AUTOMATING THE NEGOTIATION OF AGREEMENTS

A FRAMEWORK FOR DEVELOPING AUTOMATED NEGOTIATION SYSTEMS

MANUEL RESINAS

UNIVERSIDAD DE SEVILLA

DOCTORAL DISSERTATION
ADVISED BY DR. RAFAEL CORCHUELO

JUNE, 2008

First published in June 2008 by
The Distributed Group
ETSI Informática
Avda. de la Reina Mercedes s/n
Sevilla, 41012. SPAIN

Classification (ACM 1998): D.2.11 Software Architectures: Domain-specific architectures; D.2.13 Reusable Software: Domain engineering; I.2.11 Distributed Artificial Intelligence: Intelligent agents; H.4.2 Information Systems Applications: Decision support; K.4.4 Electronic Commerce: Distributed commercial transactions.

Support: Partially supported by the European Commission (FEDER) and Spanish Government under CICYT project AgilWeb (TIC-2000-1106-C02-01) and grant "Ayudas para el desarrollo de prácticas especializadas en estudios de posgrado", and by the University of Seville under grant "Plan Propio de movilidad". Complementary funding has been provided by the Spanish Government under grant TIN2007-64119 and the Andalusian Local Government under grant P07-TIC-02602.

Universidad de Sevilla

The committee in charge of evaluating the dissertation presented by Manuel Resinas in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering, hereby recommends ________________________ of this dissertation and awards the author the grade ____.

Isidro Ramos Salavert
Catedrático de Universidad
Univ. Politécnica de Valencia

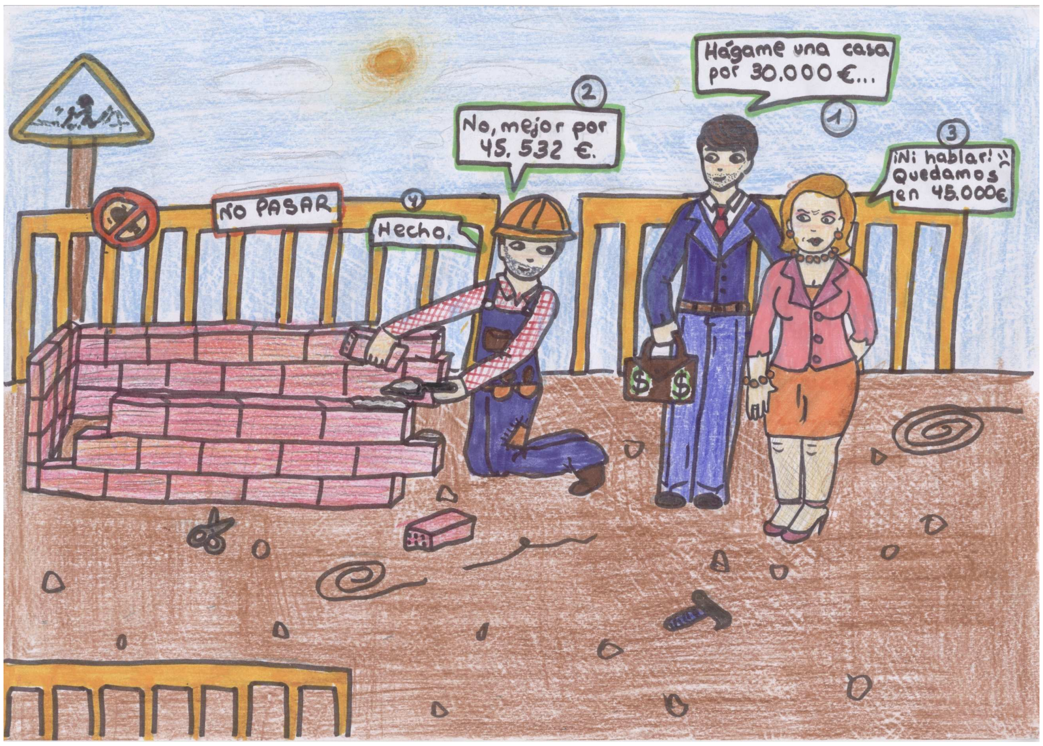
Miguel Toro Bonilla
Catedrático de Universidad
Univ. de Sevilla

Carlos Delgado Kloos
Catedrático de Universidad
Univ. Carlos III de Madrid

Mario G. Piattini Velthuis
Catedrático de Universidad
Univ. de Castilla-La Mancha

Carlos Molina Jiménez
Research Associate
Newcastle University

To put record where necessary, we sign minutes in ______________, ____________________.


NO PASAR
Hágame una casa por 30.000 €...
1
2
No, mejor por 45. 532 €.
3
¡Ni hablar! Quedamos en 45.000€
4
Hecho.

Negotiation by Ana, aged twelve.

To my parents.
To Beli.

Contents

List of Figures

List of Tables

Acknowledgements

Acknowledgement of one another's faults
is the highest duty imposed by our love of truth.

Ambrose G. Bierce, 1842–1914
Newspaper columnist

I am in no doubt that the best moment during the development of your Philosophiæ Doctor Thesis and the writing of your dissertation is when you finish them both. Not only because you reach the end of a hard way, but also because you can have a few minutes to relax, look back, and put your gratitude to many people in black ink.

Many individuals, friends and colleagues have been instrumental in making this dissertation a reality. Pivotal in this role was my research advisor, Dr. Rafael Corchuelo, since I would not have been able to finish my work without his help. Since he was the advisor of my master thesis, he has always trusted me and has taught me everything I know about research, which I believe it is the most important knowledge I have acquired while working on this dissertation.

The help of a good research team is of uttermost importance. I have been lucky since my thesis work has been developed in the bosom of a group of people, who have given me the encouragement and support I have needed since the beginning. Chiefly Pablo Fernandez, with whom I have shared so many working hours and whose arguments have provide me with valuable insights and feedback for this research work. I also would like to thank Dr. Miguel Toro for his cheerful willingness to proof-read and criticise this document. Finally, I shall never forget the breakfasts and lunches (and beers) with a list of colleagues and friends too long to mention, which made my long working hours more bearable.

Finally, and most important, I thank my parents for supporting me unconditionally in the pursuit of my life; and to Beli, without whose love and support, not a word would have been written.

Abstract

You will have only an opportunity
to make a first impression.

Popular saying

One of the major achievements of the Service-Oriented Architecture (SOA) and the Business Process Management (BPM) initiatives is providing the reference models and technologies that are necessary to help companies bridge the gap between businesses and IT, which is commonly referred to as the Business/IT misalignment. In this scenario, real-world business processes require to search, assess, and select the best available supply chain service providers, which, more often than not requires to negotiate with them the terms of a service agreement that regulates the properties and guarantees regarding the service provisioning. In this context, our research hypothesis is that the automation of service agreement negotiations shall improve the service provision process as a part of the business' supply chain, mainly in open and dynamic markets in which the flexibility in business relationships, the openness to new providers and the best exploitation of a company's resources are key aspects in the business management strategies.

Taking this hypothesis as a starting point, it follows that the development of automated negotiation systems may be a key point of the IT infrastructure of many companies, chiefly, but not limited to, those that are interested on delivering their services via the Internet. Unfortunately, developing such automated negotiation systems is a challenging task because of the following reasons: first, service negotiations involves many terms such as response time, security features or availability; second, in open environments, parties are heterogeneous, which means that automated negotiation systems must adapt to many different scenarios; third, automated negotiation systems must be able to negotiate with partial information about the other parties; and, fourth, service negotiations are usually take place in highly dynamic markets in which there are several providers and consumers at the same time and where supply and demand requirements may change dynamically depending on the availability of resources.

The goal of this dissertation is to support the thesis that it is convenient to develop a software framework that provides engineering support to make the development of automated negotiation systems easier in the context of negotiating service agreements in open and dynamic environments.

To this end, we present a strong motivation for this idea; we describe the problems that appear when negotiating in such environments; we detail why current approaches to build automated negotiation systems are not appropriate for service negotiations in open and dynamic environments; and, last but not least, we present NegoFAST, which is our approach to build such automated negotiation systems. NegoFAST is defined at three levels of abstraction: a reference architecture, a software framework and a proof-of-concept implementation. To enhance its reusability, it has been divided into a protocol-independent part, NegoFAST-Core, which is common to all negotiation protocols, and different protocol-specific extensions. In this dissertation, we have developed a bargaining-specific extension, NegoFAST-Bargaining, although other different extensions can be implemented. Furthermore, NegoFAST provides direct support to deal with the aforementioned problems of service negotiations in open and dynamic environments, which makes our proposal a novel, original contribution.

Resumen

Sólo tendrás una oportunidad
de dar una primera impresión.

Dicho popular

Uno de los mayores logros de las Arquitecturas Orientadas a Servicios (SOA) y la Gestión de Procesos de Negocio (BPM) es el de proporcionar los modelos de referencia y tecnologías que son necesarias para reducir la brecha entre el mundo empresarial y las Tecnologías de la Información (TI). En este escenario, los procesos de negocio requieren la búsqueda, asesoramiento y selección de los mejores proveedores de servicios disponibles. Este hecho requiere a menudo la negociación con los proveedores de servicio de los términos de un acuerdo de servicio que regula las propiedades y garantías del mismo. En este contexto, nuestra hipótesis de investigación es que la automatización de las negociaciones de acuerdo de servicio mejorará el proceso de aprovisionamiento del mismo como parte de la cadena de suministro del negocio. Estas ventajas se maximizarán en mercados abiertos y dinámicos en donde la flexibilidad en las relaciones de negocio, la apertura a nuevos proveedores y la obtención del máximo rendimiento posible a los recursos de una empresa son aspectos fundamentales dentro de las estrategias empresariales.

Tomando esta hipótesis como punto de partida, se deduce que el desarrollo de sistemas de negociación automática puede ser un aspecto clave de la infraestructura de Tecnologías de la Información de muchas empresas. Principalmente, aquellas que están interesadas en ofrecer sus servicios a través de Internet. Desafortunadamente, el desarrollo de sistemas de negociación automática es una tarea compleja por las siguientes razones: primero, las negociaciones de servicio incluyen la negociación de múltiples términos como el tiempo de respuesta, características de seguridad o disponibilidad; segundo, en entornos abiertos, las partes con las que se negociará son heterogéneas, lo que significa que los sistemas de negociación automática deben adaptarse a múltiples escenarios; tercer, los sistemas de negociación automática deben ser capaces de negociar con información parcial sobre las otras partes, y, cuarto, las negociaciones de servicio suelen tener lugar en mercados muy dinámicos, en los que hay muchos proveedores y consumidores al mismo tiempo y donde los requisitos de la oferta y la demanda cambian dinámicamente dependiendo de la disponibilidad de los recursos.

El objetivo de esta tesis doctoral es apoyar la idea de que es conveniente el desarrollo de un *framework* software que proporcione soporte desde el punto de vista de la ingeniería para el desarrollo de sistemas de negociación automática en el contexto de la negociación de acuerdos de servicio en entornos abiertos y dinámicos.

Con este fin, en esta memoria presentamos una motivación bien fundamentada de esta idea; describimos los problemas que aparecen en este tipo de sistemas al negociar en estos entornos; detallamos por qué las aproximaciones actuales al desarrollo de sistemas de negociación automática no son apropiadas para las negociaciones de servicios en entornos abiertos y dinámicos, y, por último, presentamos NegoFAST, que es nuestra aproximación al desarrollo de estos sistemas de negociación automática. NegoFAST está definido en tres niveles de abstracción: una arquitectura de referencia, un *framework* software y una implementación en forma de prueba de concepto. Para incrementar su reusabilidad, NegoFAST se ha divido en una parte independiente del protocolo, NegoFAST-Core, que es común a todos los protocolos de negociación, y extensiones específicas del protocolo. En este trabajo de investigación, hemos desarrollado una extensión específica para negociaciones basadas en el intercambio de oferta y contraoferta, NegoFAST-Bargaining. Sin embargo, el sistema está abierto a otras posibles extensiones. Con todo esto, NegoFAST proporciona soporte directo para tratar con los problemas, mencionados anteriormente, sobre las negociaciones de servicio en entornos abiertos y dinámicos, lo que hace de nuestra propuesta una contribución original.

Part I
Preface

Chapter 1
Introduction

There is nothing more difficult to take in hand,
more perilous to conduct, or more uncertain in its success,
than to take the lead in the introduction
of a new order of things.

Niccolo Machiavelli, 1469–1527
Italian dramatist, historian, and philosopher

In this dissertation, we report on our work to design a new framework that helps to build automated negotiation systems that are able to negotiate service agreements in open and dynamic environments. In this chapter, we first introduce the elements that constitute the context of our research work in Section §1.1; we then detail the motivation and goals of this dissertation in Section §1.2; next, we summarise our main contributions in Section §1.3; finally, we describe the structure of the dissertation in Section §1.4.

1.1 Research context

There is a well-know problem for Chief Information Officers that is commonly referred to as the Business/IT misalignment [19]. Roughly speaking, this problem can be stated as follows: *What can be done to map business needs onto IT capabilities?* Or, complementarily: *What can be done to realise business value from IT investments?* The root of this problem is the lack of communication between business and IT, which is usually caused because business and IT people speak very different languages. It is unusual that IT managers do not fully understand business or its methods and terminology, but they use techno-babble that is not commonly understood by business managers. As a result, it is hard to map business goals onto concrete IT initiatives and, hence, the results of IT are not as valuable as they could be [24].

In this context, one of the major achievements of the Service-Oriented Architecture (SOA) and the Business Process Management (BPM) initiatives has been to provide the reference models and technologies that are necessary to help companies bridge the gap between businesses and IT [16]. Particularly, the SOA initiative has emerged as a strategy for streamlining business delivery; complementarily, the BPM initiative has emerged as a strategy for streamlining supply chains. In other words, the SOA initiative provides technologies to enable companies to offer their services, *i.e.* their business delivery, whereas the BPM initiative provides technologies to make it explicit the business process that is necessary to provide a service, *i.e.* it focuses on the implementation of the supply chain [30].

Real-world business processes require to search, assess, and select the best available supply chain services providers, which, more often than not requires to negotiate the terms of a service agreement with them [18]. Automating the negotiation of agreements, so that the human participation in the process is reduced to a minimum shall bring a variety of advantages, *e.g.*, cutting the cost of reaching an agreement, increasing the speed in the contracting process, and allowing the establishment of new business relationships in a more flexible way [11, 22, 27, 39, 87, 122].

It is not surprising then that, with the advent of the Internet and electronic businesses [67, 113, 134], the need to automate such negotiation processes is becoming a must due to the general, broad availability of service providers and their need to keep their IT resources in use 24 hours a day, 7 days a week [32, 103]. Furthermore, current service-oriented architectures make automated negotiation of agreements a valuable resource even in intraorganisational environments. In a service-oriented approach, applications are developed by composing and invoking network services instead of programming new code. In this context, one of the main reasons to use service agreements is to improve the computing-resources planning inside an organisation, which argues for these service agreements to be managed as automatically as possible, including its creation through a negotiation process.

The first attempts to automate negotiation processes date back to the early 80s, being the ContractNet protocol [126] one of the most significant proposals to define an interaction protocol between the negotiating parties. Since then, much work has been done on developing algorithms, protocols and models that have desirable characteristics for automated negotiations [64]. More recently, in the web services world, WS-Negotiation [62] and WSLA [33] were the first attempts to tackle negotiations of web services agreements. However, the former allow only simple interactions, and the latter was abandoned to work in another specification: WS-Agreement [4]. Currently, WS-Agreement, which is a proposed recommendation of the Open Grid Forum, has emerged as the most significant specification for the creation of service agreements. The goal of WS-Agreement is threefold: defining a document format for the agreements, establishing a concrete agreement creation process and describing the communication interfaces that must be implemented by the parties involved in the agreement creation process. However, the negotiation capabilities of WS-Agreement are very limited. To solve this problem, an extension to enable negotiations, the so-called WS-AgreementNegotiation is being developed.

In this dissertation, we do not intend to tackle all types of automated negotiations, but we focus on automated negotiations of service agreements in open and dynamic environments. These automated negotiations have several characteristics that must be taken into consideration. First, service agreements involve multi-term negotiations, *i.e.*, although there are negotiations that only involve one term (usually the price), there are many others that involve the negotiation of many terms before reaching an agreement. Second, negotiating in open environments means that the parties with which our system negotiates may change over time. Note that this includes not only interorganisational environments, but also highly distributed intraorganisational environments such as grid services in large organisations, in which services are distributed amongst different departments and they provide services to the other parts of the organisation. And third, negotiating in dynamic markets, which is common with Internet-based businesses, involves that there are several providers and consumers at the same time and where supply and demand may change dynamically depending on the resources availability.

Since not all negotiation mechanisms are well suited to deal with all types of negotiations [64], in this dissertation, we just focus on those that are more useful for our purposes. This is particularly relevant when it comes to the negotiation protocol and their support for multi-term negotiations. Generally speaking, negotiation protocols can be categorised into auctioning and bargaining [61]. However, although auctions are very useful in many scenarios, they are not well suited to deal with agreements with multiple terms: most typical auctions just allow for the negotiation of one term, the price, and, although multi-attribute auctions allow for the negotiation of agreements with multiple terms, they require a procedure to guide the bidder in the process of improving the bid (*cf.* Appendix §D). This procedure may involve the disclosure of their own preferences, which may take the auctioneer to a weak negotiation position [108]. As a consequence, in this dissertation we just focus on bargaining protocols. Nevertheless, we acknowledge the importance of auctions; and we consider that the support for auctions requires further research work in future.

1.2 Research rationale

From the previous section we conclude the following hypothesis:

The automation of service agreement negotiations shall improve the service provision process as a part of the business' supply chain, mainly in open and dynamic markets in which the flexibility in business relationships, the openness to new providers and the best exploitation of a company's resources are key aspects in the business management strategies.

Taking this hypothesis as a starting point, it follows that the development of automated negotiation systems may be a key point of the IT infrastructure of many companies, chiefly, but not limited to, those that are interested in delivering their services via the Internet.

Unfortunately, developing such automated negotiation systems is not an easy task. It is then not surprising that very few such systems exist. Automating the negotiation of service agreements in open and dynamic environments raises several problems. On the one hand, creating service agreements involves negotiating many terms such as response time, security features or availability, not only the price as is the case of many negotiations of goods. On the other hand, negotiating in open and dynamic environments requires the automated negotiation systems to be able to deal with heterogeneous parties, which is common in open environments; to negotiate with partial information about the other parties, and to cope with dynamic ever-changing markets in which supply and demand may change quickly depending on the resources available.

Authors have followed three approaches to build automated negotiation systems: ad-hoc solutions [20, 43, 44, 69, 77, 79, 87, 104, 129], protocol-oriented frameworks [9, 75, 114] and intelligence-oriented frameworks [6, 26, 55, 74, 84, 136]. However, none of these proposals are appropriate to build systems that negotiate automatically service agreements in open environments. Ad-hoc solutions lack the reusability and adaptability that are necessary to deal with heterogeneous parties in open environments. Protocol-oriented frameworks only focus on the negotiation protocol and, hence, need to be complemented with negotiation intelligence capabilities by means of ad-hoc mechanisms or an intelligence-oriented framework. Finally, current intelligence-oriented frameworks are not well suited to deal with partial information about parties and dynamic markets.

As a consequence, our thesis is that:

It is convenient to develop a software framework that provides engineering support to make the development of automated negotiation systems easier in the context of negotiating service agreements in open and dynamic environments.

1.3 Summary of contributions

On the road towards making the development of automated negotiation systems easier, we have made the following contributions:

First, we have made a thorough analysis of the state of the art in negotiation rudiments, protocols and intelligence algorithms. The novelty of this analysis is its focus on bridging the gap between the many different terminologies and techniques useful in this field within a coherent comparison framework.

Second, we have analysed the problems of negotiating service agreements in open and dynamic environments and we have elicited the requirements that such negotiation context poses on the automated negotiation systems. Then, we have compared current approaches to build automated negotiation systems with them and concluded that they are not usually well suited to be applied to the development of automated negotiation systems able to negotiate in such environments at a sensible cost.

Third, we have developed NegoFAST, which is the main result of this dissertation. NegoFAST is our approach for developing automated negotiation systems, and it is defined at three levels of abstraction. The first one is the NegoFAST reference architecture, which provides a foundation for developing automated negotiation systems. It is divided into a protocol-independent reference architecture, the so-called NegoFAST-Core, and protocol-specific extensions. In this dissertation, we describe a bargaining-specific extension, the so-called NegoFAST-Bargaining. This allows us to deal with the different requirements posed by different protocols while keeping the reusability of the other elements of the reference architecture. The second level of abstraction is the NegoFAST framework, which provides an instantiation of the NegoFAST reference architecture by defining the interfaces and protocols amongst the elements defined in the reference architecture and the data structures they require. Its design goal has been maximising the reusability and extensibility of the framework. And the third level of abstraction is August, which is a reference implementation of the NegoFAST framework that has been used to implement several use cases that are described in the appendices of this dissertation.

1.4 Structure of this dissertation

This dissertation is organised as follows:

Part I: Preface. It comprises this introduction and Chapter §2, in which we motivate our research and conclude that current solutions are not practical enough.

Part II: Background information. Here, our goal is to provide the reader with a deep understanding of the research context in which our results have been developed, but without evaluating the different proposals. The evaluation is carried out in Chapter §2. In Chapter §3, we detail the negotiation rudiments that are the basis of all negotiation processes including different models to express agreements and preferences and the most significant negotiation protocols. In Chapter §4, we describe several approaches that have been presented to implement the decision-making and the world-modelling aspects of an automated negotiation system. In Chapter §5, we analyse the approaches that have been proposed to build automated negotiation systems and classify them.

Part III: Our approach. This is the core of our dissertation and it is organised into five chapters. In Chapter §6, NegoFAST is presented as a general approach to build such automated negotiation systems and its three levels of abstraction are outlined. In Chapter §7, we build on the Gaia methodology to detail the NegoFAST-Core reference architecture in terms of roles, interactions and environmental resources. In Chapter §8, we follow a similar approach by detailing the NegoFAST-Bargaining reference architecture, which is an extension of the NegoFAST-Core reference architecture for bargaining protocols. In Chapter §9, we specify the NegoFAST-Core framework, which refines the NegoFAST-Core reference architecture and provides a data model, the interfaces of the environmental resources and the roles, and the state machines of the coordination roles. Similarly, in Chapter §10, we describe the NegoFAST-Bargaining framework, which refines the NegoFAST-Bargaining reference architecture to extend the NegoFAST-Core framework to support bargaining protocols.

Part IV: Final remarks. It consists of one chapter in which we report on our main conclusions and future research directions.

Part V: Appendices. We illustrate the instantiation of the framework by describing the development of several August-based automated negotiation systems that implement different use cases in Appendix §A, Appendix §B and Appendix §C. In addition, we analyse auction-based protocols in Appendix §D.

Chapter 2
Motivation

Motivation will almost always beat mere talent.

Norman R. Augustine, 1935–
Chairman of Lockheed Martin Corporation

Although current work on automated negotiation provides an excellent background to build automated negotiation systems, it is still necessary to put it all together in a framework that provides engineering support to build automated negotiation systems of service agreements in open and dynamic environments. Our goal in this chapter is to present the problems that this scenario pose to automated negotiation systems; to detail to which extent current solutions deal with these problems, and to motivate the need for a new solution. It is organised as follows: in Section §2.1, we introduce the chapter; in Section §2.2, we present these problems in detail; in Section §2.3, we conclude that none of the current solutions solve these problems at a time; in Section §2.4, we discuss our results and compare them with current solutions; finally, Section §2.5 summarises the main ideas in this chapter.

2.1 Introduction

In the last years, much work has been done on automated negotiation. This work can be divided into five broad categories, namely: the agreement model (*cf.* Section §3.2), *i.e.*, the negotiation object and how it is expressed; the preferences model (*cf.* Section §3.3), *i.e.*, how to express the user preferences about the terms of the agreement; the negotiation protocol (*cf.* Section §3.4), *i.e.*, which are the rules that govern how the parties communicate with each other; the decision making (*cf.* Section §4.2), *i.e.*, which is the behaviour of the automated negotiation system during the negotiation or, in other words, which strategy it follows; and the world modelling method (*cf.* Section §4.3), *i.e.*, which models must be created about the opponents or the market to support the decision-making process. Agreement model, preferences model, and negotiation protocol are determined by the user of the automated negotiation system and the other parties, whereas decision making and world modelling constitute the negotiation intelligence of an automated negotiation system.

This work on automated negotiation chiefly focus on the development of new models, algorithms or protocols that have a number of desirable characteristics for automated negotiations, and it provides an excellent background to build automated service agreement negotiation systems. Unfortunately, the development of such systems is a challenging task because they must face the problems that are depicted in Table §2.1 and described in the next section, namely: negotiations are multi-term, parties are heterogeneous, we only have partial information about them, and markets are dynamic.

Authors have followed different approaches to build such automated negotiation systems (ad-hoc solutions, protocol-oriented frameworks, and intelligence-oriented frameworks). However, as we discuss later in this chapter, no solution seems to be appropriate enough because they do not address successfully all problems at a time. Consequently, it is still necessary to put all previous work in automated negotiations together in a framework that provides engineering support to build automated negotiation systems. This is our proposal in this dissertation.

2.2 Problems

Automated negotiation systems must cope with the problems we describe in this section when facing automated negotiations of services in open environments, *cf.* Table §2.1.

Problem	Hints
Negotiations are multi-term	(1.1) Support multi-term negotiation protocols
	(1.2) Manage expressive preferences models
Parties are heterogeneous	(2.1) Support multiple protocols
	(2.2) Negotiate the negotiation protocol
	(2.3) Support multiple negotiation intelligence algorithms
	(2.4) Support multiple agreement models
	(2.5) Support multiple preferences models
	(2.6) Allow user preferences about the negotiation process
Partial information about parties	(3.1) Manage different types of knowledge about parties
	(3.2) Gather information from different sources
	(3.3) Build analysis-based models
Markets are dynamic	(4.1) Support several simultaneous negotiations
	(4.2) Select intelligence algorithms dynamically
	(4.3) Support decommitment
	(4.4) Supervised creation of agreements
	(4.5) Build market models

Table 2.1: *Summary of problems automated negotiation systems must face.*

2.2.1 Negotiations are multi-term

Although there are negotiations that only involve one term (usually the price), negotiations of service agreements usually involves many terms such as availability, response time, security or price. However, not all negotiation protocols allow multi-term negotiations (*cf.* Appendix §D). This enables the negotiating parties to make trade-offs amongst the terms. Therefore, it would be desirable for an automated negotiation system to:

(1.1) Support multi-term negotiation protocols: The negotiation protocol must support the negotiation of multiple terms. This is the case for most bargaining protocols. However, is not so usual for other negotiation protocols such as auctioning (*cf.* Appendix §D).

(1.2) Manage expressive preferences models: As the number of terms increases, the search space for a mutual agreements becomes larger too, making it harder to find an agreement. This requires parties to use advanced negotiation intelligence algorithms in order to avoid excessively long negotiations that may finish without reaching an agreement. These algorithms require preferences to capture relations between terms and, hence, enable making trade-offs during negotiations.

2.2.2 Parties are heterogenous

In open environments, such as the Internet, we cannot make any assumptions about the negotiating parties. In other words, parties may implement a great variety of negotiation protocols, have very diverse behaviours during the negotiation (*e.g.*, some parties can concede more at the beginning of the negotiation whereas others may concede only when the deadline is approaching) and express preferences and agreements following different models. Furthermore, the best negotiation protocol, behaviour, preferences model and agreement model depends on the concrete scenario and the parties with which the automated negotiation system is negotiating [64]. To adapt to this variability, it would be desirable for an automated negotiation system to:

(2.1) Support multiple negotiation protocols: Since there is no standard negotiation protocol, different parties may implement different negotiation protocols. Furthermore, there is no best negotiation protocol, but it depends on the concrete situation [64]. Therefore, an automated negotiation system should support several negotiation protocols to avoid losing business opportunities because parties do not share the same protocol.

(2.2) Negotiate the negotiation protocol: Since the best negotiation protocol depends on the concrete situation and parties may implement different negotiation protocols, it is convenient that the automated negotiation system supports a pre-negotiation phase, in which the negotiation protocol is selected.

(2.3) Support multiple negotiation intelligence algorithms: It is convenient that the automated negotiation system supports several negotiation intelligence algorithms to face the different behaviours that may present the other parties during the negotiation. The reason is that the effectiveness of a negotiation intelligence algorithms depends, amongst other things, on the behaviour of the other parties. Furthermore, negotiation intelligence algorithms depend on a particular way of expressing agreements and preferences (*cf.* Section §4.2). Thus, simpler and more efficient algorithms can be used when the user does not require a complex model to express preferences and agreements.

(2.4) Support multiple agreement models: Agreements can be expressed using different models, *cf.* Section §3.2, and parties may choose to express agreements in a variety of models depending on the concrete situation. Furthermore, supporting multiple agreement models enables the user to make a trade-off between the expressiveness it requires and the availability and complexity of the corresponding negotiation algorithms.

(2.5) Support multiple preferences models: Like agreements, there are different models to express preferences as detailed in Section §3.3 and parties may choose one model or another depending on the situation. Particularly, two aspects must be taken into account to make this selection: the negotiation domain and the influence the preferences model has on the negotiation intelligence algorithms. The reason is that, depending on the negotiation domain, it is better to express preferences in a different way (*e.g.*, depending on whether the terms under negotiation are interrelated or not); furthermore, like agreements, a trade-off should be made between the preferences model expressiveness and the availability and complexity of the negotiation algorithms that deal with them.

(2.6) Allow user preferences about the negotiation process: Not all parties have the same preferences about the negotiation process. Some parties may have a shorter deadline, may be eager to reach an agreement or may be less strict about the agreements it accepts. Therefore, an automated negotiation system must allow users to set their preferences about the negotiation process.

2.2.3 Partial information about parties

The knowledge about a party is important to strengthen our negotiation capabilities [87, 139]. However, automated negotiation systems usually have only partial information about them [47] either because parties are not known *a priori*, or because they do not disclose any information about them since it can weaken their negotiation capabilities [87]. Therefore, it would be desirable for an automated negotiation system to:

(3.1) Manage different types of knowledge about parties: In particular, an automated negotiation system can manage knowledge about the service the other party wish to provide or consume (*i.e.* the functional and non-functional characteristics), knowledge about the other party itself (*e.g.* its reputation or its geographical situation), and knowledge about the behaviour of the party during the negotiation process (*e.g.* its temporal constraints) [111].

(3.2) Gather information from different sources: An automated negotiation system should be able to acquire information from several sources. In particular, they may query information directly to the other party (*e.g.* as a template that should be filled [4]). Furthermore, they may query third parties to obtain information related to another party (*e.g.* an automated negotiation system can ask a third party for the reputation of another party).

(3.3) Build analysis-based models of parties: An automated negotiation system can analyse previous interactions with other parties to build models and, later, use them to make better decisions during the negotiation process (*cf.* Chapter §4).

2.2.4 Markets are dynamic

It is likely that, for a single service, there are several providers and consumers competing against each other. Furthermore, these service markets can be extremely dynamic because services are not storable, which means that the idleness of a resource results in a loss of revenue [55], and, hence, providers may lower the cost of their services when their resources are idle. To deal with these changing markets, an automated negotiation system should:

(4.1) Support several simultaneous negotiations: This is desirable since it would allow so the automated negotiation system to choose the party that offers the most profitable agreement.

(4.2) Select negotiation intelligence algorithms dynamically: When dealing with simultaneous negotiations, the state of the negotiations can have an influence on the negotiation intelligence algorithms employed in a particular negotiation. Therefore, it is desirable to be able to change dynamically the negotiation intelligence algorithms to adapt to different situations. For instance, if the automated negotiation system is carrying out several simultaneous negotiations and finds a very profitable agreement, it can negotiate tougher with with the other parties.

(4.3) Support decommitment from established agreements: It may be very convenient, in dynamic markets, to be able to revoke previous agreements, possibly after paying some compensation to the other party [118].

(4.4) Supervised creation of agreements: To avoid committing to agreements that cannot be satisfied, the automated negotiation system should be supervised by external elements such as a capacity estimator to determine whether an agreement can be accepted or not [85].

(4.5) Build market models: The characteristics of the market may have an influence on the negotiation process [123]. Therefore, it is convenient for an automated negotiation system to build models of the market to obtain information such as the reservation price of a product or the probability of appearing new candidate parties during the negotiation [81].

2.3 Analysis of current solutions

Our goal in this section is to prove that none of the current solutions to build automated negotiation systems addresses the aforementioned problems at a time. In the literature, there are a few international specifications that deal with some parts of a negotiation process. Furthermore, there are three approaches to build automated negotiation systems: ad-hoc automated negotiation systems, protocol-oriented frameworks and intelligence-oriented frameworks (*cf.* Chapter §5 for a detailed description).

2.3.1 Negotiation-related specifications

The most significant specifications that have tried to standardise some parts of a negotiation process are three: FIPA protocols [49–52], OMG negotiation facility [102] and WS-Agreement [4]. Currently, the WS-Agreement specification is being extended to support negotiation (WS-AgreementNegotiation) and renegotiation, although these extensions are still at a very early stage of development. These standardisation efforts focus on facilitating the interaction amongst the negotiating parties and, more recently, they have also dealt with the structure of the agreement document.

FIPA protocols are specifications of several negotiation protocols that include Dutch [49] and English auctions [50], Contract Net [52] and iterated Contract Net [51]. They are part of the Agent Communication Language Specifications and build on other related specifications such as the FIPA Communicative Act Library Specification, which specifies the performatives that can be used in the negotiation protocols and the FIPA Content Language Specifications, which deal with different representations of the content of messages. However, the structure of agreements and negotiation messages is not detailed. Furthermore, although FIPA describes an abstract architecture for agents, this abstract architecture is generic and do not deal with the specific problems of automated negotiation systems.

The OMG Negotiation Facility [102] is a specification that builds on CORBA. It defines a protocol for bilateral negotiation and another protocol for multilateral negotiation; a language to define such protocols, and a framework that defines a processor for that language. However, the OMG Negotiation Facility does not detail the language and structure of the negotiation messages and agreements and the framework just provides support to implement the negotiation protocol, but it does not deal with providing support for the negotiation intelligence part of an automated negotiation system.

Unlike the previous specifications, WS-Agreement [4] is gaining importance and it shall likely become the *de facto* standard. The WS-Agreement specification can be divided into several parts: it specifies the structure of an agreement document, although it must be composed with several domain-specific vocabularies to give proper semantics to the terms of the agreement, and it defines a protocol and a web service-based interface to create, represent, and allow the monitoring of agreements. However, WS-Agreement only defines a take-it-or-leave-it protocol. To use more complex negotiation protocols, some extensions are required, such as the aforementioned WS-AgreementNegotiation. Furthermore, WS-Agreement does not deal with the internals of automated negotiation systems.

2.3.2 Ad-hoc automated negotiation systems

Ad-hoc automated negotiation systems [20, 43, 44, 69, 77, 79, 87, 104, 129] are designed to negotiate in specific scenarios. They usually implement only a negotiation protocol and a concrete set of negotiation intelligence algorithms. Furthermore, they only support a concrete model to express agreements and preferences (*cf.* Section §5.2 for more information about ad-hoc automated negotiation systems). The problem here is due to the heterogeneity of parties in open environments such as the Internet. A single ad-hoc automated negotiation system cannot fulfil all requirements these heterogeneous parties may pose. Moreover, since they are wired to the concrete negotiation techniques they use, it is hard to implement new negotiation protocols or negotiation intelligence algorithms without changing the other parts of the system.

2.3.3 Automated negotiation frameworks

Unlike, ad-hoc automated negotiation systems, automated negotiation frameworks focus on the reusability of the different parts of an automated negotiation system. As a consequence, a good software framework must be designed with a clear separation of concerns in mind. The separation of concerns indicates how independent each part of an automated negotiation system is from the others. A clear separation of concerns eases the addition of new aspects of the automated negotiation system such as negotiation protocols or intelligence algorithms without changing the other parts of the system. Therefore, automated negotiation frameworks must be analysed in terms of the support they provide to deal with the aforementioned problems and the separation of concerns they present.

Table §2.2 depicts how current negotiation frameworks deal with the problems that service negotiations in open and dynamic environments pose (*cf.* Section §2.2): a + sign in the table means that the proposal provides explicit support for the corresponding characteristics; a ∼ sign indicates that it addresses it partially; a blank indicates that the proposal does not support the characteristic; and NA means the information is not available.

Similarly, Table §2.3 depicts the separation of concerns of the aforementioned frameworks. Specifically, we detail the support the framework provides to decouple the five parts in which an automated negotiation system can be divided, namely: agreement model, preferences model, negotiation protocol, decision making and world modelling. For each part, we consider three degrees of coupling: loose coupling (depicted as a + sign) when the framework explicitly provides mechanism that makes a part easily changeable; medium coupling (indicated by a ∼ sign) when the framework does not explicitly provide a mechanism to change a part, but it seems not hard to do it; and tight coupling (indicated by a blank) when either the characteristics of the framework make it hard to change the part or the framework does not provide support for it (*e.g.* decision making in protocol-oriented frameworks). Finally, NA means that no information is available.

There are two kinds of negotiation frameworks depending on the parts of the negotiation system they wish to reuse: protocol-oriented frameworks and intelligence-oriented frameworks (*cf.* Section §5.3 for more information).

Protocol-oriented frameworks: They focus on the reusability of the different parts of an automated negotiation system and deal with the negotiation protocol and the problems of interoperability amongst them (*cf.* Section §5.3.1

Proposal	**1.1**	**1.2**	**2.1**	**2.2**	**2.3**	**2.4**	**2.5**	**2.6**	**3.1**	**3.2**	**3.3**	**4.1**	**4.2**	**4.3**	**4.4**	**4.5**
Protocol-oriented frameworks																
Kim and Segev [75]	+		+													
Rinderle and Benyoucef [114]	+		+													
Bartolini and others [9]	+		+													
Silkroad [128]	+		+													
Intelligence-oriented frameworks																
Ashri and others [6]	+	NA	+		NA		NA	NA	+		+					
Ludwig and others [84]	+	+	+		+					~						
PANDA [55]	+	+	+		+			+	+		~	+			+	
DynamiCS [136]	+	NA	+		+		NA	NA								
Benyoucef and Verrons [12]	NA	NA	+		+		NA	NA	NA	+	~		+		+	
Jonker and others [66]	+	+	NA		+			+		+	+					

Table 2.2: *Comparison of automated negotiation frameworks.*

for more information about protocol-oriented frameworks). However, they do not cover anything related to the development of the negotiation intelligence in the automated negotiation systems themselves. As a consequence, current protocol-oriented frameworks [9, 75, 114, 128] only deal with problems such as the support for multi-term protocols and for multiple negotiation protocols. To face the other problems, they need to be complemented with ad-hoc implementations or with an intelligence-oriented framework. Therefore, our work is closely related to protocol-oriented frameworks, but our focus is different since we are also concerned with the negotiation intelligence part of an automated negotiation system.

Intelligence-oriented frameworks: They focus on improving the reusability of automated negotiation systems by defining a software framework that supports implementing several negotiation intelligence algorithms and protocols. Intelligence-oriented frameworks [6, 12, 55, 84, 136] differ in their level of abstraction and how they support decision making and world modelling, as well as the number of negotiation protocols they support. They face some of the problems we describe in the previous section but none of them copes with all of them at the same time.

Ashri and others [6] describe two architectures for negotiating agents. However, the architecture is described from an abstract point of view. In addition, it lacks some advanced features. For instance, it does not cover a pre-negotiation phase, it does not consider information queries, it is not well suited for supporting concurrent negotiations, and it does not take into account external factors in the binding decision.

Ludwig and others [84] have developed a framework for automated negotiation of service level agreements in service grids. This framework builds on WS-Agreement and provides a protocol service provider and a decision making service provider to deal with the negotiation process. However, it does not acquire knowledge from the other parties (neither querying nor analysing previous interactions); the separation of concerns is only reached partially because all decision-making is implemented in one unique service and, hence, it makes it less reusable; and its capabilities to face a dynamic market are very limited because it does not support several negotiations simultaneously, does not create market models, does not support decommitment, and does not allow changing the negotiation intelligence algorithms dynamically.

PANDA [55] is a framework that mixes utility functions and rules to carry out the decision-making process. The decision-making component is composed of rules, utility functions and an object pool with several estimation libraries, the negotiation history and the current offer. This framework presents

Proposal	Agreement	Preferences	Protocol	Decision making	World modelling
Protocol-oriented frameworks					
Kim and Segev [75]	~		+		
Rinderle and Benyoucef [114]			+		
Bartolini and others [9]			+		
Silkroad [128]	+		+		
Intelligence-oriented frameworks					
Ashri and others [6]	NA	NA	+		
Ludwig and others [84]			+	+	
PANDA [55]			+	+	+
DynamiCS [136]	NA	NA	+	+	
Benyoucef and Verrons [12]	NA	NA	+	+	~
Jonker and others [66]	NA	NA	+	+	~

Table 2.3: *Separation of concerns in automated negotiation frameworks.*

interesting capabilities specially when dealing with dynamic markets. However, the object pool, which is an important element because it deals with all the knowledge the framework has about the other parties and the market, is not implemented, but only vaguely specified. Furthermore, it does not support querying other parties (neither third parties nor the other party directly) to get information; it does not provide automatic mechanisms to change the negotiation intelligence algorithms dynamically; neither it does support decommitment from previous agreements nor supports a pre-negotiation phase in which the negotiation protocol is decided.

DynamiCS is an actor-based framework developed by Tu and others [136]. This framework makes a clear distinction between the negotiation protocol and the decision making model and it uses a plug-in mechanism to support new protocols and strategies. Nevertheless, it is not well suited to deal with parties that are not known previously because querying information to other parties is not supported and it does not create models based on the analysis of previous interactions. In addition, the framework cannot change the negotiation intelligence algorithms dynamically. Furthermore, DynamiCS has problems when coping with dynamic markets because it does not support decommitment, does not build market models, and the creation of agreements cannot be supervised.

Benyoucef and Verrons [12] present an approach to a configurable framework for designing, implementing and deploying automated negotiation systems. Their approach is based on the separation of protocols and strategies from the system and they adopt a service-oriented architecture to make it easier the deployment and integration with current infrastructures. However, the authors do not provide a data model for the framework and do not provide any detail on the preferences the framework manages and whether it supports multi-term negotiation protocols or manages different types of knowledge about parties. Another drawback is that the services that can be composed into the system are vaguely defined. In addition, the framework does not seem to be able to build analysis-based models of parties and market models. Furthermore, it does not provide the mechanism to allow several negotiations at the same time and to permit decommitment of agreements.

Jonker and others [66] describe a component-based generic agent architecture for multi-attribute negotiation. Unlike other negotiation frameworks, it provides advanced capabilities to deal with partial information of other parties, and it also copes with multi-term negotiations successfully. However, it has problems when negotiating with heterogeneous parties in dynamic environments because it only supports one model to express agreements and preferences, and it is not clear whether it can support several different negotiation protocols. Furthermore, the framework does not support several simultaneous negotiations, the negotiation intelligence algorithms cannot be selected dynamically, it does not support decommitment, the creation of agreements cannot be supervised and no market models are built.

2.4 Discussion

From the previous sections, it follows that, although a framework to ease the development of automated negotiation systems able to negotiate service agreements in open and dynamic environments is very desirable from a practical point of view, current solutions are not well-suited. The reasons are: first, protocol-oriented frameworks need to be complemented with decision-making and world-modelling capabilities by means of ad-hoc mechanisms or an intelligence-oriented framework (*cf.* Table §2.2); second, although current intelligence-oriented frameworks deal with multi-term agreements successfully (1.1 and 1.2), they just cope with heterogeneous parties (2.1, 2.2, 2.3, 2.4, 2.5 and 2.6) and partial information about parties (3.1, 3.2 and 3.3) partially, and they have problems when dealing with dynamic markets (4.1, 4.2, 4.3, 4.4 and 4.5); and third, most frameworks are loosely coupled just regarding protocols and decision making (*cf.* Table §2.3).

The reasons behind these problems are three: first, traditionally, protocols and decision-making have been considered the only variation points in a negotiation system, instead of including agreements and preferences models as well; second, the work on modelling the other negotiating parties to deal with partial information is relatively recent, mainly if we compare it with the work on decision-making models; and, third, most negotiation frameworks, with the exception of PANDA, have been designed to support one bilateral negotiation at a time only, *i.e.*, they are not prepared to deal with dynamic markets in which several simultaneous negotiations must be carried out.

To solve the drawbacks of current negotiation frameworks, we present NegoFAST, which is defined at three levels of abstraction. The NegoFAST reference architecture (*cf.* Chapters §7 and §8) provides a foundation for developing automated negotiation systems. It defines the elements that conform an automated negotiation system as well as their interactions. Those elements can be categorised into four large blocks that are called organisations following the Gaia terminology [138]: the protocol management organisation, which deals with external negotiation protocols; the decision making organisation, which generates responses during the negotiation process and decides on the creation of agreements; the world modelling organisation, which manages the knowledge about the market and the other parties, and the coordination organisation, which coordinates the other elements of the system.

The second level of abstraction is the NegoFAST framework (*cf.* Chapters §9 and §10), which refines the NegoFAST reference architecture by defining the interfaces between the elements of the automated negotiation systems. The main design goal of the NegoFAST framework has been to promote the reusability of the elements, so that they may be freely integrated as a whole to produce automated negotiation systems. The NegoFAST framework allows to substitute them easily, which reduces the impact of changes and improves reusability. In addition, it defines a generic model for expressing preferences and agreements.

The third level of abstraction is August, which is a proof-of-concept implementation of the NegoFAST framework. August provides a reference implementation of the coordination organisation and the data structures shared by all elements of the automated negotiation systems. In order to implement the remaining parts of an automated negotiation system, existing negotiation protocol and negotiation intelligence algorithms can be used. This allows us to take advantage of all the work developed in these fields. Furthermore, protocol-oriented frameworks such as [9, 75, 114, 128] can be integrated seamlessly with our work in order to implement the support for multiple negotiation protocols. Appendices §A, §B and §C describe the development of three automated negotiation systems for three different use cases using August.

2.5 Summary

Our goal in this chapter was to motivate the reason why we embarked on this PhD project. We have analysed the problems involved in automating the negotiation of service agreements in open and dynamic environments, and have concluded that none of them succeeds in addressing them at a time. This proves that our contribution is original and advances the state of the art a step forward.

Part II

Background information

Chapter 3
Negotiation rudiments

And since geometry is the right foundation of all painting,
I have decided to teach its rudiments and principles
to all youngsters eager for art.

Albrecht Dürer, 1471–1528
German painter, engraver, and mathematician

A negotiation involves a number of interactions amongst two or more parties, following a specific protocol, in order to reach a mutually acceptable agreement. This chapter presents several proposals to express agreements and preferences and analyses the most significant negotiation protocols in the context of this dissertation. It is organised as follows: in Section §3.2 and Section §3.3, we describe proposals to express agreements and user preferences respectively; in Section §3.4, we characterise negotiation protocols and review proposals that follow the bargaining approach; finally, we summarise our main conclusions in Section §3.5.

3.1 Introduction

Negotiation is a process in which several parties interact in order to reach an agreement. The first formal analysis of the negotiation process was carried out by John Nash [95]. He pioneered the so-called game theory, which analyses negotiation processes mathematically to extract properties. This theory has been developed later and new and more complex negotiation scenarios have been analysed. However, these results are not suitable to be directly implemented in current negotiation software systems because of their assumptions, being the full rationality of the parties, the assumption that limits the most their use in real scenarios [64].

Nevertheless, some properties and conclusions derived from the game-theory are very useful to study the characteristics of negotiation processes and to create or compare negotiation protocols and systems with desirable properties. The most significant are:

Pareto-efficiency: A negotiation is Pareto-efficient if there is no agreement that improves the utility of one party while the outcome of the other parties do not decrease. This means that, if a negotiation is not Pareto-efficient, we are losing agreements that are more profitable for all the parties in the negotiation.

Stability (Nash equilibrium): Let s and s' be two strategies that determine the behaviour of a party during the negotiation. Two strategies are in Nash equilibrium if, given that the best strategy that one party is using strategy s, the best that other party can do is to use strategy s'. A negotiation process is stable if it encourages the parties to behave in a way so that their strategies are in Nash equilibrium.

Cooperative/non-cooperative: A negotiation is cooperative if the parties try to maximise the social welfare, *i.e.*, the sum of the utilities of the negotiating parties. Conversely, a negotiation is non-cooperative when parties only care of their own interests and they just try to maximise their own utility. In [87], the term semi-competitive negotiation is introduced to denote negotiations in which parties try to find a fair agreement for all parties, but maximising their own utility. This negotiation is common when the parties want the other parties to be as satisfied as possible so that long-term relationships can be established.

Success guaranteed: A negotiation process guarantees the success if it assures that an agreement is going to be reached at the end of the process. That is to say, if it guarantees that all parties will reach an agreement.

Provider/consumer bias: A negotiation process is unbiased for both provider and consumer if it is fair for both of them, *i.e.*, if being in one side is not more advantageous than being in the other.

The idea of creating software systems that support or develop negotiation processes is not new. Since the early 80s, there are systems that try either to support human decisions during a negotiation process or to carry out automatically these negotiations. The way agreements are expressed can vary from one negotiation to another. In some cases, it is an explicit document that states the terms of the agreement; in other cases, it may be only a decision on some issue. In this dissertation, we are interested in negotiations aimed at creating an explicit agreement document.

