbel and Weinhardt aim for a more generic understanding of negotiations than just claiming negotiation design is essentially equivalent to auction design [43]. They regard auctions as a particular class of negotiations; this point of view was adopted in this thesis. The authors also do not stress software agent characteristics such as agent strategy or computational complexity like Lomuscio et al. do because they consider "‘the degree of automation (...) as orthogonal to the classification criteria in [their] taxonomy"’ [28]. Thus the negotiation taxonomy presented by Ströbel and Weinhardt aims to describe and classify a multitude of negotiation protocols in a very generic, yet comprehensive way without stressing techonology-related issues. This paper therefore represents a major contribution to the work done in this thesis by providing a common terminology describing electronic negotiations and thus support the structured design of specific electronic negotiations.

After an extensive section concerning relevant definitions in the context of electronic negotiations the authors generally describe the negotiation process. They identify several phases of interaction between software agents in an electronic market in order to reach a deal: the *knowledge*, the *intention*, the *agreement* and the *settlement phase*. The *knowledge phase* comprises gathering information about products, market participants etc. During the *intention phase* supply and demand are generated by placing offers to sell or buy. Terms and conditions for a transaction, as well as signing the contract will be conducted during the *agreement phase* before the contract will be executed in the *settlement phase* together with all corresponding termination tasks, like payment or support.

Refering to the definitions given, the authors describe the application domain of this taxonomy as negotiations that are "'restricted by at least one rule that affects the decision-making or communication process"' [28]. This focus does not only cover fully automated negotiations but also negotiations, that are conducted by humans using negotiation support systems.

The taxonomy itself presents a set of criteria, each representing a "'distictive electronic negotiation scenario property"' [28]. The authors distinguish between *exogenous* and *endogenous* as well as *implicit* and *explicit criteria*. The combination of these two dimensions results in four categories. "'Exogenous criteria are (...) determined by the business context (...) and cannot be influenced by the [negotiation] designer"' [28], while negotiation parameters determined during the design of an electronic negotiation instance are called *endogenous criteria*. *Explicit criteria* represent criteria for which a formal representation can be found. On the other hand "'[i]mplicit criteria assess the consequences of explicit criteria"' [28].

The following figure depicts the abstract structure of the Montreal Taxonomy, highlighting the negotiation attributes that contributed to this thesis:

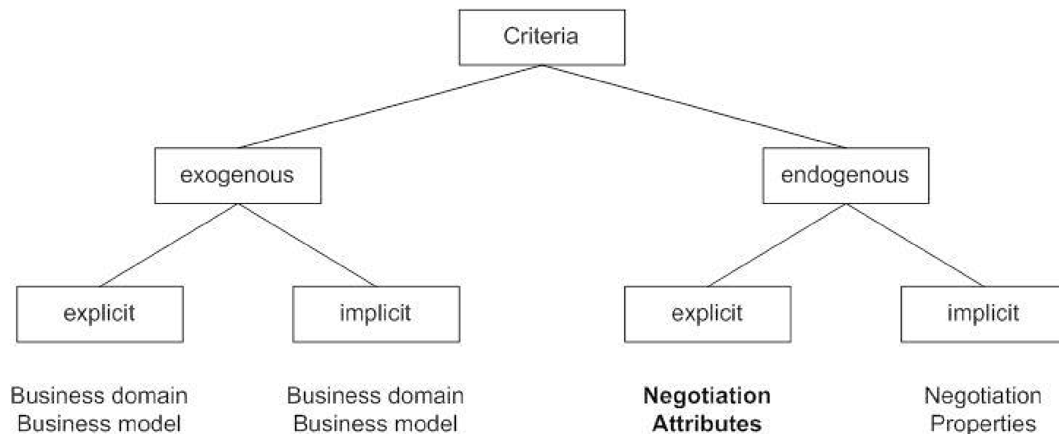

Figure 3: Montreal Taxonomy Structure

The authors propose two abstract categories for exogenous criteria (for both implicit and explicit ones): *business domain* and *business model*. The *business domain* stands for the relation between agents involved in this domain and the *business model* defines the role of a business within the described domain. These concepts therefore exhaustively describe the business environment of a negotiation. *Explicit exogenous criteria* thus describe the *business domain* in terms of transactions conducted or market configuration etc. and the *business model* in terms of the concrete role within the scenario or the value creation in the business. *Implicit exogenous criteria* include market culture (*business domain*) or strategies and goals of the business (*business model*).

Ströbel and Weinhardt focus on *explicit endogenous criteria*, in other words on criteria that have a formal representation and that a negotiation designer can use to define the one negotiation instance needed. All attributes are orthogonal in a sense, that they can be combined arbitrarily to classify a negotiation and are complemented with individual value domains (mostly only the extreme expressions, bordering the possible value ranges of the respective attribute). The *explicit endogenous criteria* are further subcategorized to enable a differentiated description of negotiations.

The *roles* category addresses admission and agent related aspects. *Process - overall rules* describes general process data, like the number of rounds or stages. The negotiated issues are covered in the *Process - offer specification* category in terms of attributes, values, relaxation etc., *Process - offer submission* attributes address permitted activities within a negotiation process. Restrictions between subsequent offers are handled in the category *Process - offer analysis* and the matching process in *Process - offer matching. Process - offer allocation* and *Process - offer acceptance* deal with the way an agreement is formed after negotiation. Information processing rules are covered in an *Information* category and finally the *Strategy* category is concerned with agent-related aspects of the general process, like rewards or punishments generated or negotiation fees. The paper is concluded by a set of *implicit endogenous criteria* like pareto-efficiency, fairness or stability. These criteria originate in game theory and mechanism design research and represent negotiation properties as presented in the last section.

This thesis mainly relies on this taxonomy, because it is the most comprehensive work of all the presented taxonomies and also provides possible value domains for the presented parameters which eases the definition of the data model. Also the categories chosen by the authors denote an appropriate and intuitive way to specify the different aspects of a negotiation. The attributes presented in this thesis are therefore structured in a similar way to these categories.

4.1.6 Summary and Comparison

The presented taxonomies provide a theoretical foundation for deriving the data structure for negotiation protocols in the next subsections. Even though some of them concentrate solely on auctions [43, 30], or stress software agent aspects of electronic negotiations more than is adequate in SLA scenarios [44] they all provide useful concepts for negotiation design in WS-Agreement environments. Every taxonomy represents a different point of view on the same topic: the description, classification and design of negotiations (or a particular group of negotiation protocols, e. g. auctions).

Bichler and Kalagnanam, in addition to the negotiation attributes, provide a description of the negotiation process and give an overview on desirable auction properties which can be viewed as a metric for the efficiency of an auction. Bartolini et al. propose a comprehensive approach of designing negotiation systems for FIPA compliant software agents. This approach, in addition to the presented taxonomy, contributed to this thesis in sketching the necessary messages and activities for software agents to perform negotiations which will be incorporated in this thesis in the next sections. Lomuscio et al. propose a quite extensive, yet informal, taxonomy for negotiations, which contributed several concepts regarding the derivation and structuring of attributes for this thesis. Wurman et al. published one of the mostly referenced papers in this research discipline, however

they focus only on auctions. Nevertheless most of the parameters they identified and formalised are as applicable for negotiations in general as they are for auctions. "‘The Montreal Taxonomy for Electronic Negotiations"’ [28], for it describes negotiations in the most comprehensive and generic way, but also with enough detail concerning the possible values for the identified attributes, extensively contributed to this thesis in terms of negotiation attributes and their structuring.

The next subsection will present the attributes and attribute categories defined for the WS-Agreement negotiation framework. References to the different taxonomies will be shown in terms of adopted or omitted attributes. After deriving the attributes needed a data model will be presented described as an Entity Relationship Model (ERM) [38] and finally an XML data structure will be derived from this data model in order to enable application in a Web Services environment.

4.2 Negotiation Attributes

As already hinted the attributes presented in this subsection allow the description of a variety of negotiation protocol. These protocols are described in a way that allows application in fully automated environments. That is the attributes comprehensively specify negotiations and provide any information needed by a software agent to participate in the corresponding negotiation instance. One should distinguish between a *negotiation type* and an actual *negotiation instance* of that type. Using the presented attributes a *negotiation type* can be specified by an agent in order to promote the negotiation protocol it can support. An *instance* is created before starting an actual negotiation. The distinction between *negotiation type* and *instance* will be explained in more detail later.

After defining some general design goals and assumptions regarding the negotiation process the different attributes will be presented in the following, categorized into distinct attribute groups.

4.2.1 General Protocol Data

The intent of this thesis is to derive a datastructure allowing to describe a multitude of particular negotiation protocols. In negotiation research negotiation protocols have been presented acually consisting of two or more individual protocols. For example one could define a negotiation starting off as an auction until only three potential business partners are left. After that the agent wanting to sell the product starts bilateral negotiations with each one of the remaining bidders. Such a negotiation does not exhibit one particular negotiation protocol but consists of a sequence of different protocols. The datastructure presented in this thesis explicitly does not cover such negotiations. Hence negotiations consisting of different protocols conducted consecutively can not be described with the means of this thesis. In order enable such scenarios the involved participants have to terminate the first negotiation, the auction, which can be specified with the data structure presented later, and start a new one. This new negotiation can again be described with the data structure provided here but has to be described seperately. The matter of transfer from one protocol flow to the next one along with such issues as information forwarding between the different protocol flows is therefore not considered here.

Attributes allowing to explicitly define the change of negotiation protocol within one negotiation can be found in the work of Lomuscio et al. [43] and Ströbel and Weinhardt [28], however.

Regarding the configuration of the negotiation processes to be described intermediaries like negotiation hosts or trusted third parties (TTP) are explicitly supported in the approach taken in this thesis. The involved parties in a negotiation are modelled using a role concept. This allows for specifying which roles are engaged in a particular negotiation and which agents adopt which role. By allowing more than two roles and more than one agent per role one can define arbitrary configurations of involved agents in a particular negotiation process. This allows for enabling distinct roles like security-related third parties etc. accordingly, one of which, the *Negotiation Coordinator*, is assumed to be mandatory for this framework.

Since SLAs inherently consist of different attributes multi-attribute and combinatorial negotiaions are crucially important in WS-Agreement scenarios. These types of negotiation allow for offers on more than just one attribute, which would most commonly be the price. Without these concepts useful SLA negotiations could not be conducted.

Multi-unit auctions, concerning the negotiation over sets of units of the same product, conducted in a traditional way will not be considered though. Each service can be unambiguously identified and these services are not homogenous items. Besides theses services are no discrete goods like books or CDs, for which multi-unit auctions normally apply. Since all services differ from each other and are identified with a unique ID each classical multi-unit auctions will not be appropriate here; each service will be referenced individually and therefore be treated in an own negotiation or as another item in a combinatorial auction.

Finally, concepts related too much to human negotiations will be omitted in this approach, because they are not applicable for automated negotiations among software agents. Examples would be withdrawal from reached agreements or fees for participating in a negotiation. These concepts can not be unambiguously interpreted and processed by software agents. Concepts like fees will also produce judicial consequences that cannot be handled without human interaction.

4.2.2 Attribute Categories

The negotiation parameters presented in this thesis will be structured according to the following categories:

- *Process Information*
- *Negotiation Context*
- *Negotiated Issues*
- *Offer Submission*
- *Offer Allocation*
- *Information Processing*

The *Process Information* category defines general information about the negotiation protocol described, like the start and the rounds of a negotiation protocol or whether the negotiation is rewarding protocol compliance or punishing protocol violation.

Negotiation Context attributes specify the agent-related aspects of a negotiation, like involved roles or admission rules. This category therefore defines the configuration of the negotiating parties.

The parameters making up the *Negotiated Issues* category specify the attributes of the negotiated service. Different services or attributes of one service can be defined to be subject of the negotiation with these attributes.

Offer Submission parameters govern the bidding process. Rules concerning the submission of bids or the relation between bids are specified with these parameters. This is used in English Auctions for example to restrict bids to be superior to the last valid bid.

The accessable information concerning the current status of the negotiation, past offers from all participating agents and the permission to access this data is handled with *Information Processing* attributes. This category therefore defines whether agents can react on other agents' actions by defining what information can be accessed by which agent. Without any access to negotiation information the participants have no knowledge about each other and can thus not react to each other's offers.

Finally *Offer Allocation* parameters govern the matching process of a negotiation, more precisely the agreement formation in SLA scenarios.

This classification does not follow one of the presented taxonomies directly, but defines a structure, based on the categories presented there, that divides up the attributes in an intuitive and unambiguous way, appropriate for SLA environments.

Bichler and Kalagnanam analogously define parameters concerning the negotiated issues, the involved agents, the bid submission and offer allocation of a negotiation process [30]. The *Payment Rules* category presented there states the focus on e-commerce auction markets. In this thesis rules concerning the payment are incorporated into the allocation parameters. Furthermore the information processing aspects of a negotiation are incorporated in the category governing the bid submission process in Bichler and Kalagnanam's work, called *Auction Protocol*. The information accessable by agents is a crucial aspect of a negotiation because it specifies what information which agent can access, and therefore to which degree agents are aware of the actions taken by their fellow negotiators. This defines the possibilities of reacting to other bids and therefore, in this thesis, this concept was modelled as a seperate category to emphasize its importance.

Lomuscio et al. present parameter categories analogously to define the negotiation context, the information processing and the allocation rules [29]. However they also introduce an additional category concerning *Agent Characteristics*. This category relates to the focus on software agent frameworks which is outside the scope of this thesis. Some of the presented attributes, however, are also applicable in WS-Agreement environments. These parameters will therefore be incorporated into this thesis within the appropriate attribute category later. The authors also present a category describing *Event Parameters*, in other words events occuring in a negotiation process. This group of parameters contains events regarding information distribution, bid submission and clearing schedules. This thesis

follows the approach of assigning the events to the appropriate parameter categories, they semantically belong to. Information distribution events therefore are assigned to the *Information Processing* category etc. This way the events are treated in a semantically consistent way because they are listed within the category they belong. Finally Lomuscio et al. present a category concerning *Environments and Goods Characteristics*. These parameters describe the dynamics of negotiation environments and nature or valuation of goods. Since these considerations denote a mainly game theoretical and economical point of view on negotiations they are not directly incorporated into this work except for some selected aspects applicable in WS-Agreement environments.

The three categories given in the work of Wurman et al. however, are not detailed enough to be directly applied in this comprehensive framework. The authors only distinguish Bidding Rules, Clearing Policy and Information Revelation Policy [43]. Even though these are the main tasks an auction or a negotiation in general has to perform it lacks the description of the context of the negotiation in terms of the involved agents and negotiated issues. The categories are presented in this thesis have been chosen to provide a broader point of view on negotiation processes.

The paper published by Bartolini et al. proposed a different structure for the negotiation parameters [44]. The *Rules for Proposal Validity* and the *Rules of Protocol Enforcement* presented there were incorporated in the *Offer submission* category of this thesis. Bartolini et al. also seperately presented a category for agent admission and termination of the negotiation which are incorporated into the *Negotiation Context* and *Process Information* categories here respectively. Analogously to this thesis Bartolini also defines categories concerning information processing and agreement formation.

At last the taxonomy presented by Ströbel and Weinhardt proposes a similar structure to the one applied in this thesis [28]. The authors also provide categories for information processing, offer matching, overall process rules, negotiated issues and involved roles. However they split up the *Offer Submission* category into *Offer submission* and *Offer Analysis* and the *Offer Allocation* category into *Offer Allocation* and *Offer Acceptance*. These concepts have been combined in this thesis for increased clarity. The additional attributes concerning rating, negotiation fees and penalties and rewards (assigned to the category *Strategy*) have been incorporated in the *Process Information* category, if applicable for WS-Agreement scenarios.

The attributes identified for WS-Agreement scenarios will be presented in the next subsections, structured according to the different attribute categories.

4.2.3 Process Information Category

- Attribute: **Rounds**

  Domain/Values: positive Integer, excluding zero

  Purpose: This attribute allows for definition of negotiation protocols proceeding over multiple rounds. Each round is conducted using the same protocol, but possibly with more available information. One example would be a sealed-bid auction with one extra round for the bidders to alter their initial bids. In the first round every agent places a bid in a sealed fashion. After completion of this round the results of the first round are promoted to all involved parties. During the second round every agent

can update its offer according to the information revealed about the other bids to improve his chances of winning the auction.

The **Rounds** parameter only defines the number of rounds of a negotiation not how each round is started or terminated. Each round is terminated, when one of the criteria specified in the **Termination** attribute applies. The start of the first round of a negotiation is defined with the **Start of Negotiation** parameter, whereas the start of the following rounds is triggered by the invocation of a *newRound()*-method as described later when the different methods and roles used for the negotiation protocol are introduced. These method calls are conducted by the *Negotiation Coordinator* which administrates the termination and starting of negotiation rounds.

- Attribute: **Start of Negotiation**

  Domain/Values: ANY CONSTRAINT EXPRESSION

  Purpose: This attribute is used to specify the starting point of a given negotiation. To enable arbitrary assertions on (possibly) external values this attribute is not restricted to one particular type. Such external values include for example certain events or actions of particular third party agents.

  Not restricting this attribute's type allows for the incorporation of external constraint languages. The content of this element can therefore be defined in terms of any constraint language applicable to express the desired restrictions. Permitting only one particular constraint language would not only narrow the application domain but is also outside of the scope of this thesis. An appropriate existing constraint language can be chosen and used to express the restriction defined in this attribute.

  To ease the definition of starting constraints a *startNegotiation()*-method is introduced later. This method can be referenced within the start element and used to express corresponding constraints, for example to describe scenarios in which the coordinating agent starts a negotiation by just notifying all participants. In this case a negotiation designer would just state that the invocation of this method starts the corresponding negotaition and no further constraints will be needed.

- Attribute: **Termination**

  Domain/Values: ANY CONSTRAINT EXPRESSION

  Purpose: Analogous to the **Start of Negotiation** attribute this parameter defines the end of a particular negotiation. However instead of providing one *terminateNegotiation()*-method similar to the *startNegotiation()*-method two different methods terminating a particular negotiation are introduced later, each stating the result of the negotiation additionally to just terminating it: *acceptAgreement()* and *rejectAgreement()*. With these two methods a negotiation can be terminated semantically correct by also providing the negotiation's result to the participating agents. Again these methods will be described in more detail later.

- Attribute: **Arbitration**

  Domain/Values: {none, punishing, rewarding, all}

  Purpose: The **Arbitration** parameter specifies whether agents are rewarded for compliance to or punished for violation of certain rules or both. This can involve rewards for acting according to the specified negotiation protocol or punishments for violation of it for example.

This attribute allows for integration of external reputation management systems [51]. Such systems generate reputation values for individual software agents based on their past behavior. Bad behavior will result in bad reputation values. These values can be queried by other agents in a MAS in order to evaluate the trustworthiness of a particular agent. Depending on some internal thresholds for reputation values of transaction partners agents can therefore exclude or at least favour other agents from some transactions because of their reputation value.

This concept is very suitable for SLA scenarios. Agents should be able to assess the reliability of a particular service offered by some other agent. With the help of reputation management systems every agent, and its offered services, can be complemented with some reputation value representing the reliability of that agent and respective the provided service. Other agents thus can find trustworthy agents evaluating their reputation values.

It is not only reasonable to generate these values based on the agents' behavior in providing and consuming services, but also based on its behavior in SLA negotiations. That is exactly what this parameter enables doing. Agents can be punished or rewarded according to their negotiation behavior. For this purpose some constraint language will be used to define the qualifying conditions for some punishment or reward and the results respectively, expressed in terms of the external reputation system used.

If such an external system is used in the negotiation it has to be referenced within the **Arbitration** attribute. This is necessary because it clearly states which external system is used to process the reputation values. This becomes crucial if there is more than one system available. The presented attribute also allows for definition of rewards and punishments without external systems. One example of a punishment would be an exclusion of an agent from the negotiation. If such free-hand definitions of rewards or punishments are stated the desired qualifying conditions and consequences have to be clearly defined, e. g. using an external constraint language again.

4.2.4 Negotiation Context Category

- Attribute: **Roles**

  Domain/Values: {service provider, service consumer, negotiation coordinator, information service, ANY ROLE}

  Purpose: This attribute defines the roles involved in a negotitation. Appart from the four compulsory roles service provider, service consumer, negotiation coordinator and information service any other role may be defined. This acts as a wildcard for third party roles in a negotiation. Examples for such roles could be TTPs verifying digital signatures or certificates for security purposes or the already mentioned reputation service.

  The default roles will be described in more detail later when the exchange and negotiation protocols are presented.

- Attribute: **Agents**

  Domain/Values: EPRs (for example according to the WS-Addressing specification [52])

Purpose: The **Agents** parameter specifies the agents involved in a negotiation. For every role defined in the **Roles** attribute the **Agents** attribute defines the agents adopting it. Each role has to have one agent assigned to it as a minimum. This parameter combined with the **Roles** attribute allows for arbitrary negotiation configurations in terms of involved parties and agents.

- Attribute **Admission Restriction**

  Domain/Values: {open, ANY CONSTRAINT EXPRESSION}

  Purpose: The **Admission Restriction** parameter can be used as a wildcard for any restriction on the admission process of a particular agent. This can be used to demand certain credentials of agents wishing to join a negotiation, like reputation values or security certificates. The requested credentials to be given or actions to be taken by agents on admission can be expressed in any constraint language suitable. Because of not restricting the type of this attribute also any restriction on external credentials can be formulated. This concept allows, for example, to restrict negotiations to agents with a reputation value of some particular reputation system above a certain threshold.

4.2.5 Negotiated Issues Category

- Attribute: **Items**

  Domain/Values: WS-Agreement with *context elements*, *service description*, *service property* and *guarantee terms*

  Purpose: This attribute allows for definition of negotiations an agreement concerning multiple services. Since SLA guarantees are often expressed as assertion over several services this parameter represents a crucial concept for WS-Agreement scenarios.

  Services in WS-Agreement environments are described as already explained in chapter 2.3. Generally WS-Agreement negotiations always concern only one particular item, the agreement. However this agreement consists of several component items or attributes to be negotiated. These elements of a WS-Agreement document representing the negotiated services are defined to be *context elements* (e. g. expiration date of the agreement), *service description terms*, *service property terms* and *guarantee terms*. A more detailed discussion about what elements of a WS-Agreement can be subject of a negotiation is given in chapter 5.3 5.3.

  Negotiating multiple units, as described earlier, is substituted by negotiating each unit, e. g. service, seperately since each service instance can be individually referenced and thus negotiated as already depicted above.

- Attribute: **Domain**

  Domain/Values: {String, Integer, ANY SPECIFICATION OF A DOMAIN}

  Purpose: Each attribute defined within the **Attributes** parameter of a negotiation is assigned a domain restricting its possible values. This restriction defines the type of the corresponding attribute.

  A domain can be ordered or not ordered. Ordered domains expose some order relation that can be expressed by the means of $<$, $\leq$, $>$ or $\geq$ as opposed to not ordered domains that do not exhibit such a relation. Integer domains would therefore

be ordered domains, whereas string domains per se do not have an order relation assigned to them.

- Attribute: **Values**

  Domain/Values: {single, multiple}

  Purpose: This parameter specifies whether only one distinct value or multiple values can be set for a specific attribute. Possible values for an Integer attribute, that allows only single values could be "'3"' or "'5869"'. An Integer attribute with possible multiple values would also allow expressions like "'$\leq$ 56"' or "'$\neq$ 44344"'.

  Intuitively not all possible relational operators used to express multiple values are applicable in every every type of domain. Not ordered domains only allow $\neq$ or a simple enumeration of possible values to express multiple possible values. In ordered domains on the other hand additionally $<$, $\leq$, $>$ and $\geq$ are permitted.

  The relationship between the **Domain** and the **Values** parameter is detailed in the following figure:

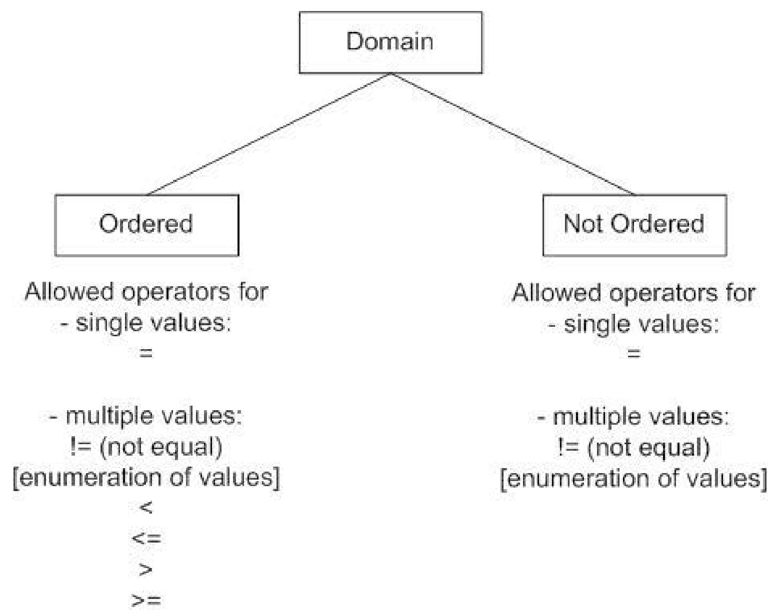

Figure 4: Relationship between *Domain* and *Values* parameter

- Attribute: **Relaxation**

  Domain/Values: Boolean

  Purpose: The Boolean attribute **Relaxation** defines whether an element or a set of elements specified in the **Negotiated Issues** parameter is fixed. Assigned to a class of negotiated issues, e. g. the element enclosing all the negotiated *guarantee terms*, it can declare if or if not this set of negotiatbable issues can be extended. Thus the **Relaxation** attribute can define whether additional *service description terms* can be introduced in a new offer, for example. This would be useful if one of the negotiators can accept the offered agreement, but only if one additional service or service attribute is provided.

  In this framework relaxation concepts are only applied to classes of issues, such as *guarantee terms*, to indicate whether this class can be extended. Individual service attributes that are fixed and therefore not subject to the negotiation will simply

not be referenced within the **Negotiated Issues** category of the negotiation type document.

4.2.6 Offer Submission Category

- Attribute: **Sides/Direction**

  Domain/Values: ALL ROLES SPECIFIED IN THE ROLES PARAMETER

  Purpose: This attribute defines which roles/sides of a negotiation are allowed to post offers. This allows the definition of negotiation protocols which allow offers to be submitted by only one side, like auctions, or by all sides, like one-on-one bargaining.

- Attribute: **Submission**

  Domain/Values: ANY CONSTRAINT EXPRESSION

  Purpose: Any restriction applied on the bidding process can be expressed with this parameter. To allow as comprehensive a set of restrictions as possible the value of this attribute will again be denoted in an external constraint language. This way negotiation designers can specify any restriction on the submission of offers possible to express with the language chosen.

  These restrictions could include, that every agent is only allowed to post one offer or when offers can be posted. Every offer is syntactically a WS-Agreement as defined in the WS-Agreement specification draft [4] This allows for a two-step validation of offers: first they are checked syntactically and then according to the restrictions stated in the **Submission** attribute of the negotiaton. An offer is only accepted when it is syntactically correct and was submitted compliant to the stated offer submission rules.

- Attribute: **Offer Progress**

  Domain/Values: {ascending, descending}×{none, ANY VALUE}

  Purpose: The **Offer Progress** attribute actually represents a particular submission restriction, but is modelled individually because it specifies a very common restriction used in negotiations. It governs the direction of the bidding process concerning a certain attribute. Negotiation designers can define, e. g. that the values for some negotiated attribute set in a new offer have to be higher than in the previous offer, much like in auctions. Also a certain minimum decrement or increment values can be specified.

  Hence the **Offer Progress** parameter combines two dimensions: first the direction of the bidding process and second the minimum increment or decrement values. Incorporating this the possible values are of a tuple form, for which the domains are given above. A negotiation designer can thus define the bidding direction and optionally augment it with some minimum increment or decrement value within the domain of the respective attribute.

  Clearly this attribute is not specified when applied to an attribute of a not ordered type.

- Attribute: **Threshold**

  Domain/Values: {none, lower bound}×{none, upper bound}

Purpose: Analogous to the **Offer Progress** attribute this parameter represents a very common offer submission restriction and is therefore modelled individually.

The Threshold parameter allows for definition of lower and upper bounds for the values some attribute is permitted to exhibit within the posted offers. Again in not ordered domains none of these are defined. In ordered domais, however, one can optionally specify a lower or an upper bound or both to restrict the possible values of this attribute during the negotiation.

4.2.7 Offer Allocation Category

- Attribute: **Offer Matching**

  Domain/Values: {forwarded, ANY CONSTRAINT EXPRESSION}

  Purpose: The **Offer Matching** attribute defines how offers are transformed into final agreements.

  Normally an agreement is formed when one of the involved parties accepts an agreement offered by some other participant (forwarded). However negotiation designers could wish to define according to which rules the agent being offered an agreement decides whether to accept it and provide this information to all negotiation participants. In that case this parameter exposes a constraint expression formalized in an external constraint language describing this offer matching rule.

  One could for example specify that in an auction-type negotiation after terminating the bid offering the highest average value of attributes A, B and C would win. This way the involved agents can assess their own offers and anticipate the winning bids before being notified by the accepting agent afterwards.