As a consequence, the first problem a negotiation software system has to face is the models in which agreements and preferences are expressed, and the specification of the mechanisms that defines the interactions that the parties carry out in a negotiation, *i.e.*, the negotiation protocol. There are many different approaches, each one with their advantages and drawbacks. For instance, user preferences can be expressed using constraints, fuzzy constraints, or utility functions amongst others. Regarding agreements, there are also different approaches. Some proposals try to standardise their structure, others focus on the interoperability of terms by defining ontologies, whereas others express the agreements in formal languages such as deontic logic in order to facilitate further analysis. Finally, most common negotiation protocols can be categorised into auctions and bargaining [61].

3.2 Agreements

An agreement defines a relationship between several parties. Its goal is to define the terms that regulate the sale of goods or the execution of a service. In our context, service agreement negotiation, the agreement specifies the functional description of the service and the non-functional guarantees. In addition, agreements may also include compensation and termination clauses: the former are used when one party does not fulfil some terms of the agreement; the latter specify when an agreement must finish.

Some authors [57, 82] use the term contract as a synonym for agreement document. However, we prefer to avoid the term contract because of the legal implications that this term involves. Therefore in this dissertation, we use the term agreement document or, simply, agreement to refer to a document with the aforementioned characteristics.

3.2.1 Foundations

Agreements are composed of the following parts. However it is not mandatory for an agreement to include them (*e.g.*, definitions may appear in other documents such as attribute catalogues).

- A specification of the *parties* involved.
- A collection of *terms* that specify the conditions and arrangements of the agreement. Some authors [82] classify them into obligation terms, permission terms and prohibition terms; other approaches, this distinction is implicit in the definition of the terms of the agreement [4, 92].
- A definition of the notions and elements that are used in the agreement. For instance, in an agreement with an Internet services provider in which it is stated that its availability is 99.9%, it is necessary to formally define what availability is and how it is measured.

Agreements are usually expressed in natural language. However, natural language is, by its own nature, imprecise and ambiguous and, thus, it is not suitable to be automatically processed by software systems. To overcome this problem, several proposals have been made: in [57, 59, 82, 92] the agreement document is used to monitor and enforce the terms specified in the agreement and they focus on the business interactions level. In addition, in Reference [56], the authors present a formal grounding for [82] using logic formalisms (*e.g.* courteous logic, deontic logic or defeasible logic). This approach makes it possible to reason about violations of obligations in agreements.

Other proposals [31, 33, 65, 132] focus on lower-level details of the agreements. They are called Service Level Agreements (SLAs) and are usually applied to non-functional attributes of services (*e.g.* response time or availability). As a consequence, these agreements tend to be a list of non-functional attributes and the values or ranges that they should take during the agreement fulfilment. This vision of agreements is also shared by other proposals [44, 68], which focus on the negotiation of the agreement. In [9, 100], the problem of interoperability is dealt with, and ontologies are used to define the agreement document and its contents. Finally, there are standardisation efforts [4] aimed at defining a general structure for an agreement document. This agreement, can later be adopted in other domains by using domain-specific vocabularies.

Note that this classification is based on the main goal of the proposal. This does not mean that the proposals that focus on the business interaction level cannot be used to specify low-level details [82], or that proposals that focus on lower-level details cannot express higher-level business details [132].

3.2.2 Proposals

Next, we analyse one representative proposal for each approach.

Linington and others: In Reference [82], the authors tackle the problem of coordinating behaviour in cross-organisational scenarios and argue that agreements (business contracts, using their terminology) can be used to solve the problem. Therefore, they focus on the business interaction level of agreements. The agreement is specified in a language called Business Contract Language (BCL), which can be seen as an event-driven language. The agreement defines a community, which is a set of roles, policies, states and related event patterns. The terms of the agreement are called policies. There are three types of policies: obligations, prohibitions and permissions. Each policy is applied to one role, which is a label for a party whose behaviour is constrained by policies, and triggered by one or more events. In addition, each policy specifies the behaviour that the role is obliged, prohibited or allowed to have. States are used to define data values shared by all the participants in the community. For instance, a policy called `MakeGoodsAvailable` is an obligation applied to role `Supplier`, that is triggered when event `PurchaseOrder` occurs and obliges it to make the goods available one day after the purchase order is issued. BCL also allows the specification of compensations in case of violations of some terms of the agreement. Specifically, they can be modelled by defining policies with guards, so that it obliges one party to follow a behaviour if a policy has not been fulfilled. Furthermore, Governatori and Milosevic [56] present a formal grounding to BCL. They define the so-called Formal Contract Logic (FCL), which is based on deontic logic but extended to support reasoning about violations, and a mapping from FCL to BCL.

Bartolini and others: In Reference [9], a framework for automated negotiation and a way to represent agreements is presented. This work is based on Reference [133], where this representation of agreements was first presented. Agreements here are focused on detailing the characteristics, both functional and non-functional, of the service or goods that are being sold. However, the novelty of this work is that interoperability issues are taken into account. In this case, the agreement document is an instance of a class of an ontology, which is defined using OWL-Lite [36]. Specifically, they define a description ontology that includes concepts such as agreement template, proposal or advertisement, and a sale ontology that defines concepts like sale, party, delivery or product. Then, they use domain-specific ontologies to define the semantics of the concrete terms of the agreement. For instance, if the product that is being sold is a PC, the domain-specific ontology includes concepts such as memory or processor. Due to the use of ontologies, the automated management of an agreement is not bounded to just simple string matches. Furthermore, to an extent, description logic reasoners may be used to validate proposals, match them or find agreements.

WS-Agreement: It is a standardisation effort developed by the GRAAP workgroup in the Open Grid Forum [4]. The goal of WS-Agreement is threefold: to define a document format for the agreements, to establish a concrete agreement creation process and to describe the communication interfaces that must be implemented by the parties involved in the agreement creation process. An agreement is an XML document specified in XML Schema and is composed of three sections: the name of the agreement, which is used to identify it, the context of the agreement, and its terms. The context of the agreement establishes the parties that are involved in the agreement, the expiration time and other general information about the agreement. The terms of an agreement may be either service terms or guarantee terms. Service terms include service description terms, which specify functional characteristics that shall present the service during its consumption; service references, which indicate how to have access to the service; and service properties, which specify properties of the service that must be measured such as response time or availability. Guarantee terms establish the non-functional characteristics of the service, *e.g.*, response time or cost. A guarantee term includes the scope of the term, when it should be applied, the condition that must be met (service-level objective), and a business value list that specifies the importance of the term, penalties and rewards. Therefore, although WS-Agreement focuses on technical details, it allows to model business-level details as well. In addition, WS-Agreement is domain independent and only specifies the structure of the agreement document. Consequently, to build actual agreements, WS-Agreement must be used in conjunction with one or several domain-specific vocabularies.

3.3 Preferences

Should we wish to automate the negotiation process, we need a means to give to the software system the knowledge that is necessary to negotiate on our behalf, *i.e.*, the preferences. Preferences express the information that is used to ensure that the agreement is reached according to our needs. Therefore, preferences guide the decisions made during the negotiation process. For instance, they can be used to evaluate and compare potential agreement proposals in order to choose the one that best fits our needs.

3.3.1 Foundations

Preferences detail the desired characteristics of three different aspects:

- **Preferences about the contents of the agreement:** Preferences express the desired values for both functional and non-functional characteristics of the service provided, such as the *service interface*, *cost* or *response time*.
- **Preferences about the other parties in the agreement:** Preferences express features, which are the capabilities and attributes that present the party on which behalf the automated negotiation system is negotiating (*e.g., it is located in Spain*), and requirements, which are the desired features in other parties (*e.g., it must have a high reputation on service Y*).
- **Preferences about the negotiation process itself:** Preferences express the desired characteristics of the negotiation process, such as the deadline to finish the negotiation [44], the eagerness of the user to obtain an agreement [121] or the willingness to cooperate with other parties [29].

Preferences typically include initial data provided by a human user, such as the requirements/features of a service demanded/offered. However, it may also include information obtained automatically. For instance, in the case of a computation service provider, the features of the service offered may be generated automatically from the current load of their servers [83, 85].

There are many different models to express preferences. The most common approach uses utility functions [37, 44, 55, 60, 70, 101]. Furthermore, many other approaches [55, 77, 116] use utility functions as a complement to the other formalisms they employ to express preferences. Others express preferences as constraints [53, 78, 88], fuzzy constraints [77, 86], a combination of attributes [39], and rules [55].

The way the user defines his or her preferences is very important to the negotiation process because decision-making algorithms depend on a particular preferences model. Furthermore, simpler and more efficient algorithms can be used when the user does not require a very expressive preferences model (*cf.* Section §4.2). As a consequence, a trade-off should be made between the preferences model expressiveness and the availability and complexity of the decision-making algorithms that deal with them.

Additionally, there is another important aspect regarding preferences: its elicitation. Obtaining an explicit representation of the preferences from the user is not easy [86]. Two different approaches have been followed in the literature to overcome these difficulties, namely: using formalisms that are natural to the user such as rules [55] and eliciting user preferences indirectly like in References [21, 60].

3.3.2 Proposals

Next, we analyse one representative proposal for each approach except for constraints because it is a particular case of fuzzy constraints:

Faratin and others (Utility functions): Faratin and others [44] use multi-attribute utility functions to define user preferences about the issues under negotiation. Specifically, they consider $a_j, j \in \{1, ..., n\}$ the set of attributes about the service, party or process defined in the preferences; $[min_{a_j}, max_{a_j}]$ as the values that are acceptable for attribute a_j; and, for each attribute, an utility function (also known as scoring function), $u_{a_j} : [min_{a_j}, max_{a_j}] \to [0, 1]$, that assigns a utility to every value of this attribute. Finally, the total utility for all attributes is defined as a weighted sum of all utility functions. The weight of each attribute (w_{a_j}) is a measure of the importance the party gives to this attribute and we assume it is normalised, which means that $\sum_{j=1,...,n} w_{a_j} = 1$. Therefore, the utility of an agreement is defined as:

$$u(X) = \sum_{j=1,...,n} w_{a_j} u_{a_j}(X[a_j])$$

and, hence, the more utility an agreement proposal has, the closer it is to our preferences. However, defining the utility of an agreement as a weighted arithmetic mean presents several drawbacks such as that the contribution of an attribute to the global score is limited to w_{a_j}, which prevents the model from dealing with mandatory attributes, for instance [37, 101]. To solve these problems, Dujmovic [37] presents the LSP (Logic Scoring of Preference) method, which defines the utility of an agreement as

$$u(X) = \sum_{j=1,...,n} (w_{a_j} u_{a_j}(X[a_j])^r)^{1/r}$$

where r is a real number selected to achieve desired logical properties in the utility function. Karp also faces these problems and propose a complex model to represent utility [70].

Kowalczyk (Fuzzy constraints): In this approach, preferences are expressed as fuzzy constraints. Therefore, the best agreement for the user is the one that maximises the satisfaction level of its fuzzy constraints. Preferences in this approach consist of the following: a set of attributes $A = \{a_1, ..., a_n\}$ that includes all decision attributes used by the party; a set of domains $D = \{D_{a_1}, ..., D_{a_n}\}$, where each domain D_{a_j} is the set of values that an attribute a_j may take, $a_j \in D_{a_j}, j = 1, ..., n$; and a set of fuzzy constraints $C = \{C_1, ..., C_k\}$, where each fuzzy constraint $C_i, i = 1, ..., k$ is characterised by a membership function $C_i : D_{a_{i_1}} \times ... \times D_{a_{i_l}} \rightarrow [0, 1], i_1, ..., i_l \in \{1, ..., n\}$, that represents the satisfaction of the constraint on a set of attributes. All constraints of the party can be represented as:

$$C(X) = \bigwedge_{i=1,...,k} C_i(X) = \bigwedge_{i=1,...,k} Cyl_X C_i(X[a_{i_1}], ..., X[a_{i_l}])$$

where $Cyl_X C_i(X[a_{i_1}], ..., X[a_{i_l}])$ is the cylindrical extension of the fuzzy relation to X. In general, it is usual to define a satisfaction threshold (α), so that an agreement proposal X is considered to be acceptable if $C(X) \geq \alpha$. In addition, the author introduces an additional constraint that represents the utility value of an agreement: $C_0(X) = u(X)$, where $u(X)$ is the final utility of an agreement as defined above. The addition of this constraint makes it possible to consider trade-offs between the issues, taking into account the utility values of the proposals.

PANDA (Rules and utility functions): PANDA [55] is a framework that mixes utility functions and rules to express preferences about the negotiation process. Utility functions are used in the same way as in the proposal by Faratin and others [44]. However, they argue that some preferences are hard to express using utility functions, for instance, a preference expressing that if the customer is a first-time customer, then we can accept less profitable agreements in order to attract new customers. Nevertheless this preference is much easier to express using rules. Gimpel and others argue that, to express rules, it is necessary to make a trade-off between expressive power and ease of use while ensuring automated execution. The authors define rules with the form condition-action and include a set of sensors to provide input to the rules (*e.g.* `LEVEL-OF-DISSENT`, `RISK-UTILITY`, `NEW-CUSTOMER`) and effectors to conduct desired actions (*e.g.* `ACCEPT-OFFER`, `FIND-TRADE-OFF-OFFER`, `MAKE-OFFER`). Hence, the previous preference can be expressed as `if LEVEL-OF-DISSENT < 0.2 and NEW-CUSTOMER then ACCEPT-OFFER`.

Elfatatry and Layzell (Combinations of attributes): Elfatatry and Layzell define a negotiation description language [39]. This language allows to define service profiles, negotiation strategies and protocols. Service profiles are a description of the functional and non-functional attributes of the service that is being offered or demanded. This description is a list of functional and non-functional attributes and their allowed values. In addition, the negotiability of each attribute is specified. For instance, `NEGOTIABLE price = 10..15 cents per minute`. Together with this description, service profiles also define the so-called selection packages. Selection packages are used to evaluate combinations of non-functional attributes of the service. The authors argue that utility functions are not always useful to specify preferences, for instance, when an attribute is considered key for choosing a particular combination of non-functional attributes. However, these conditions are easily expressed as a selection package `IF price < 10 THEN security <= 32-bit and reliability <= 2 failures per session`.

3.4 Negotiation protocols

An essential part of a negotiation is the mechanism that defines the interactions that parties carry out in a negotiation. This mechanism is the negotiation protocol. The most common negotiation protocols can be categorised into auctions and bargaining [61]. However, as discussed in Section §1.1, in this dissertation we focus on bargaining protocols because they are more suited for the type of automated negotiations we are dealing with. Nevertheless, a thorough analysis of auction protocols has been developed and it is included in this dissertation in Appendix §D.

3.4.1 Foundations

Bargaining protocols involve exchanging proposals and counterproposals between the parties. However, there are differences between them, which may be analysed in terms of five fundamental aspects: roles, performatives, rules, information exchanged, and agreement terms negotiability.

Roles: In a negotiation, there are three types of parties: consumers, providers and trusted third parties. These parties play one or more roles in a negotiation. Most bargaining protocols have two roles: initiator, which starts the negotiation, and responder, which responds to the initiator. However, some negotiation protocols require an additional role called mediator or negotiation host that facilitates the negotiation process [75, 76]. Bartolini and others [10] identifies six different sub-roles that this mediator may present. Note that, for instance, a consumer may be either the initiator or the responder of a negotiation. Furthermore, it may be even initiator and mediator at the same time. Trusted third parties just can be used as mediators.

Equally important is the number of parties that may play the roles in the negotiation. Most bargaining protocols are bilateral and, hence, the parties are limited to one party per role. However, note that parties may carry out several simultaneous bargaining negotiations, resulting a multilateral negotiation with several consumers and several providers negotiating at the same time.

Performatives: A performative is the expression of the intention of the sender of a message about it. The term performative is borrowed from the FIPA terminology and the speech act theory. The set of performatives used in a negotiation protocol may differ significantly, although all bargaining protocols use, at least, performatives `accept`, `reject negotiation` and `commit`. Table §3.1[†1] shows a description of the most common performatives in negotiation protocols, although, depending on the protocol being used, other performatives may be necessary. For instance, argumentation-approaches introduce performatives, such as `argue` and `challenge` [3], to express arguments that support our proposal.

Rules: In a negotiation protocol, there are usually some restrictions regarding a variety of aspects of the negotiation such as how the proposals must be built, when a party can send a proposal, which performative can be used in each moment, or when the negotiation finishes. Often, these rules are implicit in the description of a negotiation protocol. However, some authors [8, 75, 114] argue that to build automated negotiation systems that supports a variety of negotiation protocols, it is necessary to define them explicitly. To this end, several different formalisms can be used, being the most significant: if-then rules [8], Petri nets [34], state machines [75, 114], electronic institutions [41] and domain-specific languages [39].

Bargaining protocols usually involves an alternated exchange of proposals and there are no restrictions on their contents. However, in some bargaining protocols the terms of the proposal are negotiated one-by-

[†1]In the literature [51, 71, 119, 129], performative `propose` usually means making a binding proposal. However, we prefer to leave the term `propose` to non-binding proposals and to use `commit` for binding proposals

Performative	**Description**
`accept`	Accept a proposal
`reject proposal`	Reject the last proposal but keep negotiating
`reject negotiation`	Finishes a negotiation process unsuccessfully
`propose`	Sends a non-binding proposal, *i.e.*, a proposal which does not involve a firm commitment with the other party
`commit`	Sends a binding proposal, *i.e.*, a proposal which involve a firm commitment with the other party
`withdraw`	Withdraw the last proposal
`cfp`	Make a call for proposals to the other parties

Table 3.1: *Description of common performatives.*

one following an established agenda [47], or restricted to be narrower in the counterproposal than in the proposal [71].

Information exchanged: The approaches followed to exchange information can be classified into three broad groups: (i) the information exchanged explicitly states the parts of the agreement that are disliked by the party as well as the proposed changes. (ii) The information exchanged consists only of proposals (*i.e.*, the negotiation protocol is proposal-based). In this case, the information about the disliked parts of the agreement is implicit in the counterproposal and it is up to the parties to infer it. The advantage of this approach is that it unveils less information to the other parties. The disadvantage is that the lack of explicit information implies a blind search of a mutually acceptable agreement that may lead to longer negotiations or may even prevent an agreement from being reached. (iii) The information exchanged includes proposals, as in proposal-based protocols, and statements that are used to persuade or convince the opponent to accept our proposal [109]. This approach is called argumentation and it is a promising field that may eventually overcome the drawbacks of the proposal-based negotiation in scenarios with limited information about the environment [107]. However, the automated negotiation systems that support argumentation tend to be very complex and no argumentation approach has been applied to a realistic scenario as of the time of writing this dissertation.

Agreement terms negotiability: The simplest term negotiability involves negotiating only one term, which is usually the price. However, most bargaining protocols do not impose any restriction about the negotiability of agreement terms [44] and, hence, virtually any term can be negotiable. Nevertheless, generally speaking, increasing the term negotiability of a negotiation protocol allows causes a higher complexity in automated negotiation systems because the search space for a mutual agreement is larger and, hence, it is harder to find an agreement. In addition, this requires the parties to be given more intelligence in order to avoid excessively long negotiations that may finish without reaching an agreement. Because of this, some bargaining protocols force parties in a negotiation to follow a preestablished agenda [26, 47] during the negotiation. This means that not all terms can be negotiated at the same time, but one-by-one or in closed groups of related terms (*e.g.* terms related to the quality of the product and terms related to the delivery and payment).

3.4.2 Proposals

Next, we report on four different proposals of bargaining protocols that cover most significant approaches to bargaining (*cf.* Table §3.2): ContractNet [51], OMG bilateral negotiation [102], alternating-offers negotiations described by Sierra and others [119] and extended to allow decommitment by Nguyen and Jennings [98], and argumentation-based approaches [107].

Iterated ContractNet: The original ContractNet protocol was first described in the early 80s, and it was focused on solving resource allocation problems through negotiation. In this dissertation, we analyse the specification of this protocol developed by FIPA called Iterated Contract Net Protocol [51]. The protocol involves two roles: the initiator and the participant. Any of them can be played by either the consumer or the provider. The performatives used in the protocol are depicted in Table §3.2. Besides the common performatives, they include `inform` (informs the other party that everything finished successfully), and `failure` (indicates that there was an error at the end of the process). The negotiation protocol is developed as follows: first, the initiator sends a call for proposals to the participant. Then, the participant can either refuse the negotiation or send a concrete proposal. When the initiator receives the proposal, it has several options: it may send a reject proposal, a new call for proposals or accept the proposal. If it sends a reject proposal, the participant can either send a new proposal or refuse the negotiation. If the initiator sends a new call for proposals the whole process starts again. Finally, if it

Protocol	**Roles**	**Performatives**	**Information**	**Negotiability**
Iterated ContractNet	Bilateral	accept, reject-proposal, refuse, propose, cfp, inform/failure	Proposals/Call for proposals (content undefined)	No restrictions
OMG Bilateral	Bilateral	accept, reject, suggest, request, propose, offer	Proposals/Call for proposals (content undefined)	No restrictions
Alternating-offers	Bilateral	accept, reject negotiation, propose, decommit	Proposals (assignment of values to attributes)	No restrictions
Argumentation	Bilateral	accept, reject, withdraw, offer, request, threaten, reward, appeal	Proposals (assignments)/Logic statements	No restrictions

Table 3.2: *Summary of bargaining protocols.*

accepts the proposal, the participant must either inform that everything went fine or indicate that there was a failure that makes it impossible to fulfil the negotiation. Regarding the information exchanged during the protocol, they can be either proposals or call for proposals although the specific content of a proposal is not specified in the protocol. In addition, the protocol does not impose any constraint on the negotiability of the terms either.

OMG Bilateral Negotiation: This negotiation protocol is part of the OMG Negotiation Facility [102], which is a specification that builds upon the OMG's CORBA specification. The protocol involves two parties. The performatives employed are depicted in Table §3.2: `offer` and `propose` are commitments to proposals, whereas `suggest` is a non-committing proposal and `request` is equivalent to performative `cfp`. The negotiation starts with either a `request`, an `offer` or a `propose`. If it starts with a `request`, the protocol enters in state *requested*. In that state, parties may `suggest` new proposals, or send an `offer` or a `propose`. If an `offer` is sent, the protocol enters in state *offered*. In that state, the only options are either `accept` the offer or `reject` it. Alternatively, if a `propose` is sent, the protocol enters in state *proposed*. In that state, the options are either `accept` or `reject` the proposal or send performative `request` to get new non-committing proposal. Finally, in any state, performative `reject` may be sent to finish the negotiation. Regarding the information exchanged during the protocol, they are proposals although their specific content is not specified in the protocol. In addition, the protocol does not impose any constraint on the negotiability of the terms either.

Alternating-offers: Sierra and others described a bargaining protocol designed to develop negotiations of services [119]. The performatives the protocol includes are `propose`, `reject negotiation`, `reject proposal` and `accept` (*cf.* Table §3.2). The protocol is very simple, it starts when one party sends a proposal to the other. When it receives the proposal, it can respond with a new proposal (*i.e.* a counterproposal); it can reject the negotiation and finish it; it can reject the proposal, forcing the other party to create a new proposal; or it can accept the proposal, thus reaching an agreement. The protocol does not impose any restriction on the negotiability of the agreement terms, and the information exchanged consists of proposals exclusively.

An extension to this protocol was presented in Reference [98], in which the authors analyse the problem of several simultaneous bilateral negotiations following the protocol described above. They conclude that the best solution to deal effectively with several concurrent negotiations is to allow the decommitment of previously established agreements by paying a certain cancellation fee. To do so, they add a new performative called `decommit`. Therefore, the parties follow the previous protocol to reach an agreement and, once it is made, if they found a new proposal that improves the current agreement, they create a new agreement and decommit from the previous one by paying a penalty.

Argumentation: There are many proposals for argumentation-based negotiations [3, 91, 109] (*cf.* Reference [107] for a survey). In this dissertation, we focus on the protocol described by Sierra and others [120]. The protocol, as most argumentation-based negotiations, is bilateral. The performatives are divided into two sets: negotiation performatives, which are used to make proposals, and persuasion performatives, which are used in argumentation. The former performatives are: `request`, `offer`, `accept` , `reject` and `withdraw` (*cf.* Table §3.2). The latter are: `threaten` (threats the other party to do something if it does not accept something), `reward` (rewards the other party with something if it does something), and `appeal` (communicates something to the other party). Note that other argumentation-based approaches may have a set of performatives significantly different. The negotiation protocol always start with a request or an offer. Then, parties exchange proposals and counterproposals together with persuasion performatives. Finally, the negotiation finishes when either an accept or a withdrawal is sent by one party. The information exchanged, together with the performatives, are the main difference between argumentation-based negotiations and other negotiations. In [120], the authors do not commit to a specific language to express the information exchanged. Instead, they describe the characteristics that such language must have, namely: it must be a logical language with variables to represent the terms under negotiation, constants to represent the values for the terms under negotiation, equality to specify the value of a term under negotiation, and conjunction to define complex sentences. Regarding the negotiability of agreement terms, the protocol does not impose any restriction on it.

3.5 Summary

In this chapter, we have introduced some general negotiation properties that can been used to check if negotiation elements such as protocols or strategies lead to optimal outcomes. Furthermore, we have described different approaches to model agreements and preferences, which are a key part in every negotiation system, and we have characterised and analysed the negotiation protocols that are more relevant in the context of this dissertation.

Chapter 4
Negotiation intelligence

What a distressing contrast there is
between the radiant intelligence of the child
and the feeble mentality of the average adult.

Sigmund Freud, 1856–1939
Austrian neurologist and psychiatrist

In the previous chapter, we analyse the external parts of an automated negotiation system, i.e., the preferences as an input to the system, the agreements as the output of the system, and the negotiation protocol that guides the interaction between the negotiators. In this chapter, we move inside the automated negotiation system and describe the mechanisms that are necessary to carry out the negotiation automatically. These mechanisms are the decision-making that determines the way a certain party behaves in a negotiation process and the world-modelling that models the different elements that have an influence over the negotiation to support the decisions. The chapter is organised as follows: in Section §4.2, we describe the decision-making of an automated negotiation system; then, we analyse the approaches to create models of the environment in negotiation scenarios in Section §4.3. Finally, we summarise our main conclusions in Section §4.4.

4.1 Introduction

An automated negotiation system must be intelligent enough to carry out the negotiation process automatically. In particular, this negotiation intelligence can be divided into two tasks: first, it is necessary to make decisions autonomously during the negotiation process; second, to support this decision-making, it is necessary to create models of the elements of which the negotiation process is composed.

The decision-making model of an automated negotiation system determines its behaviour during a negotiation process. An automated negotiation system has to make the decision of which responses shall be sent to the other participants in the negotiation following a given protocol, as well as deciding on when to commit to an agreement proposal. In addition, some authors [98, 118] also include the decommit decision as a procedure to enable negotiations with several parties at the same time.

In its simplest form, decisions may be exclusively based on user preferences or other factors such as the resources available or the time remaining to reach the deadline established to achieve an agreement. However, there are other environmental aspects that can be used to make better decisions during a negotiation. For instance, knowing the preferences of the other parties may help devise more appealing proposals and, thus, increase the likelihood of reaching an agreement [139]. Therefore, modelling the world that surrounds the automated negotiation system, *i.e.*, the market, the other opponents, and the domain, plays an important role in the negotiation abilities of the system.

4.2 Decision making

The decision-making of an automated negotiation system determines the way it behaves while involved in a negotiation process. Specifically, it uses information from several sources, including preferences, world model and external factors that may prevent a party to commit to an agreement, to make, at least, two decisions: whether it is convenient to commit to an agreement and the response that shall be sent to the other parties during the negotiation process. Besides, there is another decision that may be necessary depending on the characteristics of the negotiation that is being developed, namely the possibility of decommitting from agreements as a mechanism to enable several simultaneous negotiations.

As a consequence, the decision-making of an automated negotiation system is heavily influenced by the agreement model, the preferences model and the negotiation protocol. It is influenced by the agreement model because, since the goal of the decision-making is creating agreements, the decision-making algorithms must change depending on the characteristics of the agreement model. For instance, it is not the same to create agreements in which terms are name-value pairs, than agreements in which terms can express complex relations amongst different attributes. It is influenced by the preferences model because, depending on how preferences are expressed, the decision-making algorithms may be more complex (*e.g.* they may try to find trade-offs between different terms of the agreement) or they may be simpler (*e.g.* they just concede as time goes by in the negotiation). Finally, the decision-making of an automated negotiation system is influenced by the the negotiation protocol because the automated negotiation system must behave according to the rules defined by the negotiation protocol.

4.2.1 Foundations

The decisions of automated negotiation systems are based on several different information sources. These sources can be classified into the following broad groups:

Preferences: The decision-making of an automated negotiation system must use the information provided by the preferences to behave according to the user's requirements. As detailed in Section §3.3, preferences may be related to the contents of the agreement (*e.g.* constraints on the values of the terms of the agreement or an utility function indicating the importance of these terms to the user), the party with which we are negotiating the agreement (*e.g.* we may not wish to make an agreement with a company that competes with us with another product or with a company located in a country to which we are not allowed to sell services), and the negotiation process (*e.g.* the deadline and the eagerness to reach an agreement).

World model: The knowledge about the world in which an automated negotiation system is immersed enables the decision-making of an automated negotiation system to improve its negotiation capabilities [87, 139]. As detailed in Section §4.3, the models may be related to other parties (*e.g.* we may create more appealing proposals to the other parties if we know its preferences [28], the market (*e.g.* we may be more reluctant to accept a proposal if we are likely to get a better one soon [80], or if there are many trading partners and our proposals are close to the other parties' [121]), and the negotiation domain (*e.g.* it may be necessary to measure the similarity between two different values of the same agreement term to make trade-offs that improve other party's utility).

External factors: Usually, the capability to accept new agreements depends on the current state of the resources. To avoid committing to agreements that cannot be satisfied, the automated negotiation system should use information provided by external elements such as a capacity estimator to determine whether an agreement can be accepted or not [85].

Regarding the decisions that must be made by an automated negotiation system, they can be divided into the following decisions:

Commit decision: It involves determining when to submit a binding proposal and whether a binding proposal that has been received should be accepted. Its complexity may vary significantly from one approach to another. For instance, in Reference [77], a proposal is accepted if its fuzzy constraint satisfaction is higher than a certain threshold that may change during the negotiation. In other approaches such as in Reference [44], the commit decision also depends on the time that remains to reach the deadline to finish the negotiation. Therefore, a proposal that it is not acceptable at the beginning of the negotiation may become acceptable at the end because of the time constraints. Furthermore, when multiple agreement negotiations are being carried out simultaneously, the commit decision becomes even more complex because it must coordinate the sending or acceptance of binding proposals to the other parties [98] to avoid undesirable behaviour such as a service consumer committing simultaneously to two agreements when only one is enough.

Together with the decision of submitting or accepting a binding proposal, it is necessary to establish when these decisions must be made. Usually, the decision is made as the proposals are received [97]. However, other approaches may be followed such as making the decision at some predefined points in time.

Response decision: This decision determines which messages must be sent to the other parties in the negotiation. This involves selecting the performative that is going to be used and creating the response content, which may include proposals, bids, or arguments. Note that separating commit decision and response decision is only at a conceptual level and they can be implemented together like in References [44, 77]. However, in other cases, it is convenient to implement them as separate processes. For instance, when coordinating several concurrent negotiations [98] or when the commit decision is computationally expensive (*e.g.*, it requires an analysis of the provider's capability to accept a new agreement).

The response decision is specially influenced by the negotiation protocol because it must obey the rules it imposes. Bargaining protocols are based on the exchange of proposals. Therefore, the main goal of the response decision is to create these proposals. To this end, a wide variety of techniques have been developed, being the most influentials: those that use time-dependent functions or resource-dependent functions to obtain the counterproposal by modifying the values of the terms of the offer [44]; those that try to find trade-offs and create proposals that are more appealing to the other party based on similarity between proposals [43]; those that use constraint resolution techniques [78] or that are based on fuzzy constraints [77, 87], and those that regard negotiations as games and use techniques similar to those used in chess games [72]. Genetic algorithms have also been used to calculate which the best strategy to use is [48]. In addition, in argumentation-based negotiations, the response decision is more complex because many performatives can be used (*e.g.* `reward`, `information` or `threat`) and the contents of the message may include logical arguments that must be generated and selected by the automated negotiation system [107].

Decommit decision: Decommitting an agreement involves revoking it, usually by paying a previously agreed decommit fee, and it is chiefly used as a mechanism to enable several simultaneous negotiations [98]. The first work in this direction was presented by Sandholm and Lesser [118]. The authors present a formal analysis of the advantages of decommitting from established agreements, and show how decommitment may increase the payoff obtained in several negotiations. However, they only analysed a bilateral game with fixed penalties for decommitting. Later, Nguyen and Jennings [98] have used decommitment to build a system that allows multiple concurrent negotiations and support varying decommitment fees depending on the stage of the process at which the commitment is broken. The decommit decision is closely related to the commit decision because usually the event that triggers the need for a decommitment is that we can commit to a more profitable agreement. That is the reason why both commit and decommit decisions are made by the same element in many cases [97].

4.2.2 Proposals

We focus on four decision-making models for bargaining negotiations.

Negotiation Decision Functions: Faratin and others [44] describe a decision-making model for service-oriented negotiations (*cf.* Section §3.4). They consider that each term of the agreement is an assignment of a value to an attribute. In addition, they assume the preferences include an interval of allowed minimum (min_{a_j}) and maximum (max_{a_j}) values for each attribute a_j under negotiation, together with an utility function for the attribute (u_{a_j}). In addition, there is a global utility function, which represents a weighted sum of the utility functions of each attribute.

The commit decision is made as the proposals are received and involves the following: (i) generated proposals are always binding proposals, *i.e.* they do not need to pass any approval process, and (ii) when a proposal is received, the automated negotiation system generates a counterproposal. If the utility of the received proposal is higher than the utility of its counterproposal, then the proposal is accepted; otherwise, the counterproposal is submitted.

The response decision involves: first, if the negotiation deadline is reached, the negotiation is finished; second, when a proposal is received, a counterproposal is generated and, if its utility is higher than the utility of the received proposal, then it is submitted. Counterproposals are created using the so-called negotiation decision functions. To prepare a counter proposal X', the automated negotiation system uses a set of tactics that generate new values for each of the terms in the agreement. The authors define three types of tactics: time-dependent, which depend on the progress of time during the negotiation; resource-dependent, which depend on the availability of resources (*e.g.* remaining bandwidth) and the environment (*e.g.* number of clients); and behaviour-dependent, which attempt to mimic the behaviour of the other parties. Each tactic is a function that calculates a value for each term of the agreement. For instance, in the case of time-dependent tactics, the authors propose to calculate the value of term T_{a_j} in counter proposal X' as the following function:

$$X'[T_{a_j}] = \begin{cases} \min_{a_j} + \alpha_{a_j}(t)(\max_{a_j} - \min_{a_j}), \text{ if } u_{a_j} \text{ decreases} \\ \min_{a_j} + (1 - \alpha_{a_j}(t))(\max_{a_j} - \min_{a_j}), \text{ if } u_{a_j} \text{ increases} \end{cases}$$

where α_{a_j} is a function that determines the behaviour of the tactic. Depending on its value, the authors distinguish between a conceder behaviour, which concedes quickly at the beginning of the negotiation, and the boulware behaviour, which waits until the end of the negotiation to concede. Resource-dependent tactics are implemented either by modifying dynamically the deadline of the negotiation depending on the number of simultaneous negotiations with other parties, or by making the α_{a_j} function dependent on the resources available. Regarding behaviour-dependent tactics, the authors propose three alternatives: relative tit-for-tat, where the automated negotiation system reproduces, in percentage terms, the behaviour of the opponent in the last proposals; absolute tit-for-tat, in which the behaviour of the opponent is reproduced in absolute terms; and averaged tit-for-tat, where the automated negotiation system computes the average of percentages of changes in a window of the opponent last proposals. The final value for each term in the counter proposal is calculated as a weighted sum of the values obtained applying each tactic. In this context, the evolution of these weights as a function of the mental state of the automated negotiation system is the negotiation strategy.

Faratin and others [43]: The authors present a decision-making model that, unlike the previous one, can be used to make trade-offs and is based on a similarity criteria between agreements. It has the same assumptions as negotiation decision functions [44], but it also requires the definition of a fuzzy similarity relation. A fuzzy similarity relation for an attribute a_j is a binary function $Sim_{a_j} : \mathcal{D}_{a_j} \times \mathcal{D}_{a_j} \longrightarrow [0, 1]$, where $\mathcal{D}_{a_j}$ is the domain of attribute a_j. This similarity function gives a measure of how similar two values of the domain $\mathcal{D}_{a_j}$ are. Therefore, this function can be used to measure how similar two terms of different agreements that refer to the same attribute are. In general, the similarity between two agreements is the weighted sum of the similarities of their terms: $Sim(X, Y) = \sum_{i=1..n} w_i Sim_{a_j}(X[T_{a_j}], Y[T_{a_j}])$, where $Sim_{a_j}(X[T_{a_j}], Y[T_{a_j}])$ is the similarity function of attribute a_j for terms $X[T_{a_j}]$ and $Y[T_{a_j}]$. Note that this similarity function is a subjective concept. The other party may consider a different value for the similarity function. However, it is supposed that the similarity measure of two parties is relatively close.

The commit decision in this approach is the same as in negotiation decision functions and the response decision also involves the creation of counter proposals. However, the creation process is significantly different. The idea is to create a counter proposal X' that has the same utility as our previous proposal X but that is more appealing to the other party. The way of creating more appealing proposals to the other party is based on the assumption that the more similar X' and Y are, the more appealing proposal X' is to the other party. Therefore, the goal of the algorithm is to create a counter proposal that verifies that $u(X') = u(X)$ and $Sim(X', Y)$ is as high as possible (*i.e.* the counter proposal is very similar to the other party's proposal). In particular, the algorithm implements this in S steps. It starts from the other party's proposal. In each step, it generates N agreements that have a utility $E = (u(X) - u(Y))/S$ greater than the agreement selected in the last step. Then, it selects the agreement with the highest similarity to the other party's proposal.

Kowalczyk [77]: The author views the negotiation process as a distributed fuzzy constraint satisfaction problem and, hence, describes a decision-making model for bilateral bargaining based on fuzzy constraints. The model assumes that the preferences have been defined as fuzzy constraints (*cf.* Section §3.3).

The commit decision is made as the proposals are received and is as follows: (i) generated proposals are always binding proposals, and (ii) a proposal received is accepted if its level of satisfaction is greater than or equal to a prescribed satisfaction threshold. This threshold can be fixed since the beginning of the negotiation or changed during the negotiation according to its progress and the availability of new information. In Reference [78], the author proposes this threshold to be calculated as $C^* = dC(X^*)$, where $C(X^*)$ is the level of satisfaction of the preferred solution, and d is a coefficient that expresses the desirability of a party to reach an agreement.

The response decision involves the creation of counter proposals. The idea is to start the negotiation with the most preferred agreement and to concede progressively in each counter proposal. All concessions involve decreasing the level of satisfaction of the proposal: $C(X') = C(X) + \Delta$, where X' is the new counter proposal, X is the proposal submitted previously and Δ is the concession. The negotiation strategy determines the rules of progressing negotiation towards an agreement by characterising the level of concession of one party. Kowalczyk proposed a number of basic negotiation strategies including take-it-or-leave-it, no concession, fixed concessions, and simple concessions. For instance, in a fixed concession strategy, the value of Δ is a constant and, hence, the concession made in each counter proposal is always the same. When the concession level is determined, the counter proposal is generated solving a local fuzzy constraint satisfaction problem that includes the information available to a party including the individual preferences, constraints and objectives as well as the previous proposals and counter proposals.

Nguyen and Jennings [98]: This approach deals with the problem of multiple concurrent negotiations. The decision-making model is based on the notion of leveled commitment contracts [118] in which a party can decommit from an agreement by paying a decommitment fee. Therefore, this implies that the decision-making model requires the addition of a decommit decision to decide when it is profitable for the system to decommit from an agreement. The authors assume that preferences are expressed as in negotiation decision functions [44]. In addition, a decommitment fee is established. The calculation of the decommitment fee ($\rho(t)$) takes into consideration a percentage of the utility of the agreement and the time when the agreement is broken. Therefore, as time goes by, the decommitment fee increases.

The commit decision in this proposal involves the decision of whether to commit to an agreement and the decommitment from the current agreement, if any. Let X be the proposal that is being analysed, A a previously established agreement, $u(X)$ and $u(A)$ the utility of both agreements and $\rho(t)$ the decommitment fee for the current agreement. The automated negotiation system will accept X and reject the current agreement if all the following conditions are satisfied: (i) the utility of the new agreement is higher than the sum of the current agreement and the decommitment fee: $u(X) \geq u(A) + \rho(t)$, (ii) the degree of acceptance (μ) is over a predefined threshold (τ). The degree of acceptance is calculated by comparing the utility value of X with the predicted utility of the next set of contracts from other parties that are negotiating simultaneously ($\{X_p \mid p \in Parties\}$), also taking into account the relation between the current time and the negotiation deadline ($t_{b_{max}}$). Specifically, the formula is:

$$\mu(X) = \frac{u(X) - \rho(t)}{max\{U_{exp}(X_p) \mid p \in \text{Parties}\}} \frac{t}{t_{b_{max}}}$$

where $U_{exp}(X_p)$ is the expected utility of the next proposal by the party p.

The decommit decision of this decision-making model is implicit in the commit decision because the creation of a new agreement implies the decommitment from a previous one.

The response decision in this proposal involves the creation of counter proposals following the time-dependent family of negotiation decision functions [44]. However, the authors developed a system to change the strategy during the negotiation based on the perception it has about the type of negotiation the other party uses. Specifically, they distinguish between conceder and non-conceder negotiators. At the beginning of the negotiation, the automated negotiation system assumes the other party has a probability $P(a)$ of being of type $a \in A_{types} = \{\text{conceder}, \text{non-conceder}\}$. Then, during the negotiation, it updates this probability based on how fast it concedes in two consecutive proposals. Given this, the system selects the strategy $\lambda \in S$ to use in each iteration as the strategy that maximises the expected utility $EU(\lambda) = \sum_{a \in A_{types}} PS(\lambda, a) PO(\lambda, a) P(a)$, where $PS(\lambda, a)$ is the percentage of success matrix and $PO(\lambda, a)$ is the pay off matrix. The former measures the chance of having an agreement when the automated negotiation system applies strategy λ to negotiate with a party of type a; the latter measures the average utility value of the agreement reached in such situation.

4.3 World modelling

The knowledge about the world in which an automated negotiation system is immersed is important to strengthen its negotiation capabilities [87, 139]. For instance, in the decision-making model proposed by Nguyen and Jennings [98], it is necessary to calculate the expected utility of the next proposal submitted by one party or the probability that a party is of type conceder or non-conceder.

However, this knowledge is not always available for the automated negotiation system. Negotiations can be classified into two broad groups depending on the information available to the automated negotiation system, namely: full information negotiations and partial or incomplete information negotiations. The former are those negotiations in which the automated negotiation system has full knowledge of all the factors that may affect the decisions made in a given situation. The latter are those in which the automated negotiation system has only partial knowledge of these factors and, hence, the decisions made may not be optimal.

In realistic scenarios, the most common negotiations are those with partial information [47] either because parties are not known *a priori*, or because parties in a negotiation try to minimise the amount of information they reveal about their preferences since that revelation can weaken their negotiation position [87]. As a consequence, an automated negotiation system must be provided with mechanisms to create models about the world in order to increase the partial information it has about it. This is the so-called world modelling.

4.3.1 Foundations

The models that an automated negotiation system can create about the world it is immersed can be classified into three different categories based on the subject the model is about:

Models of other parties: These models include their preferences [139], the negotiation process followed by the other party (*e.g.* whether it tends to concede [98] or its negotiation deadline) and the party itself (*e.g.* reputation or geographical location).

Market models: These models include information such as the market reservation price, the trading opportunities of a party [123], or the probability of appearing new candidate parties during the negotiation [80].

Models of the negotiation domain: These models include information about the domain of the agreement. For instance, Reference [43] requires a measure of the similarity between two different values of the same agreement term to make trade-offs that improve other party's utility.

To create these models, the automated negotiation system needs to gather information about the world, which can be gathered from the following sources:

Domain expert: The information is provided by an expert in the domain. This is the most common information source when building models of the negotiation domain. For instance, in Reference [43], a domain expert provides the automated negotiation system with the aforementioned measure of similarity between values of terms in the negotiation domain.

Directly polling the other parties: The automated negotiation system queries the other parties in the negotiation to get information about them and the characteristics of the service demanded or offered, *i.e.*, their preferences. For instance, in WS-Agreement [4], the parties may offer templates that can be used as guidelines for the other parties to know the kind of agreements they are willing to accept.

External information providers: Besides querying the other parties in the negotiation directly, automated negotiation systems may query other entities that are not directly involved in the negotiation to get information. For instance, reputation providers may be asked for the reputation of a specific party, or marketplaces may be asked for the number of providers or the results of last negotiations. This is the case of Reference [58], in which several auction sites are queried to gather information about the results of recent auctions to provide measures such as recommended maximum price for an auction.

Messages exchanged in negotiations: The messages exchanged with other parties during the negotiations provides an excellent information source that can be used to learn the behaviour and the preferences of other parties like in References [28, 139]. Note that the messages exchanged include both those exchanged in previous negotiations that have already finished and those exchanged in the current negotiation.

Depending on the information obtained from the aforementioned sources, it can can be used to support the decision-making without any further processing or it must be analysed before being used to support decisions during negotiations. The former is usually the case for information provided by domain experts or obtained by polling the other parties in the negotiation, whereas the latter is the most common case for information provided by external information providers or obtained from previous negotiations.

A variety of approaches have been proposed to analyse the information. They can be classified into the following categories:

Offline analysis: The analysis does not take into account the state of current negotiations. Therefore, it can be computationally intensive because they do not have to be computed for a specified time. An example of offline analysis is described in Reference [80], in which the history of previous negotiations is used to learn the expected utility of uncertain and dynamic potential parties that may appear in the future.

Online analysis: The analysis takes into account the state of current negotiations. Therefore, it should be performed as the negotiation progresses and needs to be fast and not very computationally intensive. Examples of online analysis can be found in Reference [139], which uses Bayesian learning to build a model of the preferences of other participants, and in Reference [122], which defines functions that measure two market conditions trading opportunity and competition based on the state of current negotiations and the number of service providers and consumers.

Note that there may be approaches that uses both an offline and an online analysis. For instance, Reference [28] uses an offline analysis to acquire a probability density function over the other party's preferences and, then, it uses an online analysis to reflects the information of the current negotiation process.

4.3.2 Proposals

Next, we report on several approaches to create models of other parties and market models that are used to help improve the decision-making of automated negotiation systems:

Zeng and Sycara [139]: In this approach, the authors model beliefs about the negotiation environment and the other negotiating parties under a probabilistic framework using Bayesian learning representation and updating mechanism. In their model, Ω represents a set of relevant information entities that may be either the parameters of the environment, which can change over time (*e.g.* demand and interest rate or overall product supply), or beliefs about the other parties: about their factual aspects (*e.g.* how many resources a party has), about their preferences (*e.g.* which is the maximum price a party is willing to pay) and about their decision-making model (*e.g.* the negotiation style of a party). Before negotiation starts, each party has a certain amount of knowledge about Ω, denoted as $P_{H_{i,0},i}$, where $P_{h,i}$ is a subjective probability distribution that represents the knowledge held by each party i in each stage of the negotiation and $H_{i,0}$ is the history at the initial step. Then, when other party sends a message, the automated negotiation system updates his subjective evaluation about the environment using Bayesian rules: given prior distribution $P_{H_{i,k-1},i}$ and the new information $H_{i,k}$, calculate the posterior distribution $P_{H_{i,k},i}$. In the article, the authors exemplify this framework by learning the reservation price of the other parties (the maximum/minimum price a buyer/seller is willing to pay).

Coehoorn and Jennings [28]: In this approach, the authors model beliefs about other negotiating parties using the kernel density estimation (KDE) method. This article takes the decision-making model proposed by Faratin and others [43] as a starting point. This model is used to make trade-offs based on a similarity criteria between agreements (*cf.* Section §4.2). One of the problems to put into practise this decision making model is the difference in importance (weight) that each party gives to the terms under negotiation. Therefore, the goal here is to learn the weights that the other parties give to the terms under negotiation. To this end, they process the data of previous negotiations offline to acquire a probability density function over the other party's weights for the various terms. In addition, this model can be improved by online learning that reflects the information of the current negotiation process. The learning is based on the difference between two consecutive proposals. Furthermore, the authors assume that the other party behaves differently as time goes by during the negotiation. Therefore, the density estimation is made for each time in the negotiation.

Li and others [80]: They focus on modelling the market instead of the parties. When an automated negotiation system is negotiating with another party, there may exist simultaneously other potential parties or some may appear in the future. These alternatives to the party that we are considering are called outside options. Outside options have an influence on the decision making model because if we expect for a new party to appear with an appealing proposal, we may be more reluctant to accept the current proposal. The problem Li and others face is to calculate the expected utility of uncertain and dynamic outside options that may appear during a negotiation process. Outside options can execute concurrently with a negotiation, or present sequentially in the future, which are uncertain outside options. The authors model outside options with two levels of complexity: synchronised negotiations, which only consider the impact of other concurrent negotiations, and dynamic negotiations, which consider outside options coming in the future. In synchronised negotiations, the authors provide four heuristic approaches to estimate the expected utility given a set of concurrent negotiations. In the first approach, the synchronised negotiation is approximated as a reverse auction. In the second one, it is approximated by an English auction followed by a bilateral negotiation between the system and the winning party. The third approach is based on the approximation of the probability distributions of negotiators' types by uniform distributions and on the application of the results of Reference [94] to calculate the probability of reaching an agreement. Finally, the fourth approach is based on learning the probability of reaching an agreement and the distribution of agreements based on previous negotiations. The case of dynamic negotiations is an extension of synchronised negotiations in which the arrival of outside options is modelled as a Poisson process.

Sim and Wang [122]: In this article, the authors consider that the automated negotiation system is carrying out several negotiations simultaneously. In this context, the authors model several market conditions such as competition and use them to make adjustable amounts of concessions and to relax trading aspirations in face of negotiation pressure. Specifically, they define functions to measure two market conditions: trading opportunity and competition. The trading opportunity function determines the bargaining power of one party based on trading alternatives (number of trading partners) and differences in proposals. To determine the trading opportunity, Sim and Wang calculate the highest probability of not reaching an agreement with any other party. This probability is based on the difference between the other party's last proposal and the counterproposal made by the negotiation system. When the probability of not reaching an agreement is nearly 0, the trading opportunity approaches 1, and hence, the automated negotiation system has more bargaining power and can make smaller concessions. Conversely, if the probability of not reaching an agreement is close to 1, the trading opportunity reduces to 0, and the bargaining power decreases. On the other hand, the competition function determines the probability that the automated negotiation system is ranked as the most preferred trading partner by at least another party in the current round. Sim and Wang assume the automated negotiation system knows, at each instant t, the number of trading partners (m_t), and the number of competitors (n_t). Then, they calculate the probability that the automated negotiation system is the most preferred trading partner by at least one party as $1-[(m_t-1)/m_t]^{n_t}$.

4.4 Summary

In this chapter, we have analysed the intelligence in automated negotiation systems. First we have described the decisions an automated negotiation system makes and, then, we have reported on models of the environment that the automated negotiation system can create to support those decisions.

Chapter 5

Automated negotiation systems

Besides black art, there is only automation and mechanisation.

Federico García Lorca, 1898–1936
Spanish poet and dramatist

Software systems can support the negotiation process to several extents. In this chapter, we review a number of software systems and frameworks that automate several parts or the whole negotiation process. These systems can be designed ad-hoc for a specific protocol and negotiation intelligence algorithms or they can be built on frameworks that make it easier to reuse protocol support and intelligence algorithms. The chapter is organised as follows: in Section §5.2, we review some ad-hoc automated negotiation systems; then, we describe several negotiation frameworks in Section §5.3. Finally, we summarise the main ideas of this chapter in Section §5.4.

5.1 Introduction

In the previous chapters, we have analysed the different parts that compose an automated negotiation system, namely: the models that are used to represent the agreements and the preferences (*i.e.* the output and the input to the automated negotiation system respectively) (*cf.* Chapter §3); the protocol that defines the rules that must be followed by the parties that participate in the negotiation (*cf.* Chapter §3), and the negotiation intelligence, which includes the decisions that an automated negotiation system makes and the models it creates to support those decisions (*cf.* Chapter §4). In this chapter we consider the automated negotiation system as a whole and we analyse how these pieces fit together.

The simplest approach to build an automated negotiation system is to make an ad-hoc design for concrete negotiation intelligence algorithms and protocols. This is the approach followed by many authors [20, 43, 44, 69, 77, 79, 87, 104, 129]. These developments cover all aspects of an automated negotiation system: from the negotiation protocol to the agreement model and the negotiation intelligence. Furthermore, they usually couple tightly these parts of an automated negotiation system.

Nevertheless, other authors focus on the reusability of the different parts of an automated negotiation system. Thus, they have came up with negotiation frameworks that can be used to build automated negotiation systems. There are two kinds of negotiation frameworks, namely: protocol-oriented frameworks [9, 75, 114, 128] and intelligence-oriented frameworks [6, 12, 26, 55, 66, 74, 84, 136]. The former focus on the interaction of the negotiation parties, whereas the latter are centred on the mechanisms that allow the automated negotiation system to carry out the negotiation process automatically. In summary, protocol-oriented frameworks deal with the external part of automated negotiation systems whereas intelligence-oriented frameworks focus on the internals of those systems.

5.2 Ad-hoc automated negotiation systems

One approach to develop automated negotiation systems is to select a concrete negotiation protocol and specific negotiation intelligence algorithms and to make an ad-hoc design for them. Therefore in such ad-hoc automated negotiation systems, the protocol and intelligence algorithms cannot be easily changed. Ad-hoc automated negotiation systems are usually employed when

Proposal	**Agreement**	**Preferences**	**Protocol**	**Decis. making**	**World model.**
Paurobally and others [104]	Assignment of values	Utility funct., intervals, deadline, behaviour	Bilateral bargaining w. non-binding proposals	Negotiation decision functions [44]	Other parties
Su and others [129]	Constraints	Constraints, utility funct., constraints relaxation	Bilateral bargaining with proposal withdrawal	Const-benefit analysis and constraints relaxation	No
FeNAs [77]	Constraints	Fuzzy constraints, utility functions	Bilateral bargaining	Concessions in fuzzy constraint satisf. problem	No

Table 5.1: *Summary of representative ad-hoc automated negotiation systems.*

we want to create a system that negotiates in a specific scenario. For instance, ad-hoc automated negotiation system have been proposed to acquire goods via a mobile device [104] or to reach service agreements with a telecommunications company [44].

In this section, we focus on a subset of all ad-hoc automated negotiation systems. Specifically, our interest is in approaches that negotiate complex agreements following a bargaining protocol. By complex agreements, we mean agreements with several negotiable terms. There are many proposals that fit into this category such as References [20, 43, 44, 69, 77, 79, 87, 104, 129]. However, they differ in the scenario they deal with and in the protocol and intelligence algorithms they employ to reach their goal. For instance, Kurbel and Loutchko [79] deals with the situation of an electronic job marketplace, Paurobally and others [104] focus on acquiring goods and services from mobile devices, Byde and others [20] focus on procurement scenarios, Su and others [129] cover both business-to-business (B2B) and business-to-consumer (B2C) scenarios, and so on.

Regarding negotiation protocols, although all of them follow a bargaining bilateral protocol, they differ in aspects such as the performatives they support and the proposals they exchange. For instance, besides the typical `commit`, `accept` and `reject` performatives, Luo and others [87] include a `reward` performative and Su and others [129] include `modify`, `acknowledge` and `withdraw`. The negotiation intelligence varies also amongst the different proposals. For example, some authors [43, 77, 87] try to find trade-offs to create proposals that are appealing to the other party, whereas others [20, 43, 44, 69, 104] just concede as the negotiation progresses.

Next, we describe several representative ad-hoc automated negotiation system proposals (*cf.* Table §5.1 for a summary). The description is made based on the parts that compose an automated negotiation system. Namely, agreement and preferences models, negotiation protocol, decision making and world modelling.

Paurobally and others [104]: The authors present an automated negotiation system for provisioning mobile services. These services include mobile shopping, location-sensitive information (*e.g.* obtaining map services or local hotels), telemetry (*e.g.* logistics tracking) and mobile banking. The particularities of this scenario is that services are context-aware and location-sensitive and the inherent limitations to mobile devices such as network availability and bounded computation capabilities and, hence, negotiation mechanisms should adapt to the varying environment conditions. The agreement data structure is simple: an agreement is a set of terms (issue in the authors' terminology) and each term is an assignment of a value to an attribute. The preferences are expressed in terms of an utility function and an interval of allowed values for each possible attribute like in Reference [44] (*cf.* Chapter §3). In addition, there are also preferences about the negotiation process such as the deadline and values to parameterise the decision-making model (*e.g.* whether the system should concede at the beginning or at the end of the negotiation). It follows a bilateral protocol based on the exchange of proposals and counterproposals. This protocol includes binding (`offer` or `propose`) and non-binding (`suggest`) proposals, a timeout to deal with network problems and the ability to resume timed-out negotiations. Regarding the negotiation intelligence, the decision-making is based on Reference [44] (*cf.* Chapter §4) but it includes a network-aware strategy to adapt the negotiation behaviour to variations in the quality of the network, and an experience strategy that uses models of other parties based on previous negotiations.

Su and others [129]: This article describes an automated negotiation system suited for negotiating in both B2B and B2C scenarios. They present an architecture with several negotiation servers in which several clients can register. These negotiation servers negotiate on behalf of their clients. The authors consider an agreement as a set of terms but, unlike Paurobally and others [104], in this one, terms can be intervals instead of only concrete values. The preferences about the agreement are expressed in two different ways: first, a set of attribute constraints and another set of inter-attribute constraints are defined. These constraints define the allowed values for the terms of the agreement. Second, an utility function is given to make a cost-benefit analysis in certain phases of the decision-making. Furthermore, preferences about the negotiation process are also specified. These preferences are expressed in the form of rules and are used to relax the constraints when no proposal meets them. The protocol supported by the automated negotiation system is bilateral and it allows the modification and withdrawal of previously submitted proposals. In addition, unlike the previous automated negotiation system [104], the values of the terms of the proposal can be intervals instead of constant values. Concerning the negotiation intelligence, when the automated negotiation system receives a new proposal, it makes a constraint verification. If constraints are satisfied, then a cost-benefit analysis is carried out to select the best agreement. Otherwise, the automated negotiation system applies the relaxation rules and creates a counterproposal that is sent back to the other party. No models of the world are created by this automated negotiation system.

FeNAs: Kowalczyk [77] presents an automated negotiation system for e-commerce scenarios. In particular, two examples are described in the article: used car trading and negotiating document translation services. However, the authors argue that it can be used in other scenarios. Agreements are viewed as a set of terms and each term is a constraint on one or several attributes. The preferences are expressed as fuzzy constraints together with an additional constraint that represents the utility value of an agreement (*cf.* Section §3.3 for a more detailed description). The automated negotiation system uses a bilateral negotiation protocol that only includes accepting or rejecting proposals, sending counterproposals and withdrawing from the negotiation. The negotiation intelligence algorithms are based on viewing the process as a distributed fuzzy constraint satisfaction problem. A proposal is accepted when its satisfaction level is higher than a predetermined threshold. Counterproposals are generated by conceding progressively following a number of negotiation strategies (*cf.* Section §4.2 for more information). No world models are created in this automated negotiation system either.

5.3 Automated negotiation frameworks

Unlike ad-hoc automated negotiation systems, frameworks focus on the reusability of the different parts of an automated negotiation system. Note that an automated negotiation framework cannot be used as a negotiation system, but it defines the elements on which an automated negotiation system can be built. The term negotiation framework is used in this dissertation in a different way than in other articles [44, 87, 104]. Here we understand negotiation framework as a software framework whereas in those articles, the term framework is used in a conceptual level to define the formal founding on which the negotiation intelligence is based.

As mentioned above, there are two kinds of negotiation frameworks depending on the parts of the negotiation system they wish to reuse: protocol-oriented frameworks and intelligence-oriented frameworks.

5.3.1 Protocol-oriented frameworks

There is a large variety of negotiation protocols (*cf.* Section §3.4). Therefore, there is a lack of a standard protocol to enable the interoperation of different automated negotiation systems, which makes it very unlikely to happen that

Proposal	Agreement	Protocol	Interop.	Mediator
Kim and Segev [75]	N/A	BPEL	Shared ontology	Yes (marketplace)
Rinderle and Benyoucef [114]	N/A	Statecharts and BPEL	N/A	Yes (marketplace)
Bartolini and others [9]	OWL-Lite [36]	Taxonomy of rules	OWL-Lite ontologies	Several deployments
SilkRoad [128]	XML Schema	State-machines	Common communication meta-model	Yes (intermediary)

Table 5.2: *Summary of protocol-oriented frameworks.*

all parties use the same negotiation protocol. Moreover, the automated negotiation of agreements between two parties that are not known *a priori* may involve semantic-interoperability problems. For instance, each party could have a different definition of what response time means. Solving these problems is the aim of protocol-oriented frameworks.

Protocol-oriented frameworks follow two different approaches. Some of them [75, 114, 128] define a negotiation host or marketplace that acts as a mediator between the negotiating parties. Others describe the elements that are required to manage negotiation protocols properly [9]. Then, these elements may be deployed in a negotiation host, in the automated negotiation systems themselves or in both of them at the same time. Next, we describe several proposals of protocol-oriented frameworks that cover both approaches:

Kim and Segev [75]: This article describes a web services-enabled marketplace architecture. This architecture enables the automation of B2B negotiations from the process management perspective by defining negotiation protocols in BPEL4WS and developing a central marketplace that controls them. Specifically, the marketplace includes a repository of shared ontology and message templates that can be used to solve semantic interoperability problems. In addition, the marketplace has a repository of executable negotiation protocols defined in BPEL and it runs a

BPEL engine that executes the process and interacts with the negotiating parties through web services. Together with this negotiation host, the authors propose a semi-automatic mechanism to build those negotiation processes in BPEL. Specifically, they identify attributes in the negotiation processes and propose a pattern-based process model in which negotiation processes are generated by the selection of the patterns in each process attribute.

Rinderle and Benyoucef [114]: In this article, the authors propose a service-oriented architecture to manage negotiation protocols. Like Kim and Segev [75], they define a marketplace, which contains a set of formally specified negotiation protocols. Then, the marketplace configures the negotiation based on the protocol and passes them on to the negotiating parties. Finally, the negotiating parties use them to carry out the negotiation. The authors formally specify negotiation protocols using statechart models and, later, map them onto BPEL processes. However, this mapping is still manual and its automation is left as future work.

Bartolini and others [9]: They argue that the rules of negotiation must be made explicit instead of being represented implicitly in the automated negotiation system's design. This allows solving maintenance and reusability problems in automated negotiation systems. To this end, the authors present a taxonomy of rules that can be used to capture a variety of negotiation mechanisms. These rules include rules for admission of participants, proposal validity, protocol enforcement, updating status and informing participants, agreement formation and management of the lifecycle of the negotiation. They also define a simple interaction protocol based on FIPA specifications that is used together with the rules in order to define negotiation protocols. Therefore, the automated negotiation systems only have to care about following this simple interaction protocol and obeying the rules. In addition, based on the rules taxonomy, they define a set of roles that must be implemented to carry out a negotiation process. However, unlike the previous frameworks [75, 114], the authors do not specify a concrete deployment of these roles. Therefore, they can be placed in a negotiation host or they can be integrated in each negotiating party. Finally, to solve the semantic-interoperability problem, the authors define a language to express negotiation proposals and agreements that is based on OWL-Lite.

SilkRoad: It is a framework developed by Ströbel [128] that consist of a meta-model, the so-called roadmap, intended to capture the characteristics of a negotiation process and an application framework, the so-called skeleton, that provides several modular and configurable negotiation service components that can be used for the implementation of automated negotiation systems. The meta-model is divided into two main constructs: organisation design meta-model to specify the structure (roles) and behaviour (protocols) of the negotiation, and communication design meta-model to express syntactical and semantical representations of the negotiation terms and agreements. The organisation structure is divided into agents (buyer, seller and intermediary) and services (match, score, mediate, bid, contract and bundle) and the organisation behaviour details the interactions amongst agents and services using a state machine that is triggered by events. At runtime, the communication meta-model is transformed into XML schemas and the organisation meta-model is expressed as a state-machine table and enforced by the policy manager that invokes the services if necessary. Note that, although they provide some services that may be considered as negotiation intelligence mechanisms, the author explicitly states that the proposal does not provide means to design or represent negotiation strategies. That is the reason why we consider SilkRoad a protocol-oriented framework.

5.3.2 Intelligence-oriented frameworks

Intelligence-oriented frameworks try to make the development of automated negotiation systems easier by giving a common founding for developing negotiation intelligence algorithms and improving reusability. Furthermore, some of these frameworks [6, 55, 136] also deal with protocol management. However, instead of dealing with the interactions amongst automated negotiation systems like protocol-oriented frameworks, they focus on decoupling the protocol and the intelligence algorithms to make the latter compatible with a number of different protocols.

There are several proposals [6, 12, 55, 66, 84, 136] of intelligence-oriented frameworks but they differ in their level of abstraction and the support they give to the negotiation intelligence as well as the number of negotiation protocols they support. Next, we report on the most significant intelligence-oriented frameworks for bargaining protocols (*cf.* Tables §5.3 for a summary):

Ashri and others [6]: In this article, two architectures for negotiating agents are described: a basic negotiating agent including those participating in auctions and bilateral bargaining, and an argumentative negotiating agent that focuses on argumentation-based negotiations. Here we focus on the first one. The framework is described at a high level of abstraction and the authors use agent-based techniques to describe it. In

Proposal	**Abstraction**	**Agreement**	**Preferences**	**Protocol**
Ashri and others [6]	High	N/A	N/A	Bilateral bargaining
PANDA [55]	Medium-Low	Constraints	Utility functions, rules	Concurrent bilateral bargaining
Ludwig and others [84]	Medium-Low	WS-Agreement [4]	Utility functions, rules	Bilateral bargaining (FIPA protocols)
DynamiCS [136]	Medium	N/A	N/A	Concurrent bilateral bargaining
Benyoucef and Verrons [12]	Medium	N/A	N/A	Any but not concurrent
Jonker and others [66]	Medium-Low	Assignment of values	Utility functions (ease and financial)	Bilateral bargaining (alternating-offers)

Table 5.3: *Summary of intelligence-oriented frameworks.*

Proposal	Decision-making	World-modelling
Ashri and others [6]	Proposal evaluator and response generation	Opponent model, mental attitudes, environment model
PANDA [55]	Rule-based, deployment planner	Object pool with estimation libraries and negotiation history
Ludwig and others [84]	Rule-based	No
DynamiCS [136]	Any (example with evolutionary strategies)	No
Benyoucef and Verrons [12]	If-then rules	N/A
Jonker and others [66]	Attribute evaluation and planning	Opponent model

Table 5.3: *Summary of intelligence-oriented frameworks (Cont'd).*

particular, they define the system in terms of a descriptive specification, a structural specification and a behavioural specification. The system specified in the framework receives incoming messages from the other negotiating parties and returns outgoing messages as responses. The protocol is managed by a protocol reasoner, which checks the incoming messages with the protocol rules and generates the available responses, and the locution generation, which encapsulates the generated responses into the proper messages. The model of the world is decomposed into three components: the opponent model, mental attitudes and the environment model. However, the article does not detail more about these components. Finally, the decision making is carried out by the proposal evaluator, and the response generator, which uses the evaluation, the possible responses and the models of the world to create a response.

PANDA: PANDA (Policy-driven Automated Negotiation Decision-making Approach) [55] is an object-oriented framework to carry out negotiations automatically. An agreement proposal is composed of non-negotiable terms such as the name of the parties and variable terms that may contain single values and may be restricted by intervals, enumerations of values, simple logical constraints and inter-attribute constraints. The preferences are a mix of utility functions and rules. The assumptions on the protocol made by the framework are weak for keeping it applicable to many different negotiation scenarios. Specifically, the authors consider bilateral bargaining protocols in which the data exchanged are proposals and the performatives supported are `accept`, `reject`, `offer`, `withdraw` and `terminate`. The decision-making component is composed of rules, utility functions and an object pool with several estimation libraries, the negotiation history and the current offer. In addition, the PANDA framework allows combining multiple decision-making components to build a more complex system. For instance, the authors propose a negotiation system architecture for infrastructure providers with three decision-makers: a negotiating agent, which develops a bilateral negotiation; a negotiation coordinator, which coordinates multiple negotiating agents to carry out multiple simultaneous negotiations; and an utility update, which updates the resource utility function as a function of the available resources in the server. Furthermore, they include a deployment planner to evaluate the feasibility of accepting an agreement and a risk function to calculate associated risks.

Ludwig and others [84]: In this article, the authors present a framework for automated negotiation of service level agreements in service grids. The underlying principle of the framework is the decomposition of the negotiation into negotiation object, negotiation protocol and decision making model [64]. More specifically, the authors take WS-Agreement [4] as a starting point and extend its agreement layer with elements that deal with the negotiation process. Thus, the agreement layer proposed is composed of three components: agreement provider and initiator, protocol service provider and decision making service provider. Each component deals with a different part of the negotiation: the agreement provider and the initiator provide the interfaces that are necessary for interacting with a provider during service negotiation and it is responsible for describing the negotiation object; the protocol service provider deals with the negotiation protocol and it is responsible for making sure the rules of the protocol are followed and the messages are correctly managed, and the decision making service provider encapsulates the decision making model. The authors propose using FIPA protocols (*cf.* Section §3.4) to implement protocol service providers and the syntax and semantics of the PANDA framework in the decision making service providers. However, the decomposition of the framework into different services allows it to use several protocol service providers with different protocols implemented in them and several decision making service providers with different levels of complexity.

DynamiCS: This is an actor-based framework developed by Tu and others [135, 136]. The framework is founded on a dynamic plug-in mechanism that allows modules to bee selected and composed dynamically depending on the requirements of the environment. Each module decomposes into a set of tasks which can be dynamically assigned to active objects (actors). The framework itself can be decomposed into four modules: the communication module, the protocol module, the strategy module and the rules module. The communication an the protocol module are in charge of managing the negotiation protocol whereas the rules and the strategy modules manage the negotiation intelligence algorithms. The rules module is used to define the preferences in the form of invariants, policies and action rules. The strategy module is composed of two kinds of actors: the coordinator and the strategy actor. The coordinator is responsible for checking the validity of execution constraints. These can be classified into strategy constraints, which control the execution of actors that model a negotiation strategy; agent constraints, which reflect the inner state of the agent; and negotiation constraints, which reflect the state of a negotiation. Moreover, the coordinator is also responsible for invoking the concrete strategy actors. Each strategy actor implements a different decision-making model. In addition, the authors argue that decision-making algorithms should deliver usable results even when interrupted at any point of computation.

Benyoucef and Verrons [12]: In this article, the authors present their approach to a configurable framework for designing, implementing and deploying automated negotiation systems. Their approach is based on the separation of protocols and strategies from the system and they adopt a service-oriented architecture to make it easier the deployment and integration with current infrastructures. The authors separate the protocol management and the negotiation intelligence into two different components, namely: the e-negotiation server and the automated e-negotiation interface, respectively. By doing so, they achieve a clear separation of concerns. Furthermore, they consider that negotiation protocols and strategies should be treated as declarative knowledge. In particular, they argue that negotiation protocols can be expressed using UML statecharts and negotiation strategies can be represented as if-then rules. Later, these UML statecharts are transformed into BPEL processes that are executed in the engine in the e-negotiation server, and the negotiation strategies are executed by software agents in the automated e-negotiation interface. This allows to modify the negotiation strategy dynamically during the negotiation. In addition, both e-negotiation server and automated e-negotiation interface can use additional services to complement them such as authentication and non-repudiation for the server, and decision support systems and advisor services for the interface. Both of them can also integrate with back office systems that may provide internal information for the decision making process such as taking provider's capacity into account in binding decisions.

Jonker and others [66]: The authors describe a component-based generic agent architecture for multi-attribute negotiation. The negotiation considered follows an alternating-offers protocol and the agreement has the form of values assigned to a number of attributes. No concurrent negotiations are supported. The negotiation component is composed of five components: negotiation coordinator, which decides the next action depending on the last events of the negotiation; attribute evaluation, which evaluates the proposals at two levels: the level of the overall proposal and the level of each of the attributes; bid utility determination, which calculates two different utilities of a proposal: financial utility, which covers the financial rationality, and ease utility, which models all other aspects within the decision making; utility planning, which determines the concession step for the next proposal; and attribute planning, which determines the proposal that shall be sent to the other party based on the concession step. The attribute planning can be composed of two or five components depending on whether the automated negotiation system create models of the other parties. If no models are created, the attribute planning is composed of target evaluation determination, which calculates the utility that each attribute must have in the proposal, and configuration determination, which calculates the values of these attributes. If the system creates models of the other parties, the attribute planning is composed of five components: target evaluation determination, configuration determination, estimation of opponent's parameters, provision of initial guess information and guess coefficients.

5.4 Summary

In this chapter, we have focused on the problem of building automated negotiation systems. First, we have analysed several ad-hoc automated negotiation systems that perform a negotiation process with a concrete protocol and specific negotiation intelligence algorithms. Then, we have reviewed a number of frameworks that have been proposed to make the development of automated negotiation systems easier. These frameworks can be classified into protocol-oriented frameworks, which focus on the external part of an automated negotiation system, and intelligence-oriented frameworks, which deal with the internals of such systems.

Part III

Our approach

Chapter 6

NegoFAST in a nutshell

O God! I could be bounded in a nutshell,
and count myself king of infinite space,
were it not that I have bad dreams.

William Shakespeare, 1564–1616
English Dramatist, and Poet

From the previous chapters, we conclude that it is appealing to build a framework that facilitates the development of automated negotiation systems of service agreements in open and dynamic environments. Our goal in this chapter is to present NegoFAST as a general approach to build such automated negotiation systems. NegoFAST is described at three levels of abstraction: the reference architecture, a framework, and an implementation. This chapter is organised as follows: in Section §6.1, we introduce NegoFAST; in Section §6.2, we detail a conceptual map to define the concepts that appear in an automated negotiation system and its general structure; in Section §6.3, we present the NegoFAST reference architecture; in Section §6.4, we describe the NegoFAST framework, which is a materialisation of the reference architecture; in Section §6.5, we outline August, our proof-of-concept implementation of the NegoFAST framework; finally, we summarise our contributions in Section §6.6.

6.1 Introduction

As we state in Chapter §1, the primary focus of our research work is to provide engineering support so that software developers can develop automated negotiation systems that negotiate service agreements in open and dynamic environments. The result of this research work is NegoFAST, which is our approach to build automated negotiation systems.

NegoFAST is defined at three levels of abstraction. The higher level of abstraction is the NegoFAST reference architecture, described in Chapters §7 and §8). It provides a foundation for developing automated negotiation systems and defines the elements and interactions that are part of them. The mid level of abstraction is the specification of the NegoFAST software framework, described in Chapters §9 and §10. It defines a set of interfaces and a data model that conforms to the reference architecture. Finally, the lower level of abstraction is August, which is a reference implementation of the NegoFAST software framework.

The main design goal of NegoFAST is twofold. On the one hand, NegoFAST has been designed to promote the reusability of the elements, so that they may be freely integrated as a whole to produce automated negotiation systems. On the other hand, NegoFAST has been designed to be flexible enough to adapt to the models, algorithms and protocols that are best suited for each negotiation scenario. To this end, NegoFAST is divided into a protocol-independent part, the so-called NegoFAST-Core, and protocol-specific extensions. In this dissertation, we describe a bargaining-specific extension, NegoFAST-Bargaining. However, other extensions to support different negotiation protocols can be added.

Furthermore, NegoFAST has been designed to be independent from whether its user acts as a service consumer or a service provider. Therefore, it is symmetric in the sense that both provider and consumer can simultaneously be negotiating with other parties that are interested in creating an agreement.

This approach, based on protocol-specific extensions, enables dealing with the different requirements that pose different negotiation protocols (*cf.* Chapters §3 and §4), while keeping the other elements of the automated negotiation system reusable.

Table §6.1 summarises the different levels of abstraction and extensions that have been defined as part of this dissertation as well as the chapters in which they are described.

Abstraction	Protocol-independent	Protocol-specific
High	NegoFAST-Core reference architecture (Chapter §7)	NegoFAST-Bargaining reference architecture (Chapter §8)
Medium	NegoFAST-Core framework (Chapter §9)	NegoFAST-Bargaining framework (Chapter §10)
Low	August	August

Table 6.1: *Summary of levels of abstraction and extensions.*

6.2 Preliminaries

From the study of the background information developed in the previous chapters, we may conclude the conceptual map of an automated negotiation system that is depicted in Figure §6.1.

The *user* of the *automated negotiation system* defines a *preferences document*, which is the set of data that it is used to ensure an agreement is reached according the *user*'s needs, and initiates the *automated negotiation system* to negotiate on his or her behalf. The *automated negotiation system* is also provided with *party references*. A *party reference* gives a means to interact with a *party*. *Party references* may be provided by the *user* or they may directly come from the *parties* as a request to start a protocol negotiation or a negotiation. *Parties* may also provide *public information about its preferences* to facilitate the negotiation process.

For each *party reference* received, the *automated negotiation system* starts a process whose goal is to execute a *negotiation protocol instance* with the *party* whose reference was received. A *negotiation protocol instance* is a concrete execution of a *negotiation protocol* played by two or more *parties*. A *negotiation protocol* belongs to a *negotiation protocol family*, which is a group of negotiation protocols with some common characteristics. Depending on the characteristics we consider, the granularity of negotiation protocol families may be higher (*e.g.* auction protocols and bargaining protocols) or lower (*e.g.* bargaining proposal-based protocols and bargaining argumentation protocols). An *abstract negotiation protocol* of a *negotiation protocol family* is a superset of all *negotiation protocols* of the *negotiation protocol family*.

Negotiation protocol instances are configured by a *protocol configuration*,

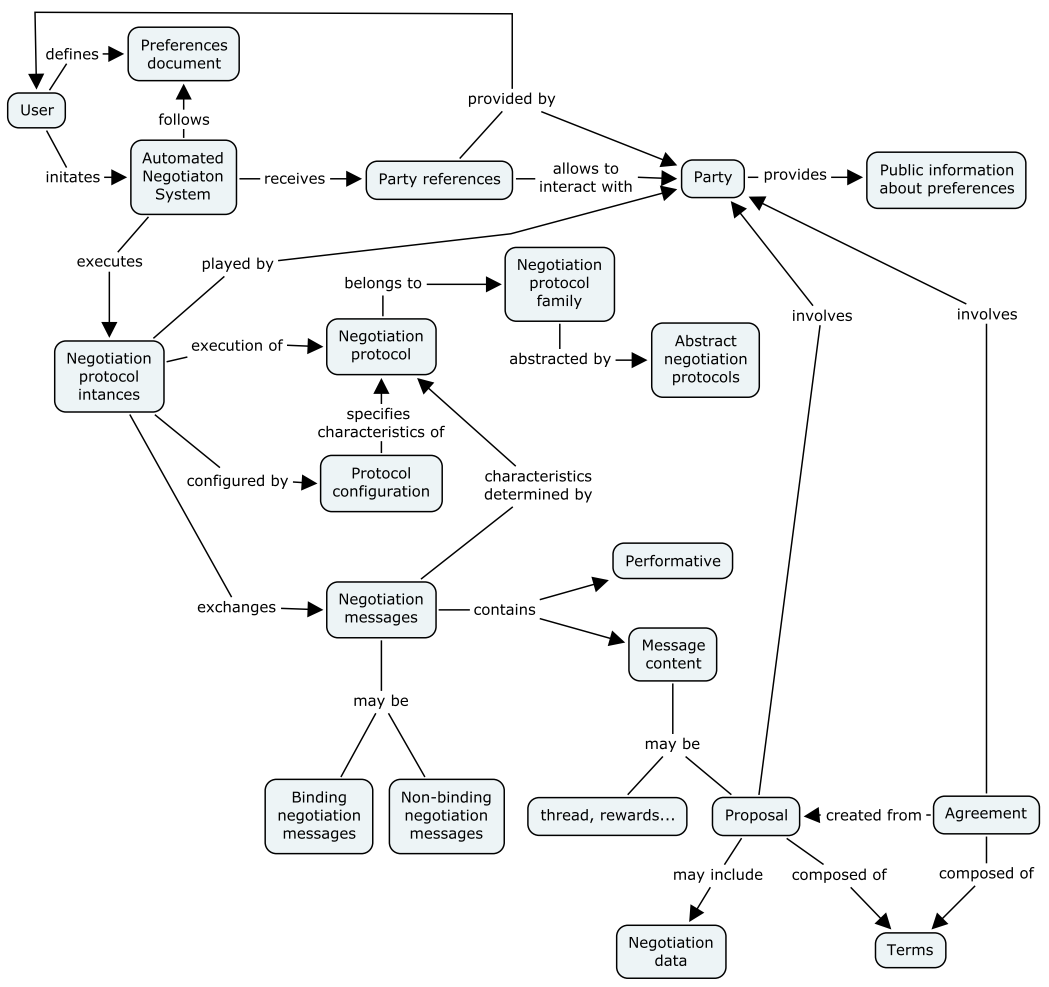


Figure 6.1: *Conceptual map of an automated negotiation system.*

which specifies characteristics of the *negotiation protocol* for a particular execution. For instance, they may specify the timeout of the protocol or its security features.

The execution of a *negotiation protocol instance* involves the exchange of *negotiation messages*, whose specific characteristics are determined by the negotiation protocol. All *negotiation messages* contain a *performative* to express the intention of the sender about the message (*cf.* Section §3.4 for more information about performatives) and a *message content*. Furthermore, depending on the *performative*, *negotiation messages* can be classified as *binding negotiation messages*, which involve a firm commitment with the other *party*, and *non-binding negotiation messages*, which do not involve such a firm commitment. *Non-binding negotiation messages* can be used to give additional information to the other party or to explore new possible areas of agreement without the need of committing to that proposals. This is very useful in dynamic contexts in which there are several simultaneous negotiations with different *parties* and the feasibility of committing to an agreement depends on the current state of the service provider's resources.

The *message content* of a *negotiation message* is usually a *proposal*. However, as described in Section §3.4, other kinds of information can be exchanged such as *threats*, *rewards* or *arguments*. A *proposal* is an offer for an agreement made by one *party* to another. A *proposal* is composed of the *parties* the proposal is about and a set of *terms*. *Terms* describe both functional descriptions and non-functional guarantees of the service. Some examples of usual *terms* are: *the service interface is specified in document http://example.org/myservice.wsdl*, *the response time is less than 20 ms*, or *the number of service requests is lower than 10 times per minute*. Additionally, *proposals* may also include *negotiation data* referring to some *terms*. *Negotiation data* expresses additional information to guide the negotiation process such as information to relax constraints specified in some *terms* of the *proposal*, to express possible trade-offs amongst *terms* (e.g. *I accept a response time higher than 20 ms provided that the price is lower than 50 cents per request*), or to indicate whether a *term* is negotiable.

Finally, when *parties* agree on a *proposal*, an *agreement* is created. An *agreement*, as stated in Section §3.2, is a document that defines a relationship between *parties*. Its goal is to define the *terms* that regulate the execution of services and it must have a specification of the parties involved in it and a collection of *terms*, such as those described in the *proposal*. These *terms* regulate how the execution of the service must be carried out in the context of the agreement. In addition, unlike *proposals*, in which the *terms* can be left open in order to be refined later, *agreement terms* must be fully specified and ambiguities must be avoided.

Together with these concepts, a general structure of an automated negotiation system can be developed, as depicted in Figure §6.2. The automated negotiation system is represented as a big white box in the middle of the figure, and the external entities that interact with it are depicted as little grey boxes. The automated negotiation system needs two inputs that are provided by the *user*: a preferences document and a set of party references with whom our system is going to negotiate. To obtain these party references, a previous discovery and pre-selection phase may have been carried out [110]. Furthermore, the automated negotiation system may also obtain party references from incoming negotiation requests made by other parties.

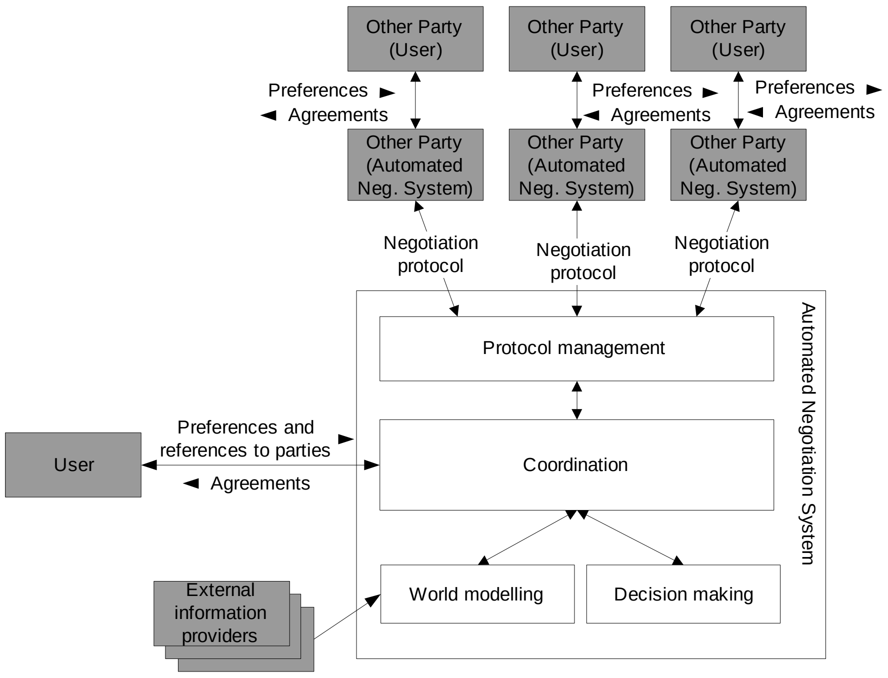


Figure 6.2: *General structure of an automated negotiation system.*

After receiving the input, the automated negotiation system starts negotiations following a negotiation protocol on which the parties have agreed previously. This negotiation protocol is implemented by the protocol management to make the system independent from concrete negotiation protocols. The behaviour of the automated negotiation system during the negotiation is determined by some decision-making algorithms that it must implement, such as those described in Section §4.2. In addition, these decision-making algorithms may be supported by models created about the other negotiating parties, the market or the negotiation domain (*cf.* Section §4.3). To create these models, the automated negotiation system may need to query *external information providers*. Protocol management, decision making and world modelling are coordinated by the coordination element.

At the end of the negotiations, the automated negotiation system reaches agreements with one or several parties. These agreements are sent back to the *user* and constitute the output of the automated negotiation system. Note that the *user* does not necessarily represent a human user, it may also represent a software system.

6.3 The NegoFAST reference architecture

The NegoFAST reference architecture is the highest level of abstraction in NegoFAST. It provides a foundation for developing automated negotiation systems and defines the elements and interactions that are part of an automated negotiation system. To model the NegoFAST reference architecture, we use the organisational metaphor proposed in the Gaia methodology [138]. The Gaia methodology decomposes an architecture into organisations, roles, interactions and environment. Organisations group several roles and have a concrete and well defined goal; roles are precisely defined tasks that must be carried out by one or more software artefacts; interactions represent the exchange of messages of two or more roles; and the environment determines the resources that roles can exploit, control or consume while working towards the achievement of the organisational goal.

6.3.1 The NegoFAST-Core reference architecture

Figure §6.3 depicts the NegoFAST-Core reference architecture. Organisations are depicted as big boxes and each corresponds to the parts of an automated negotiation system described in the previous section: protocol management, coordination, world modelling and decision making.

Each organisation is composed of several roles (depicted as small light grey boxes). Protocol management is composed of a ProtocolNegotiator and a ProtocolHandler; coordination consist of a SystemCoordinator, a PartyCoordinator and a NegotiationCoordinator; world modelling includes an Inquirer, an Informant and a WorldModeller; and decision making comprises a ResponseGenerator, a CommitHandler, and a CommitAdvisor. The arrows connecting the roles represent the interactions amongst them.

The environment is divided into several resources, which are the *AgreementsResource*, the *PreferencesResource*, the *SystemContextData*, the *PartyContextData*, the *NegotiationContextData*, the *WorldModel* and the *NegotiationHistory*. They are depicted as white boxes.

Finally, elements that are external to the architecture such as other *parties*, *external information providers*, or *user* are depicted as small dark grey boxes.

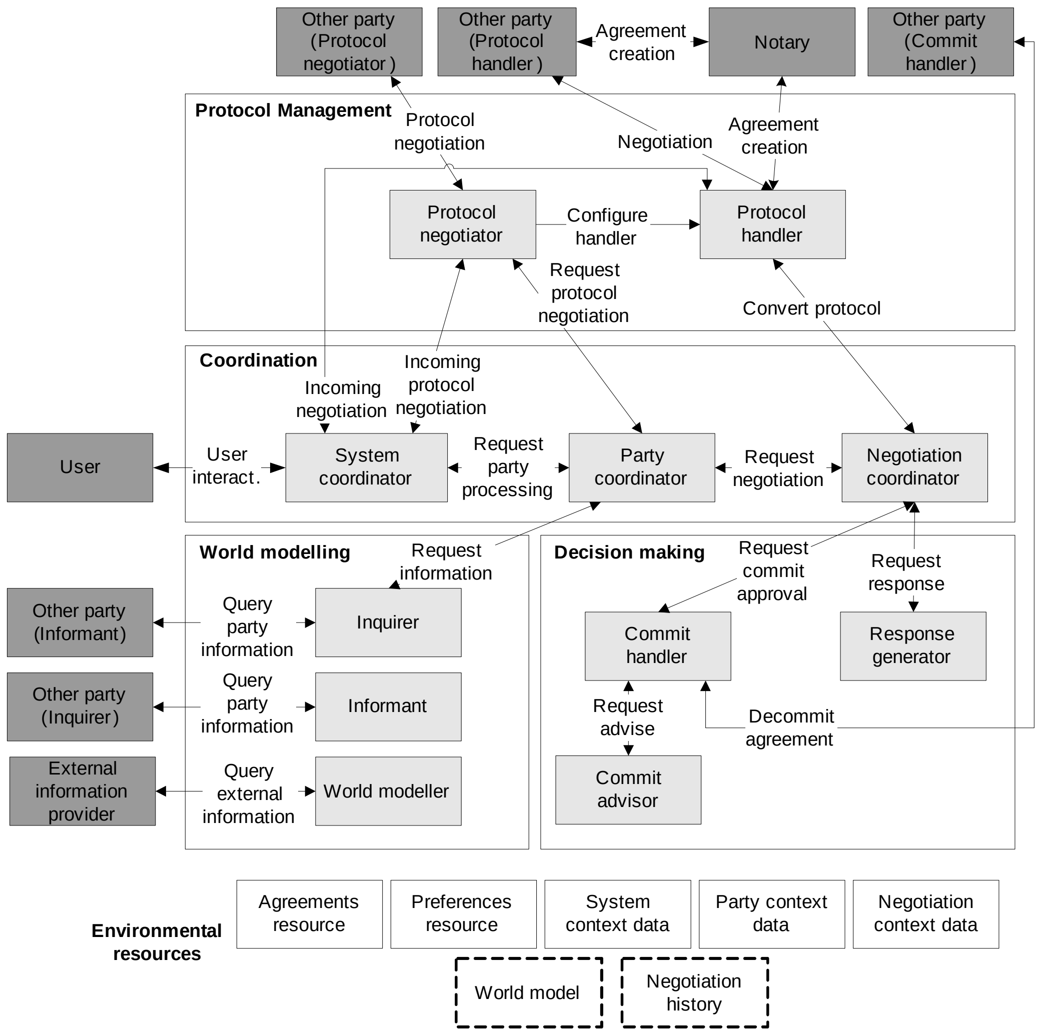


Figure 6.3: *The NegoFAST-Core reference architecture.*

6.3.2 The NegoFAST-Bargaining reference architecture

The NegoFAST-Bargaining reference architecture extends NegoFAST-Core to deal with the specific requirements of concurrent proposal-based bilateral negotiations of multi-term agreements. This is a type of bargaining negotiation in which two parties exchange proposals sequentially to reach an agreement (*cf.* Section §3.4). The extension of NegoFAST-Core to deal with these requirements involves extending roles, interactions and environmental resources.

Roles are extended by decomposing them as depicted in Figure §6.4. The

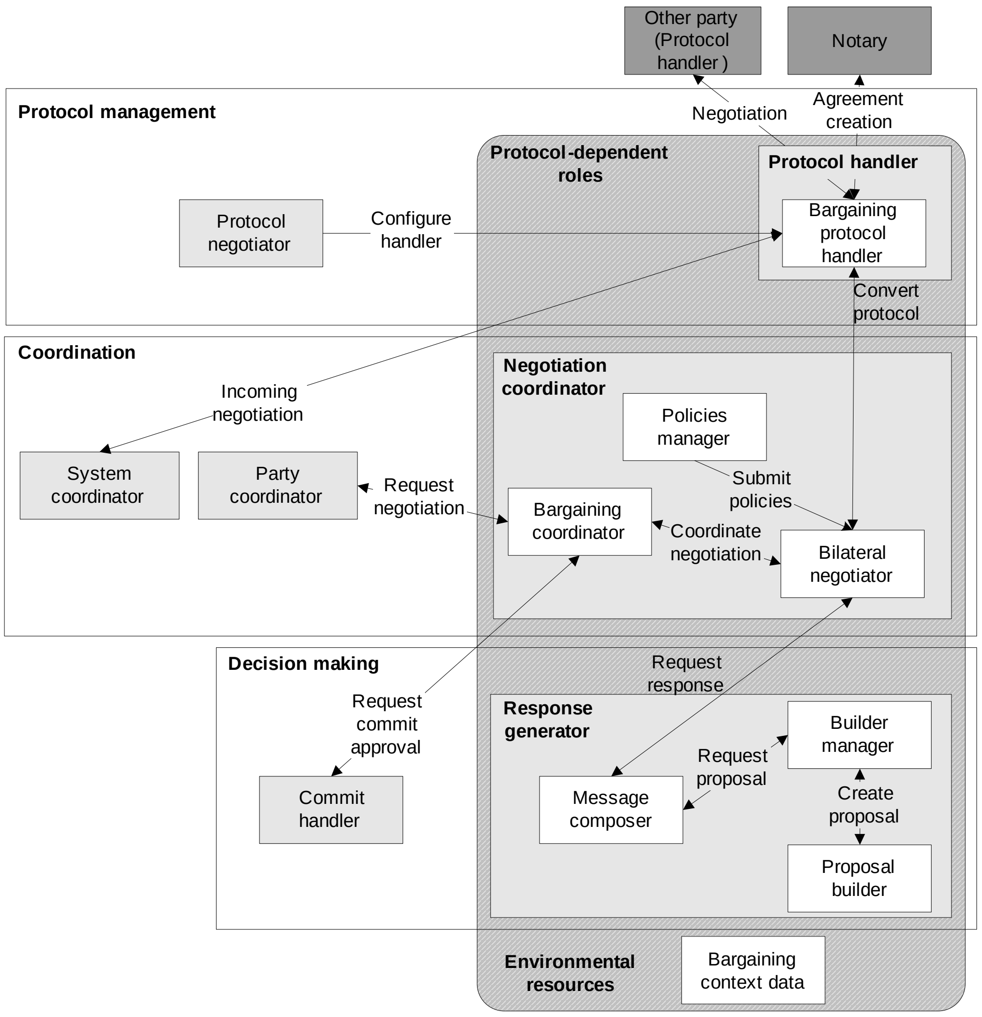


Figure 6.4: *The NegoFAST-Bargaining reference architecture.*

ProtocolHandler remains as one role, the BargainingProtocolHandler; the NegotiationCoordinator is divided into the BilateralNegotiator, the BargainingCoordinator and the PoliciesManager; and the ResponseGenerator is divided into the PerformativeSelector, the BuilderManager and the ProposalBuilder.

Regarding the interactions, NegoFAST-Bargaining defines new interactions between roles created after the decomposition (*CoordinateNegotiation*, *SubmitPolicies*, *RequestProposal* and *CreateProposal*). In addition, it also extends interaction *RequestResponse* of NegoFAST-Core to allow the submission of policies to guide the generation of responses.

Environmental resource *NegotiationContextData* is extended by resource

BargainingContextData to store and provide information regarding the current status of bargaining negotiations.

6.4 The NegoFAST framework

The medium level of abstraction is the specification of the NegoFAST software framework. The NegoFAST software framework is a materialisation of the NegoFAST reference architecture, in which the following elements are defined: a data model to represent the concepts that are managed by the roles of the reference architecture; the interfaces of the environmental resources; the interfaces of the interactions between roles; and the state machines of the roles of organisation coordination. Note that other materialisations of the NegoFAST reference architecture are also possible.

The NegoFAST framework includes the elements that conform an automated negotiation system, not the interactions with external parties. These interactions should be made using a standardised or mutually agreed protocol that fall beyond the scope of this dissertation.

The NegoFAST framework is divided into the NegoFAST-Core framework and the NegoFAST-Bargaining framework. Note that the NegoFAST-Core framework can be used without the NegoFAST-Bargaining framework and, hence, it can be reused for other protocol families (*e.g.* auctions) provided that the corresponding extension is defined.

6.4.1 The NegoFAST-Core framework

Figure §6.5 depicts the NegoFAST-Core framework, which is a materialisation of the NegoFAST-Core reference architecture. The NegoFAST-Core framework can be decomposed into four packages, namely: `model`, `environment`, `interactions` and `roles`.

Package `model` provides a data model for all protocol-independent concepts managed by the other parts of the NegoFAST-Core framework. Package `model` specifies a generic model to define the more general concepts like agreement, preferences, proposal or negotiation message. These concepts must be further refined to support concrete models for expressing agreements and preferences. This enables choosing different models to express agreements and preferences while keeping the reusability of the general concepts.