4.2.8 Information Processing Category

- Attribute: **Negotiation Transparency**

  Domain/Values: {public, protected, none}

  Purpose: The **Negotiation Transparency** attribute defines which agents can access the past offers of the negotiation. Public **Negotiation Transparency** indicates, that every agent has access to the past offers. No restrictions are defined concerning the demanding agents. Protected **Negotiation Transparency** restricts the negotiation information to agents involved in this particular negotiation instance. If none is defined no information about past offers in this negotiation is accessible at all.

- Attribute: **Content**

  Domain/Values: ANY CONSTRAINT EXPRESSION

  Purpose: This parameter defines the content of the queried past offers or the negotiation status. Thus, each of these two attributes is augmented with a **Content** parameter. With this attribute negotiation designers can not only specify whether each queried offer is returned entirely or restricted to some of its elements, but also which offers are returned at all. This is especially useful when defining the negotiation status' content: here this parameter can be used to define exactly which offers

make up the negotiation status which depends on the applied negotiation protocol. In an auction with defined offer matching rules there is always only one currently winning offer which denotes the status. If the offer matching rules are not given the status would propably consist of all current offers of all participants. ´

In order to enable flexible expressions concerning the negotiation and status content any external constraint language can be applied for this parameter.

- Attribute: **Status Transparency**

  Domain/Values: {public, protected, none}

  Purpose: The **Status Transparency** parameter defines which agents can access the current status of the negotiation. Public **Status Transparency** indicates, that every agent has access to the current status. No restrictions are defined concerning the demanding agents. Protected **Status Transparency** restricts the status information to agents involved in this particular negotiation instance. When none is defined as a value for **Status Transparency** no agent at all can access the current negotiation status. This allows for the definition of sealed-bid negotiations.

  The negotiation status is defined as the set of current offers of all agents involved in the negotiation. This concept implicitly assumes that every agent has only one valid offer at all times. Each newly posted offer therefore logically replaces the previous one.

The following attributes originating in the presented negotiation taxonomies have been omitted in this thesis. The attributes will be presented shortly by describing their purpose and the reason of not including them into this approach respectively. Also the taxonomy(ies) originally proposing the particular attribute is referenced.

4.2.9 Process Information Category

- Attribute: Concurrency

  Purpose: This attribute specifies whether an agent can get involved in more than one negotiation at a time.

  Reason for omitting: This work should present a data structure concerning negotiation protocols. From a protocol point of view the agents' activities concerning other negotitaion instances can not be checked or assessed. This thesis will derive the means for conducting a particular negotiation protocol. If an agent uses this for different negotiations or not does not make any difference for the negotiation protocol framework.

  Hence, this attribute, although important to characterize negotiation situations in general, is not considered in this approach. It is presented in the paper of Ströbel and Weinhardt, though [28].

- Attribute: Fees

  Purpose: Some negotiations demand certain fees from the participants. These fees can be due in different situations, e. g. when joining a negotiation or when posting an offer.

Reason for omitting: Since SLA negotiations will be conducted by software agents, and not human beings, the application of fees would produce severe judicial consequences. Software agents cannot take responsibility for their actions in a judicial accepted way, as humans would.

Another reason for not considering this attribute here is the sheer volume of different negotiations possibly executed in SLA scenarios which would produce to much administration effort for the application of fees. Also fees would contradict the idea of open negotiations where every agent can join and derive an agreement. If one wishes to exclude some agents from a negotiation this will be done by other means as stated in the **Admission Restriction** parameter. When applying fees possibly feasible agreements will not be negotiated just because the agents, and respective owners, cannot or do not wish to effort the negotiation fees.

Fees are defined in the taxonomies of Ströbel and Weinhardt [28] and Wurman et al. [43]

- Attribute: Activity Rules

  Purpose: Activity Rules should motivate involved agents to continue bidding or placing offers.

  Reason for omitting: Since software agents rationally place offers, until they reach an agreement they can accept or until they leave the negotiation without having created an agreement, there is no use for rules motivating agents to continue bidding.

  The concept of activity rules is considered in [30] and [43].

- Attribute: Withdrawal

  Purpose: This parameter concerns the possibility to withdraw particular offers.

  Reason for omitting: This thesis is based on the assumption that software agents act rationally. Thus an agent could only want to withdraw its offer when an external event has occured within the negotiation, like the posting of another agent's offer. If nothing would have happened the agent's decision to place the offer would still be valid, because rationality is assumed. If another agent has placed an offer better than the first agent's it would not have to withdraw because it would not win the negotiation any more. If the new offer is worse than the first again the hypothetical clear in this situation wouldn't change and the first agent would win. Therefore withdrawal of offers will not considered in this thesis.

  Withdrawal concepts or attributes can be found in [43], [29], [44] and [28].

- Attribute: Expiration

  Purpose: Expiration definition is used for defining offers to only be valid for a certain amount of time.

  Reason for omitting: The increased complexity, resulting from allowing expiring offers, would exceed the enhanced expressiveness reached with it. The datastructure presented in this thesis should encourage agents to place offers they honestly want to submit and not ones that will be retracted later. That is why this attribute is not considered here.

  Wurman et al.[43] and Bartolini et al.[44] present expiration concepts in their taxonomies.

Also other agent-related parameters, mainly presented in [29], like the agents' rationality, their bidding strategy or knowledge about other agents will not be considered in this approach. This thesis represents the protocol-related point of view on negotiatons and is not concerened with agent-internal aspects. This allows for a very generic approach allowing different kinds of agents to use this framework.

4.2.10 Negotiation Context Category

- Attribute: Coalition Formation

  Purpose: Coalition Formation defines whether or not agents are allowed to form coalitions or groups to achieve better negotiation results.

  Reasons for omitting: As with the Concurrency parameter a protocol-based approach as taken in this thesis cannot check this aspect. Since this work focuses only on the protocol part of negotiations again agent-related issues like coalition formation will not be considered, though it would be suitable for the general characterization of negotiation scenarios.

  Parameters concerning Coalition Formation can be found in [28] and [29].

- Attribute: Privacy

  Purpose: The Privacy attribute defines whether agents can be identified given the information provided in their offers or whether offers can be assigned to the agents placing them. This allows for enabling anonymous offer submission.

  Reasons for omitting: For distributed negotiations in Web Services scenarios to work, each offer must enable the identification of its originator. An agent has to commit to the offer placed. Anonymous offers, if winning, cannot be semantically processed, and therefore this parameter is not considered in this thesis.

  The Privacy concept is considered in [28].

4.2.11 Negotiated Issues Category

- Attribute: Valuation

  Purpose: This parameter specifies whether the negotiated entities have public or private valuations, that is whether they have a common value, known to everyone, or every agent prices them differently.

  Reason for omitting: Since this thesis derives a negotiation protocol framework, just concerned with the negotiation process in a narrow sense, concepts regarding the agents' valuation of the negotiated agreements are outside of the scope of this work.

  Nevertheless valuation concepts can be found in [29].

- Attribute: Object

  Purpose: The Object attribute specifies whether different service configurations can be specified in an offer. In case of single object an offer can only contain one single configuration. When multiple objects are allowed several different service configurations can be expressed in one offer. When multiple configurations can

be expressed within one offer, each augmented with some value representing its importance or preference, bundled objects are permitted.

Reason for omitting: WS-Agreement documents inherently allow to specify different agreement levels and assign some business values to them. Hence bundled objects can be assumed to be present in every WS-Agreement negotiation.

4.2.12 Offer Submission Category

- Attribute: Position

  Purpose: With the Position parameter one can specify whether agents can adopt more than one role within one particular negotiation, for example provider and consumer of a service.

  Reason for omitting: This concept can only be meaningfully applied in n:m negotaitions like CDAs [53], where the involved agents sell and buy within one negotiation process. A famous example for CDAs is the stock market, where individuals can buy and sell stocks at the same time. This negotiation concept is explicitly not covered in this thesis. This work focuses on the negotiation between agents acting as service providers and consumers, it will not provide some kind of market architecture enabling centralized marketplaces.

  Given that such negotiations are not supported by the framework proposed in this thesis, different positions to be taken by one agent within one negotiation process are not considered.

  The Position concept is mentioned in [28].

- Attribute: Bid Control

  Purpose: The Bid Control attribute defines whether the negotiating agents post offers actively or some *Negotiation Coordinator* service polls all participants for offers to be submitted.

  Reason for omitting: If the *Negotiation Coordinator* asked for offers to be posted from each agent the network traffic would dramatically increase due to the possibly high number of requests for offers. This effect is especially crucial when a large number of agents still participate in a particular negotiation, but the offers have gone too high for them to still posting bids. Normally those agents would stay participants of the negotiation but would post no more bids until this negotiation process is ended. With a polling concept each participant would be asked for a new offer everytime the polling mechanism is triggered even though none of them will respond to it any more. Allowing the involved agents to actively post their offers whenever they want also distinctly increases the flexibility of this approach.

  In order to reduce traffic and corresponding computational complexity, while increasing flexibility, polling for offers is not considered.

4.2.13 Offer Allocation Category

- Attribute: Provision

Purpose: With the Provision parameter one can specify negotiations where the conditions stated in the final agreement do not have to be the same as stated in the winning offer(s). The service provider can for example set a completely different price from the one agreed upon during the negotiation process.

Reason for omitting: The only useful way to process offers from different agents is to calculate agreements on the basis of these offers. If an agreement could be created without incorporating the values agreed upon during the negotiation an agent will not be incented to reveal its true valuations in its offers because it could offer insanely high values for some attribute (e. g. the paid price) when it would somehow know the winner only pays some particular amount. In order to ensure meaningful offers from the involved agents only offer-dependent agreements are formed.

The provision concept is incorporated in the work of Stroebel and Weinhardt [28].

- Attribute: Configuration

  Purpose: This attribute allows for definition of rules concerning how value ranges agreed upon during the negotiation can, if necessary, be reduced to one distinct value. For example the negotiating parties can agree on some range of integer values during their negotiation, e. g. "'≤ 59"' which could be transformed into "'exactly 59"' during agreement formation.

  Reason for omitting: Since in SLA negotiations distinct values as well as value ranges are feasible this parameter is of no use and will not be incorporated into this thesis.

  Again, originally this attribute was proposed by Ströbel and Weinhardt[28].

- Attribute: Evaluation

  Purpose: The way offers are presented before agreement formation is defined within the evaluation attribute. One can specify whether the received offers are presented in a ranked way or just listed.

  Reason for omitting: This concept can only be applied meaningfully for partly automated negotiations, because only human negotiators demand, possibly ranked, listings of valid offers. In this framework an agreement is always formed by an agent, with or without informing the others how it is created (**Offer Matching** parameter), so it is of no use to present the posted offers in some way. This concept focuses too much on negotiation support systems for human interaction which makes it not applicable in this approach.

  However it again helps in characterizing negotiations in general which is why it is proposed in [28].

- Attribute: Distribution

  Purpose: This attribute defines whether a uniform price is set for all units negotiated in multi-unit negotiations.

  Reason for omitting: Since classical multi-unit negotiations are not considered in this approach the distribution can not be semantically correct processed and is therefore omitted.

  Ströbel and Weinhardt [28] as well as Bichler and Kalagnanam [30] incorporate a Distribution parameter into their taxonomies.

- Attribute: Clearing Schedule

  Purpose: The Clearing Schedule defines when offer matchings take place.

  Reason for omitting: Each time a negotiation terminates the offers are matched. Hence, implicitly a triggered clearing schedule (the trigger being the conducted negotiation termination) is assumed for all negotiations within this approach.

  Clearing Schedule parameters defining additional clearing policies can be found in [43] and [28].

- Attribute: Sorting

  Purpose: The Sorting attribute specifies whether certain agents are ex ante excluded from reaching an agreement. This allows for definition of possible transaction partners and agents that can never join in such a transaction.

  Reason for omitting: Agents that are not permitted to reach an agreement should also be excluded from the negotiation process itself. That is why this exclusion will take place during the admission phase and is therefore expressed within the *Admission Restriction* parameter.

  Stroebel and Weinhardt present a sorting attribute in [28].

4.2.14 Information Processing Category

- Attribute: Trace

  Purpose: The Trace parameter specifies whether anonymous offers are allowed.

  Reason for omitting: This attribute is omitted because of similar reasons to those stated for the Privacy attribute. All offers in SLA negotiations have to state their originator for semantically correct processing.

  The Trace parameter was introduced in the Montreal Taxonomy for Electronic Negotiations [28].

- Attribute: Communication

  Purpose: Whether other messages than just offers are allowed to be exchanged between the negotiating agents is defined with the Communication attribute.

  Reason for omitting: Since this approach derives a framework within Service-Oriented Architectures the negotiations will be conducted via remote method/service invocations. Therefore the messages exchanged are always conducted as method invocations. One cannot distinguish between different kinds of messages only between different methods to invoke.

  If other messages than offers should be allowed to exchange the services would have to expose such methods for general information exchange. However, respective specifications are outside of the scope of this thesis.

  Again this concept was described in Stroebel and Weinhardt's work [28].

- Attribute: Transaction

  Purpose: The Transaction parameter defines whether past deals can be queried by the agents.

Reason for omitting: It is semantically difficult to define which service should save which past deals. It would be possible to let only service providers save past deals, some kind of *Negotiation Coordinator* or even service consumers.

Since the history of past transactions provides agents involved in current negotiations with less additional information to be processed for their internal strategies but imposes much effort to specify, this parameter is not considered in this thesis.

- Attribute: Distribution

  Purpose: The Distribution attribute defines how and when the negotiation information can be accessed.

  Reason for omitting: Since all the information concerning the negotiation can be accessed at the *Information Service* specified in the roles and agents section implicitly continuous distribution is assumed for all negotiations allowed by this approach.

4.3 Data Model of Metainformation

This subsection will derive a data model based on the attributes identified before. It will illustrate and structure all the concepts needed to describe a particular negotiation protocol. The XML representation of this data model needed for WS-Agreement scenarios will be defined subsequently in the next subsection.

4.3.1 Used Technology

The presented data model will be described with the means of the Entity-Relationship Model (ERM) [38]. This language was chosen because of different reasons: Since this model is supposed to describe negotiation data intuitively, a data modelling language was considered adequate. Data modelling languages allow for structuring of different data concepts, with according attributes and attribute domains, along with their relations. This directly suits the approach of describing a negotiation taken in this thesis.

ERM schemata distinguish between *data entities* and their *relationships*. This allows for definition of cardinalities and dependencies between *data entities*. Some of the negotiation attributes identified do have such cardinalities assigned to them: for example the **Agents** attribute can possibly define one to n agents per role which is a perfect example for a cardinality expressable in the ERM modelling language. The concept of cardinalities assigned to *relationships* in the ERM modelling language is thus very adequate to express the different quantities of distinct attribute concepts and their relation.

Each *data entity* in the following schema will consist of a set of attributes taken from the attribute list above plus some id attributes used to express the dependencies between these entities.

To formalize the cardinalities associated with a relationship an extended form of the ERM language is used. Instead of defining the complexity of a relationship whith the (1,M,N) notation (complexity of a relationship) *min,max cardinalities* will be used. *Min,max cardinalities* assign a minimum and a maximum value to every link between a *relationship*

and an *entity*. This way the *relationships* between *entities* can be defined in an unambiguous way. It also allows for modelling dependencies between entities in a more correct way.

Another reason for using the ERM language is that the instance layer of an ERM schema can easily be transferred into a database model. This way negotiation types defined as instances of this ERM schema can simply be transformed into database entries. Represented as such entries one can save negotiation types, defined with the means provided in this thesis, in some negotiation database. This allows for complex architectures incorporating negotiation servers, where the agents can retrieve different negotiation type definitions and where those definitions can be saved for further referencing. Such architectures and the use of database storage for negotiation types is outside the scope of this paper, however the development of such will be eased because of the application of an ERM schema. A short overview on how distributed look-up infrastructures can be created will be given in chapter 5.6.

4.3.2 Derivation of Data Entities and Relationships

In this subsection the attributes identified above will be structered into *data entities* and their *relationships*.

Identified Entities and assigned attributes:

- Entity: **Negotiation**

  Attributes:

  1. NegotiationID
     Domain: String
     This attribute unambiguously identifies this negotiation type.
  2. Rounds
     Domain: positive Integer, excluding zero
     The *Rounds* attribute defines the number of rounds in a negotiation.
  3. Start
     Domain: String
     This attribute describes the starting rule of the negotiation. The domain string was chosen to enable simple date/time values as well as constraint expressions.
  4. Termination
     Domain: String
     The **Termination** attribute, analogous to the **Start of Negotiation** parameter, defines the end of the negotiation or the current negotiation round respectively.
  5. ArbitrationType
     Domain: {punishing, rewarding, none, all}
     This attribute defines the type of arbitration used in this negotiation type.
  6. ArbitrationRule
     Domain: String
     The ArbitrationRule specifies the rules for arbitration in this negotiation type. To allow constraint expressions again the string domain is applied.

- Entity: **ContextElement**

  Attributes:

  1. NegotiationID
     Domain: String
     This identifier (ID) references the negotiation concerning this *context element*.
  2. ContextElementID Domain: String
     The ContextElementID identifies the corresponding *context element* defined in the negotiated WS-Agreement.

- Entity: **ReferencedService**

  Attributes:

  1. NegotiationID
     Domain: String
     This ID references the negotiation the service is involved in.
  2. ServiceID
     Domain: String
     The ServiceID unambiguously identifies the service. This value relates to the service identifier used in the WS-Agreement template to refer to the respective service.

- Entity: **ServiceAttribute**

  Attributes:

  1. ServiceID
     Domain: String
     This ID references the service exposing this attribute.
  2. AttributeID
     Domain: String
     The AttributeID unambiguously identifies the service attribute. This value is based on the ID of the respective term in the WS-Agreement template representing the negotiated attribute.
  3. Values
     Domain: {single, multiple}
     Whether only single values or value ranges are allowed is specified with this attribute.

- Entity: **Domain**

  Attributes:

  1. DomainID
     Domain: String
     This attribute unambiguously identifies the respective domain.
  2. ID
     Domain: String
     This ID references to the attribute or *context element* the domain is assigned to.

3. Type
   Domain: {ordered, not ordered}
   This attribute defines whether the domain is ordered or not.
4. DomainDefinition
   Domain: String
   This attribute allows the definition of possible domains. Different data types can be set here, e. g. string, integer or float.

- Entity: **AttributeRestriction**

  Attributes:

  1. AttributeRestrictionID
     Domain: String This ID unambiguously identifies the respective attribute restriction.
  2. AttributeID
     Domain: String
     This ID references to the attribute the restriction applies to.
  3. AttributeRestrictionRule
     Domain: String
     The AttributeRestrictionRule defines the actual restriction. To allow arbitrary constriant expressions, again the string domain is used.

- Entity: **NegotiationRestriction**

  Attributes:

  1. NegotiationRestrictionID
     Domain: String
     This ID unambiguously identifies the respective negotiation restriction.
  2. NegotiationID
     Domain: String
     This ID references to the negotiation the restriction applies to.
  3. NegotiationRestrictionRule
     Domain: String
     The NegotiationRestrictionRule defines the restriction. To allow arbitrary constriant expressions, again the string domain is used.

- Entity: **OfferAllocationPolicy**

  Attributes:

  1. NegotiationID
     Domain: String
     This ID references to the negotiation the restriction applies to.
  2. AllocationRestriction
     Domain: String
     The AllocationRestriction defines the allocation process with its restrictions. To allow arbitrary constriant expressions, again the string domain is used.

- Entity: **Role**

  Attributes:

1. NegotiationID
   Domain: String
   This ID references to the negotiation exposing this role.
2. RoleID
   Domain: {service provider, service consumer, negotiation coordinator, information service, ANY STRING}
   The RoleID unambiguously identifies the respective role.
3. AdmissionRestriction
   Domain: String
   The AdmissionRestriction specifies whether agents have to perform some action to receive admission to the negotiation. In order to enable arbitrary constraint expressions the string domain was chosen. If this attribute is left empty no admission restriction is set.

- Entity: **Agent**

  Attributes:

  1. AgentEPR
     Domain: String
     The EPR references to the agent within a distributed system. To allow different types of EPRs, like ones created according to the WS-Adressing specification [52], the string domain is applied.
  2. RoleID
     Domain: String
     This ID references to the role the respective agent adopts.

- Entity: **InformationProcessingPolicy**

  Attributes:

  1. InformationProcessingPolicyID
     Domain: String
     This ID unambiguously identifies this information processing policy.
  2. NegotiationID
     Domain: String
     This ID references to the negotiation applying this information processing policy.
  3. NegotiationTransparency
     Domain: {public, protected, none}
     This attribute defines whether past offers of the negotiation can be queried and if so which agents can do so.
  4. NegotiationContent
     Domain: String
     This attribute specifies the content of the offers logged, in case of past offers are logged. To enable arbitrary constraint expressions the string domain is used.
  5. StatusTransparency
     Domain: {public, protected, none}
     The StatusTransparency indicates which agents can access the current status of the negotiation.

6. StatusContent
   Domain: String
   This attribute specifies the content of the negotiation status that can be queried. Again, to allow expressions formalized in external constraint languages the string domain is used.

Now the relationships between the different entities will be defined in terms of cardinalities: The central data concept is the **Negotiation** entity which has relationships to different other concepts describing concrete aspects of a particular negotiation. Every **Negotiation** has exactly one (1,1) **InformationProcessingPolicy** assigned to it, whereas an **InformationProcessingPolicy** can be applied in different **Negotiations**. **InformationProcessingPolicies** can even be defined before a negotiation type using it is specified. That is why the **InformationProcessingPolicy** is assigned to zero to n Negotiations (0,*). Analogously the relationship between **Negotiation** and **OfferAllocationPolicy** is defined.

Every **Role** defined can be assigned to an arbitrary number of negotiation types (0,*), including zero because again a **Role** can be defined independently to the specification of a negotiation type exposing it. On the other hand each **Negotiation** is related to at least four roles. Other oles can be defined subsequently for each negotiation, that is why the cardinality is defined to be (4,*) (for the four default roles, see chapter 4.2.4.

Each **Role** has to have at least one agent adopting it (1,*), whereas an gent can adopt an arbitrary number of roles (0,*) at a time. This results from providing different (default) roles that cover a certain functionality used within a negotiation which can be provided by one agent (therefore adopting several roles) or by more than one, possibly adopting only one role each. The only constraint posed on the relationship between **Roles** and **Agents** is that one agent cannot act as service provider and consumer, as already described before.

The items of an agreement subject to the negotiation are represented by the **ReferencedServices**, their respective **ServiceAttributes** or the **ContextElements**. Note: Actually the referenced services and *context elements* are compontents of the one item negotiated: the WS-Agreement. For clarity reasons however, these concepts are modelled directly since a WS-Agreement entity would only aggregate those without defining any additional information.

A negotiation can refer to one to many services (1,*), whereas a service can be involved in at most one negotiation at a time (0,1). Each service exposes one to many negotiatable attributes (1,*) and each attribute is assigned to exactly one service (1,1). On the other hand each negotiation can optionally concern a distinct number of *context elements* (0,*).

Finally, exactly one **Domain** is assigned to each **ServiceAttribute** and *context element* (1,1), along with an arbitrary number of **AttributeRestrictions** (0,*). **Domains** can again be defined independently and are therefore assigned to zero to many **ServcesAttributes** or *context elements* (0,*) and each restriction is referring to exactly one **ServiceAttribute** or **ContextElement** (1,1).

General restrictions, expressed as NegotiationRestrictions are analogously assigned to exactly one negotiation (1,1), whereas each negotiation can have zero to many restrictions (0,*).

4.3.3 Entity Relationship Model

The above identified entities and relationships will result in the following ERM schema:

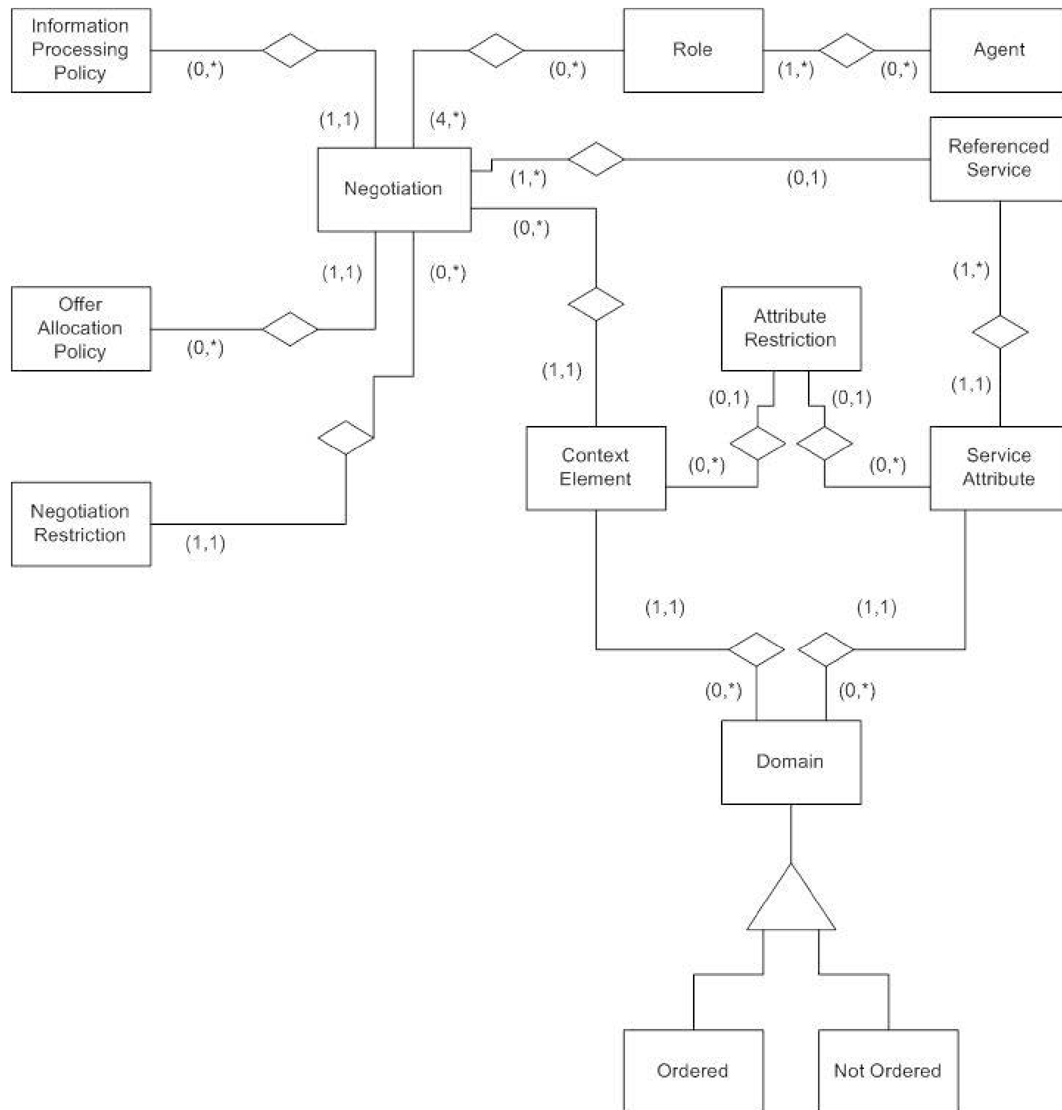

Figure 5: Entity-Relationship Schema of Negotiation Types

This data model was created using basic ERM concepts, including cardinalities for relationships and generalization of entities. The generalization of the Domain entity was used to explicitly show the two classes of Domains used in negotiations (ordered and not ordered).

4.4 Extensible Markup Language Documents

In this subsection the XML-based representation of the above introduced data model will be presented. In order to be able to process such a data structure it has to be formalized in a way that software agents can handle. Since WS-Agreement negotiation will take place in a Web Services environment XML was the intuitive language to use for the formalization of the derived datamodel. Not only do all of the Web Services-related standards utilize XML as a formalization language but XML also provides means for structuring negotiation metadata in a very suitable way.

The derived data model describes the general strcuture of negotiation types by specifying the schema layer. One particular negotiation type is therefore simply an instance of the described schema. Hence concrete negotiation type documents depict the instance layer respectively. Since a negotiation type has to be expressed with the means of XML, a document has to be derived, based on the data model presented above, that defines the structure of these XML documents. The most appropriate language used to describe the structure of XML documents is XML Schema [54]. The following document describing the structure of negotiation type documents expressed in XML will therefore be presented as an XML schema document.