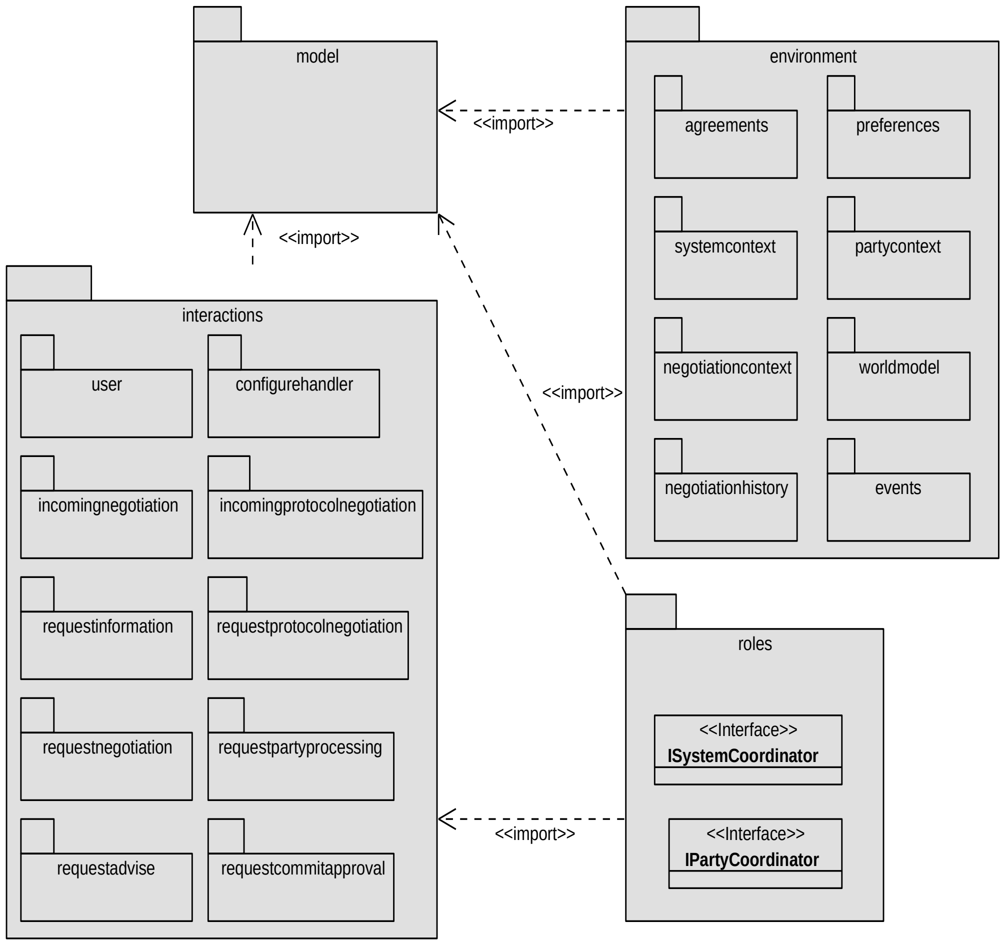


Figure 6.5: *NegoFAST-Core framework packages.*

Package `environment` includes interfaces for all environmental resources of NegoFAST-Core (*AgreementsResource*, *PreferencesResource*, *SystemContextData*, *PartyContextData*, *NegotiationContextData*, *WorldModel* and *NegotiationHistory*) as well as the data structures that contains the information stored in them. Furthermore, package `environment` also includes package `events`, which provides a publish/subscription mechanism [42] to avoid a continuous polling of the elements of the framework to detect when some event occurs. This package uses the data model provided by package `model`.

Package `interactions` includes the interfaces of the protocol-independent interactions that are internal to the automated negotiation system (*UserInteraction*, *IncomingProtocolNegotiation*, *IncomingNegotiation*, *RequestProtocolNegotiation*, *RequestNegotiation*, *ConfigureHandler*, *RequestCommitAp-*

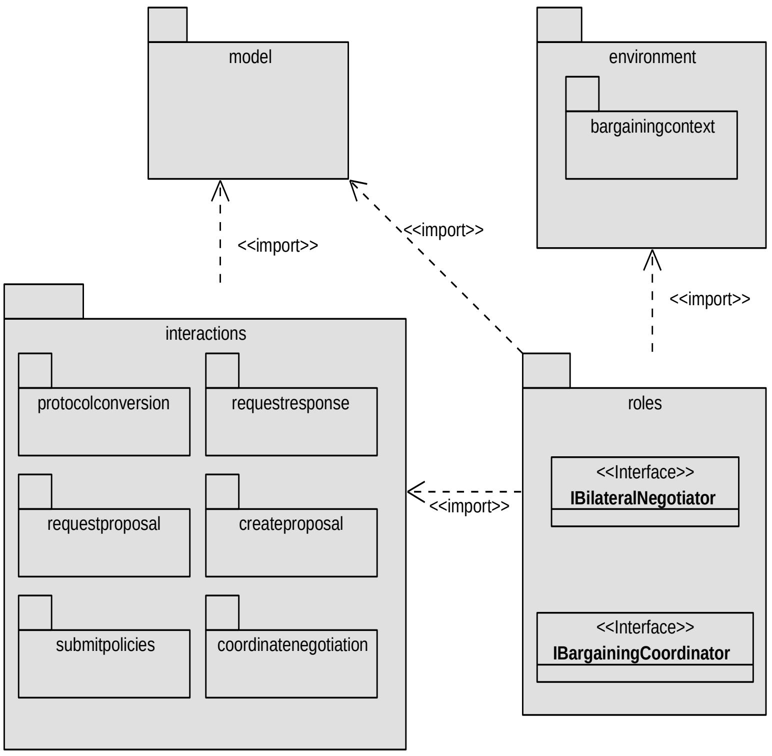


Figure 6.6: *NegoFAST-Bargaining framework packages.*

proval and *RequestInformation*). These interactions use concepts defined in package `model`.

Package `roles` provides a specification of the state machine of the SystemCoordinator and the PartyCoordinator, which orchestrates the other roles of the framework. This specification of the state machine uses both package `model` and package `interactions`.

6.4.2 The NegoFAST-Bargaining framework

The NegoFAST-Bargaining framework extends the NegoFAST-Core framework with a materialisation of the NegoFAST-Bargaining reference architecture. In other words, it provides a bargaining-specific extension to the NegoFAST-Core framework. Like the NegoFAST-Core framework, the NegoFAST-Bargaining framework can be decomposed into four packages (*cf.* Figure §6.6), namely: package `model`, which provides a data model to represent the bargaining-specific concepts that are necessary by the other packages of the NegoFAST-Bargaining framework; package `environment`, which includes resource *BargainingContextData*; package `interactions`, which includes the interfaces of the bargaining-specific interactions that are internal to the automated negotiation system (*ProtocolConversion*, *CoordinateNegotiation*, *SubmitPolicies*, *RequestResponse* *RequestProposal* and *CreateProposal*); and package `roles`, which provides a state-machine specification of the behaviour of the BargainingCoordinator and the BilateralNegotiator.

6.5 August, a proof-of-concept implementation

August is the proof-of-concept implementation of the NegoFAST framework. It has been developed using Java 1.5 and can be fetched from http://www.tdg-seville.info It provides a reference implementation of the interfaces specified in the NegoFAST framework. August has been used to implement three different automated negotiation systems for different scenarios. These implementations are described in Appendices §A, §B and §C.

6.6 Summary

In this chapter, we overview NegoFAST, which is our approach for building automated negotiation system. NegoFAST is defined at three levels of abstraction, which are outlined in this chapter and comprises the NegoFAST reference architecture, the NegoFAST framework and the August proof-of-concept implementation. The next chapters analyse them in deeper detail.

Chapter 7

The NegoFAST-Core reference architecture

If you have built castles in the air, your work need not be lost. Put the foundations under them.

Henry D. Thoreau, 1817–1862
American Philosopher

The goal of this chapter is to detail the NegoFAST-Core reference architecture as a general approach to build automated negotiation systems. It is organised as follows: in Section §7.1, we overview the main structure of NegoFAST-Core that was presented in the previous chapter; in Section §7.2, Section §7.3, Section §7.4 and Section §7.5, we detail the roles and interactions of the reference architecture; in Section §7.6, we describe the environmental resources; finally, we summarise our contributions in Section §7.7.

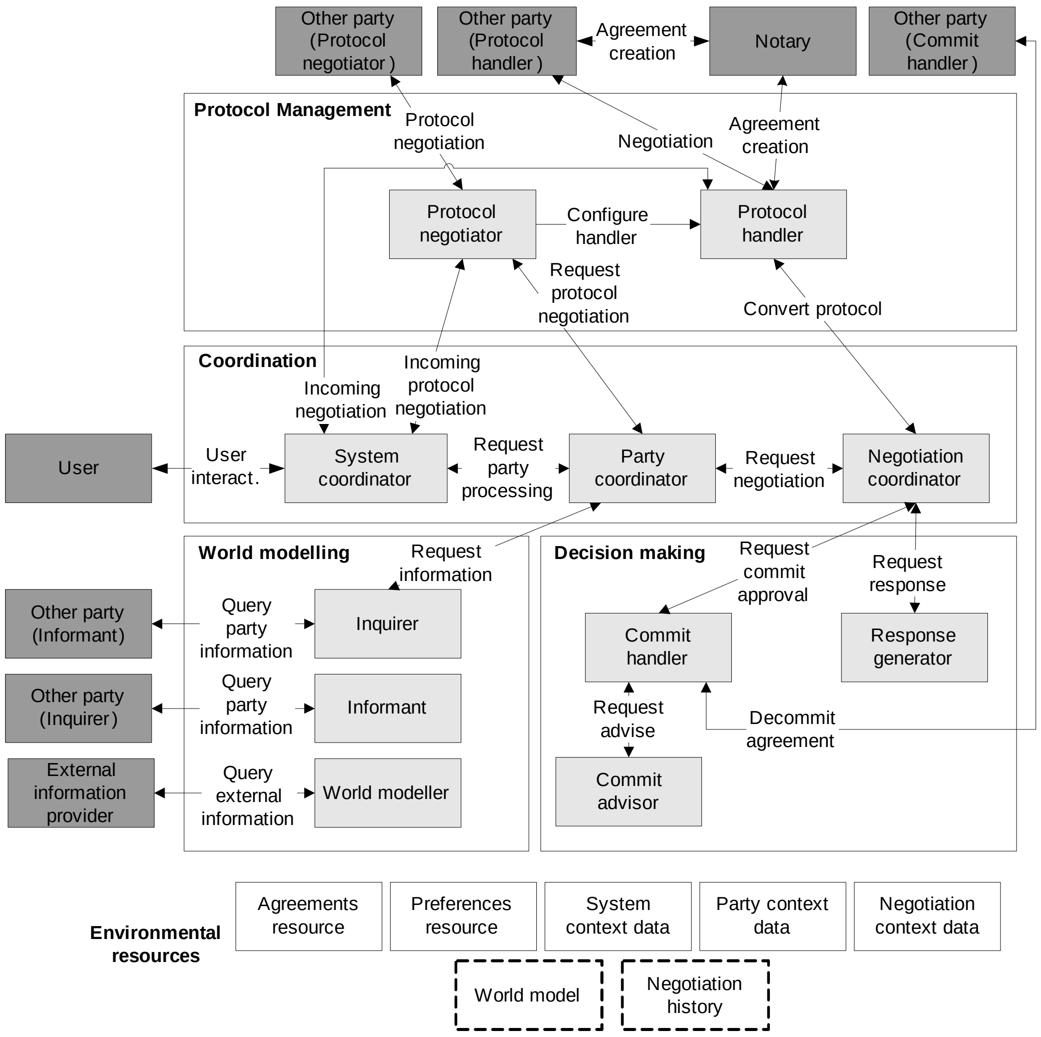


Figure 7.1: *The NegoFAST-Core reference architecture.*

7.1 Introduction

In this chapter, we detail the NegoFAST-Core reference architecture, which was briefly introduced in Section §6.3.1. The goal of NegoFAST-Core is to describe the protocol-independent elements that are necessary to develop automated negotiation systems. In the following sections, we describe the organisations and environmental resources that are defined in NegoFAST-Core (*cf.* Figure §7.1).

Context	Coordinator	Resource
System	SystemCoordinator	*SystemContextData*
Party	PartyCoordinator	*PartyContextData*
Negotiation	NegotiationCoordinator	*NegotiationContextData*

Table 7.1: *Contexts of NegoFAST-Core.*

7.2 Coordination

The coordination of an automated negotiation system is carried out by the coordination organisation. At runtime, NegoFAST-Core defines three contexts (*cf.* Table §7.1): system context, which includes the whole process of reaching agreements following the user preferences; party context, which involves the processing of a party reference and includes getting information about the party and negotiating the negotiation protocol; and negotiation context, which focuses on the execution of negotiation protocol instances.

In an automated negotiation system, there is only one system context at a time. In other words, an automated negotiation system only negotiates on behalf of one user at a time. This allows avoiding privacy concerns between users of the automated negotiation system. Regarding party contexts, there may be many different party contexts running simultaneously. Specifically, there are n_p party contexts, where n_p is the number of party references that are being processed by the automated negotiation system.

Finally, there are n_n negotiation contexts running simultaneously, one for each negotiation protocol instance that is being executed by the automated negotiation system. Each party context is associated to a negotiation context, *i.e.*, the automated negotiation system establishes just one negotiation protocol instance for each party reference it knows. However, one negotiation context may have one or several associated party contexts depending on the number of parties that participate in a negotiation protocol instance. For example, if the automated negotiation system is acting as the initiator in a bargaining protocol, in which there are two participants, the negotiation context corresponds to only one party context. Nevertheless, if the automated negotiation system is acting as the auctioneer in an auction protocol, the negotiation context corresponds to several party contexts: one for each bidder in the auction.

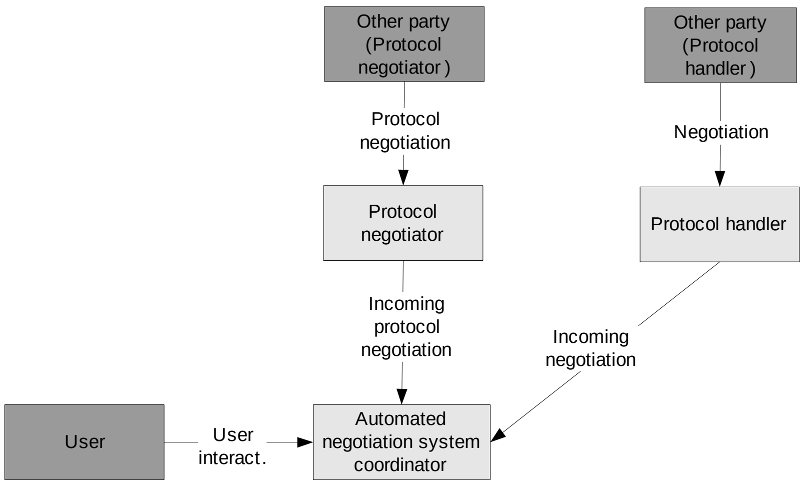

Other party (Protocol negotiator)
Other party (Protocol handler)
Protocol negotiation
Negotiation
Protocol negotiator
Protocol handler
Incoming protocol negotiation
Incoming negotiation
User
User interact.
Automated negotiation system coordinator

Figure 7.2: *Sources of party references.*

7.2.1 Roles

The organisation coordination is composed of four roles: the User, the SystemCoordinator, the PartyCoordinator and the NegotiationCoordinator. The User represents the actor on whose behalf the automated negotiation system is negotiating. The SystemCoordinator is the façade of the system for the User and is responsible for receiving the party references (*cf.* Figure §7.2). The PartyCoordinator orchestrates the other roles of the negotiation system to process those party references. The NegotiationCoordinator supports a negotiation protocol family, orchestrates the protocol management and the decision making organisation and coordinates several simultaneous negotiations with a number of parties. Note that, unlike SystemCoordinator and PartyCoordinator, which are protocol-independent, the NegotiationCoordinator is associated to one abstract negotiation protocol and, hence, it just supports one negotiation protocol family.

Role: User

Goal: It represents the external actor on whose behalf the automated negotiation system is negotiating. It provides the system with its preferences and receives the agreement documents that have been reached. In addition, it also may supply the automated negotiation system with party references.

Interactions: *UserInteraction*.

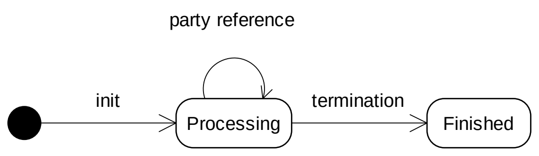


Figure 7.3: *State machine of the system context.*

Environmental resources: None.

Behaviour: The User initialises the automated negotiation system with its preferences. Since that moment, the automated negotiation system negotiates on his or her behalf following the given preferences. When it reaches an agreement with another party, it sends it back to the User. In addition, the User may provide the automated negotiation system with party references.

Note that the User may be a human user interacting with the automated negotiation system by means of a graphical user interface or it may be a software system that set the preferences of the automated negotiation system automatically.

Role: SystemCoordinator

Goal: It is responsible for coordinating the system context. This involves the interaction of the automated negotiation system and the User, the initialisation and termination of the system context and the reception of party references from the User, the ProtocolNegotiator and the ProtocolHandler.

Interactions: *UserInteraction*, *IncomingNegotiation*, *IncomingProtocolNegotiation*, and *RequestPartyProcessing*.

Environmental resources: It writes to the *PreferencesResource*, *SystemContextData*, *AgreementsResource*, and *NegotiationHistory*.

Behaviour: The SystemCoordinator is responsible for coordinating the system context and storing its status in resource *SystemContextData*. The system context starts when the User sends his or her preferences to the SystemCoordinator (*cf.* Figure §7.3). Then, the SystemCoordinator stores them into the *PreferencesResource* and initialises the other roles and resources in the architecture.

After receiving the preferences, the system context enters state *processing*. In that state, the SystemCoordinator receives party references from three different sources (*cf.* Figure §7.2), namely: the

User, the ProtocolNegotiator as a request to start a protocol negotiation from other party, and the ProtocolHandler as a request to start a negotiation from other party. For each reference received, the SystemCoordinator initiates interaction *RequestPartyProcessing* to ask the PartyCoordinator to process the party reference.

Furthermore, the SystemCoordinator checks the termination conditions of the system context and ends it if any of them holds. The termination conditions are: a preestablished negotiation deadline is reached, a desired number of agreements is achieved, or the User sends a stop signal to the negotiation system. Note that, the deadline and the desired number of agreements may be unlimited and, hence, the system context may run indefinitely until an external event stops it. When the system context finishes, the environmental resources that are specific to a system context (*AgreementsResource*, *PreferencesResource*, *SystemContextData*, *PartyContextData* and *NegotiationContextData*) are stored in the *NegotiationHistory* and it is reinitialised.

Role: PartyCoordinator

Goal: It is responsible for coordinating party contexts. In other words, it coordinates the processing of party references, since they are received, until the negotiation with the party finishes.

Interactions: *RequestPartyProcessing*, *RequestProtocolNegotiation*, *RequestNegotiation*, and *RequestInformation*.

Environmental resources: It writes to the *PartyContextData* and may read the *PreferencesResource*, the *AgreementsResource*, and the *NegotiationHistory*.

Behaviour: The behaviour of the PartyCoordinator focuses on the management of party contexts, which involves orchestrating the other roles to fulfill the states of the party context and updating the *PartyContextData*. Each party reference that the SystemCoordinator receives is sent to the PartyCoordinator to be processed. After receiving the party reference, the PartyCoordinator starts a new party context. The states of a party context are depicted in Figure §7.4:

Pending. This is the initial state and all party contexts remains in it until the PartyCoordinator decides which is their next state.

Getting information. The goal of this state is to obtain the necessary information about the party to start a negotiation with it. To this end, the PartyCoordinator uses interaction *RequestInformation* to requests the Inquirer to gather that information.

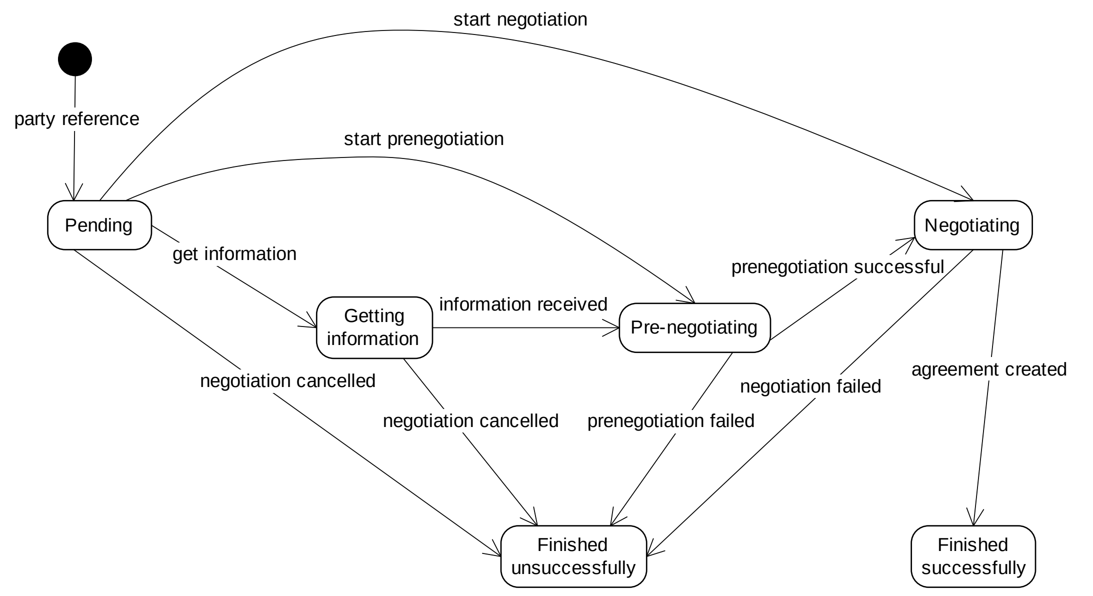


Figure 7.4: *State machine of a party context.*

Negotiating protocol. The goal of this state is to reach an agreement on the negotiation protocol that shall be used during the negotiation and its configuration. To this end, the PartyCoordinator starts interaction *RequestProtocolNegotiation* with the ProtocolNegotiator.

Negotiating. This is the state in which the actual negotiation takes place. When the party context enters this state, the PartyCoordinator delegates the negotiation to the NegotiationCoordinator by means of interaction *RequestNegotiation*. When the negotiation finishes, the NegotiationCoordinator notifies the result to the PartyCoordinator. Depending on the result, the party context enters state *finished successfully* or state *finished unsuccessfully*.

Finished successfully. When both parties reach an agreement, the PartyCoordinator sends it to the SystemCoordinator, who stores the agreement for future reference.

Finished unsuccessfully. This is the final state of all party contexts which fail to reach an agreement with the other party. In this state, the PartyCoordinator notifies the SystemCoordinator the result of the negotiation.

Decommitted. It may happen that a proposal that is more appealing than a previously created agreement is found. In this case, it may be possible to decommit from the previous agreement and, hence, the state of the party context changes from *finished successfully* to *decommitted*.

Role: NegotiationCoordinator

Goal: It coordinates the negotiation contexts by linking the ProtocolHandler with the decision making organisation.

Interactions: *RequestNegotiation*, *RequestCommitApproval*, *ProtocolConversion* and *RequestResponse*.

Environmental resources: It may be necessary to read *PreferencesResource* and *WorldModel* to coordinate the negotiations. In addition, it stores all messages exchanged in a negotiation context in the *NegotiationContextData*.

Behaviour: The NegotiationCoordinator manages negotiation contexts and stores their status in resource *NegotiationContextData*. Its behaviour is as follows: first, the NegotiationCoordinator receives a request to start a negotiation through interaction *RequestNegotiation*. Then, it either starts a new negotiation context for the request (this is the case of all bargaining negotiations) or assigns it to an existent negotiation context (for instance, if the NegotiationCoordinator acts as an auctioneer of an auction).

Figure §7.5 shows the states of a negotiation context. They are based on the distinction between binding and non-binding negotiation messages (*cf.* Section §6.2): the NegotiationCoordinator may send as many non-binding negotiation messages as it wants to; but, it needs the approval of the CommitHandler before sending a binding negotiation message. Therefore, the states of a negotiation context are as follows:

Negotiating. This is the state in which the NegotiationCoordinator coordinates the ProtocolHandler and the ResponseGenerator to exchange non-binding negotiation messages with the other negotiating parties and waits for the responses of the other parties. When the ResponseGenerator decides that a binding negotiation message should be sent, the negotiation context enters state *asked approval*.

Asked approval. In this state, the NegotiationCoordinator waits for the approval of the CommitHandler before sending a binding negotiation message. If the approval is granted, the negotiation context enters state *approved*. Otherwise, it moves back to state *negotiating*.

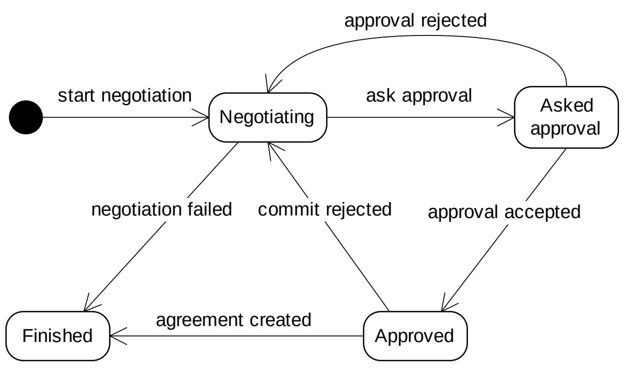


Figure 7.5: *State machine of a negotiation context.*

Approved. If the CommitHandler approves the sending of a binding negotiation message, the negotiation context enters this state and the binding negotiation message is sent. If this is the last message of the negotiation protocol (*e.g.* performative `accept`), it enters state *finished*. Otherwise, it waits in this state until the response from the other party arrives. If the other party rejects the binding negotiation message, the negotiation context moves back state *negotiating*. Else, it enters state *finished*.

Finished. This is the final state in which the negotiation protocol instance finishes.

The NegotiationCoordinator depends on the characteristics of the negotiation protocol family it supports. For instance, in a bargaining protocol it has to request a response for each negotiation message, whereas in an auction protocol, it can wait for the other parties' bids. Its behaviour also varies depending on whether it coordinates negotiation contexts independently, or it coordinates them depending on how the other negotiation contexts are performing.

7.2.2 Interactions

There are five interactions that are carried out in the coordination organisation of NegoFAST-Core. One of them (*UserInteraction*) is developed with the user of the negotiation system; other two (*IncomingProtocolNegotiation* and *IncomingNegotiation*) are incoming requests from other parties; and the last two (*RequestPartyProcessing* and *RequestNegotiation*) are the interactions between the SystemCoordinator and the PartyCoordinator, and the PartyCoordinator and the NegotiationCoordinator, respectively (*cf.* Figure §7.1).

Interaction: `UserInteraction`

Initiator: User.

Partners: SystemCoordinator.

Input: The user preferences and party references with whom to negotiate.

Output: Agreements reached with the parties according to the preferences supplied by the user.

Description: This interaction represents the interaction between the automated negotiation system and the user. It allows the User to provide the SystemCoordinator with the preferences and the parties with whom to negotiate. Furthermore, the User may also request the system to end the negotiation. The automated negotiation system returns the achieved agreements.

Interaction: `IncomingProtocolNegotiation`

Initiator: ProtocolNegotiator.

Partners: SystemCoordinator.

Input: A party reference that wishes to start a protocol negotiation process with the automated negotiation system.

Output: The acceptance or rejection to start the protocol negotiation.

Description: Through this interaction, the ProtocolNegotiator notifies that it has received a request to start a protocol negotiation and asks the SystemCoordinator for permission to proceed with it.

Interaction: `IncomingNegotiation`

Initiator: ProtocolHandler.

Partners: SystemCoordinator.

Input: A party reference that wishes to start a negotiation process with the automated negotiation system.

Output: The acceptance or rejection to start the negotiation.

Description: Through this interaction, the ProtocolHandler notifies that it has received a request to start a negotiation and asks the SystemCoordinator for permission to proceed with it.

Interaction: `RequestPartyProcessing`

Initiator: SystemCoordinator.

Partners: PartyCoordinator.

Input: A party reference.

Output: Agreements reached with the parties.

Description: By means of this interaction, the SystemCoordinator sends party references to the PartyCoordinator in order to be processed. The PartyCoordinator notifies the SystemCoordinator when an agreement has been reached. In addition, the SystemCoordinator may cancel the processing of all party references because some termination condition of the system context holds.

Interaction: `RequestNegotiation`

Initiator: PartyCoordinator.

Partners: NegotiationCoordinator.

Input: A reference to the ProtocolHandler that manages the negotiation protocol and, optionally, additional information about the other party.

Output: An agreement if the negotiation finishes successfully or nothing if it fails.

Description: By means of this interaction, the PartyCoordinator assigns a new negotiation to be carried out by the NegotiationCoordinator.

7.3 Protocol management

The aim of the protocol management organisation is to provide the elements that are necessary to deal with the selection and the execution of negotiation protocols instances and to make the other roles of the architecture independent from concrete negotiation protocols. Furthermore, it also may provide mechanisms to guarantee the reliability and non-repudiability of the agreements created. Note that the independence from concrete negotiation protocols is reached by means of the definition of abstract negotiation protocols. The protocol management organisation converts negotiation protocols into the corresponding abstract negotiation protocols that are understood by the other roles.

7.3.1 Roles

The organisation is composed of the following roles: ProtocolNegotiator, ProtocolHandler and Notary as depicted in Figure §7.1. The ProtocolNegotiator deals with the selection of the negotiation protocol; the ProtocolHandler, which is specific for each concrete negotiation protocol, interacts with the other parties following the negotiation protocol and transforms it into an abstract negotiation protocol; and the Notary guarantees that the agreement created is reliable and non-repudiable.

Role: ProtocolNegotiator

Goal: The goal of the ProtocolNegotiator is to select and configure, if necessary, in cooperation with the other parties, the protocol that will be used during the negotiation process.

Interactions: *RequestProtocolNegotiation*, *ProtocolNegotiation*, *IncomingProtocolNegotiation* and *ConfigureHandler*.

Environmental resources: The role may read the *PreferencesResource*, *WorldModel*, *SystemContextData* and *PartyContextData*.

Behaviour: The ProtocolNegotiator receives a reference to the parties as part of interaction *RequestProtocolNegotiation*. Then, through interaction *ProtocolNegotiation*, it contacts the parties to reach an agreement about the negotiation protocol that will be used and its configuration (e.g. timeouts or maximum number of interactions). To decide which protocol is the most appropriate, it may read information from the *PreferencesResource* and the *WorldModel*. After a negotiation protocol is selected, it configures a ProtocolHandler by means of interaction *ConfigureHandler* and returns a reference to the ProtocolHandler as the last part of interaction *RequestProtocolNegotiation*.

In addition, the ProtocolNegotiator receives requests to start new protocol negotiations from other parties and forwards them to the SystemCoordinator by means of interaction *IncomingProtocolNegotiation*.

Role: ProtocolHandler

Goal: The ProtocolHandler deals with the interaction with the other parties following a negotiation protocol.

Interactions: *ConfigureHandler*, *IncomingNegotiation*, *Negotiation*, and *ProtocolConversion*.

Environmental resources: None.

Behaviour: It is configured to manage a negotiation protocol instance by means of interaction *ConfigureHandler*. It adapts the negotiation protocol instance to the abstract negotiation protocol that is supported by the NegotiationCoordinator. This involves transforming the syntax of the negotiation protocol into negotiation messages that are understood by the other roles in NegoFAST and sending them by means of interaction *ProtocolConversion*; transforming negotiation messages into the concrete syntax of a negotiation protocol and sending them out to the other parties; enforcing the rules of the negotiation protocol; and coping with the errors that may appear during the interaction with the other parties such as messages missing, arriving too late or the arrival of unexpected messages.

In addition, the ProtocolHandler receives requests from other parties to start negotiations and forwards them to the SystemCoordinator by means of interaction *IncomingNegotiation*.

Role: Notary

Goal: The Notary must guarantee that the agreement created between the two parties is reliable and non-repudiable.

Interactions: *AgreementCreation*

Environmental resources: None.

Behaviour: The behaviour of the role depends on the protocol used to guarantee the reliability and non-repudiability of the process. However, it must include, at least, with cryptographic technology and algorithms to implement non-repudiation. Additionally, it can also maintain a repository of all created agreements that it has certified to resolve potential disputes.

7.3.2 Interactions

There are six types of interactions that are carried out by the roles of this organisation. Three of them: *ProtocolNegotiation*, *Negotiation* and *AgreementCreation* are interactions with external parties, whereas the other four are internal (*cf.* Figure §7.1).

Interaction: `RequestProtocolNegotiation`

Initiator: PartyCoordinator.

Partners: ProtocolNegotiator.

Input: A reference to the party or parties with whom negotiate the protocol.

Output: A reference to a ProtocolHandler role that will manage the selected negotiation protocol.

Description: This interaction allows the PartyCoordinator to request a protocol negotiation before starting the negotiation process with the other party.

Interaction: *`ProtocolNegotiation`*

Initiator: ProtocolNegotiator or other party.

Partners: Other party or ProtocolNegotiator.

Input: The proposed negotiation protocols that can be used during the negotiation.

Output: A negotiation protocol instance.

Description: This interaction represents the protocol negotiation carried out by parties before starting the negotiation.

Interaction: *`ConfigureHandler`*

Initiator: ProtocolNegotiator.

Partners: ProtocolHandler.

Input: A negotiation protocol instance.

Output: The reference to the ProtocolHandler that is configured to manage that protocol.

Description: This interactions configures the ProtocolHandler with the parameters of the negotiation protocol instance that have been previously agreed with the other parties.

Interaction: *`ProtocolConversion`*

Initiator: NegotiationCoordinator.

Partners: ProtocolHandler.

Input: A negotiation message with an initial proposal.

Output: An agreement if the negotiation finishes successfully or nothing if it fails.

Description: This interaction is an implementation of the abstract negotiation protocol and represents the exchange of messages carried out during the negotiation between the ProtocolHandler and the NegotiationCoordinator.

Interaction: `Negotiation`

- **Initiator:** ProtocolHandler.
- **Partners:** Other party.
- **Input:** A negotiation message with an initial proposal expressed in the syntax of a concrete negotiation protocol.
- **Output:** An agreement if the negotiation finishes successfully or nothing if it fails.
- **Description:** This interaction represents the negotiation protocol instance followed by the parties during the negotiation process. In Section §3.4, we present a comprehensive description of negotiation protocols that can be used to implement this interaction.

Interaction: `AgreementCreation`

- **Initiator:** ProtocolHandler or the other party.
- **Partners:** Other party or the ProtocolHandler.
- **Input:** The agreement established by both parties.
- **Output:** The assurance of the reliability and non-repudiability of the created agreement.
- **Description:** The aim of this interaction is to actually create and sign an agreement and to guarantee that the created agreement is reliable and non-repudiable. It assumes that the agreement has already been reached previously.

7.4 Decision making

The goal of the decision making organisation is to provide mechanisms to determine the behaviour of the automated negotiation system during a negotiation process. In NegoFAST-Core, we define the three different decisions that we describe in Section §4.2, namely: response decision, which involves determining which messages are sent to the other parties during the negotiation; commit decision, which includes deciding whether and when the system should send a binding negotiation message; and decommit decision, which involves deciding on the decommitment from already established agreements. This organisation also provides other elements that analyse the convenience of committing to or decommitting from an agreement.

7.4.1 Roles

The organisation is composed of three roles, namely: ResponseGenerator, CommitHandler and CommitAdvisor (*cf.* Figure §7.1). The ResponseGenerator develops the response decision by generating the message to send to the other parties; the CommitAdvisor analyses the convenience of committing to or decommitting from an agreement, and the CommitHandler deals with both commit and decommit decision because both decisions are extremely related: the automated negotiation system will only decommit from an agreement if it finds a more profitable agreement (*cf.* Section §4.2).

Role: ResponseGenerator

Goal: Its goal is to decide which responses should be sent as a response to a received negotiation message. This involves selecting the performative that is going to be used, creating proposals, bids, or arguments supporting proposals and generating any other information that can be sent to the other parties.

Interactions: *RequestResponse*.

Environmental resources: The role may read the *PreferencesResource*, the *WorldModel*, the *SystemContextData*, the *PartyContextData* and the *NegotiationContextData* to create the responses.

Behaviour: It is heavily influenced by the negotiation protocol that is being used because it must obey the rules it imposes. There are two approaches to response generation. The first one involves two phases: first, the performative has to be decided, and, then, the other parts of the negotiation message (*i.e.* proposal or additional arguments) are generated. The second approach can be divided into two phases as well: first, all possible responses (or a subset of possible responses) are generated and then, the best one is selected. The first approach is usually developed in auctions or non-argumentation bargaining protocols, whereas the latter is common in argumentation-based negotiations. Section §4.2 reports on algorithms that can be used to implement the ResponseGenerator.

Role: CommitHandler

Goal: Role CommitHandler implements the decision on whether to commit to a proposal or not and also determines when these decisions are made. In addition, it may also decide to decommit from an existing agreement if necessary.

Interactions: *RequestCommitApproval*, *RequestAdvise* and *Decommit*.

Environmental resources: It may read the *PreferencesResource*, the *WorldModel*, the *SystemContextData*, the *PartyContextData* and *NegotiationContextData* to get information in order to make the decision.

Behaviour: The CommitHandler receives proposals through interaction *RequestCommitApproval* and responds either approving or rejecting the commitment to them. To make this decision, it uses the *PreferencesResource* and the information in the *WorldModel* about the market and the other negotiating parties to decide whether to approve the proposal for commitment. It may also query one or more CommitAdvisors about the feasibility of committing to the given proposal. For instance, in the case of a service provider, the CommitHandler may inquire a CommitAdvisor (*e.g.* a capacity planner) about the provider's capability to provision the proposal.

Another aim of the CommitHandler is to determine when these approval decisions are made, *i.e.* when approvals and rejections are sent. There are several approaches to it:

- Making the decision as the proposals are received [97]. This option is easier to implement. Here, decisions are made very quickly. However, if we just wish to reach a limited number of agreements, we may miss some very good ones only because we previously accepted others.
- Making the decision at some certain points in time previously defined. These points may be dynamically selected, depending on changing conditions of the market such as the frequency of arrival of proposals, or statically determined based on temporal constraints imposed by the user in its preferences.

The last goal of the CommitHandler is to ask for the decommitment of existing agreements to commit to another one, provided it is worth. To this end, the CommitHandler may consider the *PreferencesResource*, the decommitment fee and the models of the market and the other parties. Section §4.2 outlines several algorithms that some authors have proposed to implement both commit and decommit decisions.

Role: CommitAdvisor

Goal: Role CommitAdvisor analyses the feasibility of accepting an agreement based on domain-specific knowledge (*e.g.* the provider's capacity to provision a proposal) and gives a recommendation.

Interactions: *RequestAdvise*.

Environmental resources: None.

Behaviour: The implementation of this role is domain-specific. For instance, in the case of a service provider, a CommitAdvisor can be a capacity planner that analyses the provider's capability to provision a certain agreement and recommends whether to commit to that agreement or not.

7.4.2 Interactions

There are four types of interactions amongst the roles of this organisation. Only *Decommit* is an interaction with external parties, whereas the other three are internal to the automated negotiation system.

Interaction: `RequestResponse`

Initiator: NegotiationCoordinator.

Partners: ResponseGenerator.

Input: A set of the negotiation performatives that can be used as a response and the current status of the negotiation context, including the last negotiation message received.

Output: A negotiation message to be sent as a response.

Description: Its goal is to obtain negotiation messages that the NegotiationCoordinator will send to the other parties.

Interaction: `RequestCommitApproval`

Initiator: NegotiationCoordinator.

Partners: CommitHandler.

Input: The proposal whose approval is asked for.

Output: The approval or rejection to commit to the proposal.

Description: The goal of this interaction is to get the approval, from the CommitHandler, to submit a binding proposal or to accept a binding proposal. This interaction can be very simple or more complex depending on the coupling between the NegotiationCoordinator and the CommitHandler and the way the CommitHandler determines when to make an approval decision. If they are decoupled and the CommitHandler makes decisions about the approval as it receives them, then the interaction may be just one single message to ask for approval and the response. However, it may be more complex if it uses information about the current status of the negotiations or it has to inform the NegotiationCoordinator when the following approval decision is going to take place.

Interaction: `RequestAdvise`

Initiator: CommitHandler.

Partners: CommitAdvisor.

Input: A proposal.

Output: A recommendation about the feasibility of approving a proposal.

Description: The aim of this interaction is to get specialised advise about the approval of a specific proposal.

Interaction: `Decommit`

Initiator: CommitHandler.

Partners: A party.

Input: An already established agreement.

Output: The decommitment from that agreeement

Description: The goal of this interaction is to decommit from an already established agreement. A decommitment may involve a penalisation such as the payment of a decommitment fee.

7.5 World modelling

The goal of this organisation is to obtain and manage knowledge about the world. This involves gathering information from sources such as past interactions, external information providers or the other parties and analysing that information to build models with algorithms such as those described in Section §4.3. Furthermore, the world modelling organisation must also provide the information that it is necessary to configure and start a negotiation process with other parties (*e.g.* the negotiation protocols they support or an initial template as a starting point for the negotiation).

7.5.1 Roles

As depicted in Figure §7.1, the organisation is composed of the following roles: Inquirer and Informant, which provide the information that is necessary to start the negotiation process, the ExternalInformationProviders, which provide specific information about the market or other parties, such as their reputation, and the WorldModeller, which gathers additional information from ExternalInformationProviders and previous interactions and analyses it to build models of the market and the other parties.

Role: Inquirer

Goal: The Inquirer is the role in charge of obtaining more information about the other parties by polling their Informants.

Interactions: *RequestInformation* and *QueryPartyInformation*.

Environmental resources: It may read the *PreferencesResource* to get guidelines about the information for which it must ask the other parties.

Behaviour: It receives a collection of party references by means of interaction *RequestInformation* and uses interaction *QueryPartyInformation* to get the information it considers relevant from them.

To decide which information is relevant, the Inquirer may use the *PreferencesResource*. In the process of polling the Informants, the role can select different strategies of querying, depending on the interaction standard and the type of information requested.

Role: Informant

Goal: It is responsible for publishing all public information that can be useful to other parties in order to evaluate the chances to make an agreement with it.

Interactions: *QueryPartyInformation*.

Environmental resources: It reads the *PreferencesResource*.

Behaviour: It responds the queries that have been requested using interaction *QueryPartyInformation*. Usually, the information source to create these responses is the *PreferencesResource*. However, all preferences are not intended to be public; therefore the Informant has to decide which subset of the preferences will be sent. This decision can be very simple (*e.g.* annotating when some preference is public) or more complex (*e.g.* the decision on what information is public is determined by previous experiences). The Informant may also restricts the information it sends back depending on the requester.

Role: WorldModeller

Goal: The goal is to build a model of the other parties together with a model of the market. They are based on information supplied by ExternalInformationProviders and previous negotiations.

Interactions: *QueryExternalInformation*.

Environmental resources: It may read the *NegotiationHistory*, the *SystemContextData*, the *PartyContextData*, the *NegotiationContextData* and the *WorldModel* to get information to create the models. It also writes to the *WorldModel* to store the created models.

Behaviour: As described in Section §4.3, the WorldModeller must gather information and analyse it to create models of the world. Information is gathered either by querying ExternalInformationProviders using interaction *QueryExternalInformation* to get information such as the reputation or the geographical location of a specific party, or by reading the last changes in the *NegotiationHistory*, the *SystemContextData*, the *PartyContextData* and the *NegotiationContextData*. Then, this information is analysed following some algorithm such as those described in Section §4.3.2 and the results are stored in the *WorldModel*.

Role: ExternalInformationProvider

Goal: It provides specific information about the market or other parties, such as their reputation.

Interactions: *QueryExternalInformation*.

Environmental resources: None.

Behaviour: This role is domain-specific and, hence, there is no common behaviour.

7.5.2 Interactions

There are three types of interactions in this organisation, namely: *QueryPartyInformation* and *QueryExternalInformation*, which interact with external parties, and *RequestInformation*, which is internal to the automated negotiation system.

Interaction: `RequestInformation`

Initiator: PartyCoordinator.

Partners: Inquirer.

Input: A query to get information about one party.

Output: The information requested.

Description: The aim of the interaction is to ask the Inquirer to query other parties for the information that is necessary to start a negotiation process with them.

Interaction: `QueryPartyInformation`

Initiator: Inquirer or party.

Partners: Party or Informant.

Input: A query to get information about one party.

Output: The public information about the party that was requested.

Description: The aim of the interaction is to query other parties to obtain information about them that is relevant for starting the negotiation process.

Interaction: `QueryExternalInformation`

Initiator: WorldModeller.

Partners: ExternalInformationProvider.

Input: A query for information about some party or the market.

Output: The information requested.

Description: Its goal is to query ExternalInformationProviders for information about other parties. The data used in this interaction is domain-specific.

7.6 Environmental resources

In NegoFAST-Core, we define several environmental resources that can be read, modified or both by the roles. These environmental resources are *AgreementsResource*, *PreferencesResource*, *SystemContextData*, *PartyContextData*, *NegotiationContextData*, *NegotiationHistory*, and *WorldModel*. Next, we detail them:

AgreementsResource It stores all agreements with other parties within the current system context. The goal of the *AgreementsResource* is to allow the comparison of agreements already reached with current negotiations and to allow the decommitment of one of them if necessary. The agreements stored can be in two states: committed and decommitted. When added, all agreements are in the committed state. If a decommitment from an agreement takes place, the agreement changes to the decommitted state. An agreement that is in the decommitted state cannot change back to a committed state.

Note that the *AgreementsResource* runs within a system context and, hence, when the system context finishes, the *AgreementsResource* is copied to the *NegotiationHistory* and deleted. Therefore, the responsibility of storing the agreements for future reference, *e.g.* to monitor them, must be provided by an external system.

PreferencesResource The preferences resource allows the roles in NegoFAST to have access to the user preferences and to evaluate and compare agreements and proposals.

As stated in Section §3.3, user preferences are the features (*i.e.* a set of statements that express capabilities and/or characteristics) and the requirements (*i.e.* a set of statements that specifies the needs of the user in terms of the features wished in other parties) about the agreement and the other party. Moreover, user preferences include negotiation guidelines that the automated negotiation system must follow during the negotiation (*e.g.* the negotiation deadline, the eagerness to reach a deal or specific criteria to be applied with new customers). Section §3.3 details several models to express user preferences.

Together with the user preferences, the *PreferencesResource* must provide a means to evaluate and compare agreements and proposals. This is made through the so-called assessment mechanisms. An assessment mechanism evaluates and compares agreements and proposals given a set of user preferences. Note that assessment mechanisms do not only depend on the user preferences model but on the model of agreements and proposals.

SystemContextData It stores information regarding the current system context so that all roles in NegoFAST-Core have access to the information they need about the context in which they are running. Typical information that may be stored here is: an identifier of the user of the system, the moment when the system context started and the party references that has been received and the result of their processing.

PartyContextData Its main goal is to store the state of each party context (*cf.* Figure §7.4) and the information generated along its lifecycle: the public information about the preferences of the other party (*i.e.* its features and requirements) gathered during the *getting information* state; the negotiation protocol selected together with its configuration if necessary, and the result of the *negotiating* state (*i.e.* either the agreement created or fail otherwise). It also stores the negotiation context that is negotiating this party context.

NegotiationContextData It stores information regarding each negotiation context. This information includes the current state of the negotiation context (*cf.* Figure §7.5) and the negotiation messages that have been exchanged with the other parties.

WorldModel It represents the knowledge the automated negotiation system has about the other parties, the market and the domain the negotiation is about (*cf.* Section §4.3). Therefore, the world model may include, for instance, knowledge about the preferences and negotiation style of a party and the market price for a given service. Unlike the previously described environmental resources, which run within a concrete system context, the *WorldModel* is outside it. The reason is that the models are valid across several system contexts (*e.g.* the negotiation style of a party does not depend on the system context). However the current status of the system context may have an influence on those models (*e.g.* the negotiation style of a party may be more precise if it takes the last negotiation messages exchanged into account). Therefore, the *WorldModel* may take the current system context into account to provide more accurate models.

NegotiationHistory It stores past negotiations. It is mainly intended for building models based on previous interactions. The *NegotiationHistory* can be seen as a list of all system contexts, party contexts and negotiation contexts that have been processed by the automated negotiation system.

7.7 Summary

In this chapter, we have presented NegoFAST-Core, a protocol-independent reference architecture for building automated negotiation systems. The next chapter presents an extension to NegoFAST-Core that supports bargaining protocols.

Chapter 8

The NegoFAST-Bargaining reference architecture

Every extension of knowledge arises from making the conscious the unconscious.

Friedrich Nietzsche, 1844–1900
German classical Scholar, Philosopher and Critic of culture

The goal of this chapter is to detail a protocol-specific extension to the NegoFAST-Core *reference architecture. In so doing, we provide mechanisms to tackle protocol-specific requirements with a greater detail. Specifically, we focus on bargaining protocols because we believe they are the most appropriate to accomplish the goals we set in Chapter §2. This chapter is organised as follows: in Section §8.1, we review the main structure of* NegoFAST-Bargaining*; in Sections §8.2, §8.3, §8.4 and §8.5, we extend the ProtocolHandler, NegotiationCoordinator, ResponseGenerator and* NegotiationContextData*, respectively, with elements that are specific to bargaining protocols; finally, we summarise the ideas of this chapter in Section §8.6.*

8.1 Introduction

The NegoFAST-Core reference architecture provides a protocol-independent description of the roles, interactions and resources that compose an automated negotiation system. In this chapter, we detail the NegoFAST-Bargaining reference architecture, depicted in Figure §8.1, which was briefly described in Section §6.3.2. The NegoFAST-Bargaining reference architecture extends NegoFAST-Core to deal with the specific requirements of concurrent proposal-based bilateral negotiations of multi-term agreements. This is a type of bargaining negotiation in which two parties exchange proposals sequentially in order to reach an agreement (*cf.* Section §3.4).

The requirements of concurrent proposal-based bilateral negotiations of multi-term agreements can be divided into: understanding several proposal-based bilateral negotiation protocols, generating responses composed of proposals for multi-term agreements and supporting concurrent bilateral negotiations. Next, we detail how NegoFAST-Bargaining tackles them.

The understanding of several proposal-based bilateral negotiation protocols is dealt with by specialising the ProtocolHandler in a BargainingProtocolHandler, which converts bilateral negotiation protocols in a previously defined abstract protocol that is understood by the NegotiationCoordinator.

The response generation of proposals for multi-term agreements is tackled by dividing the ResponseGenerator into the PerformativeSelector, which selects the performative that is going to be used, and the BuilderManager and ProposalBuilder, which deal with the creation of a proposal.

The support for concurrent bilateral negotiations is implemented by means of three extensions: the environmental resource *NegotiationContextData* is extended to store additional information about the negotiations; the concept of negotiation policy is introduced, and the coordination task is divided into the BilateralNegotiator and the BargainingCoordinator.

8.2 Protocol handler

The ProtocolHandler adapts a concrete negotiation protocol to an abstract negotiation protocol and, hence, it makes the architecture independent from the negotiation protocol selected. In NegoFAST-Bargaining, the abstract negotiation protocol is a proposal-based bilateral negotiation protocol. At this level of abstraction, it is not necessary to define all aspects of the abstract negotia-

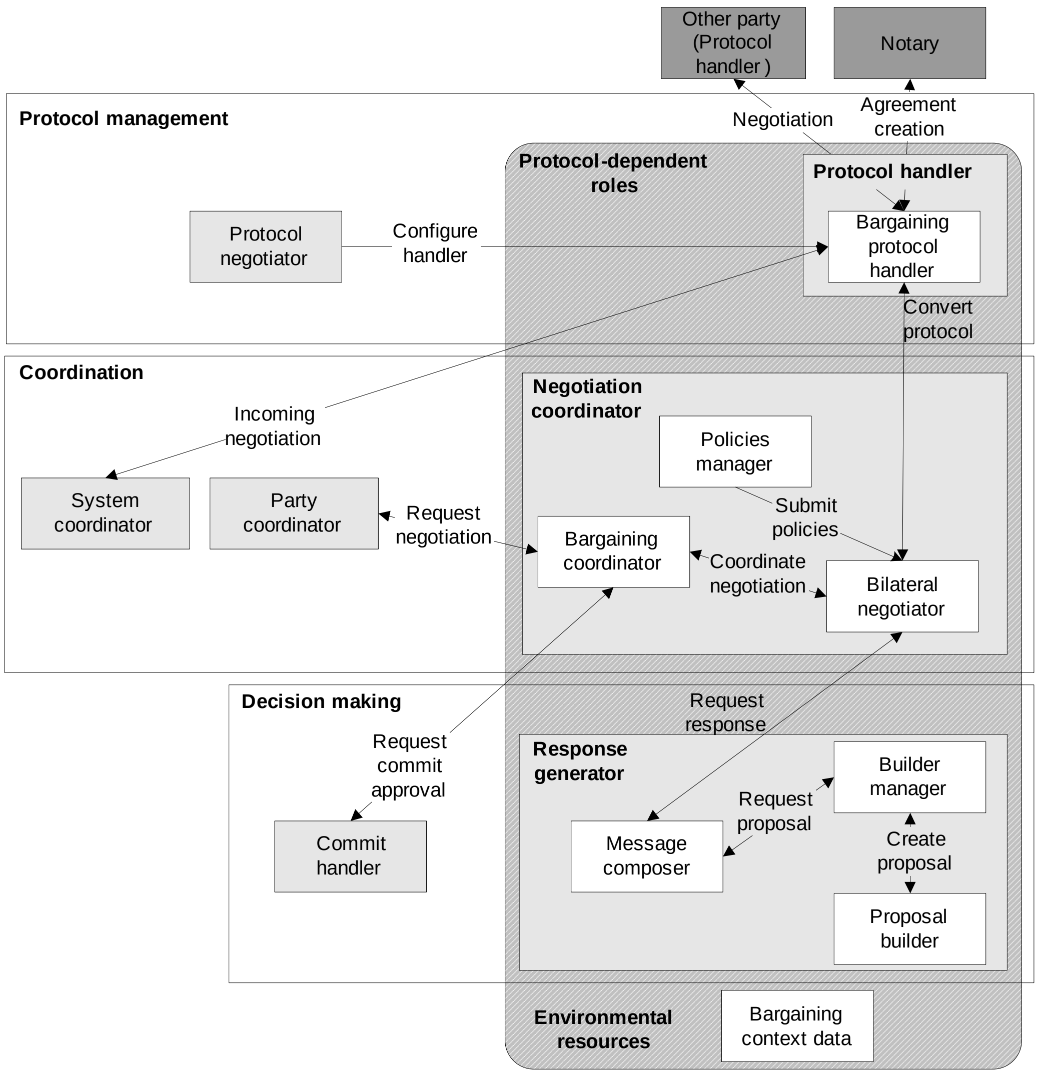


Figure 8.1: *The NegoFAST-Bargaining reference architecture.*

tion protocol (*cf.* Section §10.4). However, there are several implications that have an impact on the NegoFAST-Bargaining reference architecture:

Negotiations are bilateral, *i.e.* they are carried out between two parties only: the initiator, which is the party that initiates the negotiation, and the responder, which is the other party.

The negotiation is sequential, which means that the same party cannot send two proposals in a row. Except for some negotiation messages that include `reject-negotiation` or `withdraw` performatives that can be sent at any moment during the negotiation.

The protocol is proposal-based, *i.e.* negotiation messages contain either one performative or one performative and one or more proposals. This means that no additional arguments can be sent together with a proposal as in argumentation-based protocols (*cf.* Section §3.4).

8.2.1 Roles

In NegoFAST-Bargaining, the ProtocolHandler remains as one role: the BargainingProtocolHandler, which converts the negotiation protocol into an abstract negotiation protocol. The description of the role is basically the same but its behaviour is restricted with the aforementioned features of proposal-based bilateral negotiation protocols.

Role: BargainingProtocolHandler

Goal: The BargainingProtocolHandler deals with the interaction with the other parties following a concrete proposal-based bilateral negotiation protocol.

Interactions: *ConfigureHandler*, *IncomingNegotiation*, *Negotiation*, and *ProtocolConversion*.

Environmental resources: None.

Behaviour: It must be configured to manage a proposal-based bilateral negotiation protocol by means of interaction *ConfigureHandler*. It also transforms the syntax of a concrete negotiation protocol into negotiation messages that are understood by the other roles and sends them using interaction *ProtocolConversion*. It also implements the inverse process by transforming negotiation messages into the syntax of concrete negotiation protocols and sending them to the other parties. Moreover, it enforces the rules of the negotiation protocol and copes with the errors that may occur during the interaction with the other parties.

8.2.2 Interactions

The only interaction related to the ProtocolHandler that is extended by NegoFAST-Bargaining is the *ProtocolConversion*. This interaction is the implementation of an abstract proposal-based bilateral negotiation protocol.

Interaction: `ProtocolConversion`

> **Initiator:** BargainingCoordinator.
>
> **Partners:** BargainingProtocolHandler.
>
> **Input:** A negotiation message with an initial proposal.
>
> **Output:** An agreement if the negotiation finishes successfully or nothing if it fails.
>
> **Description:** This interaction represents the implementation of an abstract proposal-based bilateral negotiation protocol (*cf.* Section §10.4 for a materialisation of this interaction).

8.3 Negotiation coordinator

To give full support to concurrent bilateral negotiations, the NegotiationCoordinator must not just coordinate each negotiation context independently, but also must have a global vision of all negotiation contexts. In so doing, the NegotiationCoordinator must guide the behaviour of one negotiation context based on how well the other concurrent negotiations are performing. This is achieved by means of the so-called negotiation policies, which are guidelines about how to generate responses. For instance, if one negotiation is performing particularly well (*i.e.* the proposals from the other party are very appealing), the negotiation policies of the other negotiation contexts can be set to make the ResponseGenerator to concede less.

8.3.1 Roles

In NegoFAST-Bargaining, the NegotiationCoordinator is divided into several roles to support concurrent bargaining negotiations, namely: BilateralNegotiator, BargainingCoordinator and PoliciesManager.

The coordination task is divided into the BilateralNegotiator, which coordinates one negotiation context, and the BargainingCoordinator, which coordinates several BilateralNegotiators.

To add support for negotiation policies, NegoFAST-Bargaining extends interaction *RequestResponse* to enable the submission of negotiation policies to the ResponseGenerator, and introduces the PoliciesManager, which chooses negotiation policies for each negotiation to guide their behaviour.

Role: BargainingCoordinator

Goal: The BargainingCoordinator orchestrates the BilateralNegotiator, the CommitHandler and the PartyCoordinator. Furthermore, it also stores the current state of the concurrent negotiations in resource *BargainingContextData*.

Interactions: *RequestNegotiation*, *CoordinateNegotiation* and *RequestCommitApproval*.

Environmental resources: It stores the current state of concurrent negotiations in resource *BargainingContextData*.

Behaviour: Its acts as a message dispatcher amongst PartyCoordinator, BilateralNegotiator and CommitHandler. Specifically:

- When it receives a request to start a negotiation through interaction *RequestNegotiation*, it delegates it to the BilateralNegotiator by means of interaction *CoordinateNegotiation*.
- When it receives a commit approval request from a BilateralNegotiator via interaction *CoordinateNegotiation*, it updates the new state in the *BargainingContextData* uses and uses interaction *RequestCommitApproval* to forward the request to the CommitHandler.
- When it receives a response for a commit approval request from the CommitHandler, it updates the *BargainingContextData* and forwards the response to the BilateralNegotiator.
- When it receives the result of the negotiation from a BilateralNegotiator, it notifies it to the CommitHandler, the *BargainingContextData* and the PartyCoordinator.

Role: BilateralNegotiator

Goal: Its goal is to carry out a single bilateral negotiation by orchestrating the BargainingProtocolHandler and the PerformativeSelector. Furthermore, it communicates with the BargainingCoordinator to ask for approval before sending a binding negotiation message and it receives negotiation policies from the PoliciesManager.

Interactions: *CoordinateNegotiation*, *ProtocolConversion*, *RequestResponse*, *SubmitPolicies*

Environmental resources: It writes to the *BargainingContextData* the negotiation messages that have been received or sent.

Behaviour: This role implements, for each negotiation context, the state machine of a negotiation context for a bilateral protocol (*cf.* Figure §8.2). This state machine is an extension of the state machine

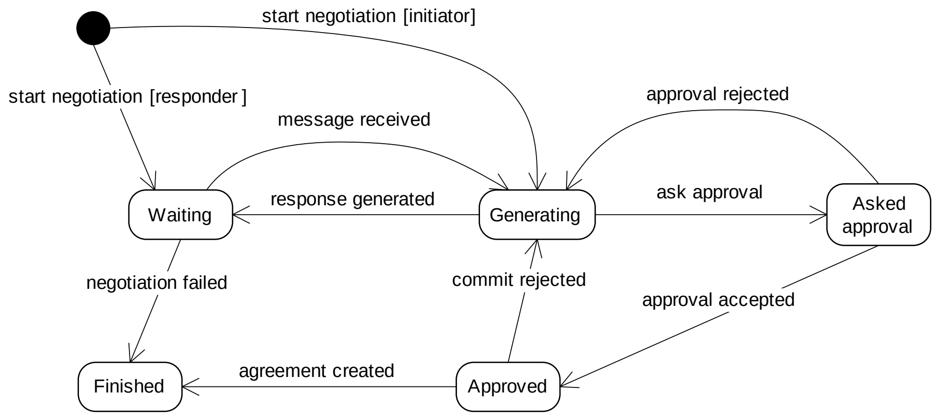


Figure 8.2: *State machine of a negotiation context for a bilateral protocol.*

of negotiation contexts described in the previous chapter (*cf.* Figure §7.5). The extension involves dividing state *negotiating* into state *waiting*, in which the negotiation context is waiting for the other party to send a negotiation message, and state *generating*, in which the automated negotiation system is building a negotiation message by means of the ResponseGenerator.

The first step is to contact the BargainingProtocolHandler to initialise it. The BargainingProtocolHandler acts as an intermediary between the BilateralNegotiator and the other party. Thus, all negotiation messages are sent and received by means of interaction *ProtocolConversion*, which makes the BilateralNegotiator independent from concrete negotiation protocols. The BargainingProtocolHandler informs the BilateralNegotiator of whether it is acting as the initiator or the responder of the interaction. In other words, if it must start the negotiation or it must wait for a negotiation message from the other party. If it is initiator, it enters state *generating*. Otherwise, it enters state *waiting*.

Waiting. This is the state in which the BilateralNegotiator is waiting for a negotiation message from the other party. When a new negotiation message is received by the BargainingProtocolHandler, it sends the message to the BilateralNegotiator via interaction *ProtocolConversion* and the state machine enters state *generating* to respond the received message.

Generating. In this state, the BilateralNegotiator interacts with the PerformativeSelector by means of interaction *RequestResponse* to obtain the negotiation message that shall be sent to the other party. If the negotiation message generated is a binding one, the BilateralNegotiator first forwards it to the BargainingCoordinator via interaction *CoordinateNegotiation* and enters state *asked approval*. Otherwise, it sends the negotiation message to the other party by means of the BargainingProtocolHandler and enters state *waiting*.

Asked approval. In this state, the BilateralNegotiator is waiting for the approval of the binding negotiation message. If the approval is granted, the negotiation context enters state *approved*. In other case, it moves back to state *generating*.

Approved. If the binding negotiation message is approved, the negotiation context enters this state and the binding negotiation message is sent. In this case, if this is the last message of the negotiation protocol (*e.g.* performative `accept`), it enters state *finished*. Otherwise, it waits in this state until the response of the other party. If the other party rejects the binding negotiation message, it moves back to state *generating*. Otherwise, it enters state *finished*.

Finished. This is the final state in which the negotiation protocol has ended.

In addition, during the whole negotiation, the BilateralNegotiator may receive negotiation policies from the PoliciesManager.

Role: PoliciesManager

Goal: The goal of the PoliciesManager is to provide the BilateralNegotiators with negotiation policies to coordinate their behaviour.

Interactions: *SubmitPolicies*

Environmental resources: It may be necessary to read the *PreferencesResource*, the *WorldModel*, the *NegotiationContextData* and the *BargainingContextData* to decide the negotiation policies.

Behaviour: The PoliciesManager determines the policies that are sent to the BilateralNegotiators based on the current status of all negotiations. It sends the negotiation policies to the BilateralNegotiator by means of interaction *SubmitPolicies*.

8.3.2 Interactions

There are two interactions between the roles described in the section: *CoordinateNegotiation* and *SubmitPolicies*.

Interaction: `CoordinateNegotiation`

Initiator: BargainingCoordinator.

Partners: BilateralNegotiator.

Input: The BargainingProtocolHandler that acts as an intermediary during the negotiation.

Output: A new agreement if the negotiation is successful or a fail otherwise.

Description: This interaction involves the request for the BilateralNegotiator to start a negotiation, the notification of the results of the negotiation, and the submissions of commit approval requests from the BilateralNegotiator to the BargainingCoordinator that are later redirected to the CommitHandler.

Interaction: `SubmitPolicies`

Initiator: PoliciesManager.

Partners: BilateralNegotiator.

Input: Negotiation policies.

Output: Nothing.

Description: This interaction implements the submission of the chosen negotiation policies to the BilateralNegotiator.

8.4 Response generator

The goal of the ResponseGenerator is to decide the negotiation messages that shall be sent to the other parties. In a proposal-based bilateral negotiation protocol this involves two tasks: selecting the performative to be used and creating the proposal to be sent if necessary (*cf.* Section §4.2).

8.4.1 Roles

In NegoFAST-Bargaining, the ResponseGenerator is divided into the following roles: the PerformativeSelector, which chooses the performative out of a set of allowed performatives, the BuilderManager, which selects the most appropriate ProposalBuilder and invokes it, and the ProposalBuilders, which are implementations of different algorithms to build proposals, such as those described in Section §4.2.

Role: PerformativeSelector

Goal: The goal of the PerformativeSelector is to generate a negotiation message by choosing the performative and requesting the accompanying proposal to the BuilderManager if necessary.

Interactions: *RequestResponse* and *RequestProposal*.

Environmental resources: The role may read the *PreferencesResource*, the *WorldModel*, the *SystemContextData*, the *PartyContextData* and the *BargainingContextData* to decide the performative.

Behaviour: The behaviour of the PerformativeSelector depends on whether the performative depends on the proposal obtained from the BuilderManager or not. If the performative is dependent, it first uses interaction *RequestProposal* to obtain a proposal from the BuilderManager and then decides the performative. If the performative is independent, the PerformativeSelector may decide the performative first and then, only if the performative involves the sending of a proposal, it requests a proposal to the BuilderManager by means of interaction *RequestProposal*. This avoids generating proposals if they are not necessary. In addition, the PerformativeSelector may also be able to cancel its operation and return a valid yet not optimal negotiation message.

Role: BuilderManager

Goal: The goals of the BuilderManager are to select a ProposalBuilder, to use it to obtain a proposal and to send this proposal back to the PerformativeSelector.

Interactions: *RequestProposal* and *CreateProposal*.

Environmental resources: This role may read the *PreferencesResource*, the *WorldModel*, the *SystemContextData*, the *PartyContextData* and the *BargainingContextData*.

Behaviour: First, it receives a request to build a proposal from the PerformativeSelector by means of interaction *RequestProposal*. Then, it selects one or several ProposalBuilders to obtain a proposal. This selection may be based on the negotiation policies, the preferences, the current state of the negotiations and world models. Next, it requests each ProposalBuilder to generate a proposal via interaction *RequestProposal*, chooses the most appropriate according to some criteria and sends it back to the PerformativeSelector. Note that the BuilderManager does not have to select several ProposalBuilders necessarily; in many cases, it can select just one and return the proposal that it generates. In addition, the BuilderManager may also be able to cancel its operation and return a valid yet not optimal negotiation message.

Role: ProposalBuilder

Goal: The goal of the ProposalBuilder is to create proposals to be sent to the other party. In Section §4.2, we report on several algorithms that have been proposed to generate such proposals.

Interactions: *CreateProposal*.

Environmental resources: The role may use the *PreferencesResource*, the *WorldModel*, the *SystemContextData*, the *PartyContextData* and the *BargainingContextData*.

Behaviour: When a proposal is requested, a proposal creation algorithm such as those described in Section §4.2 is executed and when it finishes, the proposal is returned. The ProposalBuilder may also support a cancellation of the proposal creation while the algorithm is running and returns a not-optimal-but-acceptable proposal. This is possible because some algorithms build proposals evolutionarily and, hence, they can be interrupted and return a proposal, which is not optimal, but valid.

8.4.2 Interactions

There are two interactions between the roles described in this section: *RequestResponse*, *RequestProposal*.

Interaction: `RequestResponse`

Initiator: BilateralNegotiator.

Partners: PerformativeSelector.

Input: A set of the negotiation performatives that can be used as a response and the current status of the negotiation, including the last negotiation message received and the negotiation policies.

Output: A negotiation message.

Description: The goal is to obtain negotiation messages that shall be used by the BilateralNegotiator to send them to the other parties.

Interaction: `RequestProposal`

Initiator: PerformativeSelector.

Partners: BuilderManager.

Input: The selected negotiation performatives if any, the current status of the negotiation, including the last negotiation message received and the negotiation policies.

Output: A proposal.

Description: The goal of this interaction is to obtain a proposal as part of the negotiation message that shall be sent as responses to the other parties by the BilateralNegotiator.

Interaction: `CreateProposal`

Initiator: BuilderManager.

Partners: ProposalBuilder.

Input: The selected negotiation performatives if any, the current status of the negotiation, including the last negotiation message received and the negotiation policies, and in some cases, specific configuration parameters for the proposal builder.

Output: A proposal.

Description: The goal of this interaction is to obtain a proposal as part of the negotiation message that shall be sent as a response to the other parties by the BilateralNegotiator.

8.5 Environmental resources

In NegoFAST-Core, we extend the *NegotiationContextData* with the so-called *BargainingContextData* to store the state of all concurrent negotiations so that it can be read by the roles of the reference architecture.

BargainingContextData It is a bargaining-specific extension of the resource *NegotiationContextData* of NegoFAST-Core, whose goal is to provide information about the current status of all concurrent negotiations.

The *NegotiationContextData* stores information regarding each negotiation context, such as the current state of the negotiation context and the negotiation messages that have been exchanged with the other parties (*cf.* Section §7.6). The *BargainingContextData* extends this information to support the bargaining-specific states of the negotiation context (*cf.* Figure §8.2) and to provide additional information such as the best utility value of all current negotiation parties.

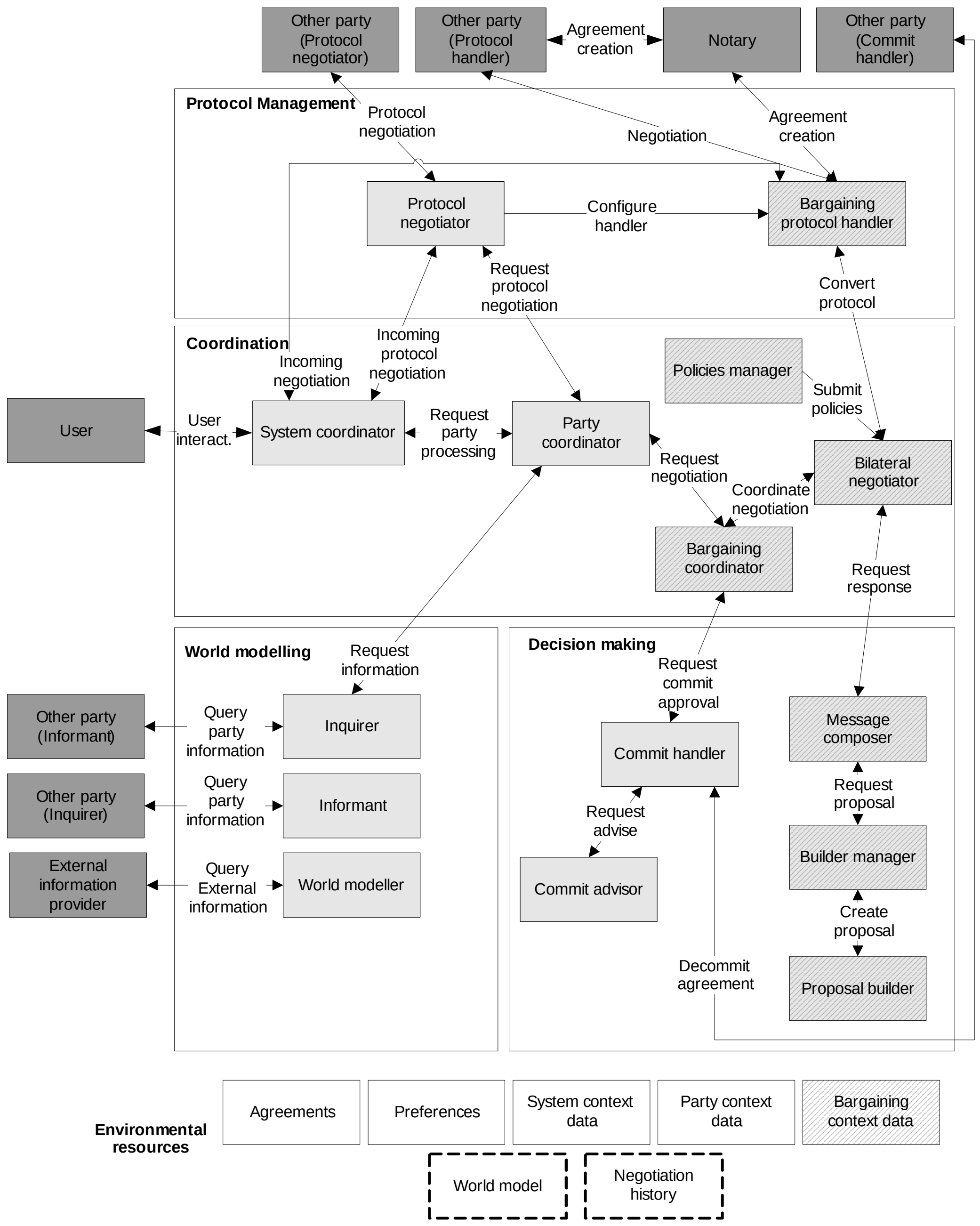


Figure 8.3: *NegoFAST-Core extended by NegoFAST-Bargaining.*

8.6 Summary

In this chapter, we have presented the NegoFAST-Bargaining reference architecture, which extends the NegoFAST-Core reference architecture with elements that are specific to bargaining protocols. Figure §8.3 depicts the whole NegoFAST reference architecture, including both NegoFAST-Core and NegoFAST-Bargaining.

Chapter 9

The NegoFAST-Core framework

If you have a knowledge framework,
let others light their candles at it.

Margaret Fuller, 1810–1850
American transcendentalist author and editor

In this chapter, we detail the NegoFAST-Core framework that we introduced in Section §6.4.1. This framework materialises the NegoFAST-Core reference architecture by providing a data model, the interfaces to environmental resources, the roles, and the state machines of the coordination roles. This chapter is organised as follows: in Section §9.1, we introduce the main ideas and overview the framework; Section §9.2 details the data model of the NegoFAST-Core framework; Section §9.3 defines the interfaces of the environmental resources of NegoFAST-Core; Section §9.4 details the interfaces of the interactions that are internal to the automated negotiation system; Section §9.5 describes the state machines of the *SystemCoordinator* and the *PartyCoordinator*; and Section §9.6 summarises the ideas of this chapter.

9.1 Introduction

The goal of this chapter is to detail a software framework that materialises the NegoFAST-Core reference architecture (*cf.* Chapter §7). The NegoFAST-Core framework provides a protocol-independent foundation for software engineers to develop automated negotiation systems. This NegoFAST-Core framework must be extended with protocol-specific frameworks that deal with concrete details of negotiation protocols, such as the NegoFAST-Bargaining framework described in the next chapter.

The NegoFAST-Core framework can be divided into the following packages (*cf.* Figure §9.1): package `model`, which defines a data model for the software framework; package `environment`, which defines the interfaces of the environmental resources; package `interactions`, which defines the interfaces of the interactions that are internal to the automated negotiation system, and package `roles`, which defines the state machines of the coordination roles. Next, we detail each of them.

9.2 Data model

The data model defines the concepts that are used in automated negotiation systems such as agreements, preferences and negotiation protocol (*cf.* conceptual map in Section §6.2). These concepts are used by the other elements of the NegoFAST-Core framework.

The key aspect is that there are many different ways of expressing agreements and preferences and there is no best way to represent them (*cf.* Sections §3.2 and §3.3). Therefore, the data model of the NegoFAST-Core framework must support different ways of expressing agreements and preferences. This is also the case of the proposals exchanged by the parties, which are heavily influenced by the agreement model. This flexibility allows the automated negotiation system to use simpler and more efficient algorithms when the user does not require a complex model to express preferences and agreements. This flexibility is achieved by specifying a generic model that defines the more general concepts such as agreements, terms, proposals, requirements and features. These concepts must be further refined to support concrete models for expressing agreements and preferences like those described in Sections §3.2 and §3.3.

The refinement of concepts can be implemented in two complementary ways. On the one hand, the concepts of the generic model are parameterised.

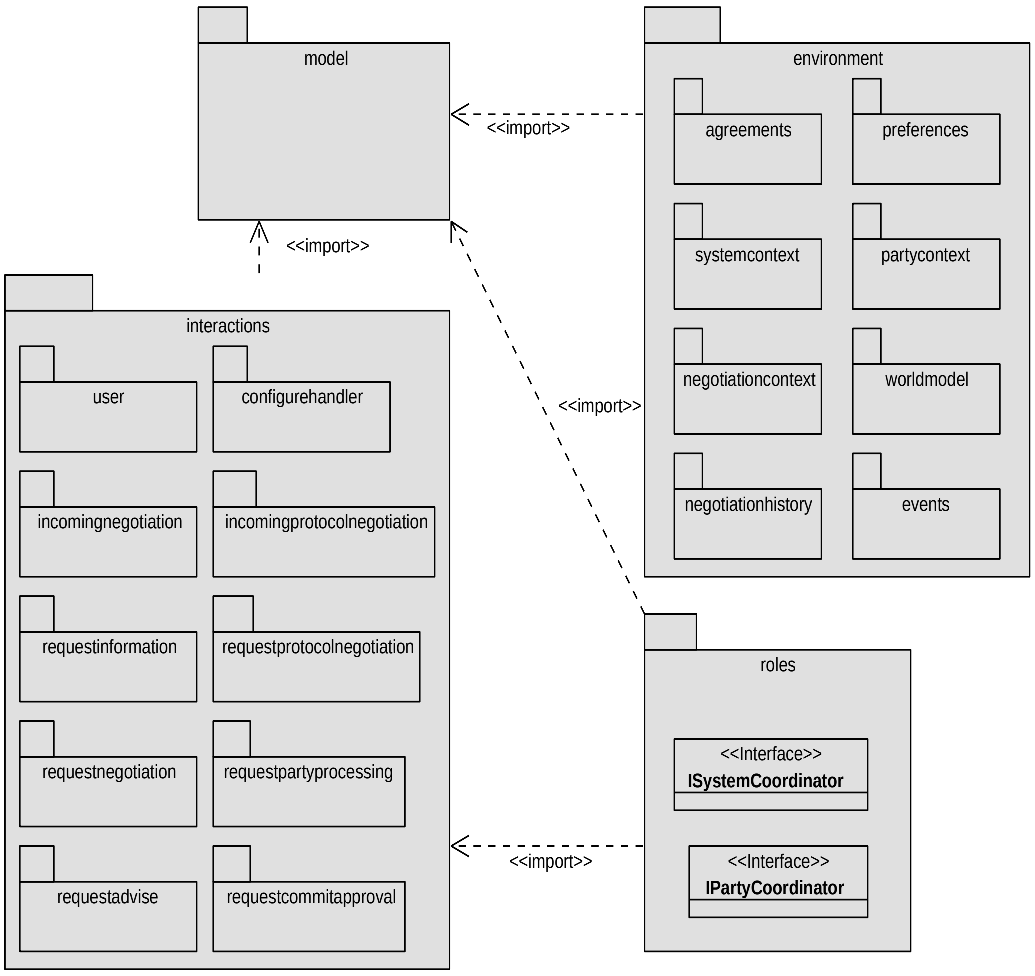


Figure 9.1: *NegoFAST-Core framework packages.*

For instance, the terms of the agreements are parameterised by the type of constraint. On the other hand, the generic model itself may also be extended to support advanced features. For instance, terms can be extended to add compensation clauses or proposals can be extended to include additional negotiation data about the terms specified in the proposal. As a proof-of-concept, in Appendix §A, we detail a concrete model to express preferences and agreements based on utility functions and assignment of values, respectively.

Preferences The preferences document is modelled by means of interface `IPreferencesDocument`. To allow different ways of modelling preferences, interface `IPreferencesDocument` (*cf.* Figure §9.2) is parameterised by the type

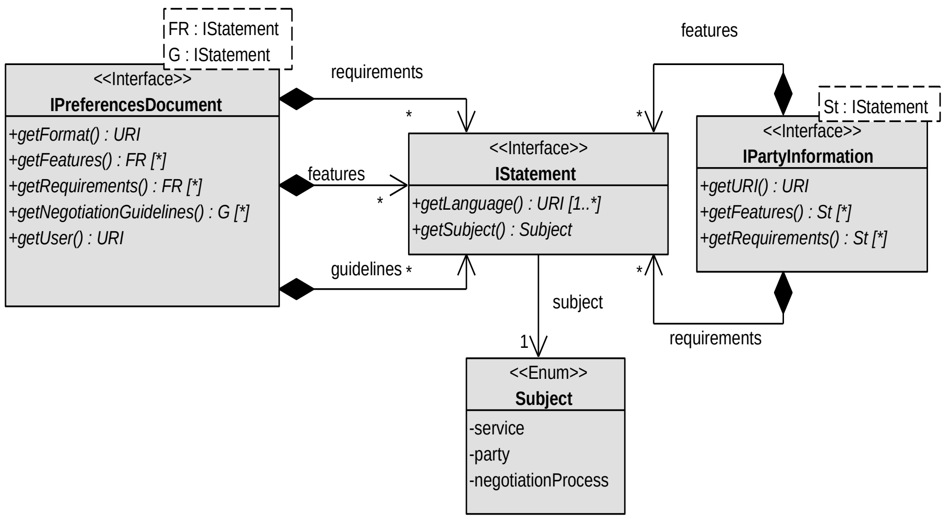


Figure 9.2: *Data model of preferences.*

of statements used to specify features and requirements and the type of statements used to specify the negotiation guidelines. For instance, in Appendix §A, features and requirements are expressed using utility functions whereas the negotiation guidelines are expressed using attribute-value pairs.

Interface `IPreferencesDocument<FR: IStatement,G: IStatement>` is as follows:

- ▷ `URI getFormat()`: Returns a URI[†1] that identifies the model used to express preferences.
- ▷ `FR[*] getFeatures()`: Returns a set of statements that express the features of a party. For instance, in the case of a service provider, they express the characteristics of the service provided.
- ▷ `FR[*] getRequirements()`: Returns a set of statements that specify requirements on other parties. For instance, the desired characteristics of the service or good consumed or bought.
- ▷ `G[*] getNegotiationGuidelines()`: Returns a set of statements that specify the negotiation guidelines that the automated negotiation sys-

[†1] A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource [13]. In NegoFAST, we use URIs to identify diverse elements of the architecture that ranges from software artifacts that implements an interface, to a concrete negotiation protocol or an external party

tem must follow. For instance, the negotiation deadline, the eagerness to reach a deal or specific criteria for new customers.

▷ `URI getUser()`: Returns a URI identifying the user who sends the preferences.

Statement A statement is a preference about one or several attributes that are defined in one or several ontologies or vocabularies. Some examples of statements are an utility function defined over an attribute, a constraint specified on several attributes, a pair name-value or a rule relating several attributes. Statements can be applied to the following subjects as defined by enumeration `Subject` depicted in Figure §9.2:

▷ `service`: If the statement is applied to the good or service offered or demanded. For instance, if it specifies the *service interface* or the *service cost*.

▷ `party`: If the statement expresses characteristics of one party, not about the good or service. Examples of this can be: *Party Z is located in Spain* or *Party X has a low reputation on service Y*.

▷ `negotiationProcess`: If the statement is applied to the negotiation process. For instance, if the statement specifies a negotiation deadline.

The interface of statements (`IStatement`) is very generic and must be extended by concrete type of statements (*cf.* Appendix §A for an example). The interface is depicted in Figure §9.2 and has the following method:

▷ `URI[1..*] getLanguage()`: Returns identifiers to the ontologies or vocabularies that are used by attributes used in the statement.

Party information Interface `IPartyInformation` models the public information offered by the parties about their preferences and is obtained by means of role Inquirer. Like preferences, the party information is composed of two different sets of statements: requirements and features and, also like preferences, it is parameterised by the type of statement used to express them. Therefore, the interface `IPartyInformation<S: IStatement>` is as follows:

▷ `URI getURI()`: Returns the identifier of the party.

▷ `S[*] getFeatures()`: Returns the statements that express the features of the party.

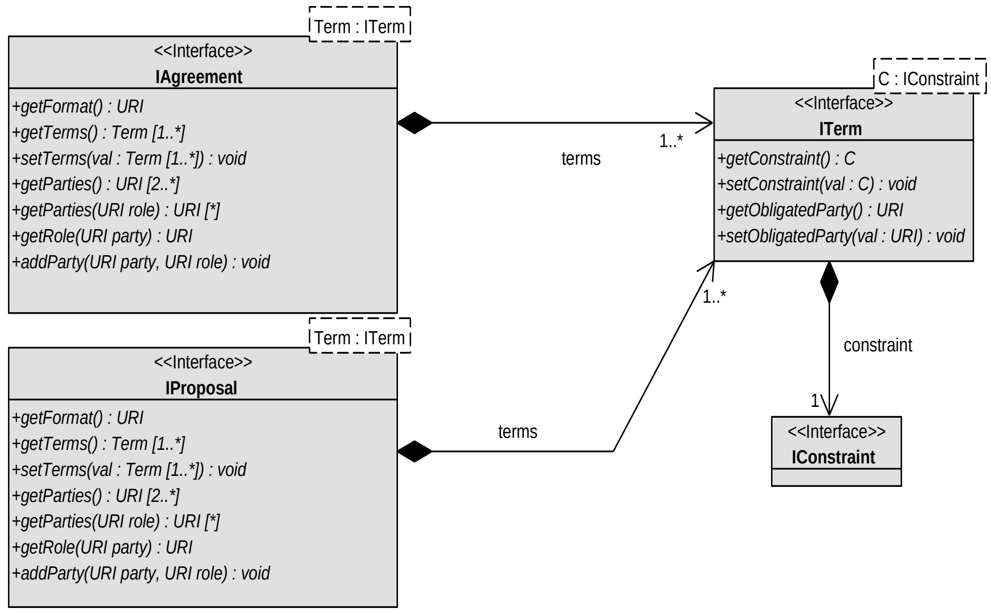

Term : ITerm
<<Interface>>
IAgreement
+getFormat() : URI
+getTerms() : Term [1..*]
+setTerms(val : Term [1..*]) : void
+getParties() : URI [2..*]
+getParties(URI role) : URI [*]
+getRole(URI party) : URI
+addParty(URI party, URI role) : void
terms
1..*
C : IConstraint
<<Interface>>
ITerm
+getConstraint() : C
+setConstraint(val : C) : void
+getObligatedParty() : URI
+setObligatedParty(val : URI) : void
1..*
constraint
1
<<Interface>>
IConstraint
Term : ITerm
<<Interface>>
IProposal
+getFormat() : URI
+getTerms() : Term [1..*]
+setTerms(val : Term [1..*]) : void
+getParties() : URI [2..*]
+getParties(URI role) : URI [*]
+getRole(URI party) : URI
+addParty(URI party, URI role) : void
terms

Figure 9.3: *Data model of agreements and proposals.*

▷ `S[*] getRequirements()`: Returns the statements expressing the requirements of the party.

Agreement Agreements are modelled by interface `IAgreement` and are composed of a set of terms. The different models of agreements are characterised by different types of terms. Therefore, interface `IAgreement` is parameterised by the type of terms used in it (*e.g.* terms as pairs attribute-value or terms as constraints).

The interface `IAgreement<Term:ITerm>` is as follows (*cf.* Figure §9.3):

▷ `URI getFormat()`: Returns a URI that identifies the model used to express the agreement.

▷ `Term[1..*] getTerms()`: Returns a collection of *terms* that must describe both functional descriptions and non-functional guarantees of the service. The terms used in an agreement must be fully specified and ambiguities must be avoided.

▷ `void setTerms(terms)`: Sets the terms that compose the agreement.

▷ `URI[2..*] getParties()`: Returns the URIs of the parties involved in the agreement.

▷ `URI[*] getParties(role)`: Returns the URIs of the parties involved in the agreement with the given role.

▷ `URI getRole(party)`: Returns the role that a given party plays in the agreement. Typically these roles are provider and consumer.

▷ `void addParty(party, role)`: Adds a new party to the agreement with the given role.

Proposal Interface `IProposal`, depicted in Figure §9.3, models proposals. Like agreements, proposals are parameterised by the type of terms it contains and its interface is exactly the same. Their main difference is that proposals may allow some terms to be left open to be refined in later interactions amongst the parties. Furthermore, proposals can include additional information, *e.g.*, data about some terms in the proposal.

Term Terms specify constraints over some attributes with which a party must comply. Therefore, they are parameterised by the type of constraint they enclose (*e.g.* equality, constraints over one attribute, constraints over several attributes). Interface `ITerm<C: IConstraint>` (*cf.* Figure §9.3) defines the generic interface of a term:

▷ `URI getObligatedParty()`: Returns the identifier of the party to whom the term is applied to. Each term is to be applied to one of the parties involved in the agreement or proposal and the party is obligated to fulfil what it is specified in it. Obviously, the party must be one of those that have been designated in the agreement or proposal as one of the parties that are involved in it.

▷ `void setObligatedParty(party)`: Sets the party that must comply with the constraint specified in the term.

▷ `C getConstraint()`: Returns the constraint specified over one or several attributes of a certain language.

▷ `void setConstraint(constraint)`: Sets the constraint to be assigned to the term.

Note that this is the generic interface of a term but it may be extended. For instance, complex terms may include a set of compensations that will be applied if the party does not comply with the constraints specified in the term.

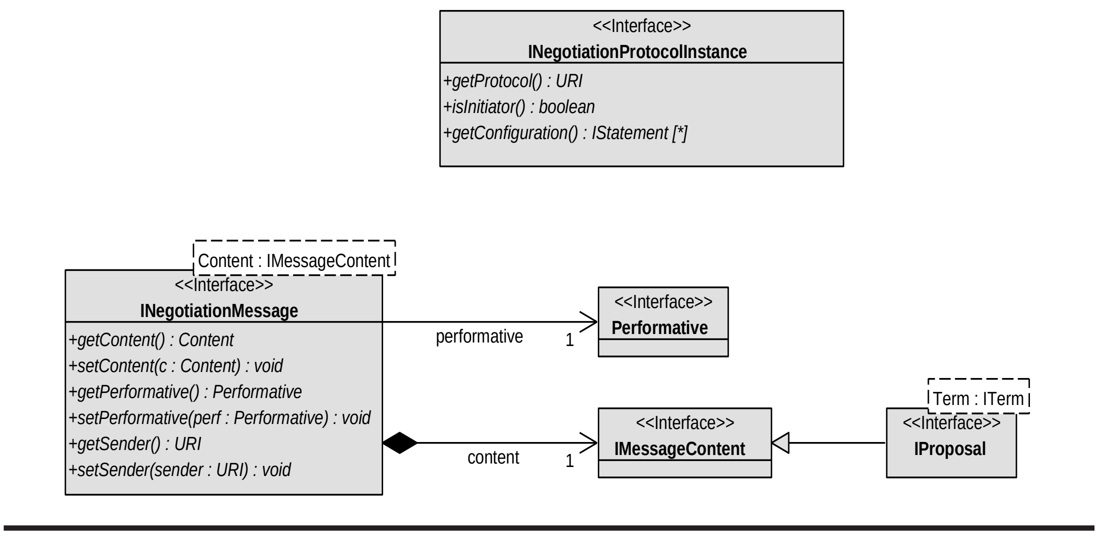


Figure 9.4: *Data model of negotiation messages.*

Negotiation protocol instance This element models the negotiation protocol instance that is performed by a negotiation context. Figure §9.4 depicts its interface (`INegotiationProtocolInstance`), and its defined as follows:

▷ `URI getProtocol()`: Returns an identifier of the negotiation protocol. This identifier may have no semantics associated to it or it may be a concept in an ontology like in References [105, 106].

▷ `boolean isInitiator()`: Returns whether the automated negotiation system is the initiator of the negotiation protocol or not, *i.e.* who sends the first message .

▷ `IStatement[*] getConfiguration()`: Returns the configuration of the negotiation protocol, which may specify some specific characteristics of the negotiation protocol such as a timeout.

Negotiation message Negotiation messages are composed of a performative that expresses the intention of the sender about the message and the content of the message. This content is usually a proposal. However, as we described in Section §3.4, other kinds of information can be exchanged such as threats, rewards or arguments. Therefore, the negotiation message is parameterised by the content of the message, which may be a proposal or other information. The interface of a negotiation message is `INegotiationMessage<Content: IMessageContent>` and has the following methods:

▷ `Content getContent()`: Returns the content of the negotiation message.

▷ `void setContent(content)`: Sets the content of the negotiation message.

▷ `Performative getPerformative()`: Returns the performative of the negotiation message.

▷ `void setPerformative(performative)`: Sets the performative of the negotiation message.

▷ `URI getSender()`: Returns the identifier of the party who sends the negotiation message.

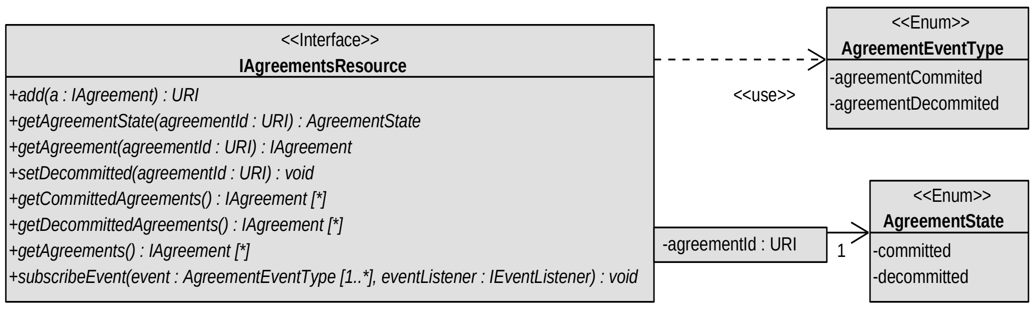
▷ `void setSender(sender)`: Sets the identifier of the party that sends the negotiation message.

Performative Performatives are modelled by tagging interface `Performative`, depicted in Figure §9.4, that must be refined by protocol-specific extensions, *cf.* Section §10.2 for a bargaining extension.

Message content The content of the message is modelled by interface `IMessageContent`, which is a tagging interface that indicates which elements may be part of a negotiation message. Interface `IProposal` is a type of `IMessageContent` (*cf.* Figure §9.4).

9.3 Environment interfaces

In the NegoFAST-Core reference architecture, we define several environmental resources that can be read, modified or both by the roles. The NegoFAST-Core framework materialises them by defining the interfaces that the software artifacts that implement them must expose. In addition, some environmental resources provide a publish/subscription mechanism [42] to avoid a continuous polling of the elements of the system to detect when some event occurs. Each environmental resources that provides a subscription mechanism must implement a method to subscribe to one or several events. Then, when one of these events takes place, the subscribers receive the notification by means of interface `IEventListener`, which is as follows:



<<Interface>>
IAgreementsResource
+add(a : IAgreement) : URI
+getAgreementState(agreementId : URI) : AgreementState
+getAgreement(agreementId : URI) : IAgreement
+setDecommitted(agreementId : URI) : void
+getCommittedAgreements() : IAgreement [*]
+getDecommittedAgreements() : IAgreement [*]
+getAgreements() : IAgreement [*]
+subscribeEvent(event : AgreementEventType [1..*], eventListener : IEventListener) : void
<<use>>
<<Enum>>
AgreementEventType
-agreementCommited
-agreementDecommited
-agreementId : URI
1
<<Enum>>
AgreementState
-committed
-decommitted

Figure 9.5: *Interface of* AgreementsResource.

▷ `void notify(source, event, eventInformation)`: Notifies that a subscribed event has occurred and includes the source of the event. Optionally, it may include additional domain-specific information related with the event.

`EventType` is a tagging interface that represents the supertype of all possible types of events.

AgreementsResource Its interface (`IAgreementsResource`) is simple, as depicted in Figure §9.5, and it includes the following methods:

▷ `URI add(agreement)`: Adds a new agreement and returns a URI that identifies it.

▷ `AgreementState getAgreementState(agreementId)`: Queries the state of an agreement given its URI (committed or decommitted).

▷ `IAgreement getAgreement(agreementId)`: Obtains a concrete agreement by its URI.

▷ `void setDecommited(agreementId)`: Marks an agreement as decommitted, hence changing its state.

▷ `IAgreement[*] getCommitedAgreements()`: Returns committed agreements.

▷ `IAgreement[*] getDecommitedAgreements()`: Returns all decommitted agreements.

▷ `IAgreement[*] getAgreements()`: Returns all agreements.

▷ `void subscribeEvent(event, eventListener)`: Subscribes to one or several events. When the event occurs, the event listener is notified.

Interface `IAgreementsResource` provides a subscription mechanism for events of type `AgreementEventType`, which are as follows:

▷ `agreementCommitted`: A new agreement has been created. The source of the notification is the URI of the agreement.

▷ `agreementDecommitted`: An agreement has been decommitted. The source of the notification is the URI of the agreement.

In addition, interface `IAgreementsResource` requires the `AgreementState` enumeration, which defines the states in which an agreement in the repository can be, namely:

▷ `committed`: The agreement was successful, and it is valid currently.

▷ `decommitted`: The agreement was successful, but it was decommitted and, hence, it is not valid currently.

PreferencesResource The *PreferencesResource* (*cf.* Figure §9.6) allow the elements in the negotiation system to have access to the user preferences. It provides access to the preferences supplied by the user (*cf.* `IPreferencesDocument` in the previous section) and methods to evaluate and compare several proposals. In addition, it also may provide mechanisms to convert the preferences supplied by the user to a different preferences model. Its interface is `IPreferencesResource`, and it is composed of the following methods:

▷ `URI getUser()`: Obtains the URI that identifies the user who sets the preferences.

▷ `IPreferencesDocument getPreferences()`: Returns the preferences as they were supplied by the user.

▷ `IPreferencesDocument getPreferences(model)`: Returns the preferences converted to the preferences model specified by `model` if possible. Note that not all conversions between preferences models are possible.

▷ `double evaluate(proposal)`: Applies an assessment mechanism to evaluate the given proposal and returns a normalised value.