This XML schema document is defined as follows:

Listing 1: Negotiation Type XML-Schema

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <xsd:schema
4     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5     xmlns:wsagn="http://xml.netbeans.org/examples/negotiation"
6     targetNamespace="http://xml.netbeans.org/examples/
         negotiation"
7     xmlns="http://xml.netbeans.org/examples/negotiation"
8     elementFormDefault="qualified">
9
10    <xsd:complexType name="TemplateType">
11        <xsd:sequence>
12            <xsd:element name="endpoint" type="xsd:anyType"/>
13            <xsd:element name="templateID" type="xsd:string"/>
14        </xsd:sequence>
15    </xsd:complexType>
16
17    <xsd:simpleType name="RoundType">
18        <xsd:restriction base="xsd:integer">
19            <xsd:minInclusive value="1"/>
20        </xsd:restriction>
21    </xsd:simpleType>
22
23    <xsd:simpleType name="ArbitrationFormType">
24        <xsd:restriction base="xsd:string">
25            <xsd:enumeration value="punishing"/>
26            <xsd:enumeration value="rewarding"/>
27            <xsd:enumeration value="all"/>
28            <xsd:enumeration value="none"/>
29        </xsd:restriction>
30    </xsd:simpleType>
31
32    <xsd:complexType name="ArbitrationType">
33        <xsd:sequence>
34            <xsd:element name="arbitrationRule" type="xsd:
                 anyType" minOccurs="0" maxOccurs="1"/>
35            <!--this element is present if punishing,
                 rewarding or all is defined as arbitrationForm
                 -->
36        </xsd:sequence>
37        <xsd:attribute name="arbitrationForm" type="wsagn:
             ArbitrationFormType"/>
38    </xsd:complexType>
```

```
39
40     <xsd:simpleType name="AdmissionRestrictionFormType">
41         <xsd:restriction base="xsd:string">
42             <xsd:enumeration value="open"/>
43             <xsd:enumeration value="restricted"/>
44         </xsd:restriction>
45     </xsd:simpleType>
46
47     <xsd:complexType name="AdmissionType">
48         <xsd:sequence>
49             <xsd:element name="admissionRestrictionRule" type=
                   "xsd:anyType" minOccurs="0" maxOccurs="1"/>
50             <!--this element is present if restricted
                   admission is defined-->
51         </xsd:sequence>
52         <xsd:attribute name="admissionRestrictionForm" type="
               wsagn:AdmissionRestrictionFormType"/>
53     </xsd:complexType>
54
55     <xsd:complexType name="RoleType">
56         <xsd:sequence>
57             <xsd:element name="maximumNumberOfAgents" type="
                   xsd:integer" minOccurs="0" maxOccurs="1"/>
58             <xsd:element name="admissionRestriction" type="
                   wsagn:AdmissionType"/>
59         </xsd:sequence>
60         <xsd:attribute name="roleName" type="xsd:string"/>
61         <xsd:attribute name="permissionToPostOffers" type="xsd
               :boolean"/>
62     </xsd:complexType>
63
64     <xsd:simpleType name="ValuesType">
65         <xsd:restriction base="xsd:string">
66             <xsd:enumeration value="single"/>
67             <xsd:enumeration value="multiple"/>
68         </xsd:restriction>
69     </xsd:simpleType>
70
71     <xsd:complexType name="ContextElementType">
72         <xsd:simpleContent>
73             <xsd:extension base="xsd:string">
74                 <xsd:attribute name="domain" type="xsd:string"
                       />
75             </xsd:extension>
76         </xsd:simpleContent>
77     </xsd:complexType>
78
79     <xsd:complexType name="TermType">
80         <xsd:simpleContent>
81             <xsd:extension base="xsd:string">
82                 <xsd:attribute name="domain" type="xsd:string"
                       />
83                 <xsd:attribute name="values" type="wsagn:
                       ValuesType"/>
84             </xsd:extension>
85         </xsd:simpleContent>
```

```
86      </xsd:complexType>
87
88      <xsd:complexType name="NegotiatedIssuesType">
89          <xsd:sequence>
90              <xsd:element name="contextElements">
91                  <xsd:complexType>
92                      <xsd:sequence>
93                          <xsd:element name="elementID" type="
                                wsagn:ContextElementType" minOccurs
                                ="0" maxOccurs="unbounded"/>
94                      </xsd:sequence>
95                      <xsd:attribute name="extendable" type="xsd
                            :boolean"/>
96                  </xsd:complexType>
97              </xsd:element>
98              <xsd:element name="serviceDescriptionTerms">
99                  <xsd:complexType>
100                     <xsd:sequence>
101                         <xsd:element name="
                                serviceDescriptionTermID" type="
                                wsagn:TermType" minOccurs="0"
                                maxOccurs="unbounded"/>
102                     </xsd:sequence>
103                     <xsd:attribute name="extendable" type="xsd
                            :boolean"/>
104                 </xsd:complexType>
105             </xsd:element>
106             <xsd:element name="servicePropertyTerms">
107                 <xsd:complexType>
108                     <xsd:sequence>
109                         <xsd:element name="
                                servicePropertyTermID" type="wsagn:
                                TermType" minOccurs="0" maxOccurs="
                                unbounded"/>
110                     </xsd:sequence>
111                     <xsd:attribute name="extendable" type="xsd
                            :boolean"/>
112                 </xsd:complexType>
113             </xsd:element>
114             <xsd:element name="guaranteeTerms">
115                 <xsd:complexType>
116                     <xsd:sequence>
117                         <xsd:element name="guaranteeTermID"
                                type="wsagn:TermType" minOccurs="0"
                                 maxOccurs="unbounded"/>
118                     </xsd:sequence>
119                     <xsd:attribute name="extendable" type="xsd
                            :boolean"/>
120                 </xsd:complexType>
121             </xsd:element>
122         </xsd:sequence>
123     </xsd:complexType>
124
125     <xsd:simpleType name="ProgressFormType">
126         <xsd:restriction base="xsd:string">
127             <xsd:enumeration value="ascending"/>
```

```
128             <xsd:enumeration value="descending"/>
129         </xsd:restriction>
130     </xsd:simpleType>
131
132     <xsd:complexType name="ProgressType">
133         <xsd:sequence>
134             <xsd:element name="progressForm" type="wsagn:
                    ProgressFormType"/>
135             <xsd:element name="delta" type="xsd:anyType"
                    minOccurs="0" maxOccurs="1"/>
136         </xsd:sequence>
137     </xsd:complexType>
138
139     <xsd:complexType name="ThresholdType">
140         <xsd:choice>
141             <xsd:element name="lowerBound" type="xsd:anyType"
                    />
142             <xsd:element name="upperBound" type="xsd:anyType"
                    />
143         </xsd:choice>
144     </xsd:complexType>
145
146     <xsd:complexType name="AttributeRestrictionType">
147         <xsd:sequence>
148             <xsd:element name="attribute" type="xsd:string"/>
149             <xsd:element name="restriction">
150                 <xsd:complexType>
151                     <xsd:choice>
152                         <xsd:element name="progress" type="
                                ProgressType"/>
153                         <xsd:element name="threshold" type="
                                wsagn:ThresholdType"/>
154                         <xsd:element name="restrictionRule"
                                type="xsd:anyType"/>
155                     </xsd:choice>
156                 </xsd:complexType>
157             </xsd:element>
158         </xsd:sequence>
159     </xsd:complexType>
160
161     <xsd:simpleType name="MatchingFormType">
162         <xsd:restriction base="xsd:string">
163             <xsd:enumeration value="forwarded"/>
164             <xsd:enumeration value="defined"/>
165         </xsd:restriction>
166     </xsd:simpleType>
167
168     <xsd:complexType name="OfferAllocationType">
169         <xsd:sequence>
170             <xsd:element name="matchingForm" type="wsagn:
                    MatchingFormType"/>
171             <xsd:element name="matchingRule" type="xsd:anyType
                    " minOccurs="0" maxOccurs="1"/>
172             <!--this element is present if defined matching is
                     specified-->
173         </xsd:sequence>
```

```
174     </xsd:complexType>
175
176     <xsd:simpleType name="TransparencyType">
177         <xsd:restriction base="xsd:string">
178             <xsd:enumeration value="public"/>
179             <xsd:enumeration value="protected"/>
180             <xsd:enumeration value="none"/>
181         </xsd:restriction>
182     </xsd:simpleType>
183
184     <xsd:complexType name="InformationProcessingType">
185         <xsd:sequence>
186             <xsd:element name="negotiationTransparency" type="
                    wsagn:TransparencyType"/>
187             <xsd:element name="negotiationContent" type="xsd:
                    anyType" minOccurs="0" maxOccurs="1"/>
188             <!--this element is present if
                    negotiationTransparency is not set to none-->
189             <xsd:element name="statusTransparency" type="wsagn
                    :TransparencyType"/>
190             <xsd:element name="statusContent" type="xsd:
                    anyType" minOccurs="0" maxOccurs="1"/>
191             <!--this element is present if statusTransparency
                    is not set to none-->
192         </xsd:sequence>
193     </xsd:complexType>
194
195     <xsd:complexType name="NegotiationType">
196         <xsd:sequence>
197             <xsd:element name="negotiationTypeID" type="xsd:
                    string"/>
198             <xsd:element name="wsAgreementTemplate" type="
                    wsagn:TemplateType"/>
199             <xsd:element name="start" type="xsd:anyType"
                    minOccurs="0" maxOccurs="1"/>
200             <xsd:element name="termination" type="xsd:anyType"
                     minOccurs="0" maxOccurs="1"/>
201             <!--start and termination are only set here if
                    they are not defined in terms of points in time
                    -->
202             <xsd:element name="rounds" type="wsagn:RoundType"
                    />
203             <xsd:element name="arbitration" type="wsagn:
                    ArbitrationType"/>
204             <xsd:element name="role" type="wsagn:RoleType"
                    minOccurs="4" maxOccurs="unbounded"/>
205             <xsd:element name="negotiatedIssues" type="wsagn:
                    NegotiatedIssuesType"/>
206             <xsd:element name="attributeRestriction" type="
                    wsagn:AttributeRestrictionType" minOccurs="0"
                    maxOccurs="unbounded"/>
207             <xsd:element name="generalRestrictionRule" type="
                    xsd:anyType" minOccurs="0" maxOccurs="unbounded
                    "/>
208             <xsd:element name="offerAllocation" type="wsagn:
                    OfferAllocationType"/>
```

```
209                <xsd:element name="informationProcessing" type="
                       wsagn:InformationProcessingType"/>
210            </xsd:sequence>
211        </xsd:complexType>
212
213        <xsd:element name="negotiation" type="NegotiationType"/>
214
215    </xsd:schema>
```

negotiationTypeID The *negotiationTypeID*-element unambiguously identifies this negotiation type. To allow a multitude of identifiers its domain was set to xsd:string. This way human-understandable names can be used as well as any machine-generated id.

wsAgreementTemplate To reference the WS-Agreement template the negotiation is based on the *wsAgreementTemplate*-element is set. This element represents a logical link to a specific template offered by a particular service. Since this template can only be uniquely identified within the service offering it it has to be referenced by using an EPR for the service and the ID of the template itself. In order to reduce traffic this template is not incorporated into the negotiation type document. Agents interested in joining the negotiation have to query the template independently using the methods already provided by the WS-Agreement standard.

start The *start*-element describes the starting conditions of a negotiation as assertions on some external or internal values or conditions. A starting condition could be, for example, the incoming of some message or the internal state transition of the *Negotiation Coordinator* service.

termination Analogously to the starting conditions the termiation of a negotiation is defined.

Start- and *termination*-elements are both optional. This is because they are only defined in the negotiation type, if they are not only time-dependend. If they only refer to some point in time for start and/or termination these elements are set in the negotiation instance document, as described in the next section.

Both elements are assigned the domain xsd:anyType, to allow arbitrary constraint expressions formalized in external constraint languages. This can involve formal condition-consequence rules as well as human-readable expressions.

rounds The *rounds*-element defines the number of negotiation rounds. A round within a negotiation is defined as one complete negotiation process as defined in the negotiation type document. Each round is terminated, when one of the defined termination rules applies as in any one-rounded negotiation. Each new round is started by the *Negotiation Coordinator* by invoking the corresponding *newRound()*-method on all participants as described in the section 6 6.

arbitration This element defines whether or not compliance to the protocol is rewarded, violation is punished or both. Hence, this element contains an attribute defining whether the protocol is punishing, rewarding, both or none of them. The element itself (called *arbitrationRule*) is defined to be of xsd:anyType. This way arbitrary constraint expressions defining what action is taken by the *Negotiation Coordinator* (e. g. punishment) under what conditions and what consequences this action has for the participant.

For example negotiation designers can specify rules stating that every agent posting offers in a way not compliant to the negotiation type definitions will get negative assessments posted to some external reputation system. The *arbitrationRule*-element is of course optional, because in case of no arbitration (arbitrationForm equals none) no rule for such actions can be specified.

role There have to be at least four *role*-elements in a negotiation type document. Each role element describes one role involved in the corresponding negotiation, and therefore the minimum amount of roles originates in the four default roles that have to be present in every negotiation: service provider, service consumer, *Negotiation Coordinator* and *Information Service*. The service provider and consumer roles originate in the common distinction between providers and consumers of services in SLA environments. The *Negotiation Coordinator* represents the agent administrating the admission process and distributing the negotiation data, this concept is explained in more deatail later. Finally the *Information Service* references the agent(s) in a negotiation process providing the others with information about the current negotiation status or past offers.

Each role exposes two attributes: *roleName* and *permissionToPostOffers*. The *roleName*-attribute defines the identifier of that particular role. Such an identifier is formalized as a string value to allow a multitude of IDs to be set. In addition to the four default-identifiers mentioned above arbitrary additional roles can be specified. The *permissionToPostOffers*-attribute defines whether this side of the negotiation (represented by this particular role) is allowed to post offers. This way a negotiation designer can allow only one side to create bids, for example to define auction protocols, as well as both sides, for example for One-on-one Bargaining.

A *role*-element consists of two child-elements: the *maximumNumberOfAgents* and the *admissionRestriction*. The optional *maximumNumberOfAgents* defines the maximum amount of participating agents for this particular role. In order to define a One-on-one Bargaining protocol each side has to be restricted to one agent, whereas in an auction protocol only one side is restricted to one and the other side possibly not restricted at all or at least to some high number of agents. If a role can be adopted by an unbounded number of agents this element is omitted, if some restriction on the maximum number exists it is expressed as an integer value in this element.

Some negotiation protocols require the joining agents to satisfy some criteria to be admitted to the negotiation. Such admission restrictions can also be specified and incorporated in this approach within the *admissionRestriction*-element. An *admissionRestriction*-element consists of one element of xsd:anyType to enable arbitrary admission restriction rules in some constraint language and one attribute defining whether admission restrictions exist or not. If open admission is defined no admission restriction rule will be specified and the whole *admissionRestriction*-element only consists of an empty ele-

ment and the attribute defining open admission. If restricted admission is defined the *admissionRestrictionRule*-element has to be filled with the restriction rules.

Note: The actual agents participating in the negotiation are referenced in the negotiation instance document as described later.

negotiatedIssues The next element of the negotiation type document defines the negotiated items it terms of a service attributes (*service description*, *service property* and *guarantee terms*) and *context elements* as described in the WS-Agreement specification. The relationship between a negotiation type and the corresponding WS-Agreement template as well as the delimitation of negotiatable service attributes is discussed in more detail in the following section.

Each of the items just mentioned exposes a unique identifier within the corresponding WS-Agreement template. Since every negotiation type always refers to one particular template within the *wsAgreementTemplate*-element, the term identifier already mentioned is sufficient to unambiguously identify the respective term.

In order to present the negotiated issues in a structured way negotiation type document contains an *negotiatedIssues*-element with four child-elements: *contextElements*, *serviceDescriptionTerms*, *servicePropertyTerms* and *guaranteeTerms*. Each of these elements contains a, possibly empty, list of references to the corresponding elements in the template.

The list is allowed to be empty because not every negotiation concerns all possibly negotiatable items. In order to define a negotiation, at least one of the listed items has to be referenced here; if not there would exist no attribute subject to the negotiation and therefore no negotiation at all.

On the other hand these item lists are of unbounded length because an arbitrary number of terms and *context elements* can possibly be referenced within each category. Each category also exposes a boolean attribute defining whether the given set of issues can be extended during negotiation. This could be useful, for example when a negotiation participant can accept the proposed agreement under the condition that one additional service parameter is guaranteed, that has not be described yet, for example response time. If the *extendable*-attribute is set true such a new *service property term* and a corresponding *guarantee term* could be inserted while negotiating.

Finally, each term-element consists of two child-elements: *domain*, *values*. The *domain*-element assigns a basic data type as a domain to the respective attribute. For example some time data type would be assigned to a term referring to maximum response time and integer would be applied as domain for size of memory in megabytes. Whether or not such a domain is ordered depends on whether the data type the domain represents is ordered or not. For example integer would be ordered whereas string would not. The *values*-element defines whether multiple values are allowed to be offered for this attribute or not. Depending on the nature of the applied domain different ways of defining multiple values are permitted as described above.

Context elements differ slightly from service term elements in that they do not allow for multiple values. Multiple values can only be used semantically correct for some value

ranges of negotiated service attributes like throughput etc. *context elements* define distinct values for attributes concerning the agreement's overall behaviour, like expiration time, for which only single values can be set. Therefore *context element* references only expose the *domain*-element without providing the *values*-element as service term references would.

attributeRestriction Each negotiation type can optionally define an arbitrary number of attribute restrictions. Each attribute restriction defines how one particular attribute (or negotiated issue, as just described) is treated within the negotiation. If a negotiation designer wants to specify that for some particular attribute a new offer always has to succeed the last offer, like in an English Auction, this would be achieved with a attribute restriction regarding this attribute.

Each *attributeRestriction*-element contains an *attribute*-child element referencing the negotiated item this restriction applies for. In addition one of three possible restriction classes can be chosen: *progress*, *threshold* or a general *restrictionRule*.

The *progress*-element defines the direction of a negotiation regarding the referenced attribute. With this optional element one can specify whether new offers have to be higher or lower than current ones. Intuitively the *progress*-element is only applicable for ordered domains. In addition to the direction, specified in the *progressForm*-child element, one can define some minimum increment or decrement with the *delta*-child element.

The *threshold*-element defines what is called reserve price in negotiation theory. A reserve price is some upper or lower bound to the overall possible values to be set for some attribute within a negotiation. Normally this concept is only applied to the price attribute, as depicted by the name, but in this more general approach it can also be applied to all other service attributes of ordered domains. A negotiation designer could for example specify that in either case at least 512 MB of memory have to be set for some service, every value below that is permitted in the subsequent negotiation.

The last possible child element of the attributeRestriction is called *restrictionRule*. This element is of xsd:anyType and can be used to express any arbitrary attribute-related constraint in addition to *thresholds* and *progress* restrictions.

generalRestriction Analogously to the *attributeRestrictions* that relate to one attribute each more general restrictions relating to the negotiation as a whole can be expressed with the *generalRestriction*-element. Such restrictions govern every constraint relating to more than one attribute of the negotiated service. If a negotiation designer wants to specify, that in a new offer at least one of two different attributes has to be offered a higher value than in the current offer this is done by a general restriction. In order to allow arbitrary external constraint languages again xsd:anyType is applied.

offerAllocation The *offerAllocation*-element defines the way the clearing of the negotiation is conducted, that is how the winning offers are identified and transformed into a WS-Agreement. The offer matching can be of forwarded or defined form as already hinted when the attribute list was presented.

In either case a valid agreement is created by one side receiving an offer and accepting it

by invoking the *acceptAgreement()*-method on the posting agent, the only difference lies in whether the other participants know how the offer allocation is conducted or not. In the forwarded matching case the rules applied in offer allocating are not explicitly defined. The accepting agent creates an agreement without letting the other participants know according to which set of rules. In the defined case the rules applied for the offer matching are explicitly defined in the negotiation type document and therefore openly available. This way every agent can predict the winning offer before the actual agreement is posted to the involved parties. The *offerAllocation*-element thus consists of a *matchingForm*-child element specifying whether defined or forwarded offer matching is applied and an optional child element called *matchingRule* defining the rules for offer matching applied. This element is only present if defined offer matching is used.

informationProcessing As described above there are two classes of negotiation data accessible to the participants: the current negotiation status and the past offers. The negotiation status is represented by all current offers of all participants allowed to post offers, whereas the past offers are all offers posted in this negotiation instance until now. For negotiation status as well as for the past offers negotiation designers can independently define the content of the accessible information. For example each offer can be specified to be accessible completely or just the originator along with the proposed quarantee terms etc.

To allow differentiated definitions of accessible negotiation data the *informationProcessing*-element contains a *negotiationTransparency*- and a *statusTransparency*-element for the past offers and the negotiation status respectively. In addition it contains two optional elements defining the respective contents for status and past offers (*negotiationContent* and *statusContent*).

The two *Transparency*-elements are restricted to three possible values each: public, protected and none. When public is specified the negotiation status, or the past offers respectively, can be queried by all agents; no restriction is applied to such information requests. Protected transparency denotes that only agents involved in the actual negotiation can do so whereas none is used, when no past offers or negotiation status can be queried. This way negotiation designers can specify sealed-bid auctions for example.

On the other hand the two *Content*-elements define the exact content that can be accessed. This involves restrictions on each offered agreement document as well as the definition what information is considered to be the negotiation status, as described before.

The XML formalization of the metadata represents a operable document structure of the ERM model to be used in WS-Agreement negotiations. XML documents suite this approach in providing dynamic structure descriptions using the XML schema language. This allows for definition of flexible element structures and numbers of elements. Also the distinction between attributes and elements represents an intuitive way to describe negotiation parameters.

However an ERM schema can easily be transformed into a database schema used for storing different negotiation types for backup reasons. Since every concept in the XML schema presented originates in the ERM model this document can easily be transformed back into an ERM and a database schema respectively. This allows for storing differend

negotiation type documents as database entries. By allowing for database use negotiation designers can reuse once implemented negotiation types in different scenarios with only few changes to be applied (concerning the referenced templetes etc.). This increases reusability and contributes to possible negotiation infrastructures to be implemented. Such infrastructure definitions are outside the scope of this thesis; a short overview on possible systems of look-up servers will be given in the next section though.

5 Exchange Protocol for Negotiation Metadata

In this section the protocol used for exchanging the negotiation metadata is presented. This protocol will be used to provide all agents, wanting to engage in a certain negotiation, with the required information for joining. As a fundamental basis for this protocol the datastructure derived in the previous section will be used. This datastructure contains all the information necessary for engaging in a negotiation by defining its type and is therefore used for communicating the negotiation metadata to the (prospective) participants.

Since this thesis provides an approach for conduction negotiations in a Web Services enwironment the exchange protocol will not focus on exchanged messages primarily, but on the provided services and respective methods to be invoked subsequently. This approach was taken due to the service oriented environment for WS-Agreement negotiations. Such service oriented environments focus inherently on provided services and define protocols in terms of method invocation seqences. The methods presented in this section will be grouped into interfaces, representing the roles that an agent can adopt within the exchange protocol. To provide a usable representation of these roles the respective WSDL [1] descriptions will be presented, after discussing the functionality provided with the respective methods.

In order to present a comprehensive specification of the exchange protocol not only the needed methods, but also the messages and datastructures used for method invocations will be presented. The proposed datastructures will allow to distinguish between negotiation types and already instantiated negotiations, a concept which will subsequently be explained. This is necessary for enabling a multitude of scenarios to be supported by this protocol. The messages and documents used will be defined in terms of SOAP message formats and XML schema definitions respectively [2, 54]. On the other hand, the scenarios to be supported by this protocol will be sketched with Unified Modelling Language (UML) Sequence Diagrams [55].

This section is organised as follows: First the negotiation type and negotiation instance concepts will be introduced for this is a crucial foundation for the exchange protocol. Then the abstract exchange process will be presented along with some scenarios to be supported. After illustrating the exchange process by describing the needed documents and data structures, the roles and respective services involved will be introduced. The second part of this section will describe some sample processes possible with the introduced methods and services. The WSDL and SOAP descriptions of the services and messages will be presented together with those for the negotiation protocol in section 6.

5.1 Negotiation Types and Instances

A basic concept needed for the exchange process of negotiation metadata is the distinction between negotiation types and instances. These concepts can be used analogously to types and instances of object-oriented programming languages, like Java or C++. A type defines a class of concrete objects by specifying their general structure. An instance, or object in object-oriented programming, is one unit of such a class exhibiting this structure. It has a unique identifier and can therefore unambiguously be referenced as a program entity.

Analogously a negotiation type describes a general class of negotiations and defines their common attributes and elements. A negotiation instance, however, stands for one particular negotiation of some type. For example, a negotiation type can define, that there is one agent involved not allowed to post offers, whereas on the other side n agents can participate posting offers, in which every offer has to succeed the last posted offer by some amount. The negotiation closes when no agent has posted an offer for some timespan. This roughly describes the class of English Auctions. One particular negotiation instance of this negotiation type represents one particular English Auction conducted at some particular point in time. This concept allows for distinction between general types of negotiation and concrete negotiation instances that can unambiguously be referenced by (potential) participants.

Regarding their elements, negotiation types and instantiated negotiations differ slightly. The main difference is that a negotiation type does not contain some sort of identifier that can be used to refer to a given negotiation instance. The other attributes identified in the previous section, however have to be set when defining a negotiation type. For example the offer submission rules and involved roles have to be set before instantiating because they represent a fundamental part of a whole class of negotiation instances, and therefore of a negotiation type.

There are only three exceptions; only three attributes do not necessarily have to be defined in a negotiation type document: the start and the termination of as well as the agents involved in a negotiation instance.

The start and the termination of a negotiation can be set in the negotiation type or in the negotiation instance document according to the same rule. Both attributes can be defined as assertions over time or as arbitrary constraint expressions not concerning some time values. When a negotiation start or termination is specified in terms of time points then it has to be set in the negotiation instance document; when defined as a constraint expression over other parameters it has to be set in the negotiation type document.

For example a negotiation designer could wish to define some negotiation type always terminating, when no offer has been posed for some time period. This would be possible by defining such a constraint expression in the negotiation type document. Every negotiation instance of this type would subsequently inherit this termination rule. When the start and termination of a negotiation are defined in terms of points in time this cannot be specified in a negotiation type attribute. Each conducted negotiation instance has a particular and unique starting time and a unique termination time. Hence such points in time would not be an attribute of a whole class of negotiation instances and therefore would have to be specified for each individual instance. Other constraints, however can apply to a whole class of negotiations, which is why such start and termination rules are defined in the type document.

The involved agents are not specified in a negotiation type document for two distinct reasons: Formost not every agent participating in the negotiation is known when a negotiation type is created. If anything only one agent will be known mostly: the agent providing the negotiation type document. During the exchange procotol described in this section other agents subsequently join this negotiation without being known to the agent that created the negotiation type. Hence not all agents involved can be set in the negotiation type document, but have to be incorporated into the datastructure representing the actual negotiation instance subsequently. Another reason for not already setting the

involved agents in the type document is that if incorporated in the negotiation type every instance of this type would have to be conducted by exactly the same agents which is not feasible. Setting the participating agents only in the negotiation instance document allows for much more flexible application of this approach.

5.2 Abstract Exchange Process

In this subsection the exchange process will be roughly described in terms of involved roles, documents and supported scenarios. This subsection is supposed to give an overview of how the distribution of the necessary negotiation information will be conducted without presenting too much details of the respective documents and definitions. This will be done in the following subsections.

In order to conduct a negotiation specified with the means provided in this thesis the involved agents have to take part in one particular negotiation instance of one particular type as defined above. The datastructure describing the negotiation instance contains a unique identifier for this instance, a reference to the negotiations type, the list of participating agents and optionally time-based start and termination parameters. This refers to the instantiation concept presented in the previous subsection. In the next subsection the XML-Schema document defining a negotiation instance data structure will be presented. Everytime an agent joins an already instantiated negotiation the datastructure is updated, by adding this agent to the role it adopts, and redistributed to all participants of the negotiation that are already known. At the end of the metadata exchange process thus every involved agent is aware of the start and termination of the negotiation, its type and the agents currently involved.

This very rough description already identifies the roles involved in the exchange protocol. First of all one role has to be defined representing the coordinator of the negotiation. This *Negotiation Coordinator* represents the agent responsible for distributing the negotiation data and handling admission of agents.

Processing admission of agents at one logical centralised coordinator service eases the integration of reputation or security related external systems involved in the admission process. This way most of the consistency problems arising when operating distributed systems can be solved in a centralised way. All agents joining a negotiation do so by invoking a corresponding method on the central coordinator service which handles the whole admission process. Another coordinator task is therefore to notify the participating agents when others have joined by posting the updated negotiation instance document to them as described above.

The other role needed is the one of a regular participant. This role is adopted by all agents actively participating in a negotiation, that is by service providers and consumers. All those agents have to offer some methods to enable negotiations because of the already mentioned service-orientation of this approach. Regular participants for example have to offer a method for updating negotiation instance data, to be called by the coordinator agent. The two roles identified will be described later in more detail.

There are different possible scenarios for the exchange of negotiation metadata that are desirable to be supported by this protocol. Although only providing two distinct roles the

exchange protocol provides a broad support for different exchange processes. A multitude of possible protocols can be constructed from only two basic logical protocol components that can be implemented using the roles and presented methods: request for and proposal of negotiation data.

A request for negotiation data represents a simple request-response protocol. Agent A requests the different negotiations supported from agent B by invoking some method on agent B. As a result A receives possibly some datastructures describing negotiation types or instances (this will be distinguished in more detail later). If one of these negotiation instances suites agent A's needs it will join this negotiation by invoking the corresponding method on the service provided by agent B. After that, agent B distributes the updated negotiation instance document to all participants of the negotiation already known to inform them about the newly joined agent A. This already shows agent B's role as a coordinator of the negotiation. If another agent C would join later, agent A would be informed about this participation analogously because now it would be one of the already known participants. This shows that agent A also has to provide some standardised methods (at least the one to update the negotiation instance status by the coordinator).

The second basic protocol component, the proposal of negotiation data, depicts the opposite scenario. Agent A creates a negotiation instance document and posts it to other agents. These other agents then decide whether to join this negotiation or not. Agent B for example decides to join, whereas agent C decides not to. In order to notify agent A of its decision regarding the negotiation, agent B has to invoke the corresponding method to accept the given negotiation proposal (when an agent does not want to join it simply omits this step, no explicit reject-operation is needed). In this scenario agent A adopts the coordinator role, because the other agents join or reject a negotiation by using the methods it offers. Note: It would also be possible to propose a negotiation to another agent stating this agent to be the coordinator for the further protocols flow. Then the second agent would have to be set as coordinator in the negotiation instance document. The respective methods for this proposal process will be presented later.

When the exchange process is mediated by some third party a slightly different and more complex protocol is used. This protocol, however can be constructed using the two basic protocol components described above (though not using the exact same methods, which will be described later). For example, agent A wants to offer some negotiation about a service it exposes. Furthermore it does not want to manage the admission of the different other participants and decides to delegate this task to some third party agent X (which would have to implement the *Negotiation Coordinator* interface). This agent then would conduct the coordinating tasks related to this negotiation without taking part in it.

Clearly in this scenario agent X adopts the role of *Negotiation Coordinator*, again responsible for admission and distribution of negotiation data. That is why agent X is set in the negotiation data structure to be coordinator, and not agent A. This way all the other agents requesting the instantiated negotiations from agent X can clearly see, that the negotiation will take place between the new participants and agent A though it is not the coordinator. Subsequently agent X can actively propose the negotiation data to already known agents (like in the proposal of negotiation data case) or passively allow other agents to query the offered negotiation (like in the request of negotiation data case) analogously to the paragraph above.

This approach allows for centralised admission handling and provides the WS-Agreement

infrastructure with a means to define logical negotiation look-up servers. As already sketched when introducing the *Negotiation Coordinator* role another useful effect of these centralised coordination services is that they provide a means for centralised security or reputation handling. Since all admission decisions will be made by this coordinators external systems regarding security or reputation concepts can easily be incorporated into this framework by pluging them into these coordinating services. Also not every agent wanting to propose a negotiation has to have access to these external systems and can delegate the verification of agents' credentials to centralised coordinators.

All of these advantages still apply when a service provider or consumer also acts as a *Negotiation Coordinator*, however this concept is of much more use, when applied to form some sort of infrastructure of *Negotiation Coordinator* servers. This can also incorporate replication concepts to assure reliability and scaling over many of such servers.

5.3 Architecture of Negotiation Objects

In this subsection the involved negotiation objects, or data structures, will be presented. Also the relationships between these data objects are detailed to describe the general architecture of documents involved in a negotiation.

WS-Agreement Template with Creation Constraints The main negotiation object is the WS-Agreement template with its corresponding creation constraints as defined in the current WS-Agreement specification. Since this thesis augments the current specification with possibilities to negotiate over a WS-Agreement this fundamental data structure is adopted for the (partial) definition of some service(s) to be negotiated. The creation constraints as part of this template are also used in this approach to give syntactical restrictions on the elements still to be filled out or altered during the negotiation.

The following parts of a WS-Agreement template represent the attributes of an agreement that can be negotiated: the *context elements*, the *service description terms*, the *service property terms* and the *guarantee terms*.

These parts intuitively describe the properties of a service guarantee negotiated over. Elements of the WS-Agreement *context* describe general information about the agreement, such as expiration time or involved agents. This element also exposes the possibility to be expanded as different scenarios might require different context information. As can be seen with the expiration time element such data is a crucial part of the agreement and should therefore be subject of the negotiation.

Service description terms describe a service in terms of functionality which inherently should be part of the negotiation. This is especially important for services that do not yet exist and are to be instantiated after the negotiation, a concept already existing in the WS-Agreement specification [4].

Service property terms represent exposed features of a service used for guarantee assertions. These terms are important for negotiations, if one of the agents wants to expand the set of *guarantee terms* and needs additional service features used in guarantee assertions.

Service reference terms are not part of the negotiated items because, if set in the template,

they refer to some already existing service and therefore cannot be altered or negotiated. If not set in a template they will be set in the offers posted by the service provider(s).

Guarantee terms denote the most important part of every SLA. Because they explicitly describe the specified guarantees in an agreement they represent the main focus of the negotiation and should therefore be part of the negotiatable attributes of a service.

Each of the items identified above exposes an id unique within the WS-Agreement template. Since every template also exposes a unique id, within the one service providing it, every negotiatable term can uniquely be identified and referenced given the templateID, the termID, and if necessary, the EPR for the service offering this template. Hence in the negotiation instance document every negotiatable element can be unambiguously referenced.

Negotiation Type Document The Negotiation Type document as described in the previous section refers to the WS-Agreement template the negotiation is defined upon for specifying the negotiated issues.

Given its content the Negotiation Type document defines which terms of a WS-Agreement can be negotiated and how to do so. Hence terms not mentioned in this document are not negotiatable. For all negotiatable terms the rules of negotiation are given in terms of negotiation parameters as identified in the previous section.

Negotiation Instance Document This document represents one particular negotiation instance of one particular negotiation type as described before. Therefore it contains a unique identifier, a reference to the negotiation's type, optional elements for start and termination of the negotiation (when time-based as already described) and an element containing all the agents that take part in the negotiation.

The XML schema of such a negotiation instance document is defined as follows:

Listing 2: Negotiation Instance XML-Schema

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <xsd:schema
4     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5     xmlns:wsagn="http://xml.netbeans.org/examples/negotiation"
6     targetNamespace="http://xml.netbeans.org/examples/
          negotiation"
7     xmlns="http://xml.netbeans.org/examples/negotiation"
8     elementFormDefault="qualified">
9
10    <xsd:include schemaLocation="NegotiationType.xsd"/>
11
12    <xsd:complexType name="NegotiationReferenceType">
13        <xsd:choice>
14            <xsd:element name="referencedNegotiationType">
15                <xsd:complexType>
16                    <xsd:sequence>
17                        <xsd:element name="endpoint" type="xsd
                              :anyType"/>
```

```
18                          <xsd:element name="negotiationTypeID"
                               type="xsd:string"/>
19                      </xsd:sequence>
20                  </xsd:complexType>
21              </xsd:element>
22              <xsd:element name="negotiationType" type="wsagn:
                   NegotiationType"/>
23              <!--here an explicit declaration of a negotiation
                   type can be inserted-->
24          </xsd:choice>
25      </xsd:complexType>
26
27      <xsd:complexType name="AgentType">
28          <xsd:sequence>
29              <xsd:element name="role" type="xsd:string"/>
30              <xsd:element name="agentEPR" type="xsd:anyType"/>
31          </xsd:sequence>
32      </xsd:complexType>
33
34      <xsd:complexType name="NegotiationInstanceType">
35              <xsd:sequence>
36                  <xsd:element name="negotiationID" type="xsd:
                       string"/>
37                  <xsd:element name="negotiationType" type="
                       wsagn:NegotiationReferenceType"/>
38                  <xsd:element name="start" type="xsd:dateTime"
                       minOccurs="0" maxOccurs="1"/>
39                  <xsd:element name="termination" type="xsd:
                       dateTime" minOccurs="0" maxOccurs="1"/>
40                  <!--start and termination are only specified
                       here when defined in terms of points in
                       time-->
41                  <xsd:element name="agent" type="wsagn:
                       AgentType" minOccurs="2" maxOccurs="
                       unbounded"/>
42              </xsd:sequence>
43          </xsd:complexType>
44
45      <xsd:element name="NegotiationInstance" type="
           NegotiationInstanceType"/>
46  </xsd:schema>
```