▷ `boolean satisfies(proposal)`: Applies an assessment mechanism to analyse whether the proposal satisfies the minimum requirements es-

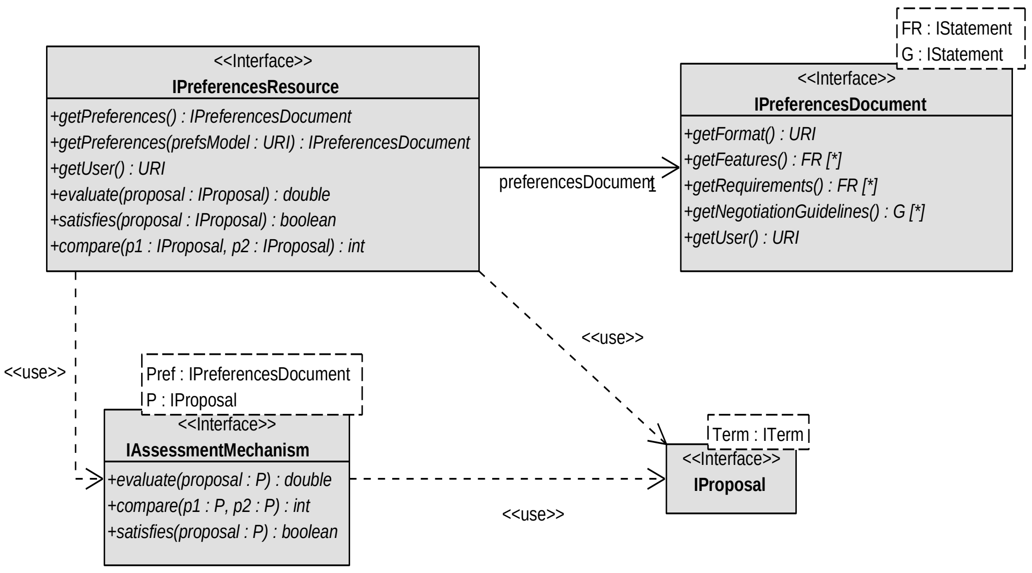

FR : IStatement
G : IStatement
<<Interface>>
IPreferencesResource
+getPreferences() : IPreferencesDocument
+getPreferences(prefsModel : URI) : IPreferencesDocument
+getUser() : URI
+evaluate(proposal : IProposal) : double
+satisfies(proposal : IProposal) : boolean
+compare(p1 : IProposal, p2 : IProposal) : int
preferencesDocument
1
<<Interface>>
IPreferencesDocument
+getFormat() : URI
+getFeatures() : FR [*]
+getRequirements() : FR [*]
+getNegotiationGuidelines() : G [*]
+getUser() : URI
<<use>>
<<use>>
Pref : IPreferencesDocument
P : IProposal
<<Interface>>
IAssessmentMechanism
+evaluate(proposal : P) : double
+compare(p1 : P, p2 : P) : int
+satisfies(proposal : P) : boolean
<<use>>
Term : ITerm
<<Interface>>
IProposal

Figure 9.6: *Interface of* PreferencesResource*.*

tablished by the user preferences. Note that this minimum is necessary but not sufficient for a proposal to be accepted.

▷ `int compare(p1, p2)`: Applies an assessment mechanism to compare two proposals. If `p1` is more appealing than `p2`, it returns a value greater than 0; if `p2` is more appealing than `p1`, it returns a value lower than 0, and if they are equally appealing, it returns 0.

To evaluate proposals, the *PreferencesResource* uses an assessment mechanism. An assessment mechanism provides a way to evaluate a proposal or an agreement and, hence, to compare several proposals to find the most appealing. Assessment mechanisms are parameterised by the model used to express the preferences (*i.e.*`IPreferencesDocument`) and the model used to express the terms of a proposal (`IProposal`). The interface (`IAssessmentMechanism<Pref: IPreferencesDocument, P: IProposal>`) is as follows:

▷ `double evaluate(proposal)`: Evaluates the given proposal.

▷ `int compare(p1, p2)`: Compares both proposals.

▷ `boolean satisfies(proposal)`: Analyses whether the proposal satisfies the minimum established by the user preferences.

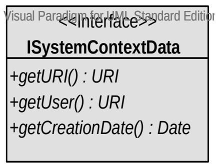

Visual Paradigm for UML Standard Edition
<<Interface>>
ISystemContextData
+getURI() : URI
+getUser() : URI
+getCreationDate() : Date

Figure 9.7: *Interface of* SystemContextData.

SystemContextData It provides access to the information stored related to the system context. Its interface, `ISystemContextData`, as depicted in Figure §9.7, includes the following methods:

▷ `URI getURI()`: Returns the URI that identifies the current system context.

▷ `URI getUser()`: Obtains the URI that identifies the user who set the preferences.

▷ `Date getCreationDate()`: Returns the moment when the system context was created.

PartyContextData Environmental resource *PartyContextData* stores information related to the party contexts. This includes the current state and the information generated while processing party references. As depicted in Figure §9.8, a party context, modelled by means of interface `IPartyContext` has one associated party and one `PartyState` and may have information about the party (`IPartyInformation`), the negotiation protocol selected for the negotiation (`INegotiationProtocolInstance`) and one negotiation context in which the negotiation takes place:

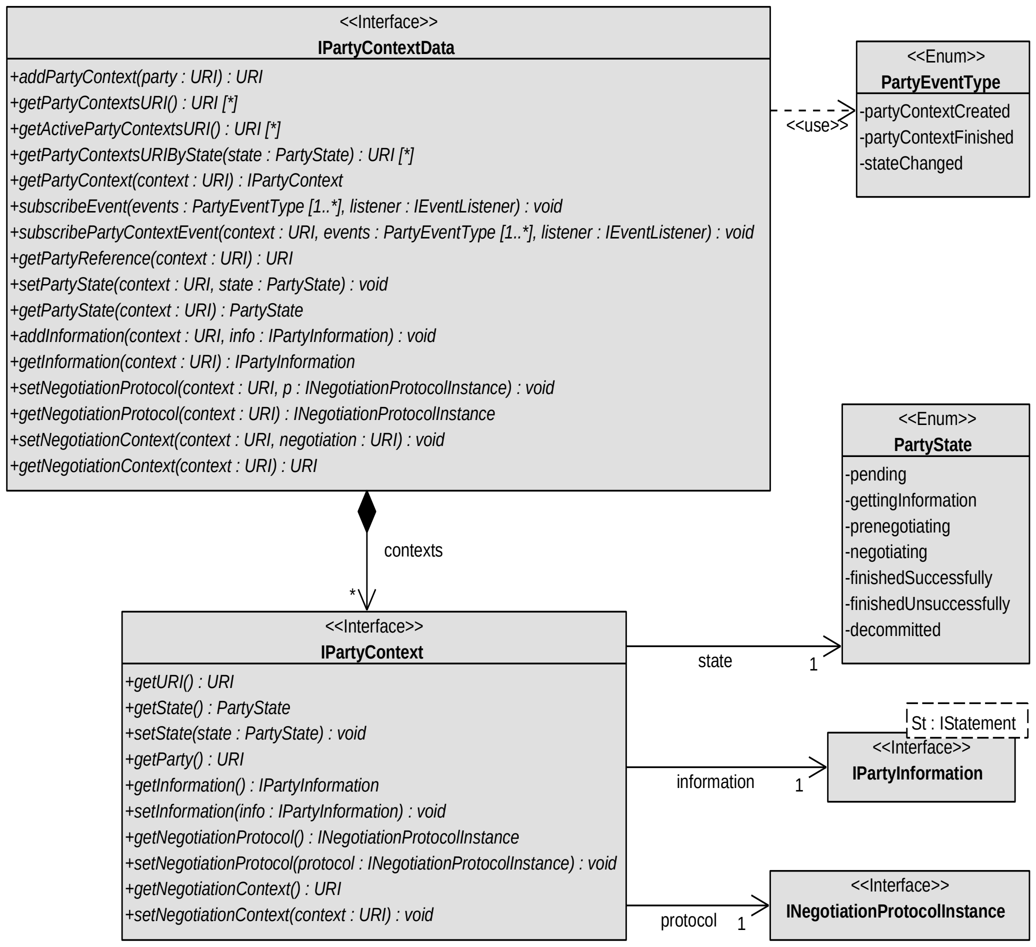
▷ `URI getURI()`: Returns an identifier for the party context.

▷ `PartyState getState()`: Returns the current state of the party context.

▷ `void setState(state)`: Sets the current state of the party context.

▷ `URI getParty()`: Returns the identifier of the party reference that is being processed by this party context.

▷ `IPartyInformation getInformation()`: Returns the information gathered about the given party for this party context. This information has been gathered in the *getting information* state.



<<Interface>>
IPartyContextData
+addPartyContext(party : URI) : URI
+getPartyContextsURI() : URI [*]
+getActivePartyContextsURI() : URI [*]
+getPartyContextsURIByState(state : PartyState) : URI [*]
+getPartyContext(context : URI) : IPartyContext
+subscribeEvent(events : PartyEventType [1..*], listener : IEventListener) : void
+subscribePartyContextEvent(context : URI, events : PartyEventType [1..*], listener : IEventListener) : void
+getPartyReference(context : URI) : URI
+setPartyState(context : URI, state : PartyState) : void
+getPartyState(context : URI) : PartyState
+addInformation(context : URI, info : IPartyInformation) : void
+getInformation(context : URI) : IPartyInformation
+setNegotiationProtocol(context : URI, p : INegotiationProtocolInstance) : void
+getNegotiationProtocol(context : URI) : INegotiationProtocolInstance
+setNegotiationContext(context : URI, negotiation : URI) : void
+getNegotiationContext(context : URI) : URI
<<use>>
<<Enum>>
PartyEventType
-partyContextCreated
-partyContextFinished
-stateChanged
contexts
*
<<Enum>>
PartyState
-pending
-gettingInformation
-prenegotiating
-negotiating
-finishedSuccessfully
-finishedUnsuccessfully
-decommitted
<<Interface>>
IPartyContext
+getURI() : URI
+getState() : PartyState
+setState(state : PartyState) : void
+getParty() : URI
+getInformation() : IPartyInformation
+setInformation(info : IPartyInformation) : void
+getNegotiationProtocol() : INegotiationProtocolInstance
+setNegotiationProtocol(protocol : INegotiationProtocolInstance) : void
+getNegotiationContext() : URI
+setNegotiationContext(context : URI) : void
state
1
St : IStatement
<<Interface>>
IPartyInformation
information
1
<<Interface>>
INegotiationProtocolInstance
protocol
1

Figure 9.8: *Interface of* PartyContextData.

▷ `void setInformation(info)`: Sets the information about the party that has been gathered via role Inquirer.

▷ `INegotiationProtocolInstance getNegotiationProtocol()`: Returns the negotiation protocol instance that has been selected for this party.

▷ `void setNegotiationProtocol(protocol)`: Sets the negotiation protocol instance.

▷ `URI getNegotiationContext()`: Gets the negotiation context that negotiates with the party

▷ `void setNegotiationContext(URI context)`: Sets the negotiation context that negotiates with the party.

The states of a party context are those depicted in Figure §7.4 and are modelled through enumeration `PartyState` (*cf.* Figure §9.8):

- ▷ `pending`: Waiting to be processed.
- ▷ `gettingInformation`: Gathering information from the other party.
- ▷ `prenegotiating`: Selecting a negotiation protocol.
- ▷ `negotiating`: Performing the negotiation.
- ▷ `finishedSuccessfully`: Negotiation with the party finishes with a new agreement.
- ▷ `finishedUnsuccessfully`: Negotiation with the party finishes without reaching an agreement.
- ▷ `decommitted`: Decommitted from a previously created agreement.

Building on those interfaces, environmental resource *PartyContextData* must implement interface `IPartyContextData`, which provides access to all party contexts of the automated negotiation system and provides a façade to query and modify the party contexts (*cf.* Figure §9.8). The methods of interface `IPartyContextData`, excluding those of the façade, which are the same as the described by interface `IPartyContext`, are:

- ▷ `URI addPartyContext(party)`: Adds a new party context with a given party and returns the URI that identifies it.
- ▷ `URI[*] getPartyContextsURIs()`: Obtains the URIs of all party contexts.
- ▷ `URI[*] getActivePartyContextsURIs()`: Obtains the URIs of all active party contexts (*i.e.* those party contexts whose state is not `finishedSuccessfully`, `finishedUnsuccessfully` or `decommitted`).
- ▷ `URI[*] getPartyContextsURIsByState(state)`: Obtains the URIs of all party contexts in a given state.
- ▷ `IPartyContext getPartyContext(context)`: Returns a reference to a party context given its URI.
- ▷ `void subscribeEvent(event, listener)`: Subscribes to one or several events. If the event is specific to a party context, it subscribes to that event in all party contexts.

▷ `void subscribePartyContextEvent(context, event, listener)`: Subscribes to one or several events of the given party context. Events that are not specific to a party context must be subscribed to using method `subscribeEvent`.

Interface `IPartyContextData` offers a subscription mechanism for events of type `PartyEventType`, which are as follows:

▷ `partyContextCreated`: A new party context is created. This is a not an event specific to a party context. The source of the notification for this event is environmental resource *PartyContextData*.

▷ `partyContextFinished`: A party context finishes either successfully or unsuccessfully. Like `partyContextCreated`, it is not specific to a party context and the source of the notification is *PartyContextData*.

▷ `stateChanged`: The state of the party context changes. This event is specific to a party context and the source of the notification is the party context that generates it.

NegotiationContextData Environmental resource *NegotiationContextData* stores information related to the negotiation contexts. A negotiation context implements a negotiation protocol instance with one or more parties. These parties are referenced by the party contexts that process each of them. Therefore, a negotiation context (interface `INegotiationContext` depicted in Figure §9.9) has one negotiation protocol, one state and several party contexts. In addition, it stores all negotiation messages that have been exchanged as part of the negotiation context:

▷ `URI getURI()`: Returns an identifier for the negotiation context.

▷ `INegotiationProtocol getProtocol()`: Gets the negotiation protocol that has been selected for the negotiation context.

▷ `Date getStartDate()`: Returns the date when the negotiation context was started.

▷ `URI[1..*] getPartyContexts()`: Returns the URIs of the the party contexts associated with the parties that are negotiating in this negotiation context.

▷ `void addPartyContext(partyContext)`: Adds the party context associated with a party that is going to participate in the negotiation of this negotiation context.

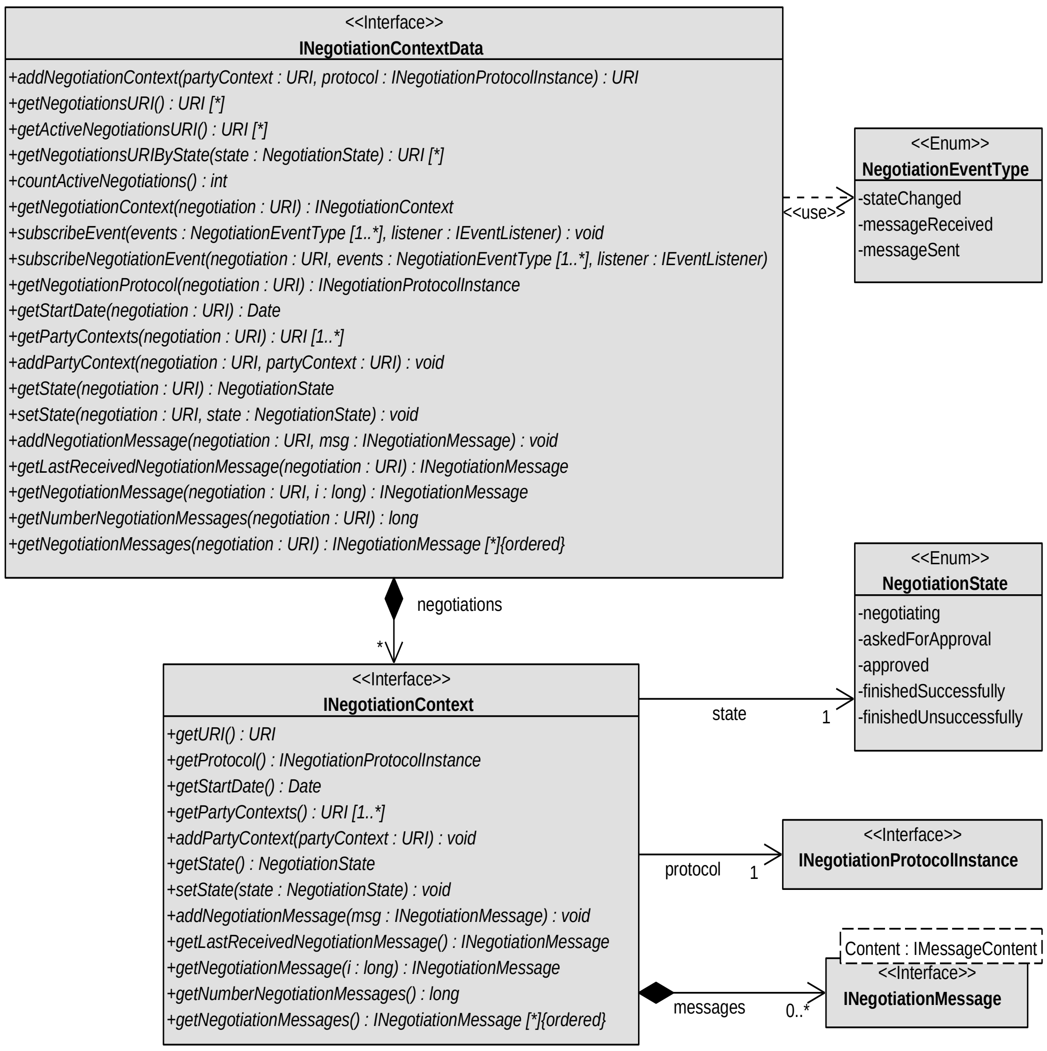
- ▷ `NegotiationState getState()`: Gets the current state of a negotiation context.
- ▷ `void setState(state)`: Changes the state of a negotiation context.
- ▷ `void addNegotiationMessage(msg)`: Adds the negotiation message that has been received or sent.
- ▷ `INegotiationMessage getLastReceivedNegotiationMessage()`: Gets the last negotiation message that has been received.
- ▷ `INegotiationMessage getNegotiationMessage(i)`: Gets the `i`-th zero-based negotiation message that has been sent or received.
- ▷ `long getNumberNegotiationMessages()`: Gets the number of negotiation messages that have been sent or received.
- ▷ `INegotiationMessage[*]{ordered} getNegotiationMessages()`: Gets all negotiation messages that have been sent or received since the negotiation started.

The states of a negotiation context are those depicted in Figure §7.5 and are modelled through enumeration `NegotiationState` (*cf.* Figure §9.9):

- ▷ `negotiating`: Exchanging non-binding negotiation messages amongst the parties.
- ▷ `askedApproval`: Requested approval to send a binding negotiation message to the CommitHandler.
- ▷ `approved`: Waiting the response of the other party after sending a binding negotiation message approved by the CommitHandler.
- ▷ `finishedSuccessfully`: Negotiation finished with an agreement.
- ▷ `finishedUnsuccessfully`: Negotiation finished without reaching an agreement.

Building on those interfaces, environmental resource *NegotiationContextData* implements interface `INegotiationContextData`, which provides access to all negotiation contexts and provides a façade to query and modify them (*cf.* Figure §9.9). The methods of interface `INegotiationContextData`, excluding those of the façade, which are the same as the described by interface `INegotiationContext`, are:

- ▷ `URI addNegotiationContext(partyContext, protocol)`: Adds a new negotiation context to negotiate with the party of the given party context following the given protocol and returns the URI that identifies it.



<<Interface>>
INegotiationContextData
+addNegotiationContext(partyContext : URI, protocol : INegotiationProtocolInstance) : URI
+getNegotiationsURI() : URI [*]
+getActiveNegotiationsURI() : URI [*]
+getNegotiationsURIByState(state : NegotiationState) : URI [*]
+countActiveNegotiations() : int
+getNegotiationContext(negotiation : URI) : INegotiationContext
+subscribeEvent(events : NegotiationEventType [1..*], listener : IEventListener) : void
+subscribeNegotiationEvent(negotiation : URI, events : NegotiationEventType [1..*], listener : IEventListener)
+getNegotiationProtocol(negotiation : URI) : INegotiationProtocolInstance
+getStartDate(negotiation : URI) : Date
+getPartyContexts(negotiation : URI) : URI [1..*]
+addPartyContext(negotiation : URI, partyContext : URI) : void
+getState(negotiation : URI) : NegotiationState
+setState(negotiation : URI, state : NegotiationState) : void
+addNegotiationMessage(negotiation : URI, msg : INegotiationMessage) : void
+getLastReceivedNegotiationMessage(negotiation : URI) : INegotiationMessage
+getNegotiationMessage(negotiation : URI, i : long) : INegotiationMessage
+getNumberNegotiationMessages(negotiation : URI) : long
+getNegotiationMessages(negotiation : URI) : INegotiationMessage [*]{ordered}
<<use>>
<<Enum>>
NegotiationEventType
-stateChanged
-messageReceived
-messageSent
negotiations
*
<<Interface>>
INegotiationContext
+getURI() : URI
+getProtocol() : INegotiationProtocolInstance
+getStartDate() : Date
+getPartyContexts() : URI [1..*]
+addPartyContext(partyContext : URI) : void
+getState() : NegotiationState
+setState(state : NegotiationState) : void
+addNegotiationMessage(msg : INegotiationMessage) : void
+getLastReceivedNegotiationMessage() : INegotiationMessage
+getNegotiationMessage(i : long) : INegotiationMessage
+getNumberNegotiationMessages() : long
+getNegotiationMessages() : INegotiationMessage [*]{ordered}
state
1
<<Enum>>
NegotiationState
-negotiating
-askedForApproval
-approved
-finishedSuccessfully
-finishedUnsuccessfully
protocol
1
<<Interface>>
INegotiationProtocolInstance
messages
0..*
Content : IMessageContent
<<Interface>>
INegotiationMessage

Figure 9.9: *Interface of* NegotiationContextData.

▷ `URI[*] getNegotiationURIs()`: Obtains the URIs of all negotiation contexts.

▷ `URI[*] getActiveNegotiationsURIs()`: Gets the URIs of the negotiation contexts that are not in state `finishedSuccessfully` or `finishedUnsuccessfully`.

▷ `URI[*] getNegotiationsURIsByState(state)`: Obtains the URIs of all negotiation contexts in a given state.

▷ `int countActiveNegotiations()`: Returns the number of active negotiations.

▷ `INegotiationContext getNegotiationContext(negotiation)`: Returns a reference to the negotiation context given its URI.

▷ `void subscribeEvent(events, listener)`: Subscribes to one or more events in every negotiation context or to events that are not specific to a concrete negotiation context, see below.

▷ `void subscribeNegotiationEvent(context, events, listener)`: Subscribes to one or several events of the given negotiation context.

These events are defined by enumeration `NegotiationEventType`:

▷ `negotiationCreated`: A new negotiation context is created.

▷ `negotiationFinished`: A negotiation context finishes either successfully or unsuccessfully.

▷ `stateChanged`: The state of the negotiation context changes.

▷ `messageReceived`: A negotiation message is received.

▷ `messageSent`: A negotiation message is sent.

Events `negotiationCreated` and `negotiationFinished` are not specific to a negotiation context and, hence, the source of the notification is environmental resource *NegotiationContextData* and must be subscribed by means of method `subscribeEvent`. Conversely, events `stateChanged`, `messageReceived` and `messageSent` are specific to a negotiation context and the source of the notification is the negotiation context that generates it.

WorldModel The *WorldModel* is implemented by means of two interfaces: interface `IEstimatorsLibrary` and interface `IEstimator` (*cf.* Figure §9.10). An estimator handles information about the parties, the market or a concrete domain, whereas the estimators library is a repository of all estimators that are available to the automated negotiation system. Each estimator covers one aspect of the *WorldModel*, *e.g.*, market reservation price, reputation, opponent behaviour or proposals similarity.

Note that this allows to have complex estimators that build on several simpler estimators and, hence, it enhances their reusability. For instance, an estimator may aggregate the information provided by other estimators and, thus, provides a more convenient way of accessing the information. In addition,

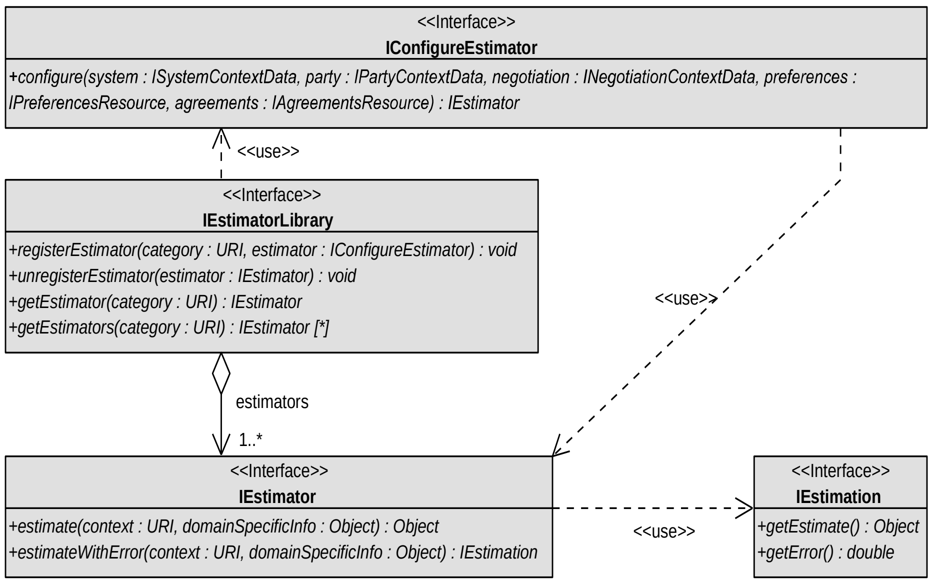

<<Interface>>
IConfigureEstimator
+configure(system : ISystemContextData, party : IPartyContextData, negotiation : INegotiationContextData, preferences : IPreferencesResource, agreements : IAgreementsResource) : IEstimator
<<use>>
<<Interface>>
IEstimatorLibrary
+registerEstimator(category : URI, estimator : IConfigureEstimator) : void
+unregisterEstimator(estimator : IEstimator) : void
+getEstimator(category : URI) : IEstimator
+getEstimators(category : URI) : IEstimator [*]
<<use>>
estimators
1..*
<<Interface>>
IEstimator
+estimate(context : URI, domainSpecificInfo : Object) : Object
+estimateWithError(context : URI, domainSpecificInfo : Object) : IEstimation
<<use>>
<<Interface>>
IEstimation
+getEstimate() : Object
+getError() : double

Figure 9.10: *Interface of* WorldModel.

estimators can act as proxies for external information providers. For instance, a reputation estimator may act as proxy for an external reputation server.

We distinguish between estimator categories, which are the different types of estimator (*e.g.* a reputation estimator), and concrete estimators, which are the actual software artifacts that provide algorithms to implement the estimator categories. The estimators library decouples the elements of the automated negotiation system from concrete estimators. For instance, an element of the automated negotiation system may ask the estimators library for a reputation estimator and the estimators library shall provide it. Estimator categories are identified using URIs.

The estimators library interface includes the following methods:

▷ `void registerEstimator(category, estimator)`: Registers a new concrete estimator that implements an estimator category and configures it with the current environmental resources.

▷ `void unregisterEstimator(estimator)`: Unregisters a estimator.

▷ `IEstimator getEstimator(category)`: Returns the default concrete estimator that implements an estimator category.

▷ `IEstimator[*] getEstimators(category)`: Obtains all concrete estimators that implement an estimator category.

Unlike the previous environmental resources which run within a system context, the *WorldModel* is outside it. The reason is that the model of the elements involved in a negotiation is valid across several system contexts. However, the system context does have an influence on those models, so the estimators may use the system context to adapt the information it provides. To this end, when an estimator is registered in the estimators library, it is configured with the current environmental resources (*SystemContextData*, *PartyContextData*, *NegotiationContextData*, *PreferencesResource* and *AgreementsResource*). Thus, the estimators library always returns estimators that have been configured with the current environmental resources.

The configuration interface of an estimator is `IConfigureEstimator`, which includes the following methods:

▷ `IEstimator configure(system, party, neg, prefs, agreem)`: Configures an estimator with the current environmental resources and returns it.

Interface `IEstimator` includes methods to obtain the estimation for a given context. The type of context (system, party or negotiation) depends on the concrete estimator. Furthermore, estimators may require additional domain-specific information that must be sent together with the context. For instance, two proposals must be sent to an estimator that measures their similarity (*cf.* Section §4.3.2 for more information about this estimator). Building on these issues, the interface `IEstimator` includes, at least, the following methods:

▷ `Object estimate(context, domainSpecificInfo)`: Obtains an estimation for a particular context. The estimation may require domain-specific information (see above). The returned value is a domain-specific structure as well.

▷ `IEstimation estimateWithError(context, domainSpecificInfo)`: Returns an estimation for a particular context with the associated error.

When the error associated with the estimation is required, an element that implements interface `IEstimation` is returned. The interface just provides two methods to access to both the estimation and the associated error:

▷ `Object getEstimate()`: Returns the estimation as a domain-specific structure.

▷ `double getError()`: Returns the error associated with the estimation.

Regarding the update of the models provided by the estimators, there are two ways of doing it, namely: online and offline estimation (*cf.* Section §4.3).

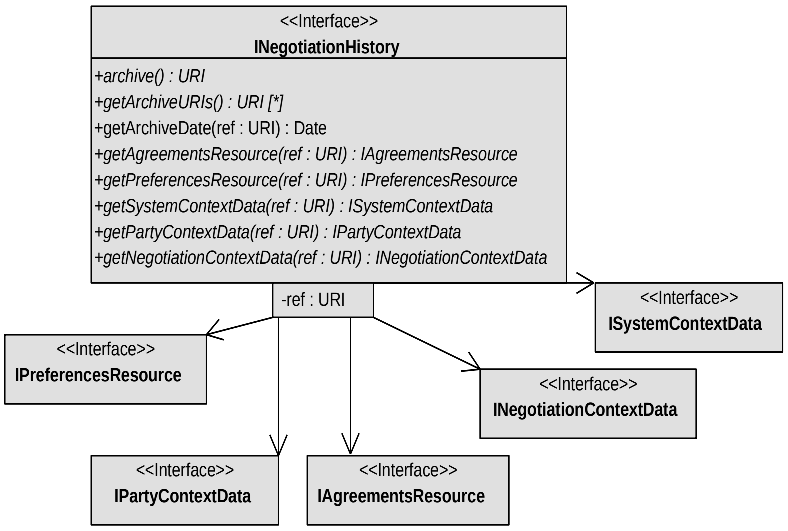

<<Interface>>
INegotiationHistory
+archive() : URI
+getArchiveURIs() : URI [*]
+getArchiveDate(ref : URI) : Date
+getAgreementsResource(ref : URI) : IAgreementsResource
+getPreferencesResource(ref : URI) : IPreferencesResource
+getSystemContextData(ref : URI) : ISystemContextData
+getPartyContextData(ref : URI) : IPartyContextData
+getNegotiationContextData(ref : URI) : INegotiationContextData
-ref : URI
<<Interface>>
ISystemContextData
<<Interface>>
IPreferencesResource
<<Interface>>
INegotiationContextData
<<Interface>>
IPartyContextData
<<Interface>>
IAgreementsResource

Figure 9.11: *Interface of* NegotiationHistory.

Offline estimation can be done by analysing previous interactions using the *NegotiationHistory*. To perform online estimations, estimators may subscribe to the notifications provided by the *SystemContextData*, *PartyContextData* and *NegotiationContextData*. In this case, the estimator has to implement interface `IEventListener`.

NegotiationHistory The *NegotiationHistory* (*cf.* Figure §9.11) is mainly intended for the estimators to develop offline estimations. However, it can be used by decision-making elements of the automated negotiation system, such as the ResponseGenerator or the CommitHandler, to support their decisions.

The *NegotiationHistory* can be seen as a list of the environmental resources of all system contexts that have been processed by the automated negotiation system. Its interface (`INegotiationHistory`) contains the following methods:

▷ `URI archive()`: Archives environmental resources *AgreementsResource*, *PreferencesResource*, *SystemContextData*, *PartyContextData* and *NegotiationContextData* and returns a reference to the system context in order access them later.

▷ `Date getArchiveDate(ref)`: Returns the date when the environmental resources were archived.

▷ `IAgreementsResource getAgreementsResource(ref)`: Gets a reference to an implementation of interface `IAgreementsResource` that provides access to all data stored in that environmental resource in the given system context.

▷ `IPreferencesResource getPreferencesResource(ref)`: Same as before, but for interface `IPreferencesResource`.

▷ `ISystemContextData getSystemContextData(ref)`: Same as before, but for interface `ISystemContextData`.

▷ `IPartyContextData getPartyContextData(ref)`: Same as before, but for interface `IPartyContextData`.

▷ `INegotiationContextData getNegotiationContextData(ref)`: Same as before, but for interface `INegotiationContextData`.

▷ `URI[*] getArchiveURIs()`: Queries the URIs of all past resources that are stored in the negotiation history.

9.4 Interactions

We define the NegoFAST-Core framework in terms of the interactions between the roles described in the NegoFAST-Core reference architecture. Therefore, for each interaction, we describe the protocol and the interfaces that must provide the roles it involves.

UserInteraction This interaction is used by the user of the automated negotiation system to provide the preferences and the references to parties with whom to negotiate. Figure §9.12 depicts the interaction in which the User participates, which must implement interface `IUser`, and the SystemCoordinator, which implements interface `ICoordinator`.

Interface `ICoordinator` is as follows:

▷ `void init(user, preferences)`: Starts a new negotiation context with the given preferences.

▷ `void startNegotiation(party)`: Requests the automated negotiation system to initiate a negotiation with the given party.

▷ `void terminate()`: Finishes the negotiation context and, hence, all active negotiation threads.

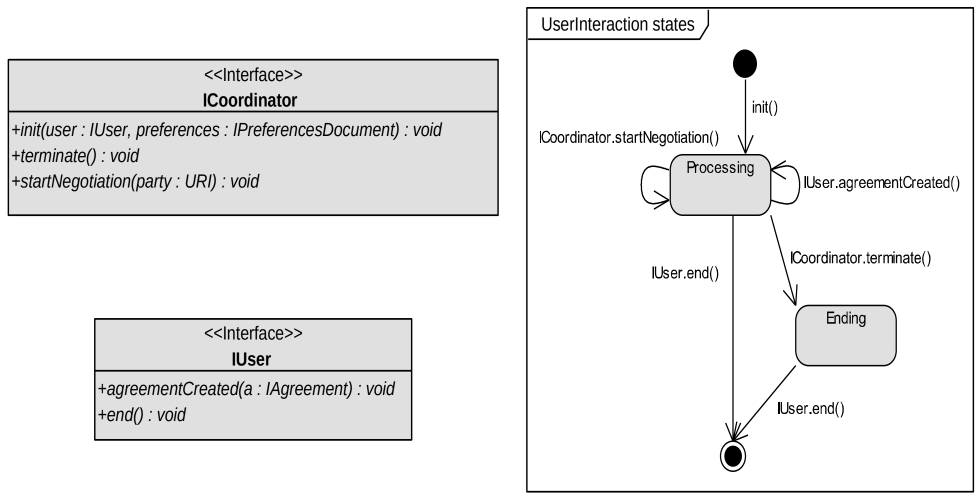


Figure 9.12: *Interaction* UserInteraction.

Interface `IUser` is composed of the following methods:

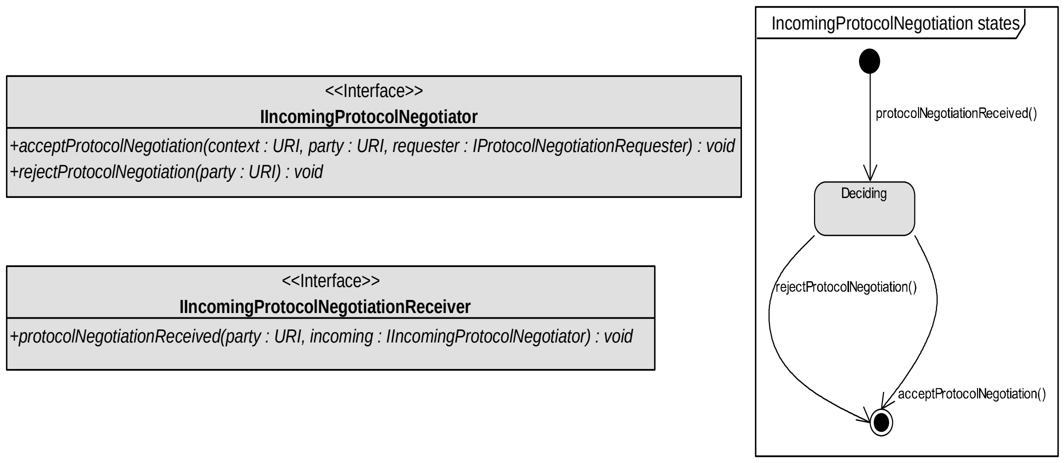
▷ `void agreementCreated(agreement)`: Notifies that the specified agreement has been reached.

▷ `void end()`: Notifies that the automated negotiation system finished working.

The interaction protocol is as follows: first, the User invokes method `init` in interface `ICoordinator` to start the system context. After that, the User may invoke method `startNegotiation` to initiate a negotiation with the given party. At the same time, the SystemCoordinator notifies the User when a new agreement has been created by invoking method `agreementCreated`.

The interaction protocol remains in that state until either the User finishes the system context by invoking method `terminate` or one of the termination conditions hold (*cf.* Section §7.2.1). In both cases, the SystemCoordinator finishes all active thread contexts and invokes method `end` to indicate the User that the automated negotiation system has finished working.

IncomingProtocolNegotiation By means of this interaction, the ProtocolNegotiator notifies the automated negotiation system that another party has sent a request to start a protocol negotiation and asks for permission to proceed with it. In this interaction, the SystemCoordinator implements interface `IIncomingProtocolNegotiationReceiver`, whereas the ProtocolNegotia-



<<Interface>>
IIncomingProtocolNegotiator
+acceptProtocolNegotiation(context : URI, party : URI, requester : IProtocolNegotiationRequester) : void
+rejectProtocolNegotiation(party : URI) : void
<<Interface>>
IIncomingProtocolNegotiationReceiver
+protocolNegotiationReceived(party : URI, incoming : IIncomingProtocolNegotiator) : void
IncomingProtocolNegotiation states
protocolNegotiationReceived()
Deciding
rejectProtocolNegotiation()
acceptProtocolNegotiation()

Figure 9.13: *Interaction* IncomingProtocolNegotiation.

tor implements interface `IIncomingProtocolNegotiator` (*cf.* Figure §9.13).

The former interface is composed of one method:

▷ `void protocolNegotiationReceived(party, incoming)`: Notifies that a protocol negotiation request from the given party has been received and asks for permission to proceed with it. It also includes a reference to the ProtocolNegotiator (`incoming`) that shall process the request.

The latter interface has the following methods:

▷ `void acceptProtocolNegotiation(ctx, party, requester)`: Grants permission to start a protocol negotiation with the given party. In addition it assigns it to a new party context and gives a reference to the element that must receive the result of the protocol negotiation.

▷ `void rejectProtocolNegotiation(party)`: Rejects starting a protocol negotiation with the given party.

The interaction protocol is simple. When the ProtocolNegotiator receives a request to start a protocol negotiation from another party, it invokes method `protocolNegotiationReceived`. Then, the SystemCoordinator decides whether to start the protocol negotiation or not and responds to the ProtocolNegotiator by invoking method `acceptProtocolNegotiation` or method `rejectProtocolNegotiation`, respectively.

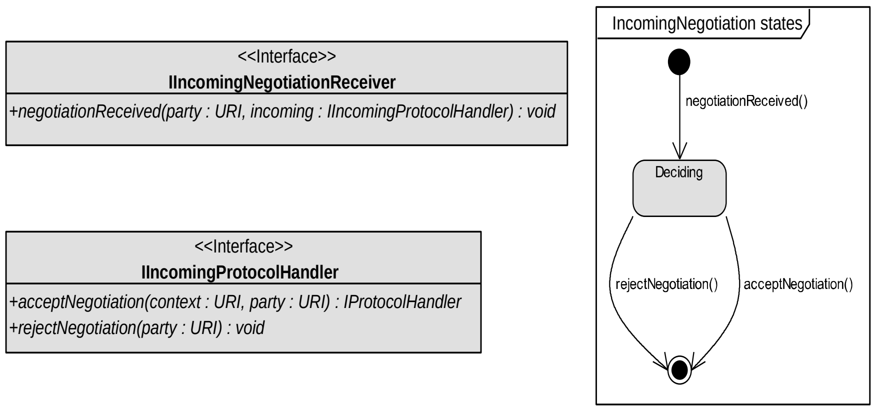

<<Interface>>
IIncomingNegotiationReceiver
+negotiationReceived(party : URI, incoming : IIncomingProtocolHandler) : void
<<Interface>>
IIncomingProtocolHandler
+acceptNegotiation(context : URI, party : URI) : IProtocolHandler
+rejectNegotiation(party : URI) : void
IncomingNegotiation states
negotiationReceived()
Deciding
rejectNegotiation()
acceptNegotiation()

Figure 9.14: *Interaction* IncomingNegotiation.

IncomingNegotiation This interaction is much the same as the previous one, the difference being that, in this case, a negotiation request is processed instead of a protocol negotiation request. As a consequence, the participants are the SystemCoordinator, which implements interface `IIncomingNegotiationReceiver` and the ProtocolHandler, which implements interface `IIncomingNegotiator` (*cf.* Figure §9.14).

Interface `IIncomingNegotiationReceiver` is composed of one method:

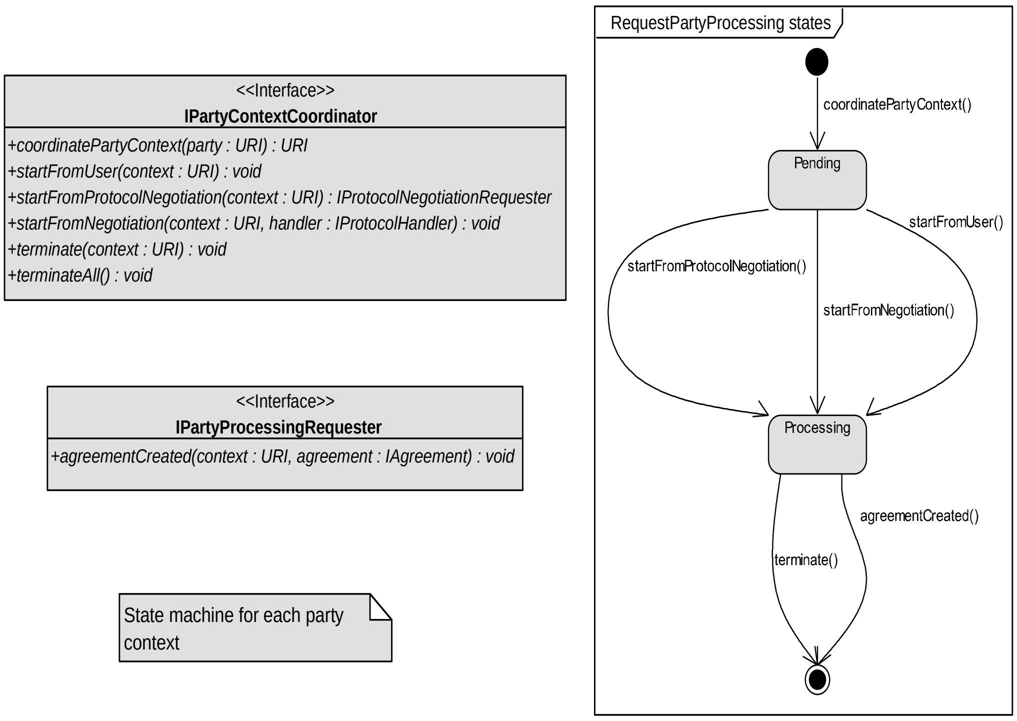
▷ `void negotiationReceived(party, incoming)`: Notifies that a negotiation request from the given party has been received and asks for permission to proceed with it. It also includes a reference to the element that shall process the request.

Interface `IIncomingNegotiator` has the following methods:

▷ `IProtocolHandler acceptNegotiation(context, party)`: Grants permission to start a negotiation with the given party. In addition it assigns it to a new party context. This method returns a reference to the element that shall handle the protocol.

▷ `void rejectNegotiation(party)`: Rejects starting a negotiation with the given party.

The interaction protocol is the same as the protocol of interaction *IncomingProtocolNegotiation*.



<<Interface>>
IPartyContextCoordinator
+coordinatePartyContext(party : URI) : URI
+startFromUser(context : URI) : void
+startFromProtocolNegotiation(context : URI) : IProtocolNegotiationRequester
+startFromNegotiation(context : URI, handler : IProtocolHandler) : void
+terminate(context : URI) : void
+terminateAll() : void
<<Interface>>
IPartyProcessingRequester
+agreementCreated(context : URI, agreement : IAgreement) : void
State machine for each party context
RequestPartyProcessing states
coordinatePartyContext()
Pending
startFromUser()
startFromProtocolNegotiation()
startFromNegotiation()
Processing
agreementCreated()
terminate()

Figure 9.15: *Interaction* RequestPartyProcessing.

RequestPartyProcessing This interaction is a request for the PartyCoordinator to initiate a party context to process a party reference that has been received by the SystemCoordinator. As depicted in Figure §9.15, there are two participants in this interaction: the PartyCoordinator, which implements interface `IPartyContextCoordinator`, and the SystemCoordinator, which implements interface `IPartyProcessingRequester`.

The PartyCoordinator implements interface `IPartyContextCoordinator`, which is composed of the following methods:

▷ `URI coordinatePartyContext(party)`: Creates a party context to process the given party reference and returns its identifier. Note that the processing does not start until any of the next three start methods is invoked.

▷ `void startFromUser(context)`: Starts the processing of the given party context and indicates that the party reference came from the User.

▷ `IProtocolNegotiationRequester startFromProtocolNegotiation(context)`: Starts the processing of the given party context and indicates that the party reference came from an incoming protocol negotiation. It returns a reference to the element that shall process the result of the protocol negotiation.

▷ `void startFromNegotiation(context, handler)`: Starts the processing of the given party context and indicates that the party reference came from an incoming negotiation. In addition, it includes the ProtocolHandler that shall manage the negotiation protocol instance.

▷ `void terminate(context)`: Terminates the given party context.

▷ `void terminateAll()`: Terminates all active party contexts.

Interface `IPartyProcessingRequester`, which has the following methods:

▷ `void agreementCreated(context, agreement)`: Notifies that an agreement was reached with the given party

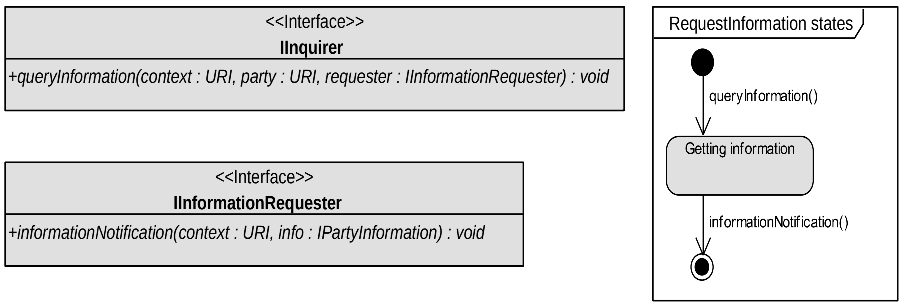
The interaction protocol for a given party context is depicted in Figure §9.15 and is as follows: first, the SystemCoordinator invokes method `coordinatePartyContext` in interface `IPartyContextCoordinator` to start a new party context for the given party reference. Then, depending on the source of the party reference, it invokes either method `startFromUser`, `startFromProtocolNegotiation` or `startFromNegotiation`. After that, either the PartyCoordinator invokes method `agreementCreated` of interface `IPartyProcessingRequester` when the agreement has been reached, or the SystemCoordinator asks for the termination of the processing by invoking method `terminate`. In addition, the SystemCoordinator may end all active party contexts by invoking method `terminateAll`.

RequestInformation This interaction is a request for the Inquirer to obtain the information from other parties that is necessary to start the negotiation. It is an asynchronous request. As depicted in Figure §9.16, the participants of this interaction are the Inquirer, which must implement interface `IInquirer`, and the PartyCoordinator, which must implement interface `IInformationRequester`.

Interface `IInquirer` has only one method:

▷ `void queryInformation(context, party, requester)`: Requests the Inquirer to get information from a party and includes the URI of the party context and an identifier to the other party, so that the Inquirer can interact with it. In addition, it includes a reference to the requester.

Interface `IInformationRequester` has one method:



<<Interface>>
IInquirer
+queryInformation(context : URI, party : URI, requester : IInformationRequester) : void
<<Interface>>
IInformationRequester
+informationNotification(context : URI, info : IPartyInformation) : void
RequestInformation states
queryInformation()
Getting information
informationNotification()

Figure 9.16: *Interaction* RequestInformation.

▷ `void informationNotification(context, partyInformation)`: It is invoked when the information has been obtained, and it includes the URI of the party context and the information gathered.

RequestProtocolNegotiation This interaction is a request for the ProtocolNegotiator to agree on the negotiation protocol the other party. It is an asynchronous request. It involves the ProtocolNegotiator, which implements interface `IProtocolNegotiator`, and the PartyCoordinator, which implements interface `IProtocolNegotiationRequester` (*cf.* Figure §9.17).

Interface `IProtocolNegotiator` is the following:

▷ `void negotiateProtocol(context, party, equester)`: Initiates a protocol negotiation and includes the URI of the party context and an identifier of the other party, so that the ProtocolNegotiator can interact with it. It also includes the requester that receives the result of the protocol negotiation

Interface `IProtocolNegotiationRequester` has two methods:

▷ `void successfulProtocolNegotiation(ctx, protocol, handler)`: Informs that the protocol negotiation finished successfully and includes the URI of the party context, the selected negotiation protocol instance and a reference to the ProtocolHandler that is configured to manage the interactions with the other party.

▷ `void failedProtocolNegotiation(context)`: Notifies that the protocol negotiation for the given party context failed.

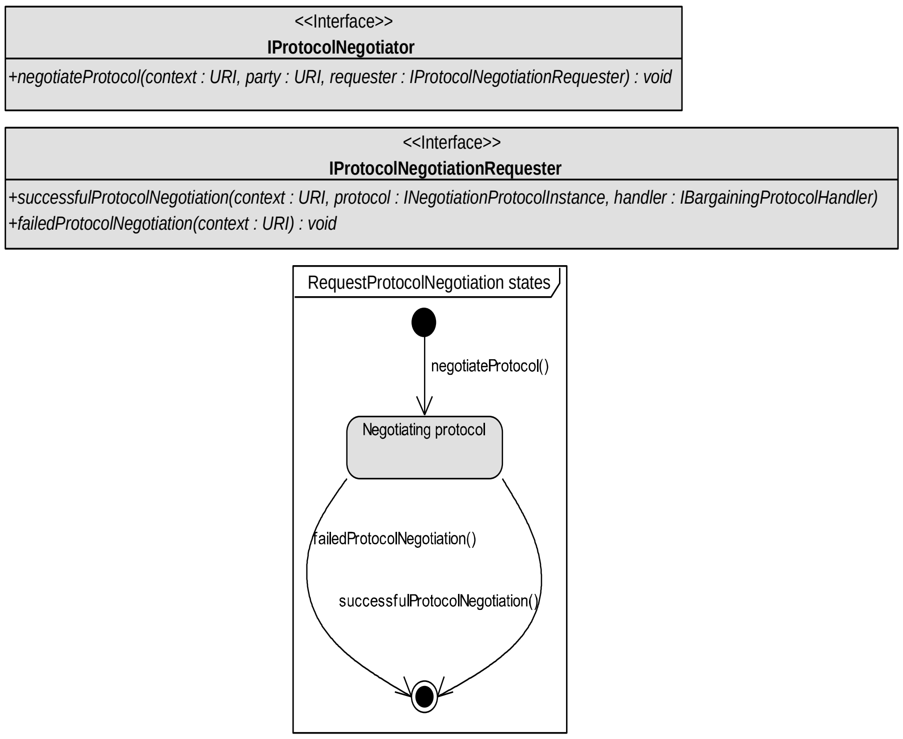

<<Interface>>
IProtocolNegotiator
+negotiateProtocol(context : URI, party : URI, requester : IProtocolNegotiationRequester) : void
<<Interface>>
IProtocolNegotiationRequester
+successfulProtocolNegotiation(context : URI, protocol : INegotiationProtocolInstance, handler : IBargainingProtocolHandler)
+failedProtocolNegotiation(context : URI) : void
RequestProtocolNegotiation states
negotiateProtocol()
Negotiating protocol
failedProtocolNegotiation()
successfulProtocolNegotiation()

Figure 9.17: *Interaction* RequestProtocolNegotiation.

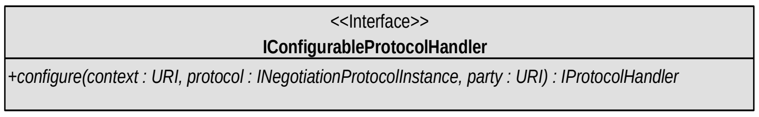
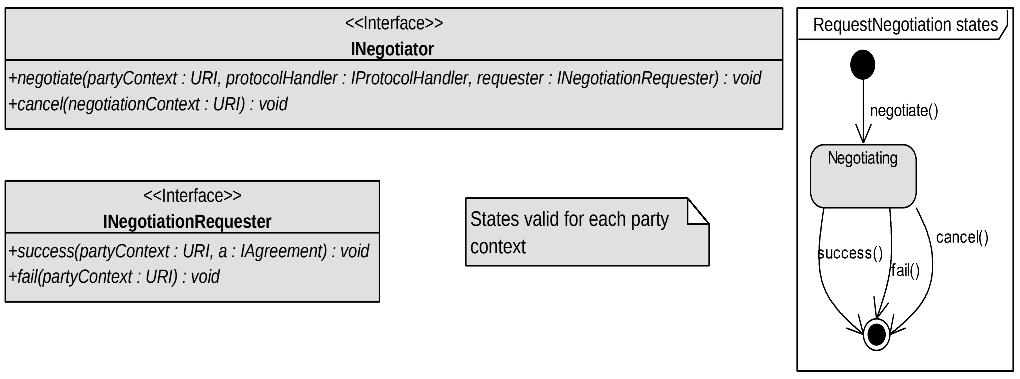

<<Interface>>
IConfigurableProtocolHandler
+configure(context : URI, protocol : INegotiationProtocolInstance, party : URI) : IProtocolHandler

Figure 9.18: *Interaction* ConfigureHandler.

ConfigureHandler This interaction configures a ProtocolHandler, which is protocol-specific, with the negotiation protocol instance and the reference to the other party that were obtained in interaction *RequestProtocolNegotiation*. It is a synchronous request. The ProtocolHandler must implement interface `IConfigurableProtocolHandler`, which has one method:

▷ `IProtocolHandler configure(context, protocol, party)`: Initialises the ProtocolHandler with the party context, the negotiation protocol instance and a URI that identifies the other party and returns a reference to the ProtocolHandler that shall manage the interaction with the other party.



<<Interface>>
INegotiator
+negotiate(partyContext : URI, protocolHandler : IProtocolHandler, requester : INegotiationRequester) : void
+cancel(negotiationContext : URI) : void
<<Interface>>
INegotiationRequester
+success(partyContext : URI, a : IAgreement) : void
+fail(partyContext : URI) : void
States valid for each party context
RequestNegotiation states
negotiate()
Negotiating
success()
fail()
cancel()

Figure 9.19: *Interaction* RequestNegotiation.

RequestNegotiation This interaction is the request for starting a negotiation context that implements the execution of a negotiation protocol instance with one or several parties. It also includes the notification of the results of this negotiation and the request for cancellation of the negotiation if necessary. As depicted in Figure §9.19, the PartyCoordinator must implement `INegotiationRequester` interface, whereas the NegotiationCoordinator must implement interface `INegotiator`, which has two methods:

▷ `URI negotiate(partyCtx, handler, requester)`: Requests the NegotiationCoordinator to start a negotiation. It includes the URI of the party context, the reference to the ProtocolHandler that shall manage the interaction with the other party and the reference to the requester to notify it about the results of the negotiation. It returns an URI to identify the negotiation context that has been created.

▷ `void cancel(negotiationContext)`: Requests the cancellation of the negotiation in the given negotiation context.

Interface `INegotiationRequester` is also composed of two methods:

▷ `void success(partyContext, agreement)`: Informs that the given agreement has been reached with the party processed by the specified party context.

▷ `void fail(partyContext)`: Indicates that the negotiation has failed for the party processed by the given party context.

The interaction protocol for one party context is as follows (*cf.* Figure §9.19): first, the negotiation requester starts a negotiation by invoking

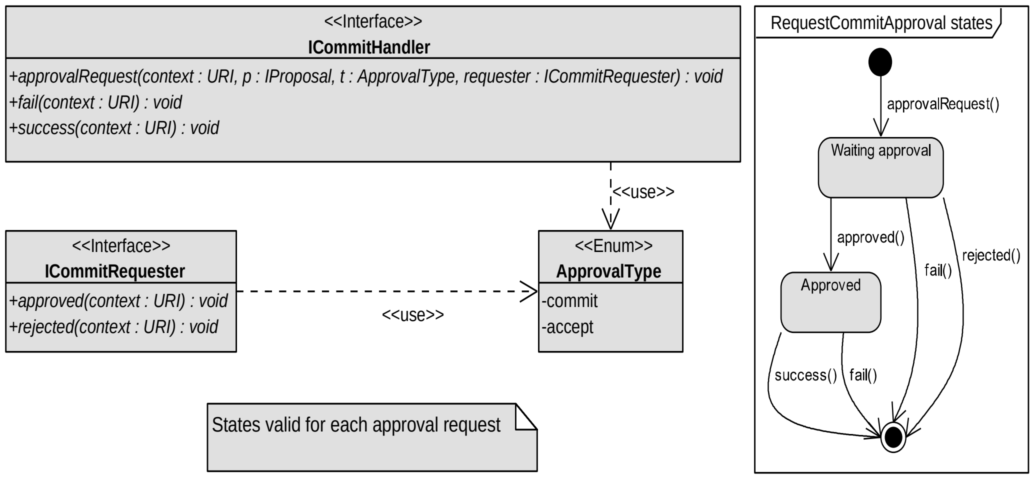

<<Interface>>
ICommitHandler
+approvalRequest(context : URI, p : IProposal, t : ApprovalType, requester : ICommitRequester) : void
+fail(context : URI) : void
+success(context : URI) : void
<<use>>
<<Interface>>
ICommitRequester
+approved(context : URI) : void
+rejected(context : URI) : void
<<use>>
<<Enum>>
ApprovalType
-commit
-accept
States valid for each approval request
RequestCommitApproval states
approvalRequest()
Waiting approval
approved()
fail()
rejected()
Approved
success()
fail()

Figure 9.20: *Interaction* RequestCommitApproval.

method `negotiate` on the negotiator. Then, the negotiation can either succeed, in which case method `success` of interface `INegotiationRequester` is invoked, or fail, in which case method `fail` is invoked. In addition, while negotiating, the PartyCoordinator may cancel the negotiation at any time by invoking method `cancel`.

RequestCommitApproval This interaction is to request approval for sending a binding negotiation message (*i.e.* sending a `commit` or an `accept`). The interaction is asynchronous. There are two participants (*cf.* Figure §9.20): the CommitHandler, which implements interface `ICommitHandler`, and the NegotiationCoordinator, which implements interface `ICommitRequester`.

Interface `ICommitHandler` has the following methods:

▷ `void approvalRequest(context, proposal, type, requester):` Requests the CommitHandler to approve the sending of a binding negotiation message and includes the negotiation context URI, the proposal, the type of approval represented by enumeration *ApprovalType*, and a reference to the requester.

▷ `void fail(context):` Notifies that an approval previously requested has failed.

▷ `void success(context):` Notifies that the given negotiation context, which received a previous approval, has successfully reached an agreement.

The type of approval is represented by enumeration `ApprovalType`, which is composed of:

▷ `commit`: The approval is asked for sending a commit. This means that the other party may or may not accept the proposal.

▷ `accept`: The approval is asked for sending an accept. This means that the other party has already committed to the proposal and, hence, if approved, an agreement shall be reached.

Interface `ICommitRequester` includes the following methods:

▷ `void approved(context)`: Notifies that the approval request has been accepted for the given negotiation context.

▷ `void rejected(context)`: Notifies that the approval request has been rejected for the given negotiation context.

The protocol works as follows (*cf.* Figure §9.20): the NegotiationCoordinator sends approval requests (`approvalRequest`) to the CommitHandler. If one of those fails while waiting for the response (*e.g.* the other party sends a reject negotiation message), the NegotiationCoordinator invokes method `fail` together with the negotiation context in which the fail took place.

When the CommitHandler makes a decision, it invokes method `approved` zero or more times with the negotiation context of the approved proposals. It also invokes method `rejected` zero or more times with the negotiation context of the rejected proposals. Note that not all proposals must be either accepted or rejected at a time. Some may be postponed for a later decision.

For each approved binding negotiation message, the NegotiationCoordinator responds by invoking method `success`, if the agreement was created, or invoking method `fail` if it was not.

RequestAdvise This interaction is used to obtain a recommendation about whether it is convenient to commit to an agreement or not. The interaction is synchronous and involves the invocation of one method of interface `ICommitAdvisor`, which is implemented by the CommitAdvisors:

▷ `double getAdvise(context, proposal)`: Returns a normalised value (between 0 and 1) that is a measure of how convenient committing to that proposal is.

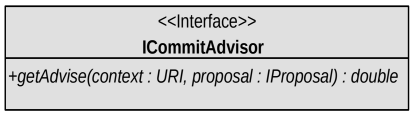


Figure 9.21: *Interaction* RequestAdvise.

9.5 State machines

There are two roles that coordinate the others in the NegoFAST-Core framework. These roles are the SystemCoordinator and the PartyCoordinator. In this section, we detail their behaviour in terms of a state-machine.

SystemCoordinator The SystemCoordinator coordinates the system context and, hence, it manages the interaction with the User and receives the party references from different sources. Its state machine is depicted in Figure §9.22 and it follows the states defined for the system context (*cf.* Section §7.2.1). The state machine of the SystemCoordinator starts when the User invokes method `init` of the SystemCoordinator. At that moment, the SystemCoordinator enters state *processing* and stays there until the user sends message `terminate` message or any other condition to terminate the system context holds (*cf.* Section §7.2.1). In that case, all current party contexts are cancelled by invoking method `terminateAll` in the PartyCoordinator.

During the *processing* state, the SystemCoordinator may receive party references from the User by means of method `startNegotiation`; from the ProtocolNegotiator by means of method `protocolNegotiationReceived`, and from the ProtocolHandler by means of method `negotiationReceived`.

If the automated negotiation system is not interested in starting new negotiations, the SystemCoordinator may reject party references by invoking method `rejectProtocolNegotiation` or method `rejectNegotiation` on the ProtocolNegotiator and the ProtocolHandler, respectively. Party references coming from the User cannot be rejected.

Alternatively, if the party reference is accepted, the SystemCoordinator first invokes method `coordinatePartyContext` on the PartyCoordinator to create a new party context; then, it invokes method `startFromUser`, `startFromProtocolNegotiator` or `startFromProtocolHandler`, depending on which was the source of the party reference; finally, if the source of the party reference was the ProtocolNegotiator or the ProtocolHandler, the SystemCo-

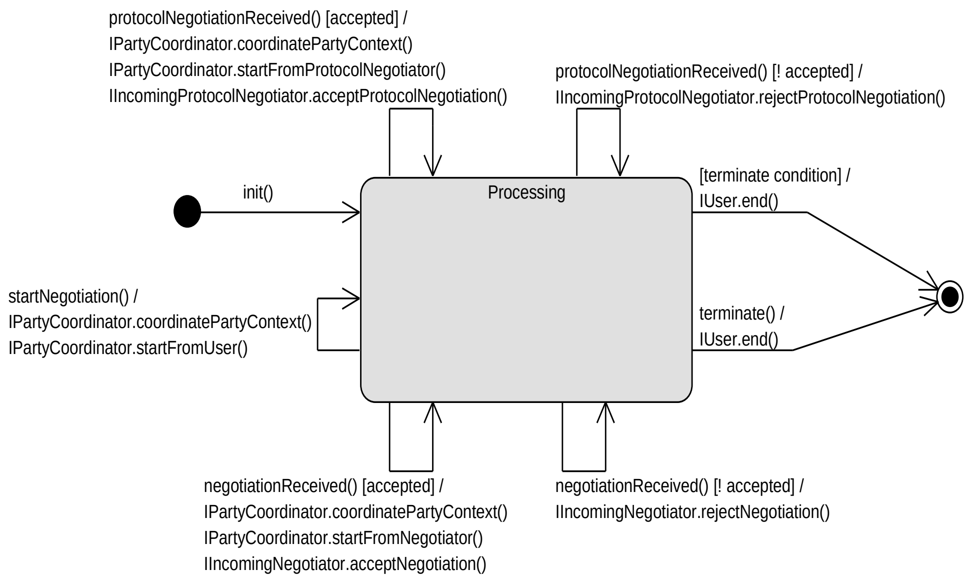

protocolNegotiationReceived() [accepted] /
IPartyCoordinator.coordinatePartyContext()
IPartyCoordinator.startFromProtocolNegotiator()
IIncomingProtocolNegotiator.acceptProtocolNegotiation()
protocolNegotiationReceived() [! accepted] /
IIncomingProtocolNegotiator.rejectProtocolNegotiation()
init()
Processing
[terminate condition] /
IUser.end()
startNegotiation() /
IPartyCoordinator.coordinatePartyContext()
IPartyCoordinator.startFromUser()
terminate() /
IUser.end()
negotiationReceived() [accepted] /
IPartyCoordinator.coordinatePartyContext()
IPartyCoordinator.startFromNegotiator()
IIncomingNegotiator.acceptNegotiation()
negotiationReceived() [! accepted] /
IIncomingNegotiator.rejectNegotiation()

Figure 9.22: *State machine of the SystemCoordinator.*

ordinator notifies them that the party reference was accepted by invoking method `acceptProtocolNegotiation` or method `acceptNegotiation`, respectively.

PartyCoordinator The PartyCoordinator coordinates all party contexts and, hence, it manages the interaction with the Inquirer, the ProtocolNegotiator and the NegotiationCoordinator. Figure §9.23 depicts the state machine implemented by the PartyCoordinator for each party context. Therefore, at run time, the PartyCoordinator contains the parallel execution of many instances of the state machines in Figure §9.23.

The state machine starts when the SystemCoordinator invokes method `coordinatePartyContext`. Then, the state machine enters state *pending*. From that state, the SystemCoordinator may receive message `startFromUser`, in which case it sends message `queryInformation` to the Inquirer and enters state *getting information*; it may receive a `startFromProtocolNegotiator` message, in which case it enters state *negotiating protocol*; or it may receive a `startFromProtocolHandler` message, in which case it invokes method `negotiate` on the NegotiationCoordinator and enters state *negotiating*.

In state *getting information*, the PartyCoordinator is waiting for the Inquirer to get the information that is necessary to start the negotiation. Therefore,

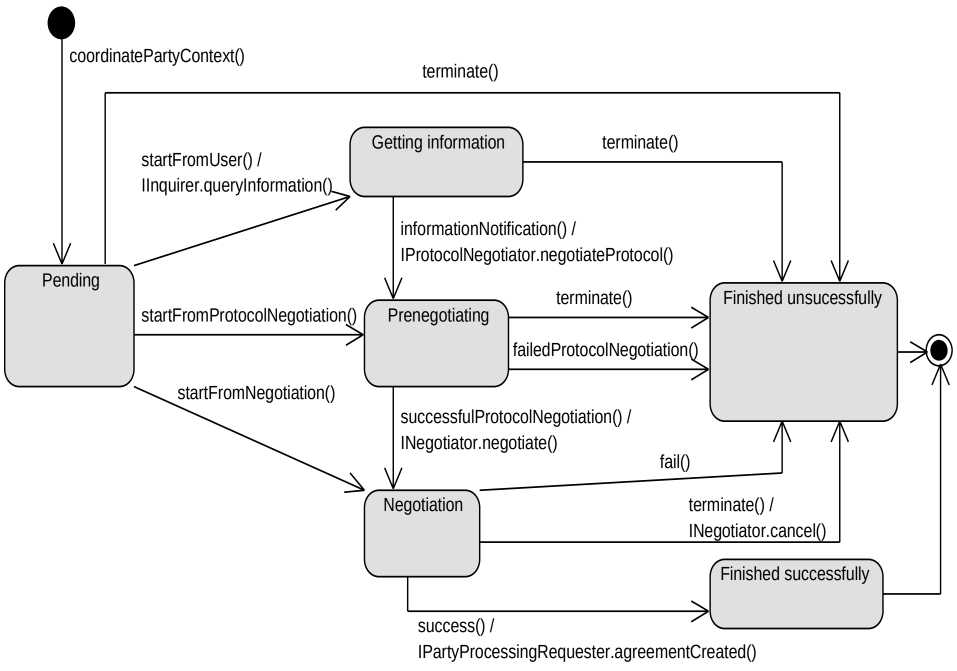


Figure 9.23: *State machine of the PartyCoordinator.*

when the Inquirer invokes method `informationNotification`, the PartyCoordinator invokes method `negotiateProtocol` of the ProtocolNegotiator and enters state *negotiating protocol*. Alternatively, the PartyCoordinator may decide to skip the protocol negotiation step. In that case, when it receives message `informationNotification`, it sends message `negotiate` to the NegotiationCoordinator and enters state *negotiating*.

In state *negotiating protocol*, the PartyCoordinator waits for the results of the protocol negotiation. If it is successful, it receives message `successfulProtocolNegotiation`, invokes method `negotiate` on the NegotiationCoordinator and enters state *negotiating*. If the protocol negotiation fails, it receives message `failedProtocolNegotiation` and enters state *end*.

Finally, in state *negotiating*, the PartyCoordinator is waiting for the results of the negotiation. If it is successful, it receives message `success` from the NegotiationCoordinator, it invokes method `agreementCreated` on interface `IUser` and enters state *end*. If the negotiation fails, it receives message `fail` and enters state *end*.

In addition, in all states, the PartyCoordinator may receive message `terminate`. In that case, the state machine enters state *end*. Additionally, if the party context was in state *negotiating*, it invokes method `cancel` on the NegotiationCoordinator before entering state *end*.

9.6 Summary

In this chapter, we have described the NegoFAST-Core framework based on the NegoFAST-Core reference architecture. We have defined the data model, the interfaces of roles and environmental resources and the state machines of the coordination roles described in the reference architecture.

Chapter 10

The NegoFAST-Bargaining framework

Here's the rule for bargains:
"Do other men, for they would do you."
That's the true business precept.

Charles Dickens, 1812–1870
British Novelist

The goal of this chapter is to detail the NegoFAST-Bargaining framework, which was introduced in Section §6.4.2. This framework extends the NegoFAST-Core framework to support bargaining protocols by materialising the NegoFAST-Bargaining reference architecture (cf. Chapter §8). This chapter is organised as follows: Section §10.1 introduces the main ideas and overviews the whole framework; Section §10.2 details the data model of the NegoFAST-Bargaining framework; Section §10.3 defines the interfaces of the environmental resources of the NegoFAST-Bargaining framework; Section §10.4 defines the interfaces and the protocol of the interactions amongst the roles of NegoFAST-Bargaining; Section §10.5 details the state machines that define the behaviour of the coordination roles; finally, Section §10.6 summarises the ideas of this chapter.

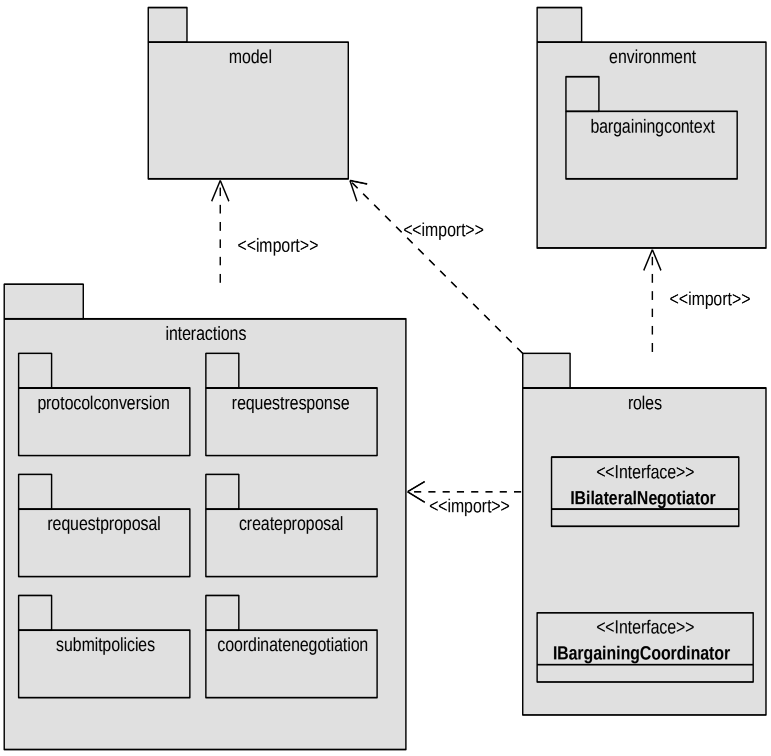


Figure 10.1: *NegoFAST-Bargaining framework packages.*

10.1 Introduction

In this chapter, we detail the NegoFAST-Bargaining framework, which extends the NegoFAST-Core framework to provide specific support for bargaining protocols (*cf.* Section §3.4). The NegoFAST-Bargaining framework takes the NegoFAST-Bargaining reference architecture (*cf.* Chapter §8) as a starting point and materialises it in a similar way as the NegoFAST-Core framework materialises the NegoFAST-Core reference architecture. The NegoFAST-Bargaining framework can be divided into the packages depicted in Figure §10.1, which includes a data model, interfaces of environmental resources, interactions and state machines for the roles that orchestrate the other roles.

In addition, as stated in Chapter §8, an abstract negotiation protocol must be defined. This abstract negotiation protocol allows to make the NegotiationCoordinator role independent from specific negotiation protocols and, hence, improve the reusability of the system. In NegoFAST-Bargaining, this protocol is an abstract proposal-based bilateral negotiation protocol, *cf.* Section §10.4.

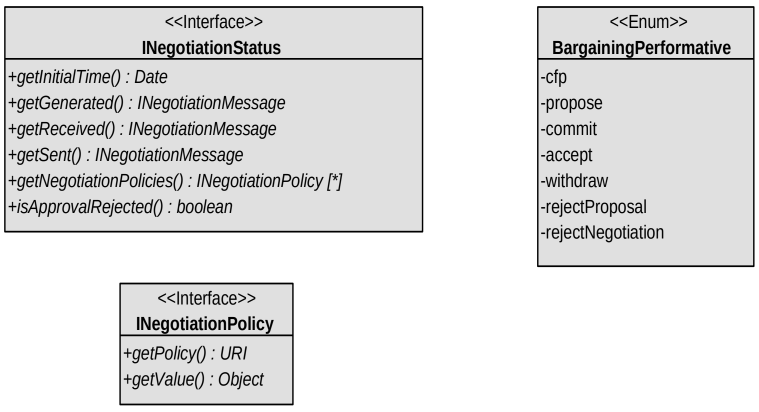


Figure 10.2: *Data model of NegoFAST-Bargaining.*

Finally, since negotiations are bilateral, there is a one-one relationship between party context and negotiation context. This allows the NegoFAST-Bargaining framework to use the same URI to identify a party context and a negotiation context.

10.2 Data model

This section extends the data model of the NegoFAST-Core framework (*cf.* Section §9.2) to represent the concepts that the NegoFAST-Bargaining framework introduces. These concepts involve aspects related to the protocol, the status of a bargaining negotiation, and the negotiation policies.

Bargaining performatives The performatives managed by the NegoFAST-Bargaining framework are those used in the abstract negotiation protocol. They are represented by enumeration `BargainingPerformative` (*cf.* Figure §10.2), which is a subtype of interface `Performative` defined in the NegoFAST-Core framework. The elements of the enumeration are `cfp`, `propose`, `commit`, `accept`, `withdraw`, `rejectProposal` and `rejectNegotiation`.

Negotiation policies The negotiation policies are modelled through interface `INegotiationPolicy`, which has the following methods, *cf.* Figure §10.2:

▷ `URI getPolicy()`: Returns a URI that identifies the negotiation policy.

▷ `Object getValue()`: Returns the value of the negotiation policy.

Status of a bargaining negotiation Interface `INegotiationStatus` represents the status of the negotiation in terms of the last negotiation messages that have been sent or received. This status constitutes the basic information used to build negotiation messages. Its methods are as follows, *cf.* Figure §10.2:

▷ `Date getInitialTime()`: Returns the time when the negotiation protocol instance started.

▷ `INegotiationMessage getGenerated()`: Returns the last negotiation message that has been generated. Note that this does not mean that this negotiation message was sent because it might have not been approved by the CommitHandler.

▷ `INegotiationMessage getReceived()`: Returns the last negotiation message that was received.

▷ `INegotiationMessage getSent()`: Returns the last negotiation message that was sent to the other party.

▷ `INegotiationPolicy[*] getNegotiationPolicies()`: Returns the set of negotiation policies that have been set for this negotiation context.

▷ `boolean isApprovalRejected()`: Returns whether the CommitHandler did not approve the last negotiation message that was generated for this negotiation or not. This information can be used to avoid entering in a loop in which the same binding negotiation message is generated and not approved by the CommitHandler.

10.3 Environmental resources

The NegoFAST-Bargaining reference architecture defines environmental resource *BargainingContextData*, which extends the *NegotiationContextData* of the NegoFAST-Core reference architecture to store bargaining-specific information. Specifically, environmental resource *BargainingContextData* must implement interface `IBargainingContextData`, which extends interface `INegotiationContextData` (*cf.* Section §9.3) with the following methods (*cf.* Figure §10.3):

▷ `void getBargainingState(context)`: Returns the bargaining-specific state of the negotiation context.

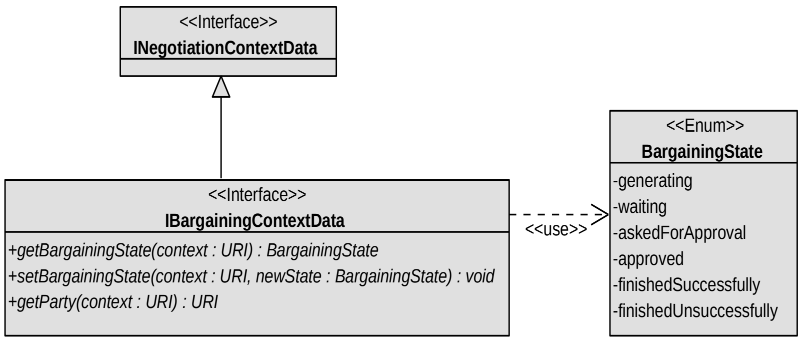


Figure 10.3: *Interface of* BargainingContextData.

▷ `void setBargainingState(context, newState)`: Sets the bargaining-specific state of the negotiation context as `newState`.

▷ `URI getParty(context)`: Returns the identifier of the party with which the automated negotiation system is negotiating.

Enumeration `BargainingState` (*cf.* Figure §10.3) models the bargaining-specific states of a negotiation context as detailed in Section §8.3.1. The states are the following:

▷ `waiting`: The negotiation context is waiting a negotiation message from the other party.

▷ `generating`: The negotiation context is generating a negotiation message to be sent to the other party.

▷ `askedForApproval`: The negotiation context is waiting for the approval of the CommitHandler to submit a binding negotiation message.

▷ `accepted`: The negotiation context has the approval to send the binding negotiation message, and it is waiting for the response of the other party.

▷ `finishedSuccessfully`: The negotiation context has finished its processing successfully (*i.e.* an agreement has been reached).

▷ `finishedUnsuccessfully`: The negotiation context has finished its processing unsuccessfully.

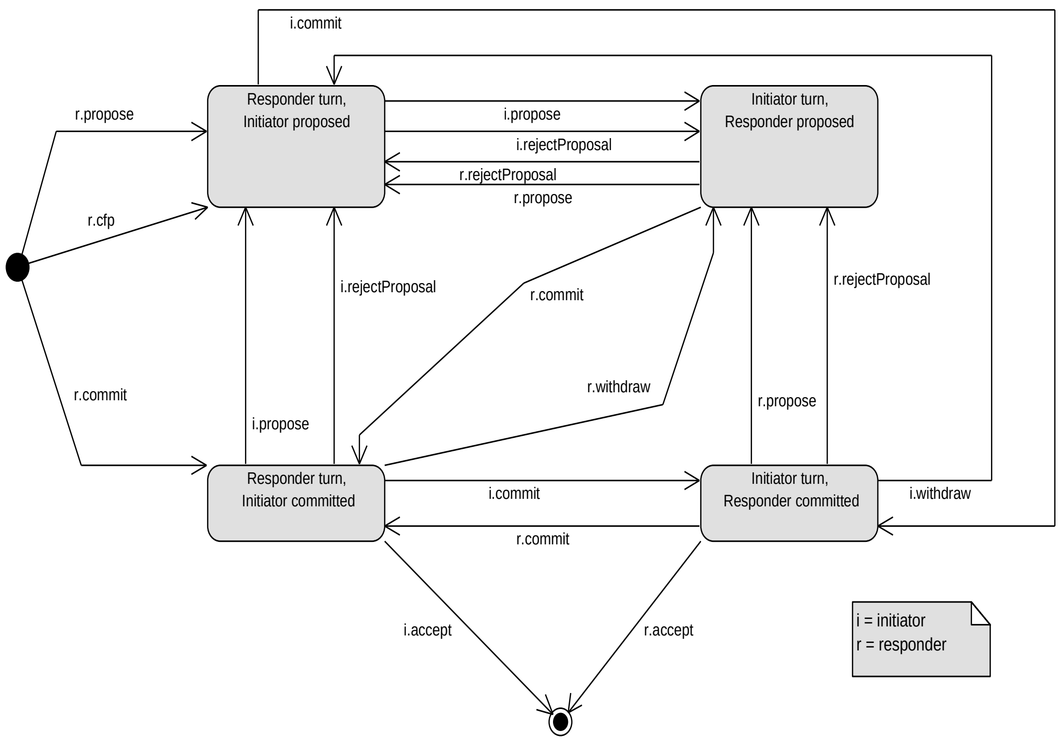


Figure 10.4: *State machine of the abstract bilateral negotiation protocol.*

10.4 Interactions

In this section, we describe the protocol and interfaces that must provide the roles involved in each interaction of the NegoFAST-Bargaining framework. However, before detailing those aspects of the NegoFAST-Bargaining framework, we must detail which is the abstract proposal-based bilateral negotiation protocol that the NegoFAST-Bargaining framework supports. In Section §3.4.1, we characterise negotiation protocols by means of five strongly-related aspects: parties, performatives, rules, information exchanged, and agreement terms negotiability. Consequently, our abstract protocol must define some of these aspects whereas others are left open to be defined by each concrete negotiation protocol. In particular, these aspects are the following:

Roles: The negotiation is carried out between two parties: the initiator, which is the party that initiates the negotiation, and the responder, which is the other party. Note that both consumer and provider can act as initiators or responders, and that we do not preclude the ability of a party to perform several simultaneous negotiations.

Performatives: The performatives allowed in our abstract protocol are: `cfp`, `reject-negotiation`, `reject-proposal`, `propose`, `commit`, `withdraw` and `accept` (*cf.* Table §3.1).

Rules: The rules imposed by the abstract protocol are as follows: it must start with a negotiation message that includes a `cfp`, a `propose` or a `commit` performative; the `cfp` performative can only be used in an initial negotiation message; only committing proposals (*i.e.* `commit` performatives) can be accepted; it is an alternating protocol, *i.e.* after one party sends a message, the other party must respond and so on. The only exception are negotiation messages that include `reject-negotiation` or `withdraw` performatives that can be sent at any time during the negotiation. These rules are formalised as the protocol state-machine in Figure §10.4.

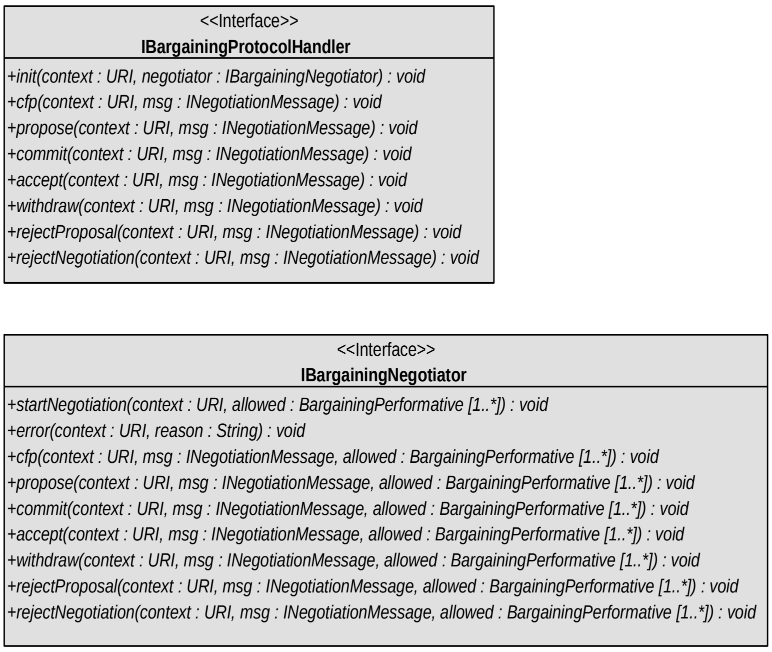
Information exchanged: The type of information exchanged is restricted to proposals and, hence, additional arguments cannot be sent together with a proposal as in argumentation-based protocols.

Agreement terms negotiability: Again, the abstract protocol does not impose any restriction on the negotiability of agreement terms. Therefore, as is the case of most bargaining protocols, any term is negotiable.

We strongly believe that this abstract protocol supports all of the bargaining protocols described in Section §3.4.2 except for the argumentation-based ones because they require additional negotiation performatives and the exchange of arguments together with the proposals during the negotiation.

ProtocolConversion This interaction is the implementation of the abstract proposal-based bilateral negotiation protocol described above, together with some control messages. To this end, both the BargainingProtocol-Handler and the BilateralNegotiator have one method in their interfaces (`IBargainingProtocolHandler` and `IBargainingNegotiator`, respectively) for each negotiation performative of the abstract proposal-based bilateral negotiation protocol, namely: `accept`, `rejectProposal`, `rejectNegotiation`, `propose`, `commit`, `cfp` and `withdraw` (*cf.* Figure §10.5). Furthermore, in interface `IBargainingNegotiator`, these methods include a set of bargaining performatives together with the negotiation message. The reason is that the set of negotiation performatives that can be used as a response changes depending on the negotiation protocol (*e.g.* a negotiation protocol may not allow non-binding proposals).

Together with these methods, interface `IBargainingProtocolHandler` includes the following one:



```
<<Interface>>
IBargainingProtocolHandler
+init(context : URI, negotiator : IBargainingNegotiator) : void
+cfp(context : URI, msg : INegotiationMessage) : void
+propose(context : URI, msg : INegotiationMessage) : void
+commit(context : URI, msg : INegotiationMessage) : void
+accept(context : URI, msg : INegotiationMessage) : void
+withdraw(context : URI, msg : INegotiationMessage) : void
+rejectProposal(context : URI, msg : INegotiationMessage) : void
+rejectNegotiation(context : URI, msg : INegotiationMessage) : void

<<Interface>>
IBargainingNegotiator
+startNegotiation(context : URI, allowed : BargainingPerformative [1..*]) : void
+error(context : URI, reason : String) : void
+cfp(context : URI, msg : INegotiationMessage, allowed : BargainingPerformative [1..*]) : void
+propose(context : URI, msg : INegotiationMessage, allowed : BargainingPerformative [1..*]) : void
+commit(context : URI, msg : INegotiationMessage, allowed : BargainingPerformative [1..*]) : void
+accept(context : URI, msg : INegotiationMessage, allowed : BargainingPerformative [1..*]) : void
+withdraw(context : URI, msg : INegotiationMessage, allowed : BargainingPerformative [1..*]) : void
+rejectProposal(context : URI, msg : INegotiationMessage, allowed : BargainingPerformative [1..*]) : void
+rejectNegotiation(context : URI, msg : INegotiationMessage, allowed : BargainingPerformative [1..*]) : void
```

Figure 10.5: *Interaction* ProtocolConversion.

- ▷ `void init(context, negotiator)`: Through this method, the BilateralNegotiator notifies the BargainingProtocolHandler that it is ready to start the negotiation and includes a reference to it.

Interface `IBargainingNegotiator` also includes additional methods:

- ▷ `void startNegotiation(context, allowed)`: By means of this method, the BargainingProtocolHandler notifies the BilateralNegotiator that it must start the negotiation by sending a negotiation message with one of the allowed performatives.

- ▷ `void error(context, reason)`: Notifies that an error has happened during the negotiation (*e.g.* a time-out or an unexpected message) and, hence, the negotiation must end.

Before starting the negotiation protocol instance, a previous exchange of messages must be carried out to setup the interaction. This is necessary because the BargainingProtocolHandler does not know the BilateralNegotiator*a priori* and because the BilateralNegotiator does not know whether it is playing role *initiator* or role *responder* in the negotiation. Therefore, the BilateralNegotiator first invokes an initialisation method (`init`) on the BargainingProtocolHandler with the negotiation context URI to initialise the interaction. This method lets the BargainingProtocolHandler know who the BilateralNegotiator is. Then, the BargainingProtocolHandler responds with either a call to methods `cfp` or `propose` or `commit`, or a call to method `startNegotiation` together with the set of allowed negotiation performatives.

Since this moment, each negotiation message coming from the other party is translated into a call to the corresponding method of interface `IBargainingNegotiator`. Together with the negotiation message coming from the other party, the BargainingProtocolHandler sends the set of allowed negotiation performatives that can be used to respond to that message.

Similarly, each negotiation message to be sent to the other party involves a call to the corresponding methods of interface `IBargainingProtocolHandler`. Then, the BargainingProtocolHandler translates the negotiation message into the concrete syntax of a negotiation protocol and sends it to the other party. These method calls follow the state machine depicted in Figure §10.4.

In addition, at any moment during the interaction, the BargainingProtocolHandler may invoke method `error` of the BilateralNegotiator together with a message describing the reason for the error (*e.g.* time out, invalid message received or communication finished). This method involves the unsuccessful finalisation of the negotiation.

CoordinateNegotiation This interaction represents the communication between the BargainingCoordinator and the BilateralNegotiator. The goal of this interaction is to allow the BilateralNegotiator to send commit or accept approval requests (*i.e.* to ask permission to send a binding negotiation message) to the BargainingCoordinator that are later redirected to the CommitHandler. Furthermore, it also involves the sending of initialisation and finalisation messages to and from the BilateralNegotiator.

Interface `IBargainingCoordinator`, which is implemented by the BargainingCoordinator, is as follows:

- ▷ `void commitApprovalRequest(context, proposal)`: Requests approval to send a negotiation message composed of the commit performative and the given proposal.
- ▷ `void acceptApprovalRequest(context, proposal)`: Requests approval to send a negotiation message composed of the accept performative and the given proposal.
- ▷ `void commitRejected(context)`: Notifies that the commit that has been sent was rejected by the other party.

▷ `void finishedUnsuccessfully(context)`: Notifies that the given negotiation context has finished unsuccessfully.

▷ `void finishedSuccessfully(context, agreement)`: Notifies that the given negotiation context has finished successfully and sends the resulting agreement.

The BilateralNegotiator implements interface `ICoordinableNegotiator`, which is composed of the following methods:

▷ `void init(context, coordinator, handler)`: Initialises the BilateralNegotiator with a reference to the coordinator and another reference to the BargainingProtocolHandler that manages the interaction with the other party. After receiving this message, the BilateralNegotiator starts the negotiation.

▷ `void reject(context)`: Notifies the BilateralNegotiator that the approval request was rejected.

▷ `void accept(context)`: Notifies the BilateralNegotiator that the approval request was accepted.

▷ `void cancel(context)`: Cancels the negotiation.

Figure §10.6 depicts the state machine of the interaction for each negotiation context. It starts entering state *negotiating* with the invocation of method `init` on the BilateralNegotiator. In state *negotiating*, the BilateralNegotiator may invoke method `commitApprovalRequest` and the interaction moves to state *commit approval request* or it may invoke method `acceptApprovalRequest` and the interaction enters state *asking approval*. In state *asking approval request*, there are two choices: the BargainingCoordinator may send message `reject` and the interaction enters back state *negotiating* or it may send message `accept` and the interaction enters state *waiting result*. In this state, the BilateralNegotiator may send message `finishedSuccessfully` and the interaction finishes, or the BilateralNegotiator may send message `commitRejected` and the interaction enters back state *negotiating*. In addition, in states *negotiating* and *waiting result*, the BilateralNegotiator may send message `finishedUnsuccessfully`, which causes the end of the interaction.

Similarly, the BargainingCoordinator may cancel the negotiation by invoking method `cancel` on the BilateralNegotiator. If method `cancel` is invoked in state *negotiating* or *asking approval*, the interaction finishes. However, if the interaction is in state *waiting result*, it means that a binding negotiation message was sent to the other party. As a consequence, if method

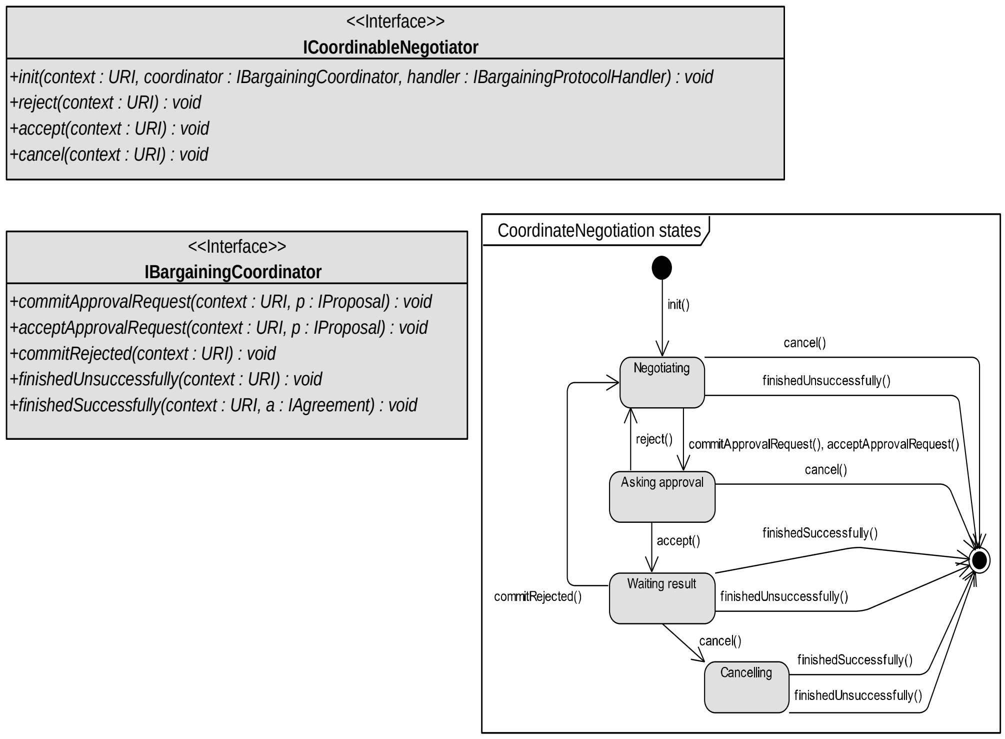

<<Interface>>
ICoordinableNegotiator
+init(context : URI, coordinator : IBargainingCoordinator, handler : IBargainingProtocolHandler) : void
+reject(context : URI) : void
+accept(context : URI) : void
+cancel(context : URI) : void
<<Interface>>
IBargainingCoordinator
+commitApprovalRequest(context : URI, p : IProposal) : void
+acceptApprovalRequest(context : URI, p : IProposal) : void
+commitRejected(context : URI) : void
+finishedUnsuccessfully(context : URI) : void
+finishedSuccessfully(context : URI, a : IAgreement) : void
CoordinateNegotiation states
init()
Negotiating
cancel()
finishedUnsuccessfully()
reject()
commitApprovalRequest(), acceptApprovalRequest()
Asking approval
cancel()
accept()
finishedSuccessfully()
commitRejected()
Waiting result
finishedUnsuccessfully()
cancel()
Cancelling
finishedSuccessfully()
finishedUnsuccessfully()

Figure 10.6: *Interaction* CoordinateNegotiation.

`cancel` is invoked in that state, the interaction must not finish immediately. Instead, the interaction enters state `cancelling`, in which it waits until the BilateralNegotiator invokes either method `finishedSuccessfully` or `finishedUnsuccessfully`.

SubmitPolicies This interaction implements the submission of negotiation policies from the PoliciesManager to the BilateralNegotiator. The PoliciesManager implements interface `IPoliciesManager`, whereas the BilateralNegotiator implements interface `IPolicyReceiver`.

Interface `INegotiationPoliciesGenerator` has only one method:

▷ `INegotiationPolicy[*] initNegotiation(ctx, policyReceiver)`: Its goal is twofold. On the one hand, it allows the BilateralNegotiator to notify the PoliciesManager that there is a new bilateral negotiation that must be provided with negotiation policies. On the other hand, it returns the initial set of negotiation policies that shall guide the new bilateral negotiation.

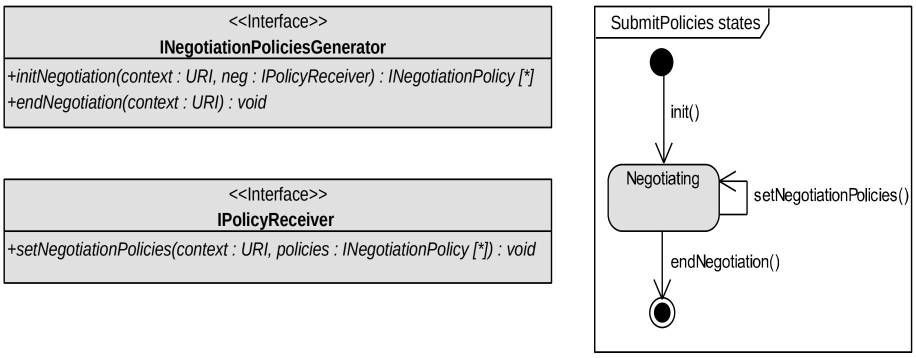

<<Interface>>
INegotiationPoliciesGenerator
+initNegotiation(context : URI, neg : IPolicyReceiver) : INegotiationPolicy [*]
+endNegotiation(context : URI) : void
<<Interface>>
IPolicyReceiver
+setNegotiationPolicies(context : URI, policies : INegotiationPolicy [*]) : void
SubmitPolicies states
init()
Negotiating
setNegotiationPolicies()
endNegotiation()

Figure 10.7: *Interaction* SubmitPolicies.