The *negotiationID*-element unambiguously identifies this negotiation instance. To allow a multitude of different identifiers its domain was set to "‘xsd:string"’. This way human-understandable names, as well as for example consecutive numbers can be used.

In the *negotiationType*-element the type of the respective negotiation is defined. This can be done in two ways: explicitly by defining the negotiation type within the negotiation instance document or implicitly by referencing a negotiation type. When specified explicitly the whole negotiation type datastructure, as described in the previous section, has to be inserted into this *negotiationType*-element. When implicit typing is used the *negotiationType*-element only consists of an EPR and a negotiation type identifier, so every agent interested in the concrete type can request it from the respective service providing it.

These two possibilities are incorporated in the schema document by definition of a choice between a *referencedNegotiationType*- and a *negotiationType*-element for implicit or explicit typing respectively. The assigned domains are chosen to suite the different purposes of the elements: a negotiation type identifier used in the *referencedNegotiationType*-element is expressed as a "‘xsd:string"’ value, whereas for an EPR the domain "‘anyType"’ is applied to allow arbitrary referencing formats; mostly WS-Adressing [52] will be used here. The *negotiationType*-element used for explicit typing on the other hand is of the type "‘wsagn:NegotiationType"’, as defined in the XML-Schema document describing negotiation types.

The *start*- and *termination*-elements are optional elements. They are set during instantiation if the negotiation's start and termination rules are expressed in terms of points in time. That is why for these elements the domain "‘xsd:dateTime"’ was set. If these negotiation parameters are expressed as assertions over other values than time points they are set in the negotiation type document as described above.

A negotiation type document does not reference any agents, all agents joining the negotiation are specified in the negotiation instance document using the *agent*-element. This element contains the agent's EPR as well as the role it adopts after joining. Each negotiation exposes at least four roles (service provider, service consumer, *Negotiation Coordinator* and *Information Service*) but only two agents must be involved as a minimum (a negotiation only exists when at least to parties are involved). That is only service provider and consumer must be adopted by different agents. The other two roles can be adopted by the same agent, perhaps also acting as service provider or consumer, or by two different agents, possibly not taking part as consumer or provider.

Each role (except the *Negotiation Coordinator* for its centralised nature) can be adopted by numberous agents, if desired. The number of participants can be restricted by the *maximumNumberOfAgents*-attribute of the *role*-element in the negotiation type document as already described, though.

These three documents mark the foundation of the negotiation data exchange process and the negotiation afterwards. First only a negotiation type is defined by one of the parties describing the general negotiation parameters. Based on this type document, a negotiation instance is specified representing the concrete negotiation process with *start* and *termination rules* as well as participating agents. Alternatively an agent can directly create an instance document and define its type within this data structure.

When new agents join the negotiation they do so by invoking the corresponding method on the coordinator service. The updated instance document is subsequently redistributed to all other participants in order to keep every agent's knowledge of the negotiation up to date. This is done by altering the negotiation instance document and invoking an *updateNegotiation()*-method on all participating agents with this updated document as a parameter. The different methods for each role will be presented in more detail subsequently.

During the negotiation the different negotiatable attributes of the service as defined in the WS-Agreement template are successively set and a concrete WS-Agreement is derived from its template. This negotiation process will be described in more detail in the next section.

5.4 Involved Roles and Methods

In this subsection the roles needed for the exchange protocol are presented along with the respective methods offered. Note: The exchange protocol and the negotiation protocol, which will be presented in the next section, both use the *Participant* role. However, they do not use exactly the same methods offered by this role. In this section the *Negotiation Participant's* methods will be described as far as they concern the exchange protocol. Those used in the negotiation protocol will be presented in the next section subsequently. Also methods, when referenced are mentioned without the required input parameters. Only if the parameters are of special interest within the context its complete signature will be presented. In case of overloaded methods (methods exposing the same name but different input parameters) the described facts refer to all methods with the corresponding name unless explicitly described otherwise.

The *Negotiation Coordinator* is only used within the exchange and the *InformationService* role only within the negotiation protocol. They are presented entirely in the corresponding sections.

5.4.1 Role: *Negotiation Coordinator*

The *Negotiation Coordinator* represents a logically centralised instance within a negotiation. This service distributes the negotiation data among the participants and manages their admission.

The *Negotiation Coordinator* role is completely independent of the two basic roles service provider and service consumer; each of these can act as *Negotiation Coordinator*. The *Negotiation Coordinator*, because of its centralised nature, if participating always represents the side of a negotiation only consisting of one agent. That is if an 1:n or n:1 configuration is given, the *Negotiation Coordinator* always stands for the side with one permitted agent, whereas in One-on-One Bargaining both sides could act as *Negotiation Coordinator*. This also gives a hint, why *Negotiation Coordinator* is defined to be independend from service consumer or service provider: only this way forward auctions, that is auctions with n buyers and one seller, are just as possible as reverse auctions, that is auctions with one buyer and n sellers. Of course the *Negotiation Coordinator* can also be an agent not involved in the negotiation at all, for example, some mediator finding and coordinating the negotiations' participants.

In the following the respective methods to be offered by the *Negotiation Coordinator* are presented.

Negotiation Coordinator methods:

- *getAllNegotiationTypes()*
  Parameter(s): none

  Return parameter(s): list of Negotiation Type documents

  Purpose: This method returns all availabe Negotiation Type documents that can be accessed from this service.

- *getAllNegotiationTypesForTemplate(EPR, templateID)*
  Parameter(s): EPR of the service offering the template; ID of the WS-Agreement template for which the possible types of negotiation should be queried.

  Return parameter(s): list of Negotiation Type documents

  Purpose: Analogous to *getAllNegotiationTypes()* this method queries all available Negotiation Type documents from a particular service. This method, however only returns the Negotiation Types applicable for the given WS-Agreement Template.

- *getCurrentNegotiations()*
  Parameter(s): none

  Return parameter(s): list of Negotiation Instance documents

  Purpose: With this method all already instantiated negotiations of the offering service can be requested.

- *getCurrentNegotiationsForTemplate(EPR, templateID)*
  Parameter(s): EPR of the service offering the template; ID of the WS-Agreement template for which the currently instantiated negotiations should be listed

  Return parameter(s): list of Negotiation Instance documents

  Purpose: This method returns all currently instantiated negotiations concerning the given template.

- *joinNegotiation(negotiationID, agentEPR, 'credentials')*
  Parameter(s): ID of the Negotiation Instance the invoking agent wants to join; the EPR of the joining agent; optional credentials needed for admission (security or reputation concepts)

  Return parameter(s): none

  Purpose: In order to join a negotiation instance an agents has to call this method on the coordinating service.

- *proposeNegotiation(NegotiationInstance-document)*
  Parameter(s): Negotiation Instance document representing the negotiation to be proposed

  Return parameter(s): none

  Purpose: This method is used to actively propose a negotiation instance to a coordinating agent. This agent can then either only act as coordinator in the respective negotiation (it is already stated to adopt this role in the instance document given) or also join the negotiation as participant by updating and redistributing the instance document.

- *publishNegotiation(NegotiationInstance-document)*
  Parameter(s): Negotiation Instance document representing the negotiation to be published

  Return parameter(s): acknowledgment, whether the publication was successful

  Purpose: This method can be used to publish negotiation instances at some coordinating service. This method differs from the *proposeNegotiation()*-method in that with the publication of a negotiation the coordinator used for publishing is not stated to act as *Negotiation Coordinator* of the respective negotiation. It only

offers this negotiation instance for look-up purposes while the actual admission and information (re)distribution tasks are conducted by the actual coordinating agent, propably the one publishing the negotiation instance. This method can be used to implement systems of distributed look-up servers as will be described later.

- *publishNegotiationToReceipients(NegotiationInstance-document, [receipients])*
  Parameter(s): Negotiation Instance document representing the negotiation to be published; list of agents the negotiation should be published to

  Return parameter(s): acknowledgment, whether the publication was successful

  Purpose: The *publishNegotiationToReceipients()*-method is used analogously to the general *publishNegotiation()*-method, except that the agents that should be actively notified of this negotiation are explicitly named. In the more generic method the publishing agent cannot specify to which agents the negotiation should be published or whether this negotiation should be published actively with the *proposeNegotiation()*-method or just be offered via the query-methods presented above.

5.4.2 Role: *Negotiation Participant*

The *Negotiation Participant* role defines the methods needed by every regular participant of a negotiation. Common *Negotiation Participants* can act as service providers or consumers. Each agent representing one of these roles within a WS-Agreement negotiation has to adopt the *Negotiation Participant* role and implement the respective methods presented below.

Negotiation Participant methods:

- *proposeNegotiation(NegotiationInstance-document)*
  Parameter(s): Negotiation Instance document representing the negotiation to be proposed

  Return parameter(s): none

  Purpose: As describe above this method is used to actively propose a negotiation instances to a potential *Negotiation Participant.* Note: As opposed to the corresponding method of the *Negotiation Coordinator* interface this method proposes a negotiation to agents to act as regular participants in the consequent negotiation.

- *updateNegotiation(NegotiationInstance-document)*
  Parameter(s): Negotiation Instance document representing the up-to-date negotiation information

  Return parameter(s): none

  Purpose: This method is called by the *Negotiation Coordinator*, when new agents have joined the negotiation and the updated instance document has to be promoted to all *Negotiation Participants*.

- *acceptNegotiation(negotiationID)*
  Parameter(s): identifier of the negotiation instance that was accepted

  Return parameter(s): none

Purpose: The *proposeNegotiation()*-method is offered to support asynchronous communication. When a negotiation is proposed to a participant this agent can decide whether to join or not. If it joins the *joinNegotiation()*-method is invoked, if not nothing more happens.

If the *Negotiation Coordinator* is proposed a negotiation this agent is supposed to act as coordinator during the respective negotiation. The participant proposing this negotiation instance has to know whether it was accepted or not, but the *joinNegotiation()*-method is either semantically not very suitable, because it doesn't join but it coordinates the following negotiation, as well as it is not present in the *Negotiation Participant* interface.

Since again asynchronous communication was considered useful here (this concept allows a coordinating agent to check the resources first before answering this request) this method was introduced. By invoking the *acceptNegotiation()*-method a coordinator accepts the negotiation instance formerly proposed to him from the respective participant.

This method is especially important during the instantiation process of a negotiation. For example, if some coordinating agent A offers several Negotiation Types and agent B queries these and wants to start a negotiation of such a type with agent A as coordinator. In this situation agent B would propose a newly created negotiation instance of some supported type to agent A. Agent A would act as coordinator in this negotiation and therefore has to notify agent B of its decision on joining. If no answer is posted to agent B yet agent A is either still deciding or decided not to coordinate this negotiation. If the *acceptNegotiation()*-method is invoked by agent A the negotiation instance moves from the pending state it was in to the accepted state and is now available for other potential participants.

5.5 Illustration of Sample Protocol Components

As already depicted above the three basic protocol scenarios (request for, proposal of negotiation data and mediated configuration) form the basic components for the exchange protocol. More sophisticated protocols can be constructed by combining these basic elements. Since the methods provided by each role have been introduced in the last subsection these protocol components can be explained in more detail now.

In the following each of the three scenarios will be described by providing UML Sequence Diagrams of the respective processes.

5.5.1 Request for Negotiation Data

The first basic protocol component describes the process of one agent requesting negotiation data from another agent. The methods described above provide two different types of negotiation data to be requested: negotiation type or negotiation instance information. The corresponding request-methods are defined in the *Negotiation Coordinator* interface for the coordinating agent stores and (re)distributes this information.

For each type of query there exist two different methods, one for querying all possible negotiation types or instances respectively and one for querying possible negotiation types

and instances for a particular service agreement template. These two distinctions lead to the four methods presented below:

- *getAllNegotiationTypes()*
- *getAllNegotiationTypesForTemplate(EPR, templateID)*
- *getCurrentNegotiations()*
- *getCurrentNegotiationsForTemplate(EPR, templateID)*

The distinction between types and instances allows agents to request actually instantiated negotiations, that are already running or that are about to start, as well as supported negotiation types in general. After requesting general types an agent can propose a concrete instance of this type to the coordinator agent in order to trigger the instantiation of a new negotiation.

Obviously there is not only one distinct way to request negotiation data from a coordinator agent. Hence in the following some sample processes depicting possible request-scenarios will be sketched with UML Sequence Diagrams [55].

The first possible request would be an agent just querying all negotiation types supported by the respective *Negotiation Coordinator* as described in the following diagram:

Figure 6: Process of requesting all negotiation types available

The agent described as "‘participant"’ here requests the possible negotiation types by invoking the method *getAllNegotiationTypes()* on the coordinator service. As a result the coordinator returns a list of negotiation type documents. This allows the *Negotiation Coordinator* to generally define the supported negotiation protocols without currently instantiating one particular negotiation. The coordinator agent can thus wait until other agents have requested these available types and proposed a particular type to be instantiated.

Analogously to requesting all possible types agents can also query all possible negotiaotion types for a particular WS-Agreement template by invoking the *getAllNegotiationTypesForTemplate()*-method. This method exposes the same signature as the general request-method mentioned above except for two additional input parameters referencing the template for which the negotiation types should be returned.

As already described, this template represents a generic WS-Agreement, with references to (possibly already existing) services, elements yet to be filled in and corresponding restrictions. Hence this method provides agents with a means to inspect all possible negotiation types available for some service they already know.

On the other hand agents can query already instantiated negotiations with the *getCurrentNegotiations()*. As described earlier a negotiation instance is created from a certain type by specifying the involved agents and optionally the start and termination rules of the negotiation. Hence a negotiation instance can be already running when queried. The request process for already instantiated negotiations is described in the following diagram.

Figure 7: Process of requesting all instantiated negotiations

The requesting agent invokes the *getCurrentNegotiations()*-method on the coordinating service. As a result the *Negotiation Coordinator* returns a list of negotiation instance documents describing the currently available negotiation instances.

Analogously instantiated negotiations for a particular template can be queried by invoking the *getCurrentNegotiationsForTemplate()*-method.

If only the possible negotiation types or current negotiations are to be queried, the exchange protocol only consists of one method invocation and respective return parameter as just described. If one of the returned negotiation types or instances is appealing for the requesting agent and it wishes to take part in the respective negotiation the involved agents have to conduct an additional step.

In case of the request for negotiation types an agent can identify a negotiation type it wishes to instantiate and propose the created instance to the coordinator by invoking the *proposeNegotiation()*-method. This process is described in the following diagram:

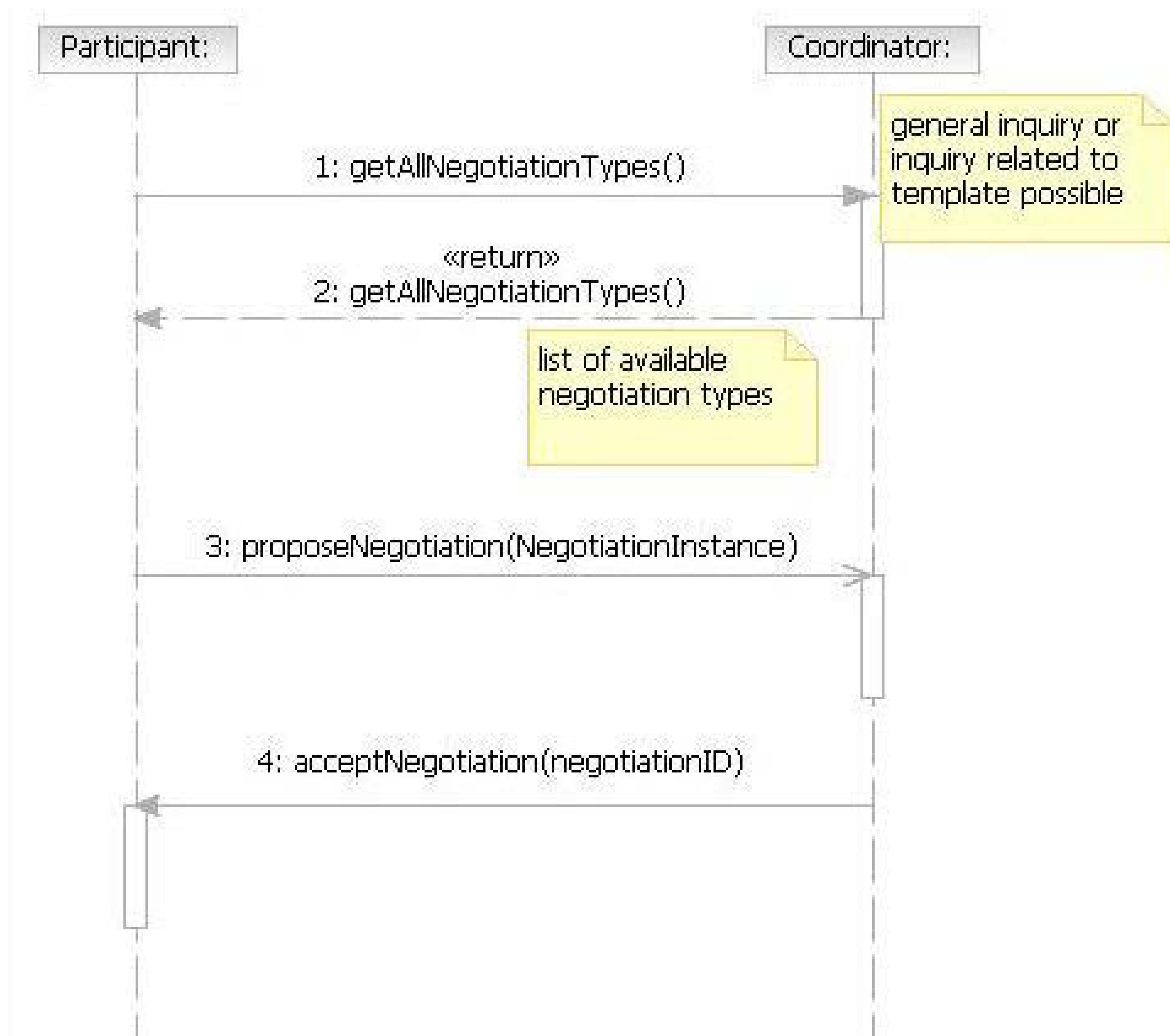

Figure 8: Process of requesting available negotiation types and instantiating a negotiation

In such cases the coordinating agent would be set as *Negotiation Coordinator* within the instance document. After such a negotiation is proposed, the coordinator would check it's resource situation for deciding whether to accept such a negotiation or not. The diagram shown above depicts the situation where the *Negotiation Coordinator* accepts the negotiation instance. This is done by invoking the *acceptNegotiation()*-method on the proposing agent. The proposing agent can join this negotiation instance by invoking the *joinNegotiation()*-method afterwards.

If an agent wants to join an already instantiated negotiation (queried before by invoking the corresponding method as described above) the *proposeNegotiation()*-step is omitted. The agent first requests the currently available negotiation instances, chooses an appropriate one and invokes the *joinNegotiation()*-method on the coordinator.

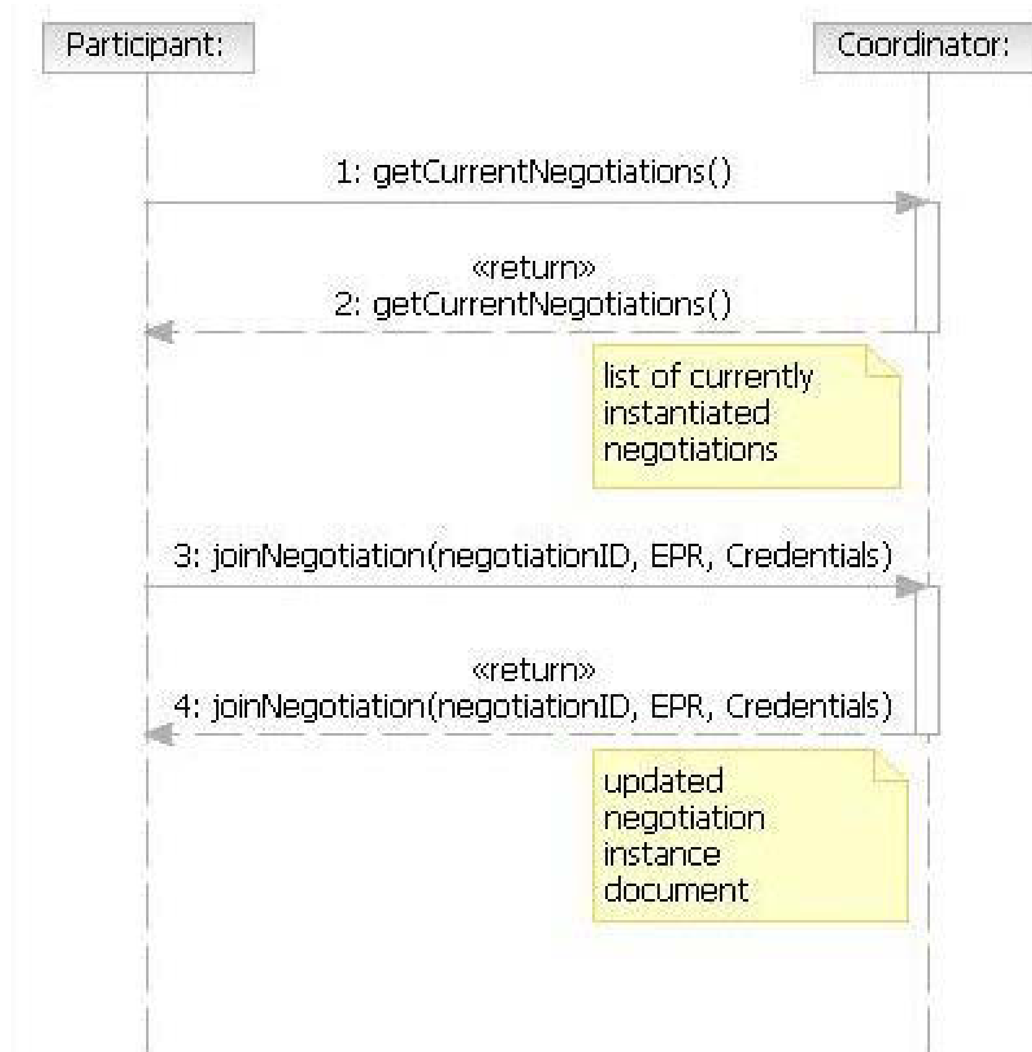

Figure 9: Process of requesting current negotiations and joining of the respective agent

Again the agent can also join negotiation instances that have been queried for some particular template before.

5.5.2 Proposal of Negotiation Data

The second basic protocol component is the proposal of a negotiation instance to other agents. This process was already incorporated in one of the described request protocol components. In order to actively propose a negotiation to other agents an agent creates a negotiation instance document and then offers it to several other agents it wants to get to join this negotiation.

In order to offer a negotiation instance to another agent the *proposeNegotiation()*-method is invoked.

After receiving a negotiation instance document the receiving agent decides whether to

join this negotiation. If this negotiation is appealing and the agent decides to join, it invokes the *joinNegotiation()*-method subsequently. The following diagram shows this process of proposal and subsequent joining of the other agent:

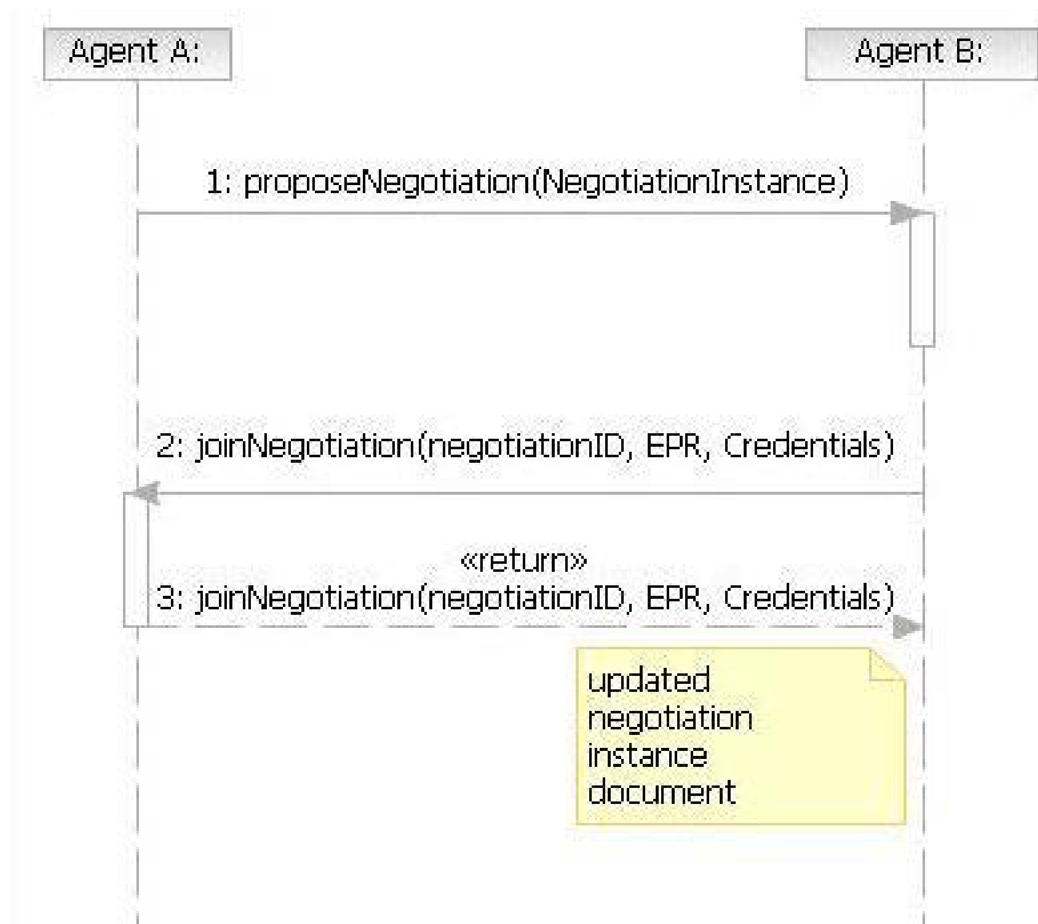

Figure 10: Process of proposing a Negotiation Instance and joining of the other agent

5.5.3 Mediated Configuration

The two basic protocol components just explained, together with one additional method offered by the *Negotiation Coordinator* role allow for definition of a commonly used pattern in distributed software systems: a configuration of software systems coordinated by some centralised entity, a mediator.

A *Negotiation Coordinator* does not have to join the negotiation as a service provider or consumer, but can act as a third party only responsible for administrative tasks, as already depicted. This concept enables the specification of centralised look-up servers only distributing negotiation data without taking part in any of these negotiations. These coordinators hence act as mediating third parties within the exchange protocol.

To enable publish/subscribe-like functionality agents should be able to publish negotiation instances to such look-up servers to promote their desired negotiation protocol. On the other hand agents requesting available protocols should be able to query these submitted protocols and search for appropriate ones.

In order to implement such architectures the *publishNegotiation()*-method was introduced.

This method allows for publication of instantiated negotiations at some *Negotiation Coordinator* service. The other agents requesting the available protocols again query these by invoking the already introduced request-methods. As described before, the *Negotiation Coordinator* can also actively suggest negotiation instances to other agents using the *proposeNegotiation()*-method ,which is therefore also present in the *Negotiation Participant* role.

The following diagram shows an exchange process with an agent A publishing a negotiation instance to the coordinator that proposes this negotiation to two different agents B and C of which only agent B joins. This is of course just a fragement of the complete exchange process because certainly the coordinator would propose this negotiation much more other agents and also other agents requesting this document by using request-methods could join respectively.

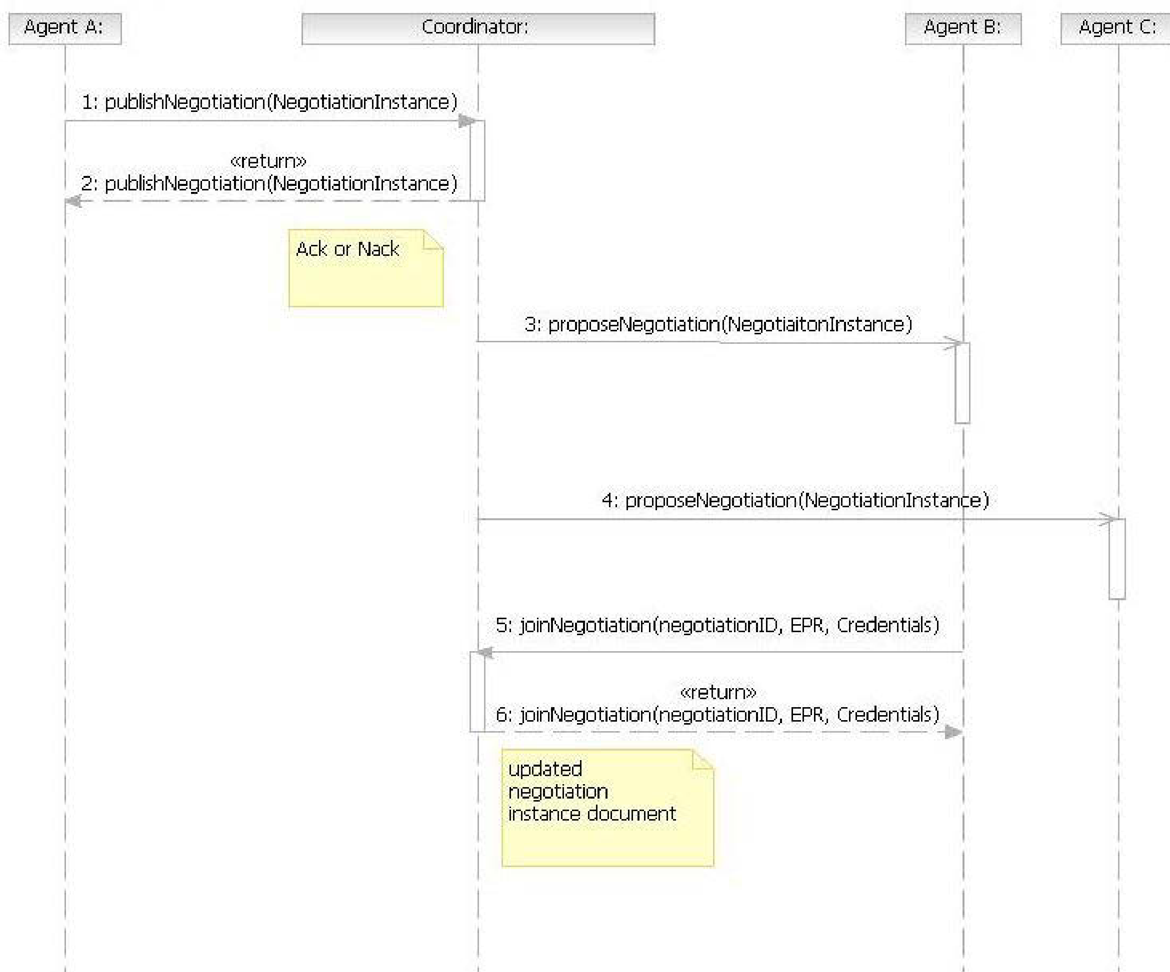

Figure 11: Mediated Exchange Process

Note: If the proposed negotiation would only allow one more additional agent, for example with One-on-One Baragining the process described in this diagram would of course represent the complete exchange protocol. In this case only one other agent could join which has already happend with agent B and hence no other agents could participate in the negotiation and the coordinator would not propose this document to other agents.

The syntanctical difference between the *proposeNegotiation()*- and the *publishNegotiation()*-method is that the first is used via an asynchrouous method call and the latter explicitly returns an acknowledgement or a rejection concerning the publication. Apart

from that the coordinating agent is set as *Negotiation Coordinator* in instances proposed to it while in such documents just published this is not the case (here the publishing agent would act as *Negotiation Coordinator* most likely).

Apart from that, these two methods provide quite similar functionality, they both suggest a negotiation instance document to another agent. The reason why not one of these methods is omitted for simplicity reasons and both scenarios are conducted using the same remaining method is that when publishing a negotiation the publishing agent has to know whether the coordinator accepts this request. If an asychronous method (like *proposeNegotiation()*) was applied here the only notification the publishing agent would get about the negotiation's status is the update-messages when new agents join this negotiation, as will be described subsequently. If the coordinator didn't accept the proposed negotiation (which would have its EPR set as *Negotiation Coordinator* in order to differ from a negotiation that is proposed to it so it would join as a regular participant) the publishing agent would not be aware of that. On the other hand a synchronous method would not be applicable for proposition of negotiation instances to potential participants. These have to be able to assess the proposed negotiation instance which perhaps includes other requests, for example for querying the referenced negotiation type, which would take some time. In order to incoporate these two distinct scenarios with their different requirements, although exposing similar functionality, the two different methods *proposeNegotiation()* and *publishNegotiation()* were introduced for the *Negotiation Coordinator* interface.

5.5.4 Update Process of Negotiation Instances

Whenever a new agent joins an existing negotiation instance the *Negotiation Coordinator* updates the respective instance document and redistributes it to all other participanting agents known so far. Thus the participant interface exhibits an *updateNegotiation()*-method.

The following diagram shows a scenario during which agent A and agent B have already joined a negotiation as service consumers, agent C as service provider and *Negotiation Coordinator*. Then agent D joins the negotiation by invoking the respective methods on the service exposed by agent C, for it is the coordinating agent. After that all already known agents (namely agents A, B, and C) have to be informed of the new participant. Since agent C already acts as coordinator, and therefore already knows the new participant, it only has to invoke the *updateNegotiation()*-method on agents A and B.

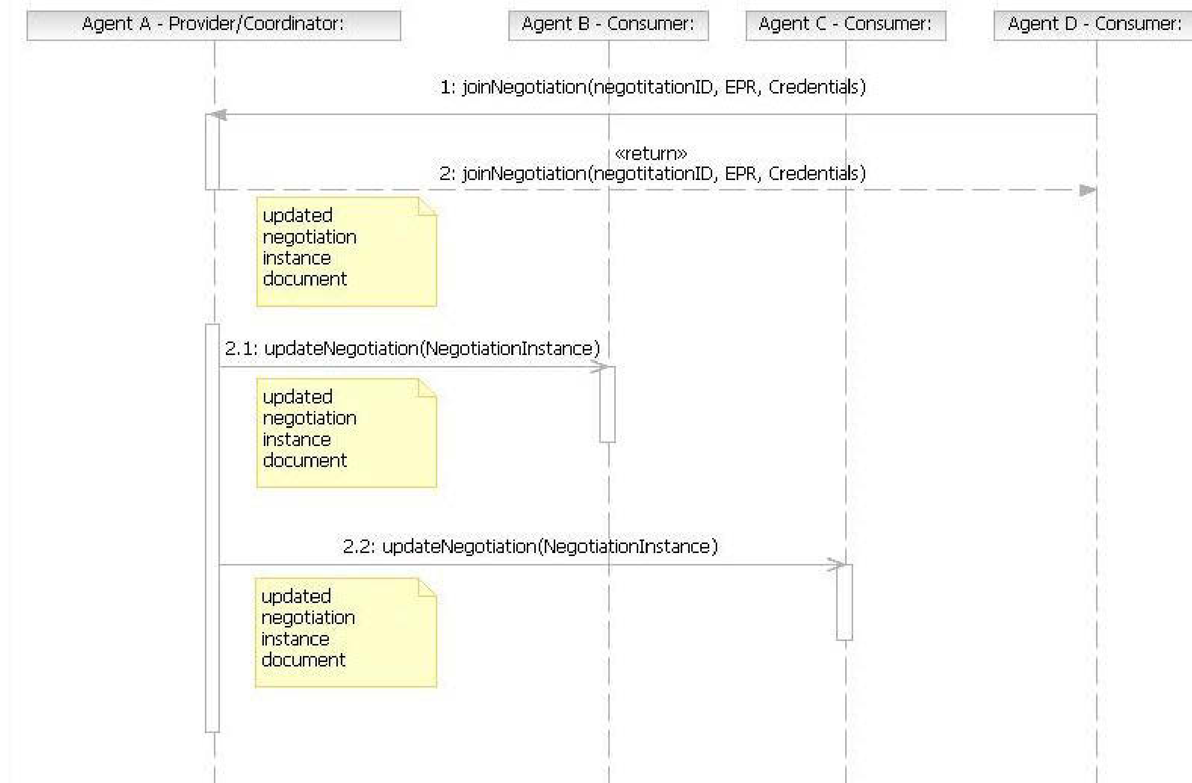

Figure 12: Process of updating a Negotiation Instance and redistributing it

5.6 Look-up Server Infrastructure

This subsection will shortly describe how infrastructures of distributed look-up servers providing coordinator functionality can be created with the roles and methods presented so far.

A *Negotiation Coordinator* does not have to be involved in the actual negotiation as a participant posting offers, but only as a independent third party. If so, this coordinating agent only handles the admission of agents as well as the (re)distribution of the negotiation data as already described without actually posting offers. Service providers or consumers wanting to propose some negotiation publish it at one of these coordinating agents. These *Negotiation Coordinator*s in turn offer this negotiation to other potential participants with the means presented above, although they do not necessarily adopt the *Negotiation Coordinator* role whithin the individual negotiation instances.

With this concept of independent coordinators it is possible to implement an architecture of distributed *Negotiation Coordinators* only concerned with admission and information distribution tasks, that are related to each other to achieve desireable performance properties, like robustness, availability or maximum response-time values. In order to set up such a high-performance distributed look-up system the involved coordinators have to deal with some of the most fundamental problems in distributed software systems: redundancy of data in order to increase availability and decrease response-time, synchronization and consistency issues.

The published negotiation instance documents have to be replicated on different individual look-up servers to guarantee redundancy and availability even if some of these servers are

down. Especially when a negotiation type defines some maximum number of agents for example it is very crucial to synchronize the passed messages between these look-up servers in order to get consistent copies of this negotiation instance documents on each server. Fundamental concepts used in distributed systems to keep distributed copies of some data entity consistent can and have to be applied here analogously [56].

To ensure such criteria like robustness or availability and consistency, the individual look-up servers have to exchange messages for synchronization and information distribution purposes. These message definitions and potential protocols, including the just mentioned distributed algorithms for consistent data for example, are outside of the scope of this thesis. They will probably be subject to future work, though.

6 Negotiation Protocol

In this section a generic negotiation protocol will be presented. Using this protocol the different negotiation types that can be specified with the presented data structure can be executed. Since this protocol again will take place in service-oriented environments it will be defined in terms of roles and respective methods analogously to the exchange protocol presented in the last section.

First the abstract process of a negotiation is described. After presenting the involved roles and respective methods used in this negotiation protocol some sample negotiation processes will be illustrated.

6.1 Abstract Negotiation Process

In general every negotiation can be described as a bidding process. Each party involved in a negotiation offers an agreement to the other party, concerning the issues subject to the negotiation, that is currently acceptable for them. Then the other party assesses the offered agreement and generates a counter-offer, accepts the offer of rejects it and terminates the negotiation. This way the two parties involved move from a conflict situation concerning some (logical) resource(s) to a consensus represented by the resulting agreement. Since SLA scenarios only exhibit two logical positions actively involved in a negotiation, the service providers and consumers, this thesis only considers such two-sided negotiation protocols.

In One-on-One Bargaining each side consist of only one agent. These two agents take turns in posting offers and counter-offers until one of these posted bids is accepted by the other agent. On the other hand negotiation protocols, such as auctions let all agents involved on one side post offers to the one agent representing the other side. Each agent can offer more than just one bid, but every agent can have only one currently valid bid. Hence, when posting a new offer this offer replaces the last one posted by the same agent. When the negotiation is terminated the currently best valid bid will be transfered into the resulting agreement.

The presentedd data structure allows for definition of a multitude of auction- and bargaining-like negotiation types. In order to support such processes the generic negotiation protocol introduced now has to provide the agents with means to post offers and to promote the decision made about a concrete offer. Since this framework will be applied in service-oriented environments this is done by defining respective methods to be exposed by the involved agents.

In the following these methods will be introduced. They are assigned to the two roles involved in the actual negotiation protocol: *Negotiation Participant* and *Information Service*.

6.2 Involved Roles and Methods

In this subsection the roles and respective methods used during an actual negotiation process will be described. This will augment the *Negotiation Participant* role, presented in the last section, with additional methods only used during the actual negotiation protocol. Furthermore one additional role will be introduced which will be used for the information processing during a negotiation process: the *Information Service*.

6.2.1 Role: *Negotiation Participant*

As already stated the methods presented here are only used during the actual negotiation.

Negotiation Participant methods:

- *placeOffer(agentEPR, WS-Agreement-document)*
  Parameter(s): EPR of the posting agent; complete WS-Agreement document representing the offered SLA

  Return parameter(s): none

  Purpose: As described above this method is used for posting offers within the actual negotiation protocol.

- *newRound(negotiationID, 'Information')*
  Parameter(s): ID of the negotiation instance for which a new round should be started; datastructure containing some information about the bidding process, that was not available in the last round (push-distribution), for example the current negotiation status etc.

  Return parameter(s): none

  Purpose: This method enables centralised coordination of multi-round negotiations. Each round is started when the *Negotiation Coordinator* invokes this method on all particpants (except the first one whose start is defined in the **Start**-attribute of the Negotiatoin Type/Instance). Multi-rounded negotiations consist of several phases, which are all conducted according to the same protocol. The only difference between the individual rounds is the knowledge of the involved agents. At the beginning of each round some negotiation information is revealed to the participants that was not available in the previous round. This way the involved agents can alter their offers according based on their increased knowledge about the negotiation. Negotiation designers could for example define a negotiation protocol, that allows only for one sealed bid from each participant in each round. After each round the bids from all other agents are promoted to all participants and every agent can post another offer in round two, and so on. In order to promote the newly accessible information the push-concept is applied. When invoking the *newRound()*-method the coordinator service posts this information to all agents along with the identifier of the negotiation the new round is started for.

- *acceptAgreement(negotiationID, agreementID)*
  Parameter(s): ID of the negotiation instance that was terminated; ID of the agreement offer that was accepted

Return parameter(s): none

Purpose: In order to promote a negotiation's outcome to all participants this and the next method are introduced: the winner(s) of the negotiation and therefore the agent(s) involved in the resulting agreement are notified by invoking this method. Since each agent could be involved in multiple negotiations the ID of the negotiation instance is given as a parameter along with the id of the accepted agreement offer.

- *rejectAgreement(negotiationID)*
  Parameter(s): ID of the negotiation instance that was terminated

  Return parameter(s): none

  Purpose: This method is in invoked after the termination of the negotiation analogously to the *acceptAgreement()*-method. However this method is invoked on all agents that did not win in the negotiation, that is on all agents not being involved in the resulting agreement.

6.2.2 Role: *Informationservice*

In order to access the current negotiation status or past offers of some negotiation instance the role *Information Service* is introduced. The *Information Service* interface defines methods used to access information about a particular negotiation process. Although this role must be adopted by (at least) one agent in a negotiation it is not strictly defined which agent has to do so. The intuitive way would be to also let the coordinating agent adopt this role but other configurations are explicitly allowed. Hence the methods used for coordinator and information processing tasks have been structured into two distinct interfaces to increase flexibility.

Analogously to the look-up infrastructure hinted in the previous section this concept allows for distributed systems of information servers. However, corresponding protocol and architecture specifications are again outside of the scope of this thesis.

Informationservice methods:

- *getStatus(negotiationID)*
  Parameter(s): ID of the negotiation nnstance, the status of which is queried

  Return parameter(s): datastructure containing the current negotiation status, which is denoted by all current offers of all parties allowed to post offers

  Purpose: This method is used by all negotiation participants to access the current negotiation status. This allows for example to assess which offer is currently winning the negotiation and if necessary to adopt the own offer.

  Note: The currently winning bid can of course only be identified if the offer matching rules of this negotiation are given in the negotiation type document, otherwise the requesting agent can not anticipate the current winner.

- *getPastOffers(negotiationID)*
  Parameter(s): ID of the negotiation instance, the past offers of which are queried

  Return parameter(s): datastructure containing all past offers of this negotiation

Purpose: This method lets participating agents access all past offers of a negotiation. This information can be used for internal decision making of the negotiating agents.

- *getPastOffers(negotiationID, agentID)*
  Parameter(s): ID of the negotiation instance, the past offers of which are queried; ID of the agent for which the past offers should be listed

  Return parameter(s): datastructure containing the past offers of some particular agent

  Purpose: Analogous to the general *getPastOffers()*-method presented before this method retrieves past offers of a negotiation, restricted however to offers posted by the specified agent.

6.3 Illustration of Sample Protocol Components

In this subsection the two abstract negotiation classes, 1:1 and 1:n negotiations, will be illustrated. For each class one particular example process will be described by presenting the sequence of method invocations used to conduct such a negotiation.

After presenting a simplified auction-like process involving a small number of participants for scope reasons, the special situation of One-on-One Bargaining will be described. Both scenarios will be supplemented by respective UML Sequence Diagrams. Finally the use of the *Information Service* within the two negotiation processes detailed before will be sketched.

6.3.1 1:n Negotiations

A very popular class of negotiations are those exhibiting a 1:n configuration. One participant offers some product to or wants to purchase something from n other agents, representing the other side of this negotiation. Regularly only these n agents being on the same side of a negotiation are allowed to post offers, whereas the one agent being on the other side choses the one winning offer according to some rules. This type of negotiation is called auction.

Analogously to the illustration of exchange processes as given in the last section the following diagram describes a sample auction process using the methods introduced above:

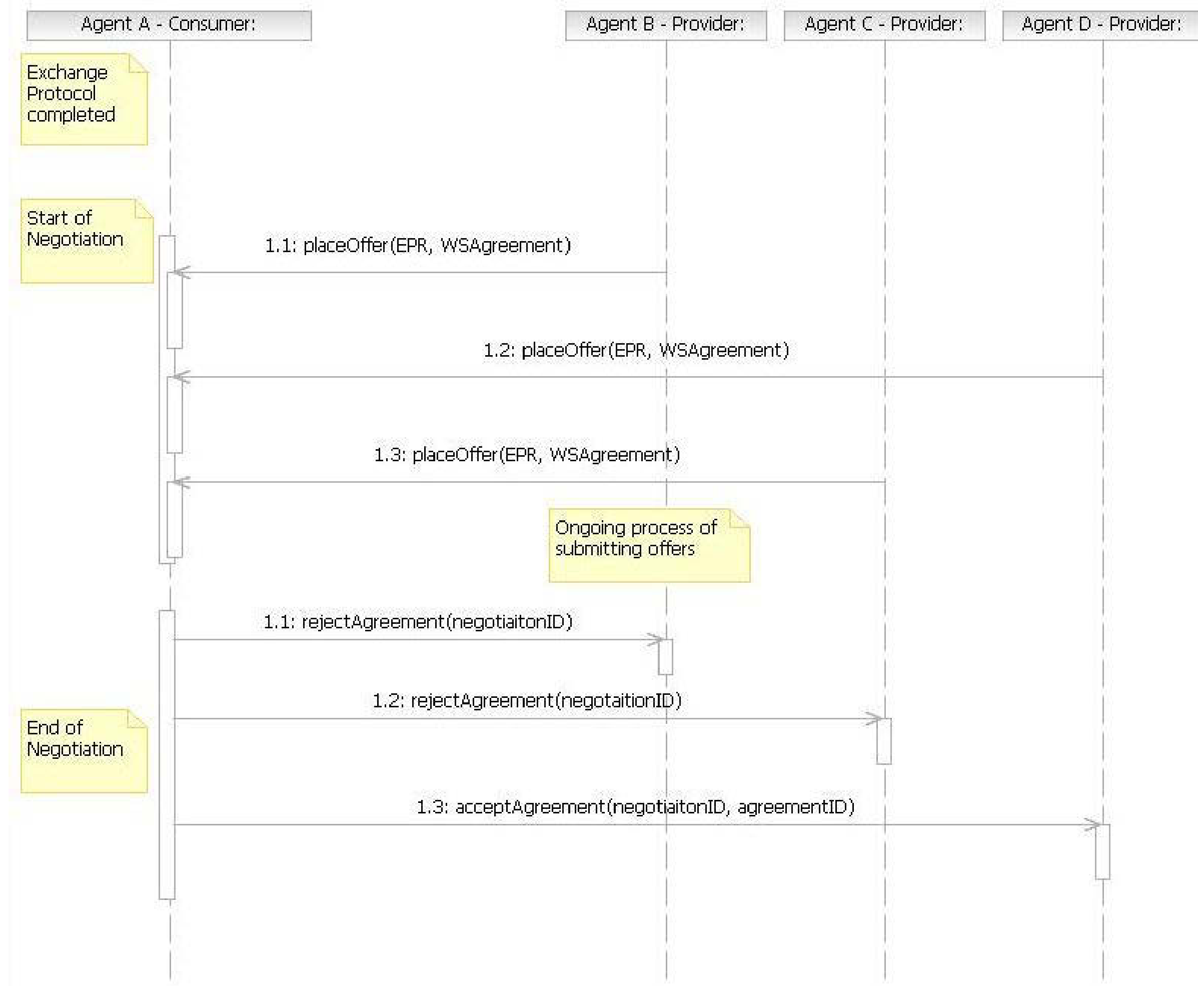

Figure 13: Auction Process

In this negotiation process agent A acts as service consumer requesting offers from different service providers. Agents B, C and D represent those service providers posting offers to agent A.

Initially the necessary negotiation data has to be distributed among the participants according to the exchange protocol defined in the last section. Given this phase of negotiaiton is already completed the actual negotiation starts as defined in the corresponding start-rule. After the negotiation started the bidding process takes place. Agents B, C and D subsequently post offers to agent A. This is depicted in the diagramm by explicitly showing the submission of offers by each of these agents. As also hinted in the diagram this bidding process will go on for some amount of time resulting in much more offer postings than actually shown in the diagram.

After the negotiation is terminated (the termination condition again stated in the negotiation type or instance document), its result is communicated to all participants.

In this case agent D offered the best agreement of all bidding agents and therefore wins this auction. Agent A subsequently promotes the result to all participants by invoking the *acceptAgreement()*- or *rejectAgreement()*-methods respectively as shown in the diagram.

As a result of this negotiation agents A and D engage in an agreement with each other, whereas agents B and C do not take part in an agreement because of loosing the negotiation.

Even though this negotiation process only shows a very simplified auction because of scope reasons it already sketches how even more complex negotiations can be conducted using the roles and respective methods defined above.

Note: Method invocations related to information queries have been omitted in this diagram for clarity reasons. A modification of this diagram incorporating such an information request will be presented in the subsections dealing with the *Information Service* usage.

6.3.2 1:1 Negotiations

The second class of negotiations to be suppported by this framework are 1:1 negotiations, or bargaining processes.

In such One-on-One Bargaining situations both sides are allowed to post offers. Each side submits an agreement offer representing the agreement currently acceptable. The other agent, after receiving the offered agreement, decides whether this agreement is acceptable for him. Subsequently it accepts the offer or generates a counter offer describing the agreement currently acceptable for it. This way both parties move from some conflict situation to an agreement by making concessions to reach a compromise.

The following diagram describes such a One-on-One Bargaining process with two agents (A and B) that negotiate over some issue, for example a SLA as assumed for this framework. Again every offer is submitted by invoking the *placeOffer()*-method. For clarity reasons only a few offer submissions are shown, an actual bargaining process however would regularly consist of much more proposed offers until the agreement is finally reached.

Again first the negotiation data has to be exchanged according to the exchange protocol defined in the previous section. For this negotiation process a negotiation type is assumed that specifies two sides of the negotiation, each consisting of only one agent. Both are allowed to post offers.

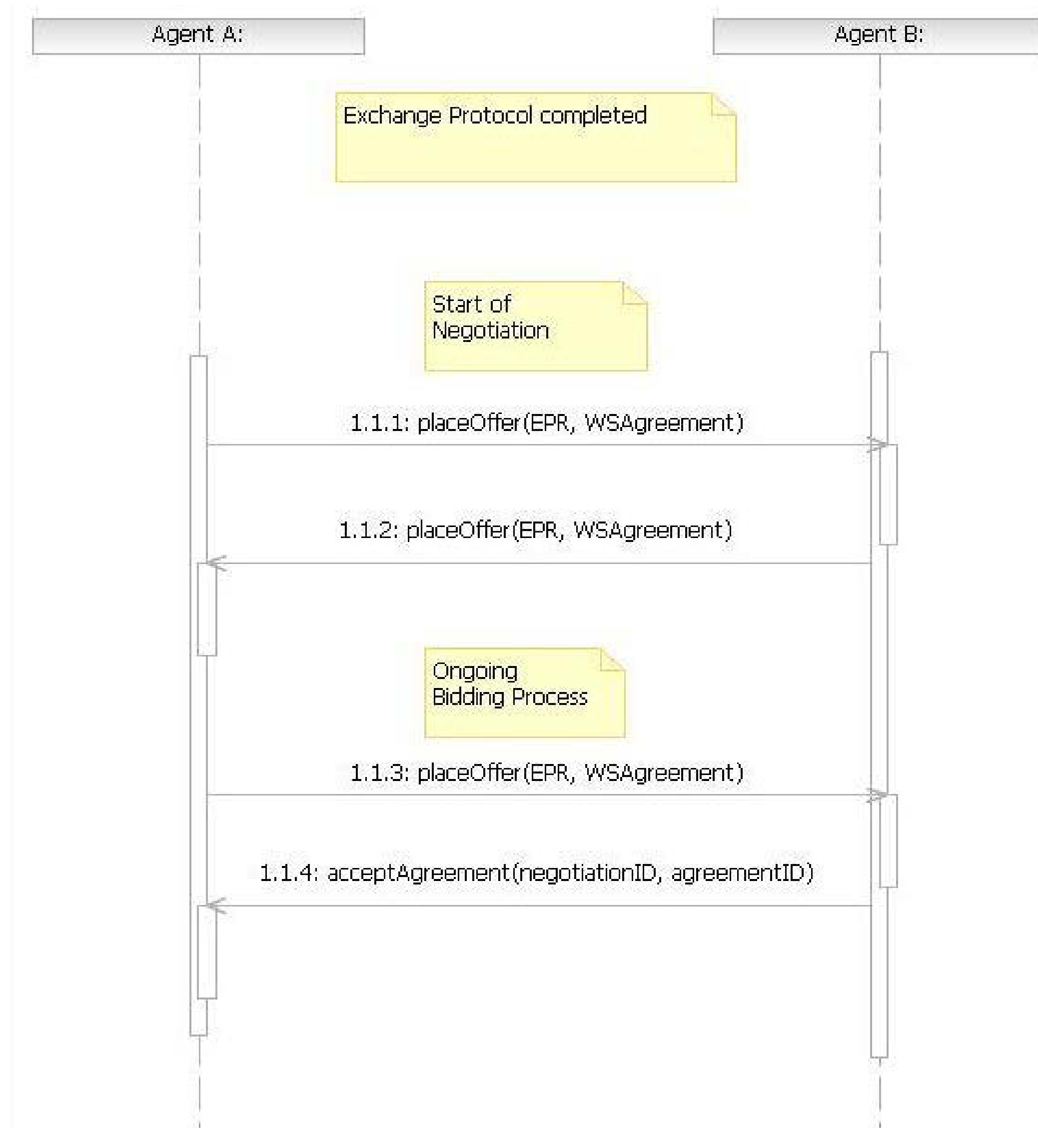

Figure 14: One-on-One Bargaining Process

In the negotiation shown in the diagram agent A starts with offering an agreement for which agent B creates a counter offer. After several rounds of offers and counter offers agent B proposed an agreement agent A could accept. Consequently agent A invoked the *acceptAgreement()*-method on agent B to communicate this decision.

Note: This of course assumes that invoking this method was set as a termination rule for this negotiation.

6.3.3 Use of the Information Service

Until now the sample processes have been described without detailing the information processing component. Every information about a negotiation like past offers or the current negotiation status has to be queried by invoking the corresponding methods offered by the *Information Service* role.

This way all agents, that are allowed to as specified in the negotiation type document, can access this information at any point in time.

In One-on-One Bargaining situations information concerning a negotiation is not as important as in auctions. The actual negotiation status, that is the currently offered agreement is always known by the other party it was offered to, because it was posted by invoking the corresponding method on exactly this agent. Thus only past offers can be meaningfully queried.

In an auction the current status is not known to every agent deserving this information. When one agent posts an offer only the one agent on the other side of the negotiation/auction is aware of that because it was posted by invoking one of the methods it exposes. All other agents have to request the current negotiation status to know what the other participants offered and, if necessary, post an improved offer exceeding the currently winning offer.

The following diagram represents the modified auction process as described above. However, in this version one information request was added. By retrieving the negotiation status from the *Information Service* (in this case also offered by agent A) agent B realizes that agent D posted an offer exceeding it's initial offer. If the negotiation would end at that point agent D would engage in an agreement with agent A and the other participants would loose the negotiation. As a reaction to this negotiation status agent B creates another, better offer and posts it to agent A to succeed the formerly winning offer. However, after an ongoing process of offer submission agent D still wins the negotiation in the process described in this diagram.

This way all agents can query the current negotiation status and, if necessary, post better offers as their response.

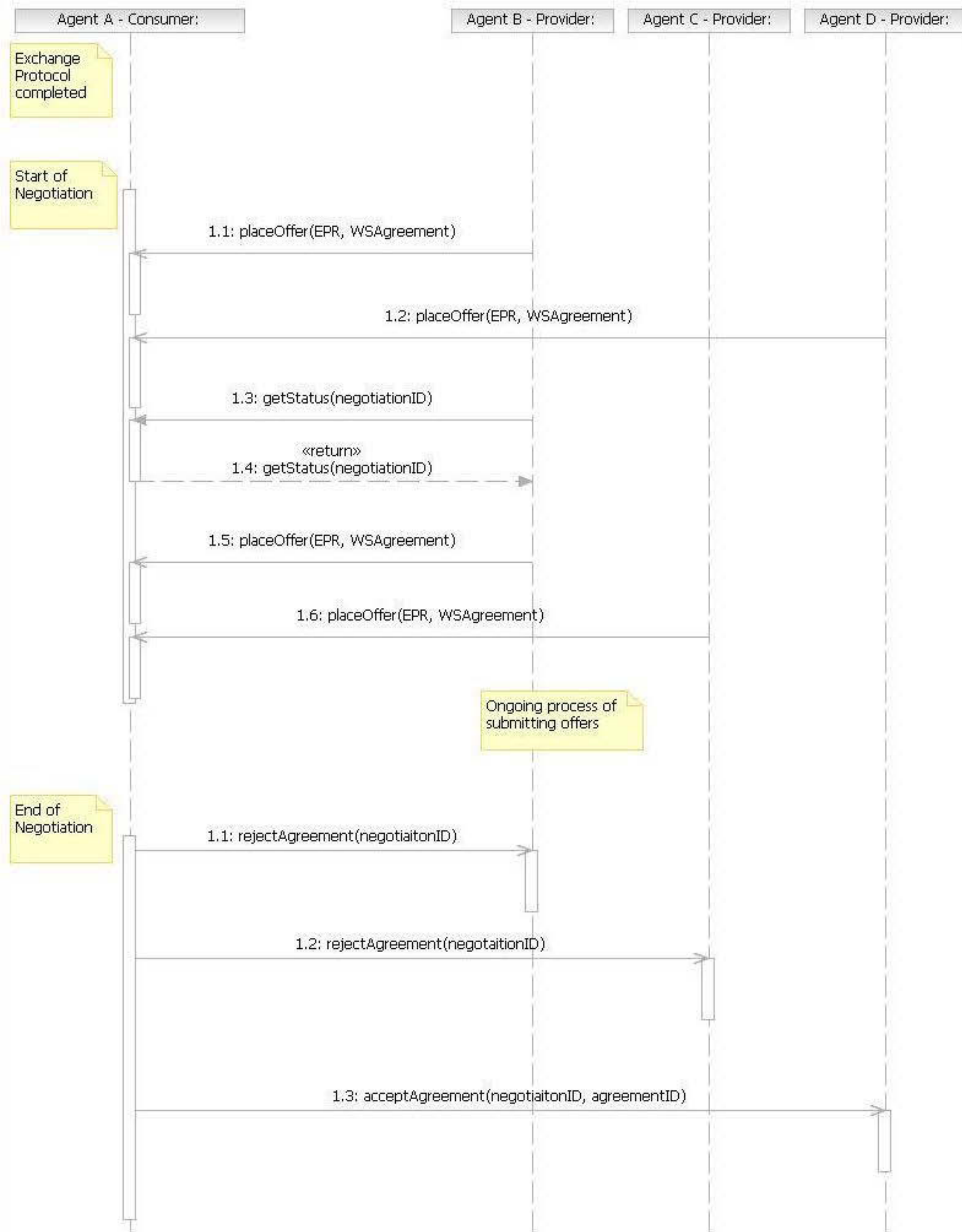

Figure 15: Auction Process with usage of the *Information Service*

Note: Which negotiation is accessible for which agents is defined in the information processing rules of the respective negotiation type.

7 Web Service Description Documents

In this section for each role identified before (*Negotiation Coordinator*, *Participant* and *Information Service*) the corresponding interface definition will be presented. These interfaces will be described using WSDL [1] documents.

Since all methods along with their input and output parameters for each role have already been described informally before, the following documents represent a WSDL formalization of the abstract interface descriptions given in sections 5 and 6.

Note: WSDL has been recently been proposed as Version 2.0 [1]. However most of current applications still use WSDL 1.1 [57] to describe Web Services. Also the tool support for WSDL 2.0 is not as comprehensive as for the former version. Since the wide-spread use of this version the following documents will be described in WSDL 1.1. For the WSDL 2.0 correspondents see Appendix A A. These 2.0 documents, however have not been machine-validated because of the lack of tool support, as opposed to the WSDL 1.1 documents presented in this section.

7.1 Fault Messages

Before discussing the individual interface descriptions a short overview on the possible fault messages used in these is given in this subsection.

In order to increase reusability only two different fault message types are defined: fault messages indicating invalid input parameters and those used for promoting denied access to the respective operation. All fault messages used in the following interface descriptions are of one of these two types.

Each of the two message types contains exactly one element describing the occurred fault in more detail. For the *InvalidInputMessage* this element is called *inputError* and its possible values are: *syntacticalError*, *inputParametersMissing* or *referencedEntityNotFound*. If some of the input parameters do not confirm to their required type *syntacticalError* and if some of them are missing *inputParametersMissing* is chosen. *referencedEntityNotFound* is set if one or more of the references set as an input cannot be resolved, which would be the case, for example, when defining a negotiationID in the input message that is not known by the respective agent.

Analogously the *AccessDeniedMessage* exposes an element called *accessError*. The possible values of this elemnet are defined to be two distinct string values: *coordinatorOnlyMethod* and *violationOfNegotiationRestriction*. When non-coordinator agents try to invoke a method that can only be called by the *Negotiation Coordinator* *coordinatorOnlyMethod* is specified, while *violationOfNegotiationRestriction* is chosen whenever the method invocation is not compliant to the restrictions defined in the negotiation type document.

Clearly not all operations defined in the following interfaces use all possible fault message variants. Each of the two types is set as a possible fault message for every operation possibly producing at least one variant of this fault message type. That is a particular operation might not use all three InvalidInputMessage variants, but exposes this general

message type as a possible fault message.

7.2 Negotiation Coordinator

The following WSDL document represents the formalised description of the formerly introduced *Negotiation Coordinator* interface:

Listing 3: WSDL 1.1: Negotiation Coordinator

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions
3     name="NegotiationCoordinator"
4     targetNamespace="http://www.mycomp.org/schemas/
        NegotiationCoordinator"
5     xmlns:tns="http://www.mycomp.org/schemas/
        NegotiationCoordinator"
6     xmlns:wsag="http://schemas.ggf.org/graap/2005/09/ws-
        agreement"
7     xmlns:ns="http://xml.netbeans.org/examples/negotiation"
8     xmlns="http://schemas.xmlsoap.org/wsdl/"
9     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
10    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
11    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
12
13    <!-- Type definitions -->
14    <types>
15        <xsd:schema>
16            <xsd:import namespace="http://schemas.ggf.org/
                graap/2005/09/ws-agreement" schemaLocation="
                agreement_types.xsd"/>
17            <xsd:import namespace="http://xml.netbeans.org/
                examples/negotiation" schemaLocation="
                NegotiationInstance.xsd"/>
18
19            <xsd:complexType name="TemplateType">
20                <xsd:sequence>
21                    <xsd:element name="endpoint" type="xsd:
                        anyType"/>
22                    <xsd:element name="templateID" type="xsd:
                        string"/>
23                </xsd:sequence>
24            </xsd:complexType>
25
26            <xsd:complexType name="NegotiationsType">
27                <xsd:sequence>
28                    <xsd:element name="negotiation" type="ns:
                        NegotiationType" minOccurs="0"
                        maxOccurs="unbounded"/>
29                </xsd:sequence>
30            </xsd:complexType>
31
32            <xsd:complexType name="NegotiationInstancesType">
33                <xsd:sequence>
34                    <xsd:element name="negotiation" type="ns:
                        NegotiationInstanceType" minOccurs="0"
```

```
                        maxOccurs="unbounded"/>
35                  </xsd:sequence>
36              </xsd:complexType>
37
38              <xsd:complexType name="CredentialsType">
39                  <xsd:sequence>
40                      <xsd:element name="credentials" type="xsd:
                        anyType" minOccurs="0" maxOccurs="1"/>
41                  </xsd:sequence>
42              </xsd:complexType>
43
44              <xsd:complexType name="AgentsType">
45                  <xsd:sequence>
46                      <xsd:element name="receivingAgent" type="
                        xsd:anyType" minOccurs="1" maxOccurs="
                        unbounded"/>
47                  </xsd:sequence>
48              </xsd:complexType>
49
50              <xsd:simpleType name="InvalidInputMessageType">
51                  <xsd:restriction base="xsd:string">
52                      <xsd:enumeration value="syntacticalError"
                        />
53                      <xsd:enumeration value="
                        inputParametersMissing"/>
54                      <xsd:enumeration value="
                        referencedEntityNotFound"/>
55                  </xsd:restriction>
56              </xsd:simpleType>
57
58              <xsd:simpleType name="AccessDeniedMessageType">
59                  <xsd:restriction base="xsd:string">
60                      <xsd:enumeration value="
                        coordinatorOnlyMethod"/>
61                      <xsd:enumeration value="
                        violationOfNegotiationRestriction"/>
62                  </xsd:restriction>
63              </xsd:simpleType>
64
65          </xsd:schema>
66      </types>
67
68      <!-- Message definitions -->
69      <message name ="GetAllNegotiationTypesRequest">
70      </message>
71
72      <message name ="GetAllNegotiationTypesForTemplateRequest">
73          <part name="template" type="tns:TemplateType"/>
74      </message>
75
76      <message name ="GetCurrentNegotiationsRequest">
77      </message>
78
79      <message name ="GetCurrentNegotiationsForTemplateRequest">
80          <part name="template" type="tns:TemplateType"/>
81      </message>
```

```
82
83     <message name ="JoinNegotiationRequest">
84         <part name="negotiationID" type="xsd:string"/>
85         <part name="agentEPR" type="xsd:anyType"/>
86         <part name="credentials" type="tns:CredentialsType"/>
87     </message>
88
89     <message name ="ProposeNegotiationRequest">
90         <part name="proposedNegotiation" type="ns:
               NegotiationInstanceType"/>
91     </message>
92
93     <message name ="PublishNegotiationRequest">
94         <part name="publishedNegotiation" type="ns:
               NegotiationInstanceType"/>
95     </message>
96
97     <message name="PublishNegotiationToReceipientsRequest">
98         <part name="publishedNegotiation" type="ns:
               NegotiationInstanceType"/>
99         <part name="receipients" type="tns:AgentsType"/>
100    </message>
101
102    <message name ="GetNegotiationTypesResponse">
103        <part name="negotiations" type="tns:NegotiationsType"
               />
104    </message>
105
106    <message name ="GetCurrentNegotiationsResponse">
107        <part name="negotiationInstances" type="tns:
               NegotiationInstancesType"/>
108    </message>
109
110    <message name ="JoinNegotiationResponse">
111        <part name="updatedNegotiationInstance" type="ns:
               NegotiationInstanceType"/>
112    </message>
113
114    <message name ="PublishNegotiationResponse">
115        <part name="publicationAcknowledged" type="xsd:boolean
               "/>
116    </message>
117
118    <message name="InvalidInputMessage">
119        <part name="inputError" type="InvalidInputMessageType"
               />
120    </message>
121
122    <message name="AccessDeniedMessage">
123        <part name="accessError" type="AccessDeniedType"/>
124    </message>
125
126    <!-- Port type definitions -->
127    <portType name="NegotiationCoordinatorPortType">
128        <operation name="GetAllNegotiationTypes">
```

```
129             <input message="tns:GetAllNegotiationTypesRequest"
                    />
130             <output message="tns:GetNegotiationTypesResponse"
                    />
131         </operation>
132 
133         <operation name="GetAllNegotiationTypesForTemplate">
134             <input message="tns:
                    GetAllNegotiationTypesForTemplateRequest"/>
135             <output message="tns:GetNegotiationTypesResponse"
                    />
136             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
137         </operation>
138 
139         <operation name="GetCurrentNegotiations">
140             <input message="tns:GetCurrentNegotiationsRequest"
                    />
141             <output message="tns:
                    GetCurrentNegotiationsResponse"/>
142         </operation>
143 
144         <operation name="GetCurrentNegotiationsForTemplate">
145             <input message="tns:
                    GetCurrentNegotiationsForTemplateRequest"/>
146             <output message="tns:
                    GetCurrentNegotiationsResponse"/>
147             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
148         </operation>
149 
150         <operation name="JoinNegotiation">
151             <input message="tns:JoinNegotiationRequest"/>
152             <output message="tns:JoinNegotiationResponse"/>
153             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
154             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
155         </operation>
156 
157         <operation name="ProposeNegotiation">
158             <input message="tns:ProposeNegotiationRequest"/>
159             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
160         </operation>
161 
162         <operation name="PublishNegotiationOperation">
163             <input message="tns:PublishNegotiationRequest"/>
164             <output message="tns:PublishNegotiationResponse"/>
165             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
166         </operation>
167 
168         <operation name="PublishNegotiationToReceipients">
169             <input message="tns:
                    PublishNegotiationToReceipientsRequest"/>
```

```
170             <output message="tns:PublishNegotiationResponse"/>
171             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
172         </operation>
173 
174     </portType>
175 </definitions>
```

All three WSDL descriptions omit the binding- and service-elements for these are only present if a particular Web Service is described and therefore referenced in the WSDL documents. Since the interface descriptions presented here only generally define the offered methods and message formats these elements are not considered.

Within the portType-element the different methods, or operations, are defined along with their input and, if needed, output or fault messages.

As already described in section 5 all coordinator methods exhibit the request-response pattern (synchronous communication), except for *proposeNegotiation*, which is defined to be asynchronous, and therefore does not specify an output message.

Analogously to these method descriptions all the offered operations are specified. For each input, output or fault message a corresponding type is defined containing the respective parameters to be transmitted. These parameters are defined using the part-element within the message definitions. Respectively, messages not transmitting any parameters only consist of the top-level element exposing the message's name.

For this particular interface description some additional types were introduced. The TemplateType is used for references to WS-Agreement templates, for example in the *getAllNegotiationTypesForTemplate()*-method. Therefore this type defines an element pointing to the service providing the template (by specifying its EPR) and an element containing the template's id. This way arbitrary templates can be referenced.

The remaining types introduced in this document only define (possibly empty) sets of elements, whose types are already introduced in the negotiation type and instance schemata in the previous sections.

7.3 Participant

The *Negotiation Participant* interface is formalised with the following WSDL description:

Listing 4: WSDL 1.1: Negotiation Participant

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions
3     name="NegotiationParticipant"
4     targetNamespace="http://www.mycomp.org/schemas/
          NegotiationParticipant"
5     xmlns:tns="http://www.mycomp.org/schemas/
          NegotiationParticipant"
6     xmlns:wsag="http://schemas.ggf.org/graap/2005/09/ws-
          agreement"
7     xmlns:ns="http://xml.netbeans.org/examples/negotiation"
8     xmlns="http://schemas.xmlsoap.org/wsdl/"
```

```
9      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
10     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
11     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
12 
13     <!-- Type definitions -->
14     <types>
15         <xsd:schema>
16             <xsd:import namespace="http://xml.netbeans.org/
                   examples/negotiation" schemaLocation="
                   NegotiationInstance.xsd"/>
17             <xsd:import namespace="http://schemas.ggf.org/
                   graap/2005/09/ws-agreement" schemaLocation="
                   agreement_types.xsd"/>
18 
19             <xsd:complexType name="OffersType">
20                 <xsd:sequence>
21                     <xsd:element name="negotiationID" type="
                           xsd:string"/>
22                     <xsd:element name="offer" minOccurs="0"
                           maxOccurs="unbounded">
23                         <xsd:complexType>
24                             <xsd:sequence>
25                                 <xsd:element name="agentID"
                                       type="xsd:anyType"/>
26                                 <xsd:element name="timeStamp"
                                       type="xsd:dateTime"/>
27                                 <xsd:element name="
                                       offeredAgreement" type="
                                       wsag:AgreementType"/>
28                             </xsd:sequence>
29                         </xsd:complexType>
30                     </xsd:element>
31                 </xsd:sequence>
32             </xsd:complexType>
33 
34             <xsd:simpleType name="InvalidInputMessageType">
35                 <xsd:restriction base="xsd:string">
36                     <xsd:enumeration value="syntacticalError"
                           />
37                     <xsd:enumeration value="
                           inputParametersMissing"/>
38                     <xsd:enumeration value="
                           referencedEntityNotFound"/>
39                 </xsd:restriction>
40             </xsd:simpleType>
41 
42             <xsd:simpleType name="AccessDeniedMessageType">
43                 <xsd:restriction base="xsd:string">
44                     <xsd:enumeration value="
                           coordinatorOnlyMethod"/>
45                     <xsd:enumeration value="
                           violationOfNegotiationRestriction"/>
46                 </xsd:restriction>
47             </xsd:simpleType>
48         </xsd:schema>
49     </types>
```

```
50
51     <!-- Message definitions -->
52     <message name="ProposeNegotiationRequest">
53         <part name="offeredNegotiation" type="ns:
               NegotiationInstanceType"/>
54     </message>
55
56     <message name="UpdateNegotiationRequest">
57         <part name="updatedNegotiation" type="ns:
               NegotiationInstanceType"/>
58     </message>
59
60     <message name="AcceptNegotiationRequest">
61         <part name="negotiationID" type="xsd:string"/>
62     </message>
63
64     <message name="StartNegotiationRequest">
65         <part name="negotiationID" type="xsd:string"/>
66     </message>
67
68     <message name ="PlaceOfferRequest">
69         <part name="agentID" type="xsd:anyType"/>
70         <part name="offer" type="wsag:AgreementType"/>
71     </message>
72
73     <message name="NewRoundRequest">
74         <part name="negotiationID" type="xsd:string"/>
75         <part name="processInformation" type="tns:OffersType"
               />
76     </message>
77
78     <message name="AcceptAgreementRequest">
79         <part name="negotiationID" type="xsd:string"/>
80         <part name="agreementID" type="xsd:string"/>
81     </message>
82
83     <message name="RejectAgreementRequest">
84         <part name="negotiationID" type="xsd:string"/>
85     </message>
86
87     <message name="InvalidInputMessage">
88         <part name="inputError" type="InvalidInputMessageType"
               />
89     </message>
90
91     <message name="AccessDeniedMessage">
92         <part name="accessError" type="AccessDeniedType"/>
93     </message>
94
95     <!-- Port type definitions -->
96     <portType name="NegotiationParticipantPortType">
97         <operation name="ProposeNegotiation">
98             <input message="tns:ProposeNegotiationRequest"/>
99             <fault name="Invalid Input" message="tns:
                   InvalidInputMessage"/>
100        </operation>
```

```
101
102         <operation name="UpdateNegotiation">
103             <input message="tns:UpdateNegotiationRequest"/>
104             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
105             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
106         </operation>
107
108         <operation name="AcceptNegotiation">
109             <input message="tns:AcceptNegotiationRequest"/>
110             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
111             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
112         </operation>
113
114         <operation name="StartNegotiation">
115             <input message="tns:StartNegotiationRequest"/>
116             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
117             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
118         </operation>
119
120         <operation name="PlaceOffer">
121             <input message="tns:PlaceOfferRequest"/>
122             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
123             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
124         </operation>
125
126         <operation name="NewRound">
127             <input message="tns:NewRoundRequest"/>
128             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
129             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
130         </operation>
131
132         <operation name="AcceptAgreement">
133             <input message="tns:AcceptAgreementRequest"/>
134             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
135             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
136         </operation>
137
138         <operation name="RejectAgreement">
139             <input message="tns:RejectAgreementRequest"/>
140             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
141             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
142         </operation>
```

```
143     </portType>
144 </definitions>
```

Analogously to the *Negotiation Coordinator* interface description each operation offered by the *Negotiation Participant* interface is defined. As already stated in sections 5 and 6 all of these methods do not offer output messages (asynchronous communication).

Again for each message the respective parameters are defined using XML Schema standard types or ones defined in the already presented schema files for negotiation types and instances.

However, one additional type is introduced: the *OffersType*. This type is used for the information package distributed to all participants when starting a new round of the negotiation. Hence it contains an ID used to identify the negotiation it concerns and an unbounded number of offers, along with their originator and the timestamp referring to when the offer was posted. This way the *Negotiation Coordinator* can supply the *Negotiaton Participants* with additional information for the new round.

7.4 Information Service

Finally the *Information Service* interface is described with the following WSDL document:

Listing 5: WSDL 1.1: Information Service

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions
3     name="InformationService"
4     targetNamespace="http://www.mycomp.org/schemas/
        InformationService"
5     xmlns:tns="http://www.mycomp.org/schemas/
        InformationService"
6     xmlns:wsag="http://schemas.ggf.org/graap/2005/09/ws-
        agreement"
7     xmlns="http://schemas.xmlsoap.org/wsdl/"
8     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
9     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
10    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
11
12    <!-- Type definitions -->
13    <types>
14        <xsd:schema>
15            <xsd:import namespace="http://schemas.ggf.org/
                graap/2005/09/ws-agreement" schemaLocation="
                agreement_types.xsd"/>
16
17            <xsd:complexType name="OffersType">
18                <xsd:sequence>
19                    <xsd:element name="negotiationID" type="
                        xsd:string"/>
20                    <xsd:element name="offer" minOccurs="0"
                        maxOccurs="unbounded">
21                        <xsd:complexType>
22                            <xsd:sequence>
```

```
23                              <xsd:element name="agentID"
                                    type="xsd:anyType"/>
24                              <xsd:element name="timeStamp"
                                    type="xsd:dateTime"/>
25                              <xsd:element name="
                                    offeredAgreement" type="
                                    wsag:AgreementType"/>
26                          </xsd:sequence>
27                      </xsd:complexType>
28                  </xsd:element>
29              </xsd:sequence>
30          </xsd:complexType>
31
32          <xsd:simpleType name="InvalidInputMessageType">
33              <xsd:restriction base="xsd:string">
34                  <xsd:enumeration value="syntacticalError"
                        />
35                  <xsd:enumeration value="
                        inputParametersMissing"/>
36                  <xsd:enumeration value="
                        referencedEntityNotFound"/>
37              </xsd:restriction>
38          </xsd:simpleType>
39
40          <xsd:simpleType name="AccessDeniedMessageType">
41              <xsd:restriction base="xsd:string">
42                  <xsd:enumeration value="
                        coordinatorOnlyMethod"/>
43                  <xsd:enumeration value="
                        violationOfNegotiationRestriction"/>
44              </xsd:restriction>
45          </xsd:simpleType>
46      </xsd:schema>
47  </types>
48
49  <!-- Message definitions -->
50  <message name="GetStatusRequest">
51      <part name="negotiationID" type="xsd:string"/>
52  </message>
53
54  <message name ="GetPastOffersRequest">
55      <part name="negotiationID" type="xsd:string"/>
56  </message>
57
58  <message name ="GetPastOffersFromAgentRequest">
59      <part name="negotiationID" type="xsd:string"/>
60      <part name="agentID" type="xsd:anyType"/>
61  </message>
62
63  <message name ="GetStatusResponse">
64      <part name="currentOffers" type="tns:OffersType"/>
65  </message>
66
67  <message name ="GetPastOffersResponse">
68      <part name="pastOffers" type="tns:OffersType"/>
69  </message>
```

```
70
71     <message name="InvalidInputMessage">
72         <part name="inputError" type="InvalidInputMessageType"
               />
73     </message>
74
75     <message name="AccessDeniedMessage">
76         <part name="accessError" type="AccessDeniedType"/>
77     </message>
78
79     <!-- Port type definitions -->
80     <portType name="InformationServicePortType">
81         <operation name="GetStatus">
82             <input message="tns:GetStatusRequest"/>
83             <output message="tns:GetStatusResponse"/>
84             <fault name="Invalid Input" message="tns:
                   InvalidInputMessage"/>
85             <fault name="Access Denied" message="tns:
                   AccessDeniedMessage"/>
86         </operation>
87
88         <operation name="GetPastOffers">
89             <input message="tns:GetPastOffersRequest"/>
90             <output message="tns:GetPastOffersResponse"/>
91             <fault name="Invalid Input" message="tns:
                   InvalidInputMessage"/>
92             <fault name="Access Denied" message="tns:
                   AccessDeniedMessage"/>
93         </operation>
94
95         <operation name="GetPastOffersFromAgent">
96             <input message="tns:GetPastOffersFromAgentRequest"
                   />
97             <output message="tns:GetPastOffersResponse"/>
98             <fault name="Invalid Input" message="tns:
                   InvalidInputMessage"/>
99             <fault name="Access Denied" message="tns:
                   AccessDeniedMessage"/>
100        </operation>
101    </portType>
102 </definitions>
```

The three operations exposed by the *Information Service* are again described by specifying their input and output messages respectively.

In order to transmit past offers of a negotiation the *OffersType* introduced for the *Negotiation Participant* interface is used.

8 Example Negotiations

The exchange and negotiation processes to be supported by this framework have already been detailed in sections 5 and 6. However these protocol descriptions only showed possible sequences of method invocations along with the transmitted messages in a very abstract way.

Now two distinct negotiation protocols, an auction-like protocol and a specific One-on-One Bargaining process, are implemented using the definitions made in sections 4 and 5. The presented documents illustrate how respective negotiation types and instances can be exhaustively defined with the data structures presented in this thesis.

Such negotiation type or instance documents have been part of the process descriptions, concerning the exchange and the negotiation protocol, before. However these data structures have only been referenced to illustrate what data has to be provided for or is returned after a particular method invocation. Now two sample documents are presented as examples of such data structures to be transmitted within the respective protocols. Note: Not only would such an example data structure represent some input or output parameter of the corresponding method(s), but also it defines the negotiation protocol as a whole by specifying the restrictions placed on the bidding process. Therefore not only the parameters but the sequences of method invocations are crucially dependent on the defined negotiaion types and instances as already depicted.

8.1 Used Constraint Language

Some of the elements specified in the negotiation type document are defined to contain restriction expressions formalised in external constraint languages. In order to implement example negotiation types such an external language had to be chosen to express the respective restrictions.

Since only very simple restrictions are set in the respective elements of the example documents, no actual external constraint lanaugage was used. For simplicity purposes a rudimentary rule-based language was created to express the few needed restriction assertions.

In the following documents only **Attribute** or **Negotiation Restrictions** and the **Content** of past offers(**Negotiation Transparency**) or the negotiation status (**Status Transparency**) are defined using this language. Hence only very few constructs are needed. These language components will be presented shortly in this subsection.

For the *negotiationContent* attribute the value complete is defined denote all past offers can be queried entirely, that is not restricted to some subset of the available elements. Complete statusContent means that all currently valid offers are returned as negotiation status.

For the restrictions four operators are introduced, to be used for the respective rule-expressions: **geq** meaning greater or equal than, **gt** meaning greater than, **neq** meaning not equal than and **lt** standing for less than. Furthermore each rule is of the form "‘function-name(parameters):= rule, defining when the function evaluates to true"’. "‘abc(d,

e):= true"', for example would always evaluate to true, whereas "'abcde(x):= x"', would evaluate to true if x=true, otherwise it would evaluate to false.

Also two functional operators were introduced: **iff** means "'if and only if"' and **forAll()** denotes an expression concerning all possible values that can be reference within the brackets. For example forAll(offer) is set before some rule that evaluates to true if it always evaluates to true for all available offers to be set as the offer-parameter. This concept is used to describe restrictions for the auction protocol.

By using this (fragmentary) language all restrictions used in the following examples can be defined.

8.2 Auction Protocol concerning a Middle-Tier Storage Service

As already described a negotiation type document is needed to describe the respective bidding process. The negotiation type document for the example auction protocol is defined as follows:

Listing 6: Auction Type Document

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <negotiation
4      xmlns='http://xml.netbeans.org/examples/negotiation'
5      xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
6      xsi:schemaLocation='http://xml.netbeans.org/examples/
          negotiation NegotiationType.xsd'>
7
8      <negotiationTypeID>
9          storageServiceAuctionType
10     </negotiationTypeID>
11     <wsAgreementTemplate>
12         <endpoint>
13             http://www.abc.com/data
14         </endpoint>
15         <templateID>
16             storageServiceTemplate
17         </templateID>
18     </wsAgreementTemplate>
19     <!-- start and termination is set in the negotiation
          instance document -->
20     <rounds>
21         1
22     </rounds>
23     <arbitration arbitrationForm="none"/>
24     <role roleName="serviceProvider" permissionToPostOffers="
          true">
25         <admissionRestriction admissionRestrictionForm="open"
              />
26     </role>
27     <role roleName="serviceConsumer" permissionToPostOffers="
          false">
28         <maximumNumberOfAgents>
29             1
```

```
30          </maximumNumberOfAgents>
31          <admissionRestriction admissionRestrictionForm="open"
                />
32      </role>
33      <role roleName="coordinator" permissionToPostOffers="false
            ">
34          <maximumNumberOfAgents>
35              1
36          </maximumNumberOfAgents>
37          <admissionRestriction admissionRestrictionForm="open"
                />
38      </role>
39      <role roleName="informationService" permissionToPostOffers
            ="false">
40          <maximumNumberOfAgents>
41              1
42          </maximumNumberOfAgents>
43          <admissionRestriction admissionRestrictionForm="open"
                />
44      </role>
45      <negotiatedIssues>
46          <contextElements extendable="false"/>
47          <serviceDescriptionTerms extendable="false"/>
48          <servicePropertyTerms extendable="false"/>
49          <guaranteeTerms extendable="false">
50              <guaranteeTermID domain="xsd:integer" values="
                    single">
51                  storageSizeInGBGuarantee
52              </guaranteeTermID>
53              <guaranteeTermID domain="xsd:integer" values="
                    single">
54                  parallelConnectionsGuarantee
55              </guaranteeTermID>
56              <guaranteeTermID domain="xsd:float" values="single
                    ">
57                  maximumResponseTimeInSecondsGuarantee
58              </guaranteeTermID>
59          </guaranteeTerms>
60      </negotiatedIssues>
61      <attributeRestriction>
62          <attribute>
63              storageSizeInGBGuarantee
64          </attribute>
65          <restriction>
66              <threshold>
67                  <lowerBound>
68                      30
69                  </lowerBound>
70              </threshold>
71          </restriction>
72      </attributeRestriction>
73      <attributeRestriction>
74          <attribute>
75              parallelConnectionsGuarantee
76          </attribute>
77          <restriction>
```

```
78              <threshold>
79                  <lowerBound>
80                      50
81                  </lowerBound>
82              </threshold>
83          </restriction>
84      </attributeRestriction>
85      <attributeRestriction>
86          <attribute>
87              maximumResponseTimeInSecondsGuarantee
88          </attribute>
89          <restriction>
90              <threshold>
91                  <upperBound>
92                      10,5
93                  </upperBound>
94              </threshold>
95          </restriction>
96      </attributeRestriction>
97      <generalRestrictionRule>
98          currently_valid_offer:= a iff: forAll(offer): better(a
                , offer);
99          other_offer:= a iff a neq currently_valid_offer;
100         valid(offer):= better(offer, currently_valid_offer);
101         better(a, b):= (a.storageSizeInGBGuarantee geq b.
                storageSizeGuarantee OR
102         a.parallelConnectionsGuarantee geq b.
                parallelConnectionsGuarantee OR
103         a.maximumResponseTimeInSecondsGuarantee leq b.
                maximumResponseTimeInSecondsGuarantee) AND
104         (a.storageSizeInGBGuarantee gt b.storageSizeGuarantee
                OR
105         a.parallelConnectionsGuarantee gt b.
                parallelConnectionsGuarantee OR
106         a.maximumResponseTimeInSecondsGuarantee lt b.
                maximumResponseTimeInSecondsGuarantee)
107     </generalRestrictionRule>
108     <offerAllocation>
109         <matchingForm>defined</matchingForm>
110         <matchingRule>
111             currently_valid_offer
112         </matchingRule>
113     </offerAllocation>
114     <informationProcessing>
115         <negotiationTransparency>none</negotiationTransparency
                >
116         <statusTransparency>public</statusTransparency>
117         <statusContent>currently_valid_offer</statusContent>
118     </informationProcessing>
119     </negotiation>
```

After specifying this auction type's id (storageServiceAuctionType) and referencing the corresponding template (storageServiceTemplate, available at http://www.abc.com/data) the actual bidding process is described.

Start and termination of this auction are defined time-dependent and are therefore given in the negotiation instance document. This auction only consists of one bidding phase and therefore defines only one round. After setting the arbitration rules for this protocol (none was used to indicate that no rewards or punishments are generated during protocol execution) the different roles are specified.

For this example only the four default roles are defined (service consumer and provider, *Negotiation Coordinator* and *Information Service*), each with open admission indicating that no particular credentials have to be provided by the agents joining a specific role. All roles but one are restricted to at most one agent (this is always the case for the *Negotiation Coordinator* because of its centralised nature; more than one coordinators would not be semantically correct). The only role admitting more than one is the service provider. This already shows the configuration of the auction: one service consumer accepts offers from a multitude of service providers, hence this document represents a reverse auction.

The auction protocol only concerns three *guarantee terms* as negotiatable items: *storageSizeInGBGuarantee*, *parallelConnectionsGuarantee* and *maximumResponseTimeInSecondsGuarantee*, each of which is accompanied with a corresponding domain. Since all of these items should be set to distinct values single is defined in each *values*-attribute. Other possibly extisting elements of the WS-Agreement template are not subject of the negotiation because they are not mentioned in this document. Also no additional terms or *context elements* can be specified in potential offers as stated in the *extendable*-attributes.

The bidding process is defined by providing several attribute and one negotiation restriction, meaning the following: For example storageSizeInGBGuarantee-elements within new offers always have to be higher than 30 (lower bound). Analogously, the other attribute restrictions are defined. The general negotiation restriction states that a new offer is valid if the storageSizeInGBGuarantee- and the parallelConnectionsGuarantee-elements are at least as much as and the maximumResponseTimeInSecondsGuarantee-element is at most as much as stated in the current best offer. Additionally one of the first two items mentioned has to be higher or the last one has to be lower than the equivalent in the current offer. For example the current bid would define the individual bounds of each attribute as offered values: 30, 50 and 10,5. A new valid offer would be 40, 50 and 10,5 or 30, 50, and 9 but not 30, 50, 10,5 because the second part of the restriction (requiring at least one changed parameter) would be violated.

The offer matching element defines that the currently_valid_offer, will win when the negotiation terminates. Additionally past offers can not be queried, whereas the negotiation status is also defined to be the currently_valid_offer and is publicly accessible.

Such a type document could then be used to create a corresponding auction instance described in an instance document, for example like this:

Listing 7: Auction Instance Document

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <NegotiationInstance
4     xmlns="http://xml.netbeans.org/examples/negotiation"
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6     xsi:schemaLocation="http://xml.netbeans.org/examples/
        negotiation NegotiationInstance.xsd">
7     <negotiationID>
```

```
8          storageServiceAuction
9      </negotiationID>
10     <negotiationType>
11         <referencedNegotiationType>
12             <endpoint>
13                 http://www.abc.com/data
14             </endpoint>
15             <negotiationTypeID>
16                 storageServiceAuctionType
17             </negotiationTypeID>
18         </referencedNegotiationType>
19     </negotiationType>
20     <start>
21         2006-09-30T13:30:00
22     </start>
23     <termination>
24         2006-09-30T23:30:00
25     </termination>
26     <agent>
27         <role>
28             serviceConsumer
29         </role>
30         <agentEPR>
31             http://www.abc.com/data
32         </agentEPR>
33     </agent>
34     <agent>
35         <role>
36             coordinator
37         </role>
38         <agentEPR>
39             http://www.abc.com/data
40         </agentEPR>
41     </agent>
42     <agent>
43         <role>
44             informationService
45         </role>
46         <agentEPR>
47             http://www.abc.com/data
48         </agentEPR>
49     </agent>
50 </NegotiationInstance>
```

This would be the initial instance document setting the agent with the EPR "‘http://www.abc.com/data"’ to act as service consumer, *Negotiation Coordinator* and *Information Service*, referencing the respective negotation type document (by providing an EPR and the document’s id) and specifying start and termination dates.

After joining of additional agents this document would be updated and redistributed as described in section 5 and would lead to a negotiation process like the one described in section 6 6.

8.3 One-on-One Bargaining in a Job-Scheduling Scenario

A One-on-One Bargaining protocol used for negotiations in a job-scheduling environment could be defined as schown in the following negotiation type document:

Listing 8: Bargaining Type Document

```
<?xml version="1.0" encoding="UTF-8"?>

<negotiation
    xmlns='http://xml.netbeans.org/examples/negotiation'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://xml.netbeans.org/examples/
        negotiation NegotiationType.xsd'>

    <negotiationTypeID>
        jobSchedulingBargainingType
    </negotiationTypeID>
    <wsAgreementTemplate>
        <endpoint>
            http://www.abc.com/job
        </endpoint>
        <templateID>
            jobSchedulingTemplate
        </templateID>
    </wsAgreementTemplate>
    <start>
        invocation of startNegotiation()
    </start>
    <termination>
        invocation of accept/rejectAgreement()
    </termination>
    <rounds>
        1
    </rounds>
    <arbitration arbitrationForm="none"/>
    <role roleName="serviceProvider" permissionToPostOffers="
        true">
        <maximumNumberOfAgents>
            1
        </maximumNumberOfAgents>
        <admissionRestriction admissionRestrictionForm="open"
            />
    </role>
    <role roleName="serviceConsumer" permissionToPostOffers="
        true">
        <maximumNumberOfAgents>
            1
        </maximumNumberOfAgents>
        <admissionRestriction admissionRestrictionForm="open"
            />
    </role>
    <role roleName="coordinator" permissionToPostOffers="false
        ">
        <maximumNumberOfAgents>
            1
        </maximumNumberOfAgents>
```

```
45          <admissionRestriction admissionRestrictionForm="open"
                />
46      </role>
47      <role roleName="informationService" permissionToPostOffers
            ="false">
48          <maximumNumberOfAgents>
49              1
50          </maximumNumberOfAgents>
51          <admissionRestriction admissionRestrictionForm="open"
                />
52      </role>
53      <negotiatedIssues>
54          <contextElements extendable="true">
55              <elementID domain="xsd:dateTime">
56                  wsagn:ExpirationTime
57              </elementID>
58          </contextElements>
59          <serviceDescriptionTerms extendable="true">
60              <serviceDescriptionTermID domain="xsd:string"
                    values="multiple">
61                  javaRuntimeEnvironment
62              </serviceDescriptionTermID>
63          </serviceDescriptionTerms>
64          <servicePropertyTerms extendable="true"/>
65          <guaranteeTerms extendable="true">
66              <guaranteeTermID domain="xsd:integer" values="
                    single">
67                  averageResponseTimeInSecondsGuarantee
68              </guaranteeTermID>
69              <guaranteeTermID domain="xsd:integer" values="
                    single">
70                  minimumMemoryInMBGuarantee
71              </guaranteeTermID>
72          </guaranteeTerms>
73      </negotiatedIssues>
74      <attributeRestriction>
75          <attribute>
76              averageResponseTimeInSecondsGuarantee
77          </attribute>
78          <restriction>
79              <progress>
80                  <progressForm>descending</progressForm>
81              </progress>
82          </restriction>
83      </attributeRestriction>
84      <attributeRestriction>
85          <attribute>
86              averageResponseTimeInSecondsGuarantee
87          </attribute>
88          <restriction>
89              <threshold>
90                  <upperBound>
91                      40
92                  </upperBound>
93              </threshold>
94          </restriction>
```

```
95      </attributeRestriction>
96      <attributeRestriction>
97          <attribute>
98              minimumMemoryInMBGuarantee
99          </attribute>
100         <restriction>
101             <threshold>
102                 <lowerBound>
103                     1024
104                 </lowerBound>
105             </threshold>
106         </restriction>
107     </attributeRestriction>
108     <offerAllocation>
109         <matchingForm>forwarded</matchingForm>
110     </offerAllocation>
111     <informationProcessing>
112         <negotiationTransparency>protected</
                negotiationTransparency>
113         <negotiationContent>complete</negotiationContent>
114         <statusTransparency>protected</statusTransparency>
115         <statusContent>complete</statusContent>
116     </informationProcessing>
117 </negotiation>
```

This negotiation type is described analogously to the auction type before, exept for a few particular elements. Now all sides are restricted to one participant, as needed for One-on-One negotiations. Also the start and termination attributes are defined as method invocations to occur during the negotiation process and are therefore set in the type document instead of the instance document.

Again some netotiated items and corresponding attribute restrictions are defined. This time however, no general restriction concerning the bidding process is set. In this protocol additional terms and *context elements* can be specified in an offer as defined in the *extendable*-attributes. Also the progression is defined to be *descending* for one particular item (descending); this is similar to the auction protocol, but there this was either concerning a bundle of attributes and also it was described using a general restriction.

This protocol sets offer matching to be *forwarded*, so no matching rule is publicly available. An agreement is simply reached when one of the parties accepts a received offer.

Finally information concerning the negotiation is only available for the two involved agents. However, they can access all past offers (complete) and both current offers posted by the two participants as negotiation status (complete).

Analogously this type document can be used to instantiate actual negotiations as described before. As opposed to the auction presented before a negotiation instance of this type would not have to specify the *start* and *termination*-elements because they are already set in the type document.

9 Conclusion and Prospects

This thesis presents a comprehensive framework for automated SLA negotiations. This framework is based on the WS-Agreement specification draft [4], proposing means to formalise SLA contracts, and therefore enables automated negotiations on such WS-Agreements. First the SLA concept and its applications in business integration scenarios are sketched, before a short review on the WS-Agreement specification is given.

Next some basic concepts regarding negotiations and negotiation protocols originating in the multi-disciplinary field of negotiaiton research are presented. By listing a small subset of possible negotiation protocols the need for different protocols in different situations is shown. In order to support such a variety of different negotiation protocols this framework is based on comprehensive work in negotiation research. Negotiation taxonomies originating in either economics, e-commerce or multi-agent research are presented and compared. Based on these taxonomies a suitable set of attributes and attribute categories are identified, used to exhaustively describe negotiation protocols in terms of their defining parameters. With these attributes market and negotiation designers can define negotiation protocols in a machine-processable manner, which is crucially important for this framework for its intended use within multi-agent systems. This is software agents should be enabled to negotiate SLAs with the definitions provided by this framework without any human interaction. Because of the sheer volume of conducted SLA negotiations only such a totally automated approach is considered appropriate for efficiency reasons. Accordingly, for each individual negotiation situation the appropriate negotiation protocol can be specified and applied afterwards.

Next, these attributes are used to define a data model representing the meta information of negotiation protocols in general. This data model is given in terms of the ERM modelling language [38], to ease the persistent storage of such negotiation types in relational databases; well-known techniques to convert ERM data models into database models can be applied here. The distinction between negotiation types and instances as given in this thesis also contributes to this, since instances can be derived from negotiation types in some database.

Finally the set of attributes and attribute categories are operationalised for WS-Agreement and Web Services scenarios in general by defining corresponding XML Schema [54] documents representing the structure of negotiation protocol types or instances. With theses Schema documents negotiation designers can define negotiation types and instances as XML documents that can easily be exchanged among the different negotiation participants using Web Services related technologies.

By utilising these documents an exchange protocol for negotiation meta data is introduced. This protocol is used to distribute all information, necessary to take part in a respective negotiation, to all prospective participants. That is this protocol distributes the negotiation type and instance documents defined with the means presented in this thesis. To enable the application of this framework in service oriented environments this protocol is defined by proposing involved roles and methods respectively along with possible method invocation sequences used for exchanging the negotiation information. This way the abstract exchange process is described in terms of simple protocol components that can be combined to create more sophisticated ones, if desired. To define such roles and methods WSDL documents are used [1]; both WSDL 1.1 and WSDL 2.0 documents

are given to support broad application of this framework in different environments.

Finally a generic negotiation protocol is defined analogously. This protocol is capable to process all negotiation protocols that can be defined with the formerly proposed data model. Again this protocol is described by presenting involved roles and methods as well as a description of the abstract negotiation process employing these methods.

This thesis concludes by presenting two example negotiation type and instance documents for an auction and a bargaining protocol respectively.

With this framework fully automated SLA negotiations between software agents can be executed. Negotiation designers can specify a multitude of different protocols with the presented data structures, let the software agents exchange the respective information and conduct the actual negotiations to reach agreements. The respective exchange and negotiation protocols as well as the presented data structure are defined in very generic, yet detailed way to support a large set of different negotiation protocols to be conducted. This way the appropriate protocol can be chosen for each negotiation situation in order to reach optimal results. Each negotiation procotol results in a valid WS-Agreement document involving the respective service providers and consumers. The proposed negotiation protocol framework can be used to specify negotiation protocols that can substitute the simple agreement creation process as described in the current specification draft. This way the simple request-response mechanism is replaced by complex negotiation processes. The needed interfaces and methods are defined using Web Services related standards in order to comply to the technologies used in the WS-Agreement specification for agreement monitoring purposes.

The proposed framework only supports 1:1 and 1:n or n:1 negotiations at the moment. N:m negotiations like CDAs could be incorporated into this framework by extending the coordinator role appropriately. For such negotiations centralised market instances would have to be defined according with detailed matching algorithms used in that market. The *Negotiation Coordinator* role, for already representing a centralised entity within a negotiation protocol would therefore represent an appropriate concept to use for defining such market instances.

Also a new level of abstraction between negotiation types and instances is likely to be introduced in the future. At the moment each negotiation type always refers to one particular WS-Agreement template which inhibits the definition of very general negotiation protocol types, as a reverse auction for example, without referring to particular negotiated issues and therefore distinct template documents. Such general protocol types could then be held in persistent storage services, like distributed data bases, and referenced within newly created negotiation instances. This way negotiation designers would be able to define negotiation instances using already known negotiation protocol types and only defining the negotiated issues and participants. This approach would provide software agents as well as negotiation designers with a common vocabulary of negotiation types. Such negotiation types made available within an infrastructure of distributed negotiation information servers would highly increase reusability and ease the definition of concrete negotiation protocols.

This framework will also probably be extended to provide means for defining distributed systems of negotiation servers as just hinted. Such infrastructures of look-up or information servers would not only increase availability and reduce response time within electronic

negotiations by storing the relevant information in a redundant way but also ease automated negotiations in general by providing a common platform used to store relevant information.

References

[1] R. Chinnici, J.-J. Moreau, A. Ryma, and S. Weerawarana, “Web services description language (wsdl) version 2.0 part 1: Core language,” *W3C*, March 2006. [Online]. Available: http://www.w3.org/TR/2006/CR-wsdl20-20060327/<2006-09-21>

[2] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, “Soap version 1.2 part 1: Messaging framework,” *W3C*, June 2003. [Online]. Available: http://www.w3.org/TR/2003/REC-soap12-part1-20030624/<2006-09-21>

[3] T. Bellwood, S. Capell, L. Clement, J. Colgrave, M. J. Dovey, D. Feygin, A. Hately, R. Kochman, P. Macias, M. Novotny, M. Paolucci, C. von Riegen, T. Rogers, K. Sycara, P. Wenzel, and Z. Wu, “Universal description, discovery and integration v3.0.2,” *OASIS UDDI Specification Technical Committee*, October 2004. [Online]. Available: http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm<2006-09-21>

[4] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, “Web services agreement specification draft, version 09/2005,” 2005.

[5] P. H. Gulliver, *Disputes and Negotiation: A Cross-Cultural Perspective.* New York: Academic Press, 1979.

[6] P. McAfee and J. McMillan, “Auctions and bidding,” *Journal of Economic Literature*, vol. 25, pp. 699–738, 1987.

[7] E. Wolfstetter, “Auctions: An introduction,” *Journal of Economic Surveys*, vol. 10, pp. 367–420, 1996.

[8] N. R. Jennings, K. Sycara, and M. Wooldridge, “A roadmap of agent research and development,” *Int Journal of Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, pp. 7–38, 1998.

[9] N. R. Jennings and M. Wooldridge, “Applications for intelligent agents,” *Agent Technology: Foundations, Applications, and Markets (editors N. R. Jennings and M. Wooldridge)*, pp. 3–28, 1998.

[10] A. Keller, G. Kar, H. Ludwig, A. Dan, and J. L. Hellerstein, “Managing dynamic services: A contract based approach to a conceptual architecture,” *Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS 2002), Florence Italy*, April 2002.

[11] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck, “A service level agreement language for dynamic electronic services,” *Journal of Electronic Commerce Research*, vol. 3, pp. 43–59, March 2003.

[12] D. Booth, H. Haas, F. McGabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, “Web services architecture,” *W3C*, February 2004. [Online]. Available: http://www.w3.org/TR/ws-arch/<2006-09-21>

[13] A. Dan, H. Ludwig, and G. Pacifici, “Web services differentiation with service level agreements,” *IBM Software Group Web Services Web site*, 2003. [Online]. Available: ftp://ftp.software.ibm.com/software/websphere/webservices/webserviceswithservicelevelsupport.pdf<2006-09-21>

[14] H. Ludwig, A. Dan, and R. Kearney, “Cremona: An architecture and library for creation and monitoring of ws-agreements,” *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC 2004), ACM Press, New York*, pp. 65–74, 2004.

[15] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible markup language (xml) 1.0 (fourth edition),” *W3C*, August 2006. [Online]. Available: http://www.w3.org/TR/2006/REC-xml-20060816/<2006-09-21>

[16] T. Banks, “Web services resource framework (wsrf) v1.2 - primer, draft 02,” *OASIS Web Services Resource Framework Technical Committee*, May 2006. [Online]. Available: http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf<2006-09-21>

[17] J. Gray, “Learning from the amazon technology platform - a conversation with werner vogels,” *ACM Queue*, vol. 4, no. 4, May 2006. [Online]. Available: http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=388<2006-09-21>

[18] D. G. Pruitt, *Negotiation Behavior.* Academic Press, New York, 1981.

[19] G. E. Kersten, “The science and engineering of e-negotiation: An introduction,” *Proceedings of the Hawaii International Conference on System Sciences*, 2003.

[20] M. Bichler, G. Kersten, and S. Strecker, “Towards a structured design of electronic negotiations,” *Group Decision and Negotiation*, vol. 12, no. 4, pp. 311–335, 2003.

[21] J. Nash, “The bargaining problem,” *Econometrica*, vol. 18, pp. 155–162, 1950.

[22] ——, “Two-person cooperative games,” *Econometrica*, vol. 21, pp. 121–140, 1953.

[23] A. Roth, *Axiomatic Models of Bargaining*, ser. Lecture Notes in Economics and Mathematical Systems. Berlin: Springer, 1979, no. 170.

[24] R. K. Dash, N. R. Jennings, and D. C. Parkes, “Computational-mechanism design: A call to arms,” *IEEE Intelligent Systems*, vol. 18, no. 6, pp. 40–47, 2003.

[25] R. Fisher, E. Kopelman, and A. K. Schneider, *Beyond Machiavelli: Tools for Coping with Conflict.* Cambridge: Harvard University Press, 1994.

[26] W. Ury, *Getting past No: Negotiating your way from Confrontation to Cooperation.* Bantam Books, 1993.

[27] D. Druckman, *Negotiations: Social-Psychological Perspectives.* Beverly Hills: Sage Publications, 1977.

[28] M. Stroebel and C. Weinhardt, “The montreal taxonomy for electronic negotiations,” *Journal of Group Decision and Negotiation*, vol. 12, pp. 143–164, 2003.

[29] A. R. Lomuscio, M. Wooldridge, and N. R. Jennings, "A classification scheme for negotiation in electronic commerce," *Int Journal of Group Decision and Negotiation*, vol. 12, no. 1, pp. 31–56, 2003.

[30] M. Bichler and J. R. Kalagnanam, "Software frameworks for advanced procurement auction markets," *Communications of the ACM (CACM)*, vol. 49, no. 12, 2006. [Online]. Available: http://ibis.in.tum.de/staff/bichler/docs/050404_bichler_kalagnanam_cacm.pdf<2006-09-21>

[31] P. Cramton, Y. Shoham, and R. Steinberg, "Introduction to combinatorial auctions," *Combinatorial Auctions, MIT Press 2006*, pp. 1–14, 2006.

[32] M. Bichler, *The Future of eMarkets - Multi-Dimensional Market Mechanisms.* Cambridge: Cambridge University Press, 2001.

[33] J. Teich, H. Wallenius, and J. Wallenius, "Multiple issue action and market algorithms for the world wide web," no. ir98109, December 1998, available at http://ideas.repec.org/p/wop/iasawp/ir98109.html.

[34] G. E. Kersten and J. Teich, "Are all e-commerce negotiations auctions?" *Fourth International Conference on the Design of Cooperative Systems, Sophia-Antopolis, France*, 2000.

[35] R. E. Walton and R. B. McKersie, *A Behavioral Theory of Labor Negotiations: An Analysis of a Social Interaction System.* New York: McGraw-Hill Book Company, 1965.

[36] S. Parsons and N. R. Jennings, "Negotiation through argumentation - a preliminary report," *Proceedings of the Second International Conference on Multi-Agent Systems*, pp. 267–274, 1996.

[37] S. Parsons, C. Sierra, and N. R. Jennings, "Agents that reason and negotiate by arguing," *Journal of Logic and Computation*, vol. 8, no. 3, pp. 261–292, 1998.

[38] P. P.-S. Chen, "The entity-relationship model - toward a unified view of data," *ACM Transactions on Database Systems (TODS) archive*, vol. 1, no. 1, pp. 9–36, March 1976.

[39] A. R. Lomuscio, M. Wooldridge, and N. R. Jennings, "A classification scheme for negotiation in electronic commerce," *Agent-Mediated Electronic Commerce: A European AgentLink Perspective (eds. F. Dignum and C. Sierra)*, pp. 19–33, 2001.

[40] S. Kraus, K. Sycara, and A. Evenchik, "Reaching agreements through argumentation: a logical model and implementation," *Artificial Intelligence*, vol. 104, no. 1-2, pp. 1–69, September 1998.

[41] C. Sierra, N. R. Jennings, P. Noriega, and S. Parsons, "A framework for argumentation-based negotiation," *Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages (ATAL-97)*, pp. 167–182, 1997.

[42] P. R. Wurman, M. P. Wellman, and W. E. Walsh, "The michigan internet auctionbot: A configurable auction server for human and software agents," *Second International Conference on Autonomous Agents*, May 1998.

[43] ——, "A parametrization of the auction design space," *Games and Economic Behavior*, vol. 35, no. 1-2, pp. 304–338, 2001.

[44] C. Bartolini, C. Preist, and N. R. Jennings, "A software framework for automated negotiation," *Software Engineering for Multi-Agent Systems III: Research Issues and Practical Applications (eds. R. Choren, A. Garcia, C. Lucena, and A. Ramonovsky)*, pp. 213–235, 2005.

[45] E. Friedman-Hill, "Jess: The java expert system shell," *Sandia Laboratories.* [Online]. Available: http://www.jessrules.com/jess/index.shtml<2006-09-21>

[46] F. for Intelligent Physical Agents (FIPA), "Fipa communicative act library specification," *FIPA TC Communication*, no. SC00037J, December 2002. [Online]. Available: http://www.fipa.org/specs/fipa00037/SC00037J.pdf<2006-09-21>

[47] D. L. McGuinness and F. van Harmelen, "Owl web ontology language overview," *W3C*, February 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-owl-features-20040210/<2006-09-21>

[48] C. Bartolini, C. Preist, and N. R. Jennings, "A framework for automated negotiation," Trusted E-Services Laboratory, Hewlett Packard Laboratories Bristol, Tech. Rep. HPL-2001-90, April 2001.

[49] T. Bellwood, S. Capell, L. Clement, J. Colgrave, M. J. Dovey, D. Feygin, A. Hately, R. Kochman, P. Macias, M. Novotny, M. Paolucci, C. von Riegen, T. Rogers, K. Sycara, P. Wenzel, and Z. Wu, "A generic software framework for automated negotiation," Trusted E-Services Laboratory, Hewlett Packard Laboratories Bristol, Tech. Rep. HPL-2002-2, January 2002.

[50] C. Bartolini, C. Preist, and N. R. Jennings, "Architecting for reuse: A software framework for automated negotiation," *Proc. 3rd Int Workshop on Agent-Oriented Software Engineering, Bologna, Italy*, pp. 87–98, 2002.

[51] A. Gruenert, S. Hudert, S. König, S. Kaffille, and G. Wirtz, "Decentralized reputation management for cooperating software agents in open multi-agent systems," in *Inproceedings of the German conference on Multi-Agent system TEchnologieS (MATES), September 19 - 20, 2006 in Erfurt, Germany*, 2006.

[52] M. Gudgin, M. Hadley, and T. Rogers, "Web services addressing 1.0 - core," *W3C*, May 2006. [Online]. Available: http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/<2006-09-21>

[53] D. Friedman and J. Rust, Eds., *The Double Auction Market: Institutions, Theories, and Evidence.* Boston: Addison-Wesley Publishing, 1993.

[54] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, "Xml schema part 1: Structures, second edition," *W3C*, October 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/<2006-09-21>

[55] O. M. Group, "Unified modelling language (uml) 2.0," 2005. [Online]. Available: http://www.omg.org/technology/documents/formal/uml.htm<2006-09-21>

[56] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, March 2005.

[57] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (wsdl) version 1.1," *W3C*, March 2001. [Online]. Available: http://www.w3.org/TR/2001/NOTE-wsdl-20010315<2006-09-21>

A Web Services Description Language 2.0

Negotiation Coordinator:

Listing 9: WSDL 2.0: Negotiation Coordinator

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions
3     name="NegotiationCoordinator2.0"
4     targetNamespace="http://j2ee.netbeans.org/wsdl/
          NegotiationCoordinator2.0"
5     xmlns:tns="http://j2ee.netbeans.org/wsdl/
          NegotiationCoordinator2.0"
6     xmlns:wsag="http://schemas.ggf.org/graap/2005/09/ws-
          agreement"
7     xmlns:ns="http://xml.netbeans.org/examples/negotiation"
8     xmlns="http://schemas.xmlsoap.org/wsdl/"
9     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
10    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
11    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
12
13    <!-- Type definitions -->
14    <types>
15        <xsd:schema>
16            <xsd:import namespace="http://schemas.ggf.org/
                  graap/2005/09/ws-agreement" schemaLocation="
                  agreement_types.xsd"/>
17            <xsd:import namespace="http://xml.netbeans.org/
                  examples/negotiation" schemaLocation="
                  InstantiatedNegotiation.xsd"/>
18
19            <xsd:complexType name="TemplateType">
20                <xsd:sequence>
21                    <xsd:element name="endpoint" type="xsd:
                          anyType"/>
22                    <xsd:element name="templateID" type="xsd:
                          string"/>
23                </xsd:sequence>
24            </xsd:complexType>
25
26            <xsd:complexType name="NegotiationsType">
27                <xsd:sequence>
28                    <xsd:element name="negotiation" type="ns:
                          NegotiationType" minOccurs="0"
                          maxOccurs="unbounded"/>
29                </xsd:sequence>
30            </xsd:complexType>
31
32            <xsd:complexType name="NegotiationInstancesType">
33                <xsd:sequence>
34                    <xsd:element name="negotiation" type="ns:
                          NegotiationInstanceType" minOccurs="0"
                          maxOccurs="unbounded"/>
35                </xsd:sequence>
36            </xsd:complexType>
37
38            <xsd:complexType name="CredentialsType">
```

```
39                  <xsd:sequence>
40                      <xsd:element name="credentials" type="xsd:
                            anyType" minOccurs="0" maxOccurs="1"/>
41                  </xsd:sequence>
42              </xsd:complexType>
43
44              <xsd:complexType name="AgentsType">
45                  <xsd:sequence>
46                      <xsd:element name="receivingAgent" type="
                            xsd:anyType" minOccurs="1" maxOccurs="
                            unbounded"/>
47                  </xsd:sequence>
48              </xsd:complexType>
49
50              <xsd:simpleType name="InvalidInputMessageType">
51                  <xsd:restriction base="xsd:string">
52                      <xsd:enumeration value="syntacticalError"
                            />
53                      <xsd:enumeration value="
                            inputParametersMissing"/>
54                      <xsd:enumeration value="
                            referencedEntityNotFound"/>
55                  </xsd:restriction>
56              </xsd:simpleType>
57
58              <xsd:simpleType name="AccessDeniedMessageType">
59                  <xsd:restriction base="xsd:string">
60                      <xsd:enumeration value="
                            coordinatorOnlyMethod"/>
61                      <xsd:enumeration value="
                            violationOfNegotiationRestriction"/>
62                  </xsd:restriction>
63              </xsd:simpleType>
64
65              <xsd:complexType name="
                    GetAllNegotiationTypesRequestType">
66                  <xsd:all/>
67              </xsd:complexType>
68
69              <xsd:complexType name="
                    GetAllNegotiationTypesForTemplateRequestType">
70                  <xsd:all>
71                      <xsd:element name="template" type="tns:
                            TemplateType"/>
72                  </xsd:all>
73              </xsd:complexType>
74
75              <xsd:complexType name="
                    GetCurrentNegotiationsRequestType">
76                  <xsd:all/>
77              </xsd:complexType>
78
79              <xsd:complexType name="
                    GetCurrentNegotiationsForTemplateRequestType">
80                  <xsd:all>
```

```
81                          <xsd:element name="template" type="tns:
                                TemplateType"/>
82                     </xsd:all>
83                </xsd:complexType>
84
85                <xsd:complexType name="JoinNegotiationRequestType"
                      >
86                     <xsd:sequence>
87                          <xsd:element name="negotiationID" type="
                                xsd:string"/>
88                          <xsd:element name="agentEPR" type="xsd:
                                anyType"/>
89                          <xsd:element name="credentials" type="
                                CredentialsType"/>
90                     </xsd:sequence>
91                </xsd:complexType>
92
93                <xsd:complexType name="
                      ProposeNegotiationRequestType">
94                     <xsd:all>
95                          <xsd:element name="proposedNegotiation"
                                type="ns:NegotiationInstanceType"/>
96                     </xsd:all>
97                </xsd:complexType>
98
99                <xsd:complexType name="
                      PublishNegotiationRequestType">
100                    <xsd:all>
101                         <xsd:element name="publishedNegotiation"
                                type="ns:NegotiationInstanceType"/>
102                    </xsd:all>
103               </xsd:complexType>
104
105               <xsd:complexType name="
                      PublishNegotiationToReceipientsRequestType">
106                    <xsd:sequence>
107                         <xsd:element name="publishedNegotiation"
                                type="ns:NegotiationInstanceType"/>
108                         <xsd:element name="receipients" type="tns:
                                AgentsType"/>
109                    </xsd:sequence>
110               </xsd:complexType>
111
112               <xsd:complexType name="
                      GetNegotiationTypesResponseType">
113                    <xsd:all>
114                         <xsd:element name="negotiations" type="tns
                                :NegotiationsType"/>
115                    </xsd:all>
116               </xsd:complexType>
117
118               <xsd:complexType name="
                      GetCurrentNegotiationsResponseType">
119                    <xsd:all>
120                         <xsd:element name="negotiationInstances"
                                type="tns:NegotiationInstancesType"/>
```

```
121                     </xsd:all>
122                 </xsd:complexType>
123 
124                 <xsd:complexType name="JoinNegotiationResponseType
                        ">
125                     <xsd:all>
126                         <xsd:element name="
                                updatedNegotiationInstance" type="ns:
                                NegotiationInstanceType"/>
127                     </xsd:all>
128                 </xsd:complexType>
129 
130                 <xsd:complexType name="
                        PublishNegotiationResponseType">
131                     <xsd:all>
132                         <xsd:element name="publicationAcknowledged
                                " type="xsd:boolean"/>
133                     </xsd:all>
134                 </xsd:complexType>
135 
136                 <xsd:element name="GetAllNegotiationTypesRequest"
                        type="tns:GetAllNegotiationTypesRequestType"/>
137                 <xsd:element name="
                        GetAllNegotiationTypesForTemplateRequest" type=
                        "tns:
                        GetAllNegotiationTypesForTemplateRequestType"/>
138                 <xsd:element name="GetCurrentNegotiationsRequest"
                        type="tns:GetCurrentNegotiationsRequestType"/>
139                 <xsd:element name="
                        GetCurrentNegotiationsForTemplateRequest" type=
                        "tns:
                        GetCurrentNegotiationsForTemplateRequestType"/>
140                 <xsd:element name="JoinNegotiationRequest" type="
                        tns:JoinNegotiaitonRequestType"/>
141                 <xsd:element name="ProposeNegotiationRequest" type
                        ="tns:ProposeNegotiationRequestType"/>
142                 <xsd:element name="PublishNegotiationRequest" type
                        ="tns:PublishNegotiatioinRequestType"/>
143                 <xsd:element name="
                        PublishNegotiationToReceipientsRequest" type="
                        tns:PublishNegotiatioinToReceipientsRequestType
                        "/>
144                 <xsd:element name="GetNegotiationTypesResponse"
                        type="tns:GetNegotiationTypesResponseType"/>
145                 <xsd:element name="GetCurrentNegotiationsResponse"
                         type="tns:GetCurrentNegotiationsResponseType"
                        />
146                 <xsd:element name="JoinNegotiationResponse" type="
                        tns:JoinNegotiationResponseType"/>
147                 <xsd:element name="PublishNegotiationResponse"
                        type="tns:PublishNegotiationResponseType"/>
148                 <xsd:element name="InvalidInputMessage" type="tns:
                        InvalidInputMessageType"/>
149                 <xsd:element name="AccessDeniedMessage" type="tns:
                        AccessDeniedMessageType"/>
150             </xsd:schema>
```

```
151     </types>
152
153     <interface name="NegotiationCoordinatorInterface">
154         <operation name="GetAllNegotiationTypes" pattern="http
                ://www.w3.org/2006/01/wsdl/in-out">
155             <input message="tns:GetAllNegotiationTypesRequest"
                    />
156             <output message="tns:GetNegotiationTypesResponse"
                    />
157         </operation>
158
159         <operation name="GetAllNegotiationTypesForTemplate"
                pattern="http://www.w3.org/2006/01/wsdl/in-out">
160             <input message="tns:
                    GetAllNegotiationTypesForTemplateRequest"/>
161             <output message="tns:GetNegotiationTypesResponse"
                    />
162             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
163         </operation>
164
165         <operation name="GetCurrentNegotiations" pattern="http
                ://www.w3.org/2006/01/wsdl/in-out">
166             <input message="tns:GetCurrentNegotiationsRequest"
                    />
167             <output message="tns:
                    GetCurrentNegotiationsResponse"/>
168         </operation>
169
170         <operation name="GetCurrentNegotiationsForTemplate"
                pattern="http://www.w3.org/2006/01/wsdl/in-out">
171             <input message="tns:
                    GetCurrentNegotiationsForTemplateRequest"/>
172             <output message="tns:
                    GetCurrentNegotiationsResponse"/>
173             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
174         </operation>
175
176         <operation name="JoinNegotiation" pattern="http://www.
                w3.org/2006/01/wsdl/in-out">
177             <input message="tns:JoinNegotiationRequest"/>
178             <output message="tns:JoinNegotiationResponse"/>
179             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
180             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
181         </operation>
182
183         <operation name="ProposeNegotiation" pattern="http://
                www.w3.org/2006/01/wsdl/robust-in-only">
184             <input message="tns:ProposeNegotiationRequest"/>
185             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
186         </operation>
187
```

```
188         <operation name="PublishNegotiation" pattern="http://
                www.w3.org/2006/01/wsdl/in-out">
189             <input message="tns:PublishNegotiationRequest"/>
190             <output message="tns:PublishNegotiationResponse"/>
191             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
192         </operation>
193 
194         <operation name="PublishNegotiationToReceipients"
                pattern="http://www.w3.org/2006/01/wsdl/in-out">
195             <input message="tns:
                    PublishNegotiationToReceipientsRequest"/>
196             <output message="tns:PublishNegotiationResponse"/>
197             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
198         </operation>
199     </interface>
200 </definitions>
```

Negotiation Participant:

Listing 10: WSDL 2.0: Negotiation Participant

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    name="NegotiationParticipant2.0"
    targetNamespace="http://j2ee.netbeans.org/wsdl/
        NegotiationParticipant2.0"
    xmlns:tns="http://j2ee.netbeans.org/wsdl/
        NegotiationParticipant2.0"
    xmlns:wsag="http://schemas.ggf.org/graap/2005/09/ws-
        agreement"
    xmlns:ns="http://xml.netbeans.org/examples/negotiation"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <!-- Type definitions -->
    <types>
        <xsd:schema>
            <xsd:import namespace="http://schemas.ggf.org/
                graap/2005/09/ws-agreement" schemaLocation="
                agreement_types.xsd"/>
            <xsd:import namespace="http://xml.netbeans.org/
                examples/negotiation" schemaLocation="
                NegotiationInstance.xsd"/>

            <xsd:complexType name="OffersType">
                <xsd:sequence>
                    <xsd:element name="negotiationID" type="
                        xsd:string"/>
                    <xsd:element name="offer" minOccurs="0"
                        maxOccurs="unbounded">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="agentID"
                                    type="xsd:anyType"/>
                                <xsd:element name="timeStamp"
                                    type="xsd:dateTime"/>
                                <xsd:element name="offer" type
                                    ="wsag:AgreementType"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:complexType>

            <xsd:simpleType name="InvalidInputMessageType">
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="syntacticalError"
                        />
                    <xsd:enumeration value="
                        inputParametersMissing"/>
                    <xsd:enumeration value="
                        referencedEntityNotFound"/>
                </xsd:restriction>
```

```
40             </xsd:simpleType>
41 
42             <xsd:simpleType name="AccessDeniedMessageType">
43                 <xsd:restriction base="xsd:string">
44                     <xsd:enumeration value="
                           coordinatorOnlyMethod"/>
45                     <xsd:enumeration value="
                           violationOfNegotiationRestriction"/>
46                 </xsd:restriction>
47             </xsd:simpleType>
48 
49             <xsd:complexType name="
                   ProposeNegotiationRequestType">
50                 <xsd:all>
51                     <xsd:element name="offeredNegotiation"
                           type="ns:NegotiationInstanceType"/>
52                 </xsd:all>
53             </xsd:complexType>
54 
55             <xsd:complexType name="
                   UpdateNegotiationRequestType">
56                 <xsd:all>
57                     <xsd:element name="updatedNegotiation"
                           type="ns:NegotiationInstanceType"/>
58                 </xsd:all>
59             </xsd:complexType>
60 
61             <xsd:complexType name="
                   AcceptNegotiationRequestType">
62                 <xsd:all>
63                     <xsd:element name="negotiationID" type="
                           xsd:string"/>
64                 </xsd:all>
65             </xsd:complexType>
66 
67             <xsd:complexType name="StartNegotiationRequestType
                   ">
68                 <xsd:all>
69                     <xsd:element name="negotiationID" type="
                           xsd:string"/>
70                 </xsd:all>
71             </xsd:complexType>
72 
73             <xsd:complexType name="PlaceOfferRequestType">
74                 <xsd:sequence>
75                     <xsd:element name="agentID" type="xsd:
                           string"/>
76                     <xsd:element name="offer" type="wsag:
                           AgreementType"/>
77                 </xsd:sequence>
78             </xsd:complexType>
79 
80             <xsd:complexType name="NewRoundRequestType">
81                 <xsd:sequence>
82                     <xsd:element name="negotiationID" type="
                           xsd:string"/>
```

```
83                    <xsd:element name="processInformation"
                          type="tns:OffersType"/>
84                 </xsd:sequence>
85             </xsd:complexType>
86
87             <xsd:complexType name="AcceptAgreementRequestType"
                  >
88                 <xsd:sequence>
89                     <xsd:element name="negotiationID" type="
                          xsd:string"/>
90                     <xsd:element name="agreementID" type="xsd:
                          string"/>
91                 </xsd:sequence>
92             </xsd:complexType>
93
94             <xsd:complexType name="RejectAgreementRequestType"
                  >
95                 <xsd:all>
96                     <xsd:element name="negotiationID" type="
                          xsd:string"/>
97                 </xsd:all>
98             </xsd:complexType>
99
100            <xsd:element name="ProposeNegotiationRequest" type
                  ="tns:ProposeNegotiationRequestType"/>
101            <xsd:element name="UpdateNegotiationRequest" type=
                  "tns:UpdateNegotiationRequestType"/>
102            <xsd:element name="AcceptNegotiationRequest" type=
                  "tns:AcceptNegotiationRequestType"/>
103            <xsd:element name="StartNegotiationRequest" type="
                  tns:StartNegotiationRequestType"/>
104            <xsd:element name="PlaceOfferRequest" type="tns:
                  PlaceOfferRequestType"/>
105            <xsd:element name="NewRoundRequest" type="tns:
                  NewRoundRequestType"/>
106            <xsd:element name="AcceptAgreementRequest" type="
                  tns:AcceptAgreementRequestType"/>
107            <xsd:element name="RejectAgreementRequest" type="
                  tns:RequestAgreementRequestType"/>
108            <xsd:element name="InvalidInputMessage" type="tns:
                  InvalidInputMessageType"/>
109            <xsd:element name="AccessDeniedMessage" type="tns:
                  AccessDeniedMessageType"/>
110
111        </xsd:schema>
112    </types>
113
114    <interface name="NegotiationParticipantInterface">
115        <operation name="ProposeNegotiation" pattern="http://
              www.w3.org/2006/01/wsdl/robust-in-only">
116            <input message="tns:ProposeNegotiationRequest"/>
117            <fault name="Invalid Input" message="tns:
                  InvalidInputMessage"/>
118        </operation>
119
```

```
120         <operation name="UpdateNegotiation" pattern="http://
                www.w3.org/2006/01/wsdl/robust-in-only">
121             <input message="tns:UpdateNegotiationRequest"/>
122             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
123             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
124         </operation>
125 
126         <operation name="AcceptNegotiation" pattern="http://
                www.w3.org/2006/01/wsdl/robust-in-only">
127             <input message="tns:AcceptNegotiationRequest"/>
128             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
129             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
130         </operation>
131 
132         <operation name="StartNegotiation" pattern="http://www
                .w3.org/2006/01/wsdl/robust-in-only">
133             <input message="tns:StartNegotiationRequest"/>
134             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
135             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
136         </operation>
137 
138         <operation name="PlaceOffer" pattern="http://www.w3.
                org/2006/01/wsdl/robust-in-only">
139             <input message="tns:PlaceOfferRequest"/>
140             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
141             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
142         </operation>
143 
144         <operation name="NewRound" pattern="http://www.w3.org
                /2006/01/wsdl/robust-in-only">
145             <input message="tns:NewRoundRequest"/>
146             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
147             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
148         </operation>
149 
150         <operation name="AcceptAgreement" pattern="http://www.
                w3.org/2006/01/wsdl/robust-in-only">
151             <input message="tns:AcceptAgreementRequest"/>
152             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
153             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
154         </operation>
155 
156         <operation name="RejectAgreement" pattern="http://www.
                w3.org/2006/01/wsdl/robust-in-only">
```

```
157                 <input message="tns:RejectAgreementRequest"/>
158                 <fault name="Invalid Input" message="tns:
                        InvalidInputMessage"/>
159                 <fault name="Access Denied" message="tns:
                        AccessDeniedMessage"/>
160             </operation>
161         </interface>
162 </definitions>
```

Information Service:

Listing 11: WSDL 2.0: Information Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    name="InformationService2.0"
    targetNamespace="http://j2ee.netbeans.org/wsdl/InformationService2.0"
    xmlns:tns="http://j2ee.netbeans.org/wsdl/InformationService2.0"
    xmlns:wsag="http://schemas.ggf.org/graap/2005/09/ws-agreement"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <!-- Type definitions -->
    <types>
        <xsd:schema>
            <xsd:import namespace="http://schemas.ggf.org/graap/2005/09/ws-agreement" schemaLocation="agreement_types.xsd"/>

            <xsd:complexType name="OffersType">
                <xsd:sequence>
                    <xsd:element name="negotiationID" type="xsd:string"/>
                    <xsd:element name="offer" minOccurs="0" maxOccurs="unbounded">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="agentID" type="xsd:anyType"/>
                                <xsd:element name="timeStamp" type="xsd:dateTime"/>
                                <xsd:element name="offeredAgreement" type="wsag:AgreementType"/>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:complexType>

            <xsd:simpleType name="InvalidInputMessageType">
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="syntacticalError"/>
                    <xsd:enumeration value="inputParametersMissing"/>
                    <xsd:enumeration value="referencedEntityNotFound"/>
                </xsd:restriction>
            </xsd:simpleType>

            <xsd:simpleType name="AccessDeniedMessageType">
```

```
41                  <xsd:restriction base="xsd:string">
42                      <xsd:enumeration value="
                            coordinatorOnlyMethod"/>
43                      <xsd:enumeration value="
                            violationOfNegotiationRestriction"/>
44                  </xsd:restriction>
45              </xsd:simpleType>
46
47              <xsd:complexType name="GetStatusRequestType">
48                  <xsd:all>
49                      <xsd:element name="negotiationID" type="
                            xsd:string"/>
50                  </xsd:all>
51              </xsd:complexType>
52
53              <xsd:complexType name="GetPastOffersRequestType">
54                  <xsd:all>
55                      <xsd:element name="negotiationID" type="
                            xsd:string"/>
56                  </xsd:all>
57              </xsd:complexType>
58
59              <xsd:complexType name="
                    GetPastOffersFromAgentRequestType">
60                  <xsd:sequence>
61                      <xsd:element name="negotiationID" type="
                            xsd:string"/>
62                      <xsd:element name="agentID" type="xsd:
                            anyType"/>
63                  </xsd:sequence>
64              </xsd:complexType>
65
66              <xsd:complexType name="GetStatusResponseType">
67                  <xsd:all>
68                      <xsd:element name="status" type="tns:
                            OffersType"/>
69                  </xsd:all>
70              </xsd:complexType>
71
72              <xsd:complexType name="GetPastOffersResponseType">
73                  <xsd:all>
74                      <xsd:element name="pastOffers" type="tns:
                            OffersType"/>
75                  </xsd:all>
76              </xsd:complexType>
77
78              <xsd:element name="GetStatusRequest" type="tns:
                    GetStatusRequestType"/>
79              <xsd:element name="GetPastOffers" type="tns:
                    GetPastOffersRequestType"/>
80              <xsd:element name="GetPastOffersFromAgent" type="
                    tns:GetPastOffersFromAgentRequestType"/>
81              <xsd:element name="GetStatusResponse" type="tns:
                    GetStatusResponseType"/>
82              <xsd:element name="GetPastOffersResponse" type="
                    tns:GetPastOffersResponseType"/>
```

```
83              <xsd:element name="InvalidInputMessage" type="tns:
                    InvalidInputMessageType"/>
84              <xsd:element name="AccessDeniedMessage" type="tns:
                    AccessDeniedMessageType"/>
85          </xsd:schema>
86      </types>
87
88      <interface name="InformationServiceInterface">
89          <operation name="GetStatus" pattern="http://www.w3.org
                /2006/01/wsdl/in-out">
90              <input message="tns:GetStatusRequest"/>
91              <output message="tns:GetStatusResponse"/>
92              <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
93              <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
94          </operation>
95
96          <operation name="GetPastOffers" pattern="http://www.w3
                .org/2006/01/wsdl/in-out">
97              <input message="tns:GetPastOffersRequest"/>
98              <output message="tns:GetPastOffersResponse"/>
99              <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
100             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
101         </operation>
102
103         <operation name="GetPastOffersFromAgent" pattern="http
                ://www.w3.org/2006/01/wsdl/in-out">
104             <input message="tns:GetPastOffersFromAgentRequest"
                    />
105             <output message="tns:GetPastOffersResponse"/>
106             <fault name="Invalid Input" message="tns:
                    InvalidInputMessage"/>
107             <fault name="Access Denied" message="tns:
                    AccessDeniedMessage"/>
108         </operation>
109     </interface>
110 </definitions>
```

B Erklärung

Ich erkläre hiermit gemäß § 27 Abs. 2 APO, dass ich die vorstehende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Bamberg, den 22. 09. 2006