▷ `void endNegotiation(context)`: Notifies the PoliciesManager that the given negotiation context has finished and, hence, that it is not necessary to provide new negotiation policies.

Interface `IPolicyReceiver` has also only one method:

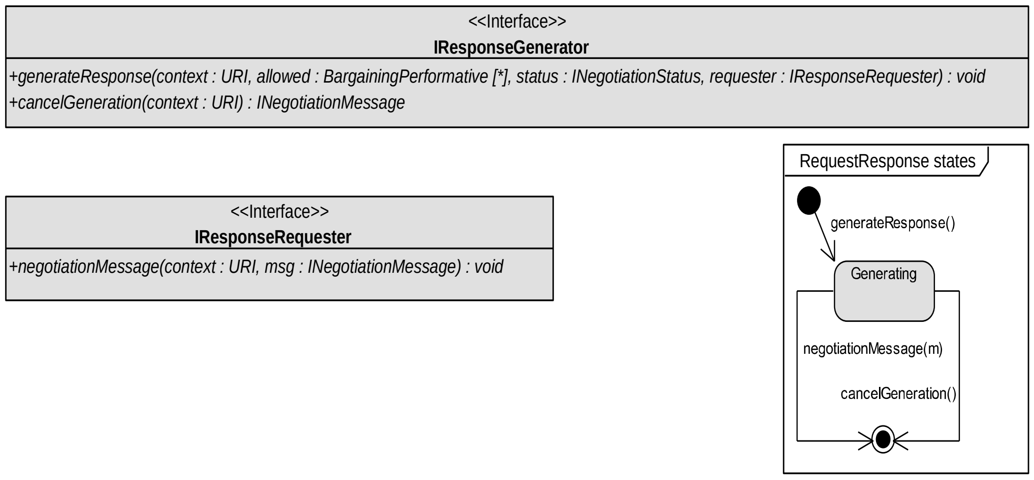
▷ `void setNegotiationPolicies(context, policies)`: Sets the negotiation policies of the given negotiation context.

The interaction protocol is simple. When a new bilateral negotiation starts, the BilateralNegotiator invokes method `initNegotiation` on the PoliciesManager to notify it and to receive the initial set of negotiation policies. Then, when the PoliciesManager finds it convenient, it submits a new set of negotiation policies by invoking method `setNegotiationPolicies`. Finally, when the negotiation context finishes, the BilateralNegotiator invokes method `endNegotiation` to notify that no new negotiation policies are needed.

RequestResponse The goal of this interaction is to obtain a negotiation message that shall be sent as a response to the other negotiating party. The interaction is asynchronous and has two participants: the BilateralNegotiator, which requests the negotiation message and implements interface `IResponseRequester`, and the PerformativeSelector, which returns the generated negotiation message and implements interface `IResponseGenerator`.

Interface `IResponseGenerator` has the following methods:

▷ `void generateResponse(context, allowed, status, requester)`: Requests the generation of a new negotiation message for the given negotiation context. It also provides the bargaining performatives that can be



<<Interface>>
IResponseGenerator
+generateResponse(context : URI, allowed : BargainingPerformative [*], status : INegotiationStatus, requester : IResponseRequester) : void
+cancelGeneration(context : URI) : INegotiationMessage
<<Interface>>
IResponseRequester
+negotiationMessage(context : URI, msg : INegotiationMessage) : void
RequestResponse states
generateResponse()
Generating
negotiationMessage(m)
cancelGeneration()

Figure 10.8: *Interaction* RequestResponse.

used in the negotiation message, the current status of the negotiation and a reference to the requester that shall receive the generated negotiation message.

▷ `INegotiationMessage cancelGeneration(context)`: Cancels the generation of the message and returns either a valid negotiation message or a null value if no valid negotiation message could be generated.

Interface `IResponseRequester` has only one method:

▷ `void negotiationMessage(context, msg)`: Receives the generated negotiation message from the PerformativeSelector.

The interaction takes place as follows: when the BilateralNegotiator needs a negotiation message as response, it invokes method `generateResponse` on the PerformativeSelector. Then, the PerformativeSelector starts creating the negotiation message and, when it finishes, it invokes method `negotiationMessage` on the BilateralNegotiator with the generated negotiation message.

In addition, at any moment, the BilateralNegotiator may cancel the generation by invoking method `cancelGeneration`. In that case, the PerformativeSelector must respond synchronously with a valid negotiation message or a null value if no valid negotiation message could be generated.

RequestProposal The goal of this interaction is to obtain a proposal that shall be sent as part of a negotiation message to the other negotiating party.

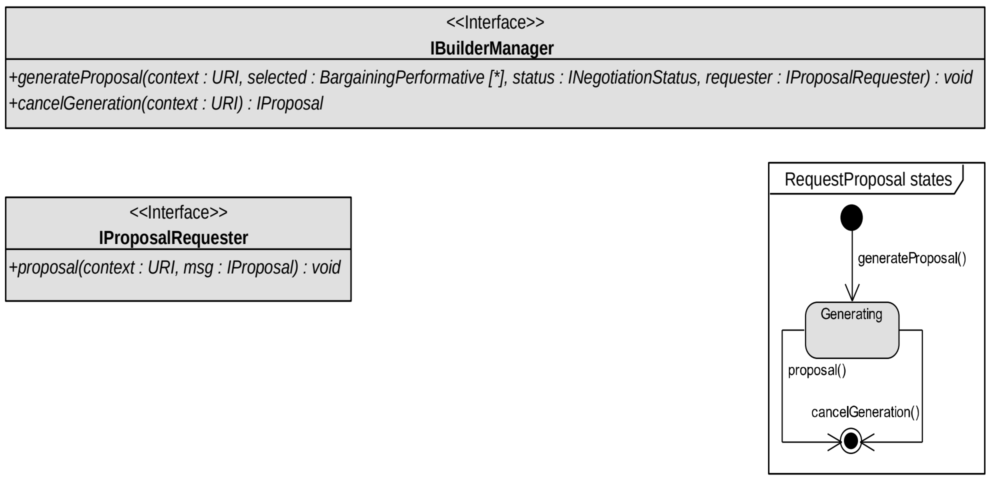

<<Interface>>
IBuilderManager
+generateProposal(context : URI, selected : BargainingPerformative [*], status : INegotiationStatus, requester : IProposalRequester) : void
+cancelGeneration(context : URI) : IProposal
<<Interface>>
IProposalRequester
+proposal(context : URI, msg : IProposal) : void
RequestProposal states
generateProposal()
Generating
proposal()
cancelGeneration()

Figure 10.9: *Interaction* RequestProposal.

This interaction is much the same as the previous one, the difference being that the goal of the former is to obtain a whole negotiation message (*i.e.* a performative and a proposal in this case), whereas the goal of the latter is just to obtain the proposal. The interaction has two participants: the BuilderManager, which provides the proposal and implements interface `IBuilderManager`, and the PerformativeSelector, which requests the proposal and implements interface `IProposalRequester`.

Interface `IBuilderManager` has the following methods:

▷ `void generateProposal(ctx, performat, status, requester)`: Requests the generation of a new proposal for the given negotiation context. It also provides the bargaining performatives that may be sent together with this proposal (`selected`), the current status of the negotiation and a reference to the requester to send the proposal back.

▷ `IProposal cancelGeneration(context)`: Cancels the generation of the proposal and returns either a valid proposal or a null value if it could not be generated.

Interface `IProposalRequester` has just one method:

▷ `void proposal(context, proposal)`: Receives the proposal from the BuilderManager.

The interaction protocol is exactly the same as the previous one.

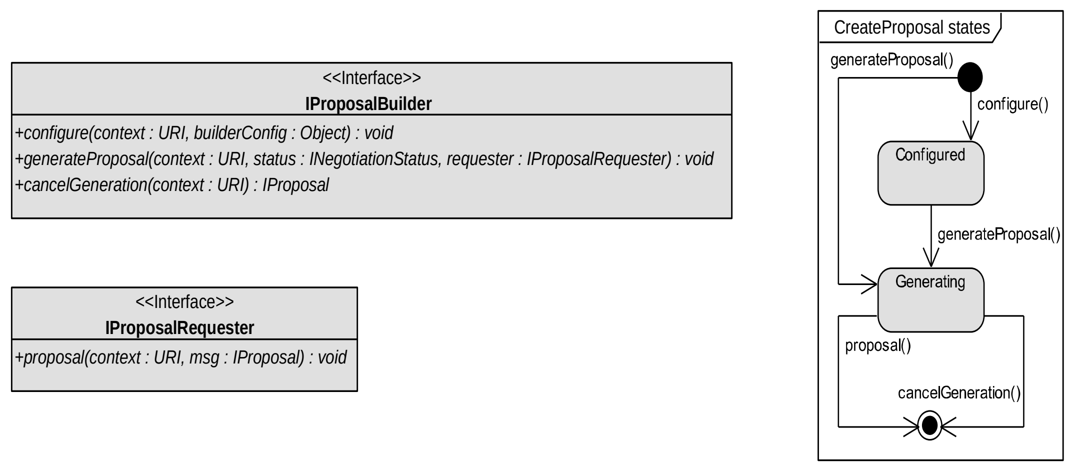

<<Interface>>
IProposalBuilder
+configure(context : URI, builderConfig : Object) : void
+generateProposal(context : URI, status : INegotiationStatus, requester : IProposalRequester) : void
+cancelGeneration(context : URI) : IProposal
<<Interface>>
IProposalRequester
+proposal(context : URI, msg : IProposal) : void
CreateProposal states
generateProposal()
configure()
Configured
generateProposal()
Generating
proposal()
cancelGeneration()

Figure 10.10: *Interaction* CreateProposal.

CreateProposal The goal of this interaction is to obtain a proposal that shall be sent as part of a negotiation message to the other negotiating party. The difference is that this interaction allows a lower-level configuration prior to the request of the new proposal. In addition, the participants are also different. In this case, the ProposalBuilder is the creator of the proposal and implements interface `IProposalBuilder`, whereas the BuilderManager is now the requester of the proposal (*cf.* Section §8.4.2) and, hence, it implements interface `IProposalRequester`.

Interface `IProposalBuilder` has three methods:

▷ `void configure(context, builderConfig)`: Configures the ProposalBuilder for the given negotiation context. The configuration is specific to each ProposalBuilder, and it is mainly intended to convert negotiation policies into builder-specific configuration parameters.

▷ `void generateProposal(context, status, requester)`: Requests the generation of a new proposal for the given negotiation context. It also provides the current status of the negotiation and a reference to the requester to send the proposal back.

▷ `IProposal cancelGeneration(context)`: Cancels the generation of the proposal and returns either a valid proposal or a null value if it could not be generated.

Interface `IProposalRequester` was described in the previous interaction.

The interaction protocol is similar to the previous one, the difference being that before invoking method `generateProposal`, the BuilderManager may configure the ProposalBuilder by invoking method `configure`. However, this step is optional; if the BuilderManager does not configure the ProposalBuilder, it shall take its default configuration.

10.5 State machines

In this section, we report on the state machines of the BilateralNegotiator and the BargainingCoordinator roles, which coordinate the others.

BilateralNegotiator The goal of the BilateralNegotiator is to carry out a single bilateral negotiation by orchestrating the BargainingProtocolHandler and the response generation roles. Furthermore, it must communicate with the BargainingCoordinator to coordinate with the other simultaneous bilateral negotiations and to ask for approval before committing to a proposal, and with the PoliciesManager to receive negotiation policies that shall guide the generation of responses.

Figure §10.11 depicts the state machine of the BilateralNegotiator. It consists of the following states: four states (*waiting initial*, *waiting*, *waiting accept* and *cancelling*) in which the BilateralNegotiator is waiting for a message from the other party, one state (*generating response*) in which the BilateralNegotiator is waiting for the PerformativeSelector to generate a response, and two states (*approving commit* and *approving accept*) in which the BilateralNegotiator is waiting for the CommitHandler to decide on whether to send a binding negotiation message or not.

The state machine starts when the BargainingCoordinator initiates the BilateralNegotiator by invoking method `init` with the URI of the negotiation context and a reference to the BargainingProtocolHandler that manages the communication with the other party. Then, the BilateralNegotiator invokes method `init` of the BargainingProtocolHandler to initialise the interaction and waits for its response in state *waiting initial*. The response can be `startNegotiation` or a negotiation performative. In any case, when the response is received the BilateralNegotiator invokes method `generateResponse` on the PerformativeSelector and waits for the response in state *generating response*. In addition, in state *waiting initial*, the BilateralNegotiator may also receive message `cancel`. In this case, it enters state *cancelling*.

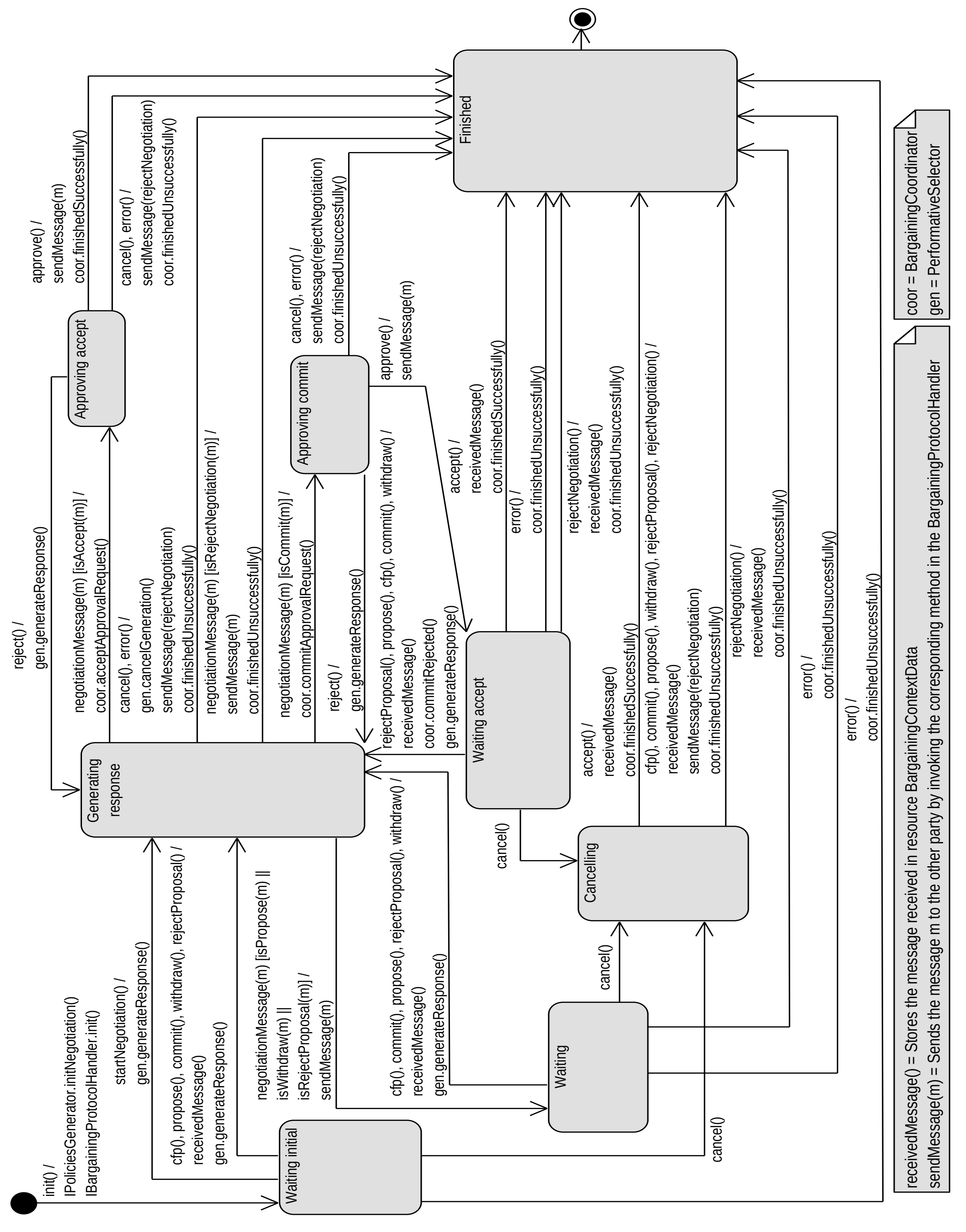

Finished
Approving accept
Approving commit
Generating response
Waiting accept
Cancelling
Waiting
Waiting initial
init() / IPoliciesGenerator.initNegotiation() IBargainingProtocolHandler.init()
startNegotiation() / gen.generateResponse()
cfp(), propose(), commit(), withdraw(), rejectProposal() / receivedMessage() gen.generateResponse()
negotiationMessage(m) [isPropose(m) || isWithdraw(m) || isRejectProposal(m)] / sendMessage(m)
cfp(), commit(), propose(), rejectProposal(), withdraw() / receivedMessage() gen.generateResponse()
reject() / gen.generateResponse()
negotiationMessage(m) [isAccept(m)] / coor.acceptApprovalRequest()
cancel(), error() / gen.cancelGeneration() sendMessage(rejectNegotiation) coor.finishedUnsuccessfully()
negotiationMessage(m) [isRejectNegotiation(m)] / sendMessage(m) coor.finishedUnsuccessfully()
negotiationMessage(m) [isCommit(m)] / coor.commitApprovalRequest()
reject() / gen.generateResponse()
rejectProposal(), propose(), cfp(), commit(), withdraw() / receivedMessage() coor.commitRejected() gen.generateResponse()
approve() / sendMessage(m) coor.finishedSuccessfully()
cancel(), error() / sendMessage(rejectNegotiation) coor.finishedUnsuccessfully()
cancel(), error() / sendMessage(rejectNegotiation) coor.finishedUnsuccessfully()
approve() / sendMessage(m)
accept() / receivedMessage() coor.finishedSuccessfully()
error() / coor.finishedUnsuccessfully()
rejectNegotiation() / receivedMessage() coor.finishedUnsuccessfully()
accept() / receivedMessage() coor.finishedSuccessfully()
cfp(), commit(), propose(), withdraw(), rejectProposal(), rejectNegotiation() / receivedMessage() sendMessage(rejectNegotiation) coor.finishedUnsuccessfully()
rejectNegotiation() / receivedMessage() coor.finishedUnsuccessfully()
error() / coor.finishedUnsuccessfully()
error() / coor.finishedUnsuccessfully()
cancel()
cancel()
cancel()
coor = BargainingCoordinator
gen = PerformativeSelector
receivedMessage() = Stores the message received in resource BargainingContextData
sendMessage(m) = Sends the message m to the other party by invoking the corresponding method in the BargainingProtocolHandler

Figure 10.11: *State machine of the BilateralNegotiator.*

The BilateralNegotiator leaves state *generating response* when either the BargainingProtocolHandler invokes method `error`, the BargainingCoordinator invokes method `cancel` or the PerformativeSelector invokes method `negotiationMessage`. In the first two cases, the BilateralNegotiator invokes method `cancelGeneration` in the PerformativeSelector and method `finishedUnsuccessfully` in the BargainingCoordinator and enters state *finished*. In the third case, the transition of the BilateralNegotiator depends on the performative of the generated negotiation message: if the performative is `accept`, it invokes method `acceptApprovalRequest` in the BargainingCoordinator and enters state *approving accept*; if the performative is `rejectNegotiation`, it invokes method `rejectNegotiation` in the BargainingProtocolHandler and method `finishedUnsuccessfully` in the BargainingCoordinator and moves to state *finished*; if the performative is `commit`, it invokes method `commitApprovalRequest` in the BargainingCoordinator and enters state *approving commit*; and if the performative is either `propose`, `withdraw` or `rejectProposal`, it invokes the corresponding method in the BargainingProtocolHandler and enters state *waiting*.

In state *approving accept*, the BilateralNegotiator waits for an `approve` or `reject` message. If the message is `approve`, the BilateralNegotiator invokes method `accept` in the BargainingProtocolHandler and method `finishedSuccessfully` in the BargainingCoordinator and enters state *finished*. If the message is `reject`, the BilateralNegotiator invokes again method `generateResponse` in the PerformativeSelector and moves back to state *generating response* to wait for another response. In addition, the BilateralNegotiator may also receive message `cancel` or message `error`. If this is the case, it invokes method `rejectNegotiation` in the BargainingProtocolHandler and method `finishedUnsuccessfully` in the BargainingCoordinator and enters state *finished*.

Similarly, in state *approving commit*, the BilateralNegotiator waits for the BargainingCoordinator to respond with an `approve` or `reject` message. If the response is `approve`, the BilateralNegotiator invokes method `commit` in the BargainingProtocolHandler and enters state *waiting accept*. If the response is `reject`, the BilateralNegotiator behaves as in state *approving accept*: invokes method `generateResponse` and enters state *generating response*. In addition, the BilateralNegotiator may also receive message `cancel` or message `error`, in which case the BilateralNegotiator behaves as in state *approving accept* as well.

In states *waiting* and *waiting accept*, the BilateralNegotiator waits for a new negotiation message. The difference being that in state *waiting accept*, the BilateralNegotiator has sent a binding negotiation message and it is waiting for its acceptance whereas in state *waiting*, it has not. In both states, if the negotiation message is `rejectNegotiation` or `error`, the BilateralNegotiator invokes method `finishedUnsuccessfully` in the BargainingCoordinator and enters state *finished*. If the negotiation message is either `cfp`, `commit`, `propose`, `rejectProposal` or `withdraw`, the BilateralNegotiator invokes method `generateResponse` in the PerformativeSelector and enters state *generating response*. In this case, if the BilateralNegotiator is in state *waiting accept*, it also invokes method `commitRejected` in the BargainingCoordinator. In state *waiting accept*, the negotiation message may also be `accept`, in which case it invokes method `finishedSuccessfully` in the BargainingCoordinator and enters state *finished*. Besides a negotiation message, the BilateralNegotiator may also receive message `cancel`. In this case, it enters state *cancelling*.

In state *cancelling*, the negotiation has been cancelled but by the BargainingCoordinator but the other party has not been notified (*i.e.* message `rejectNegotiation` has not been sent) because the BilateralNegotiator was waiting for a negotiation message from the other party. When this negotiation message arrives, if it is `accept`, the BilateralNegotiator invokes method `finishedSuccessfully` in the BargainingCoordinator and enters state *finished*. Otherwise, the BilateralNegotiator invokes method `rejectNegotiation` in the BargainingProtocolHandler and method `finishedUnsuccessfully` in the BargainingCoordinator and enters state *finished*.

In addition, at any moment, the PoliciesManager may send negotiation policies to the BilateralNegotiator (`setNegotiationPolicy`).

BargainingCoordinator The BargainingCoordinator is chiefly a message dispatcher amongst the BilateralNegotiator, the CommitHandler, the PartyCoordinator and the *BargainingContextData*. Its tasks include: initialising and finalising negotiations and coordinating the approval requests between the CommitHandler and the BilateralNegotiators.

The first task involves the following: whenever the PartyCoordinator invokes method `negotiate`, it creates a new negotiation context and invokes method `init` on a BilateralNegotiator. Similarly, when a BilateralNegotiator invokes either method `fail` or method `succeed`, it forwards them to the PartyCoordinator. Finally, if the PartyCoordinator invokes the `cancel` method, it forwards it to the corresponding BilateralNegotiator.

The second task involves coordinating the approval requests: the BargainingCoordinator receives commit approval requests and accept approval requests from the BilateralNegotiators. For each of them, it forwards them to the CommitHandler by invoking the `approvalRequest` method. Similarly, it forwards messages `accept` and `reject` from the CommitHandler to the corresponding BilateralNegotiators. Finally, it forwards the `commitRejected` or `success` message from the BilateralNegotiator to the CommitHandler.

10.6 Summary

In this chapter, we have described the NegoFAST-Bargaining framework, which refines the NegoFAST-Bargaining reference architecture to extend the NegoFAST-Core framework to support bargaining protocols. To this end, we have defined the data model, the interfaces of the roles and environmental resources and the state machines of the coordination roles described in the reference architecture.

Part IV

Final remarks

Chapter 11

Conclusions

I know not what I may appear to the world,
but to myself I have been only like a boy playing on the seashore,
diverting myself in now and then finding a smoother
pebble or a prettier shell than ordinary,
whilst the great ocean of truth lay all undiscovered before me.

Sir Isaac Newton, 1642–1727
English scientist and philosopher

Both SOA and BPM initiatives are enabling technologies that help companies bridge the gap between businesses and IT. In a service-oriented approach, agreements play a major role to regulate the functional, non-functional properties, and guarantees of a service [4, 83, 93]. Reaching such agreements usually require the provider and the consumer to negotiate on multiple terms. It is not surprising then that many researchers have focused on automating such negotiation [11, 22, 38, 44, 104, 122]. In particular, they have developed negotiation models, protocols and algorithms, usually implemented as software agents, that try to mimic some aspects of human behaviour, while removing an often-cited disadvantage [27], namely, the slowness of human negotiations.

More specifically, we believe that the automation of service agreement negotiations shall improve the service-provisioning process as a part of the business' supply-chain, mainly in open and dynamic markets in which the flexibility in business relationships, the openness to new providers and the best exploitation of company's resources are key aspects in the business management strategies.

The goal of this dissertation was to support the idea that it is convenient to develop a software framework that provides engineering support to make the development of automated negotiation systems easier in the context of negotiating service agreements in open and dynamic environments. In this dissertation, we presented a strong motivation for this idea; described the problems that appear when negotiating in such environments, and detail why current approaches to build automated negotiation systems are not appropriate for service negotiations in open and dynamic environments (*cf.* Chapter §2).

NegoFAST is our approach to building such automated negotiation systems. It is defined at three levels of abstraction: a reference architecture, a software framework and a proof-of-concept implementation. To enhance the reusability of NegoFAST, it has been divided into a protocol-independent part, NegoFAST-Core, and protocol-specific extensions. In this dissertation, we have developed a bargaining-specific extension, NegoFAST-Bargaining. However, other extensions can be implemented.

NegoFAST provides direct support to deal with the problems of service negotiations in open and dynamic environments described in Chapter §2, namely: multi-term negotiations, heterogeneous parties, partial information about parties and dynamic markets.

Multi-term negotiations are dealt with by providing generic data structures that support multi-term agreements and negotiation proposals and expressive preferences models (*cf.* Section §9.2).

NegoFAST tackles the heterogeneity of parties in open environments by promoting a clear separation of concerns thanks to the generic data model and to well-defined interfaces between the elements that conform the system (*cf.* Chapter §9 and Chapter §10). This allows an automated negotiation system to support easily multiple protocols and negotiation intelligence algorithms by implementing several ProtocolHandlers (*cf.* Section §7.3) and ResponseGenerators or CommitHandlers (*cf.* Section §7.4), respectively. In addition, NegoFAST gives support to allow the negotiability of protocols by means of the ProtocolNegotiator (*cf.* Section §7.3).

The problem of negotiating with partial information about parties is dealt with as follows: first, the generic data model makes it explicit the difference between different types of knowledge about parties (*cf.* Section §9.2); second, Inquirer, Informant (*cf.* Section §7.5), and estimators acting as proxies for external information providers (*cf.* Section §9.3) allow the automated negotiation system to get information from different sources; and third, the events mechanism implemented on several environmental resources (*cf.* Section §9.3) and the *NegotiationHistory* that enables offline estimation (*cf.* Section §7.5) makes it easier the development of estimators that build world models.

Finally, the problem of coping with dynamic markets is faced with the following mechanisms: first, the NegotiationCoordinator (BargainingCoordinator, PoliciesManager and BilateralNegotiator in NegoFAST-Bargaining) makes it possible to carry out several negotiations at the same time (*cf.* Section §7.2 and Section §8.3); second, the BuilderManager enables the dynamic selection of different intelligence algorithms, implemented in several ProposalBuilders, depending on the current state of the negotiations (*cf.* Section §8.4); third, the CommitHandler and its CommitAdvisors facilitate an assessed creation of agreements to avoid committing to more agreements than the system can afford (*cf.* Section §7.4); and fourth, the events mechanism and the *Negotiation-History* make it possible the development of models of characteristics of the market that may have an influence on the negotiation process.

The only aspect identified in Section §2.2 that is not fully supported by NegoFAST is the decommitment. Currently, the decommitment support in NegoFAST is kind of naive and it is just detailed at a high level of abstraction. The main reason is that decommitment is still a novel topic that deserves a specific research line on its own.

Anyhow, the results in this dissertation cannot be seen as the concluding end of a path, but as the motivation for further research on this topic. Amongst the many issues that remain open or can be improved, we think that the most exciting is to develop more protocol-specific extensions to NegoFAST-Core. The most interesting protocol-specific extensions are an auctioning extension and an argumentation-based extension. The auctioning extension implies extending the ProtocolHandler, NegotiationCoordinator and ResponseGenerator to the specific requirements of auctions. This involves deciding not only which proposal must be send to the other parties, like in bargaining protocols, but also, when this proposal shall be made. The argumentation-based extension is more challenging because it requires an extension of the data model to support the exchange of logical arguments together with proposals. Furthermore, a language to express these logical arguments should be developed.

In addition, another interesting topic that deserves further research is the study of more advanced ways of expressing preferences related to the negotiation process so that the user can give higher-level business rules that regulate the behaviour of the automated negotiation system. This research involves reviewing the interaction between the user and the automated negotiation system, the data model to express preferences, and an analysis of how preferences should affect the behaviour of the elements that compose the automated negotiation system.

Part V
Appendices

Appendix A

Use case: Submitting computing jobs

In the previous chapters, we detail the NegoFAST-Core and NegoFAST-Bargaining frameworks, which provide an infrastructure to ease the development of automated negotiation systems. The goal of this chapter is to exemplify the use of these frameworks to build an automated negotiation system in a computing job submission scenario. The chapter is organised as follows: in Section §A.1, we motivate and describe the scenario; Section §A.2, Section §A.3, Section §A.4 and Section §A.5 detail the steps that are necessary to build the automated negotiation system; finally, Section §A.6 summarises the ideas of this chapter.

A.1 Introduction

This use case focuses on the submission of a computing job to a job hosting service. This is a common scenario in grid environments in which several computational resources are used to execute the requests of other elements of the system. Furthermore, this scenario is also becoming increasingly popular in interorganisational scenarios, in which companies that have intensive computational needs outsources them to computing services companies.

In this context, service agreements must be created between job submitters and job hosting services that set the terms under which the job shall be executed. These terms may include details such as the nature of the process to be executed, the resources required for the execution or any scheduling requirements such as job start or job completion deadlines [4]. As a consequence, it is appealing for both job submitters and job hosting services to use automated negotiation systems in order to reach these agreements on their behalf.

Note that the negotiations in this scenario are multi-term because there are more than one term involved in the negotiation. In addition, chiefly in interorganisational scenarios but also in grid environments of large corporations, it is likely to find heterogeneous job hosting services that may implement a great variety of negotiation protocols and present very diverse behaviours during the negotiation. Finally, it is an intrinsically dynamic scenario because the execution conditions that a job hosting service can offer depend heavily on the current resource availability. Since the availability of resources is likely to change frequently, agreements should be created on demand. Furthermore, if agreements are created automatically, the usage of resources can be fine-tuned. This is specially relevant for computing services companies, which may use their idle time by selling it at a lower cost.

A variety of different use cases can be defined in this scenario. For instance, job submitters may initiate bilateral negotiations with job hosting services or job hosting services may auction available resources and job submitters bid for them. However, since the goal of this chapter is not to discuss the different choices to implement this scenario, but to exemplify the instantiation of the NegoFAST framework, we focus on a particular use case. This use case is depicted in Figure §A.1 and is as follows: first, the job submitter sends its agreement preferences to its automated negotiation system. These preferences may include both requirements about the job execution and guidelines regarding the negotiation process. Then, when the automated negotiation system receives references to job hosting services, it starts a bilateral negotiation with them. When an agreement is reached, the automated negotiation system notifies the job submitter and sends it the agreement. Finally, the job submitter

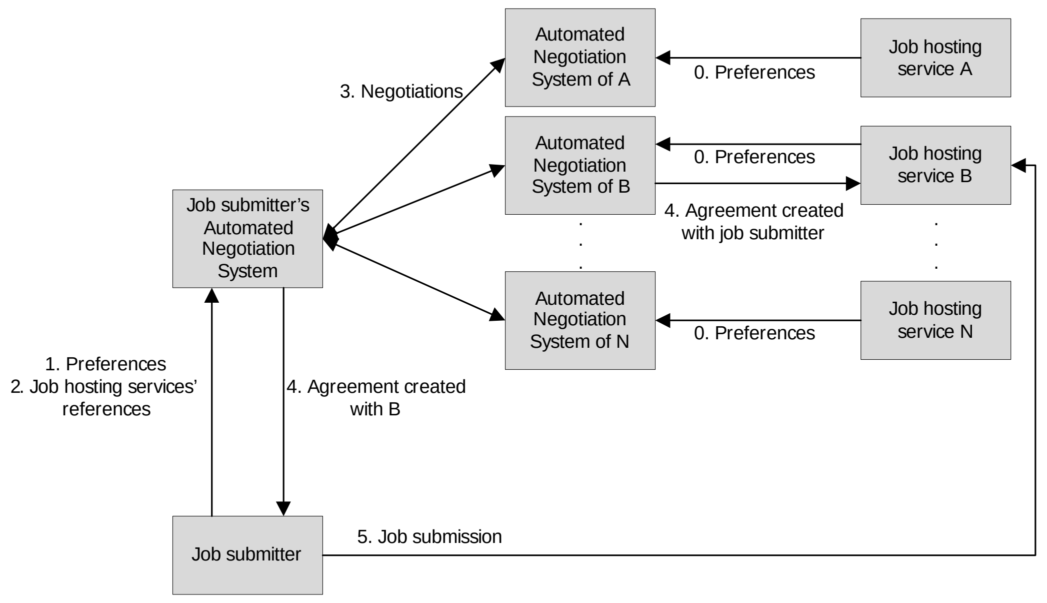

Automated Negotiation System of A
0. Preferences
Job hosting service A
3. Negotiations
Automated Negotiation System of B
0. Preferences
Job hosting service B
4. Agreement created with job submitter
Job submitter's Automated Negotiation System
Automated Negotiation System of N
0. Preferences
Job hosting service N
1. Preferences
2. Job hosting services' references
4. Agreement created with B
Job submitter
5. Job submission

Figure A.1: *Computing job submission scenario.*

sends the job to the job hosting service, and it shall be executed following the terms established in the agreement.

To build the automated negotiation system, we take August as a starting point and we extend it to fulfil the requirements of this use case. August is a proof-of-concept implementation of the NegoFAST framework (both NegoFAST-Core and NegoFAST-Bargaining) that provides an implementation of the coordination roles (SystemCoordinator, PartyCoordinator, BargainingCoordinator and BilateralNegotiator), the environmental resources and the generic data model. In addition, it provides an infrastructure that implements the communication amongst the roles of the system.

However, there are still several parts of the system that need to be detailed, namely: the concrete model in which preferences, agreements and proposals are expressed; an implementation of the roles that implement negotiation protocols (*i.e.* at least one BargainingProtocolHandler); and an implementation of the decision making roles (*i.e.* at least one PerformativeSelector, one BuilderManager, one ProposalBuilder and one CommitHandler). Optionally, other roles can be implemented to negotiate protocols (a ProtocolNegotiator), to get and provide information (an Inquirer and an Informant), to model the world (a WorldModeller implemented by means of several estimators), and to improve the coordination of concurrent negotiations (a PoliciesManager).

Next, we detail the decisions and implementations that have been made to build an automated negotiation system for the job submitter. A similar development process and sample execution could have been defined for the job hosting service.

A.2 Preferences, agreements and proposals

The generic data model of NegoFAST implemented in August must be further refined to a particular way of expressing preferences, agreements and proposals. This refinement can be implemented in two complementary ways. On the one hand, the concepts of the generic model are parameterised by concrete aspects. Particularly, preferences are parameterised by the type of statements, and agreements and proposals are parameterised by the type of terms. On the other hand, the generic model itself may also be extended to support advanced features. For instance, terms could be extended to add compensation clauses or proposals could be extended to include additional negotiation data about the terms specified in the proposal. Therefore, in its simplest form, this refinement involves only the definition of the different types of statements that shall be used in the preferences by extending interface `IStatement`, and the definition of the types of terms that shall be used in agreements and proposals by extending interface `ITerm`.

As a consequence, the first step is on deciding which models are appropriate to express the preferences, agreements and proposals. Before making this decision, it is important to bear in mind that the models selected have a strong influence on the algorithms that must be implemented by the decision making roles (chiefly PerformativeSelector, BuilderManager and ProposalBuilder). Generally speaking, the more expressive the model, the less variety of algorithms are available to negotiate with them. Therefore, a trade-off shall be made between the expressiveness of the models and the availability of negotiation algorithms that deal with them.

In our case, we have opted for a commonly used model, for which there are a variety of algorithms available. Specifically, terms of agreements and proposals are name-value pairs and statements can be of two types: name-value pairs for negotiation guidelines and weighted utility functions for requirements and features. This decision is implemented as follows (*cf.* Figure §A.2): regarding agreements and proposals, interface `ITerm` is extended by interface `ITermLite`, which is parameterised by interface `IEquals`. `IEquals` is a type of constraint that supports equality constraints only; regarding preferences, interface `IPreferencesDocument` is parameterised to use statements of

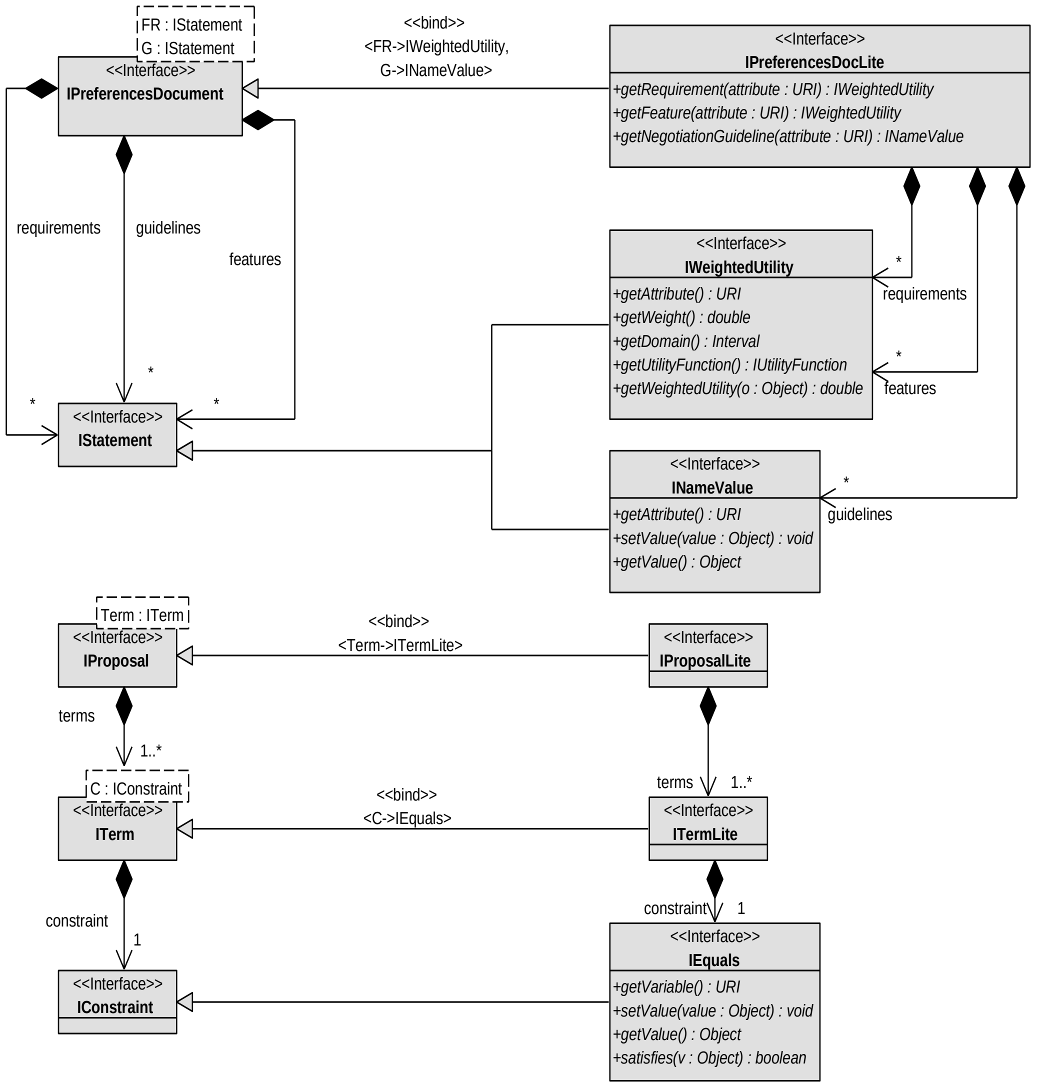


Figure A.2: *Preferences, agreements and proposals models.*

type `IWeightedUtility` to express requirements and features, and statements of type `INameValue` to express negotiation guidelines. The former implements utility functions as those described in Section §3.3, whereas the latter implements name-value pairs.

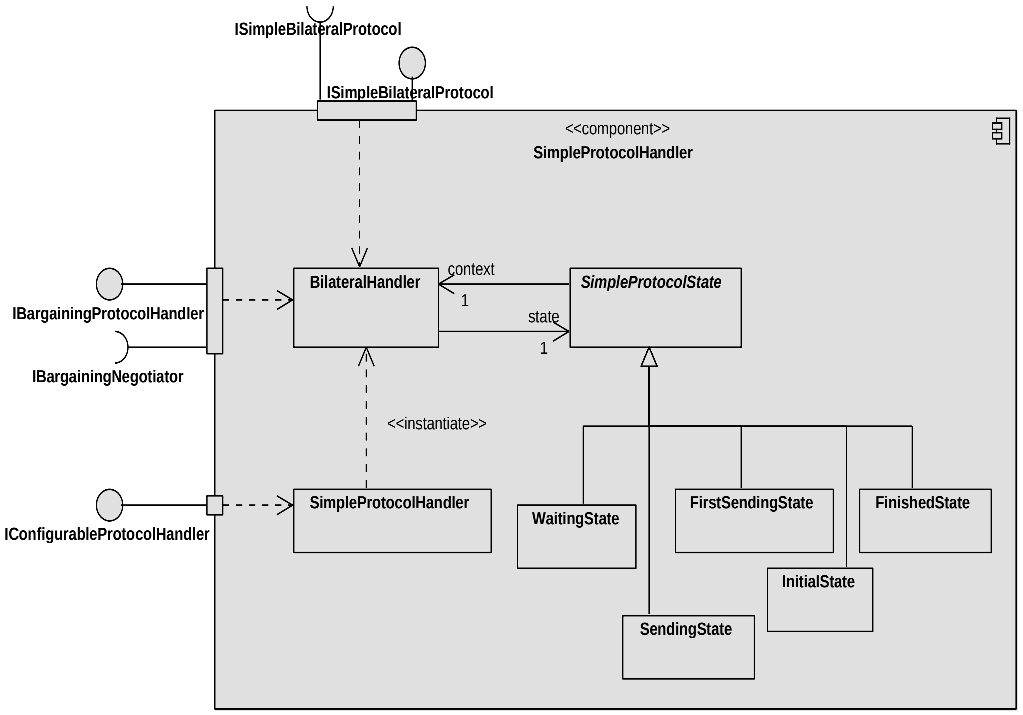


Figure A.3: *Implementation of negotiation protocol roles.*

A.3 Negotiation protocols

The NegoFAST-Bargaining framework defines an abstract negotiation protocol and lets role BargainingProtocolHandler to deal with particular bargaining protocols. This role must deal with the errors, *e.g.* delays or unknown messages, convert the negotiation messages of the bargaining protocol to the data models provided by the NegoFAST framework and its extensions, and implement interaction *ProtocolConversion* to communicate with the Bilateral-Negotiator. It must also provide the BilateralNegotiator with the performatives that can be used as response to the current negotiation message.

Consequently, the first step is to decide which the most appropriate protocols are. To make this decision, we must pay attention to the expressiveness of the proposals for which it allows, the restrictions that it poses over the contents of the proposals and the performatives it allows, *cf.* Section §3.4. In addition, the negotiation protocol has also an influence on the decision-making algorithms.

We have selected a common negotiation protocol described by Sierra and others and detailed in Section §3.4. The implementation of this protocol (*cf.* Figure §A.3) involves the definition of interface `ISimpleBilateralProtocol` that represents the interface with the other party; the implementation of class `BilateralHandler`, which implements both `ISimpleBilateralProtocol` and `IBargainingProtocolHandler` and uses the state pattern [54] to deal with the protocol; and the implementation of class `SimpleProtocolHandler`, which implements interface `IConfigurableProtocolHandler` and simply returns new instances of class `BilateralHandler`.

A.4 Decision-making roles

The NegoFAST framework specifies the interface and protocol of decision-making roles, which include PerformativeSelector, BuilderManager, ProposalBuilder and CommitHandler. However, neither the NegoFAST framework nor August provide any implementation. To build an automated negotiation system, at least an implementation of each of them must be provided. Specifically, an automated negotiation system must provide one implementation of the PerformativeSelector and the CommitHandler, and must provide one or more implementations of the BuilderManager and the ProposalBuilder. Nevertheless, there is usually just one BuilderManager that selects, configures and invokes several ProposalBuilders.

As a consequence, the next step is on deciding which algorithms should implement these decision-making roles. Section §4.2 reviews many of these algorithms and Section §7.4.1 and Section §8.4.1 outline some alternatives to implement them. This decision may be influenced by the model used to express agreements, proposals and preferences and the bargaining protocol because the use of a particular algorithm may depend on a certain preferences model or a concrete bargaining protocol. However, generally speaking, only ProposalBuilders and BuilderManager are influenced by them. In particular, ProposalBuilders are heavily influenced because their algorithms are based on the models of proposals and preferences and they must obey the restrictions that the negotiation protocol poses over the contents of the proposals. Instead, BuilderManager just must be aware of them to select and configure the appropriate ProposalBuilder.

In our case, the PerformativeSelector (class `PerformativeSelector` in Figure §A.4) implements a simple algorithm that selects performative `accept` if the utility of the received proposal exceeds an user-defined threshold. Otherwise, it selects performative `commit`. There are two Propos-

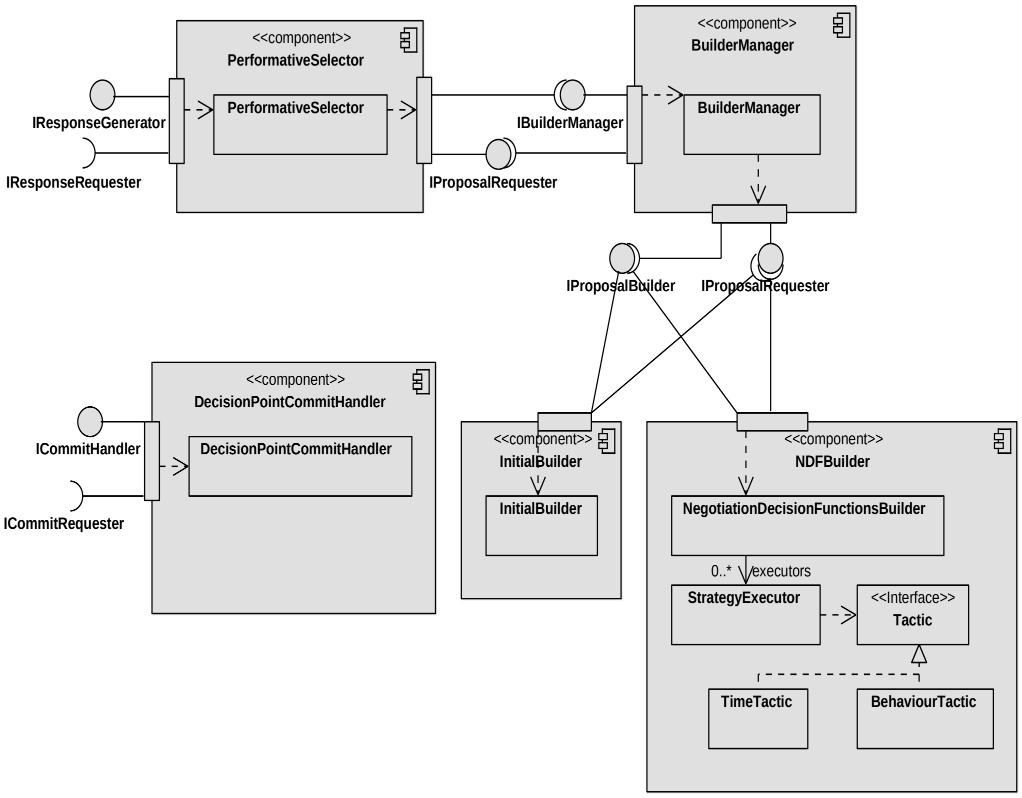


Figure A.4: *Implementation of decision-making roles.*

alBuilders implemented (*cf.* Figure §A.4), `InitialBuilder`, which creates the initial proposal by selecting the values that maximise the utility, and `NegotiationDecisionFunctionsBuilder`, which implements the decision-making algorithms proposed by Faratin and others (*cf.* Section §4.2). As a consequence, the implementation of the BuilderManager is also simple: if it is the first negotiation message, it selects `InitialBuilder`, in other case, it selects `NegotiationDecisionFunctionsBuilder`. The implementation of the CommitHandler (*cf.* Figure §A.4) is based on decisions points, *i.e.* binding negotiation messages are waiting for approval until a decision point takes place. In that case, a number of binding negotiation messages are approved and the others are rejected. In our implementation, decision points take place when the number of binding negotiation messages waiting for approval exceeds a threshold or the negotiation deadline is close.

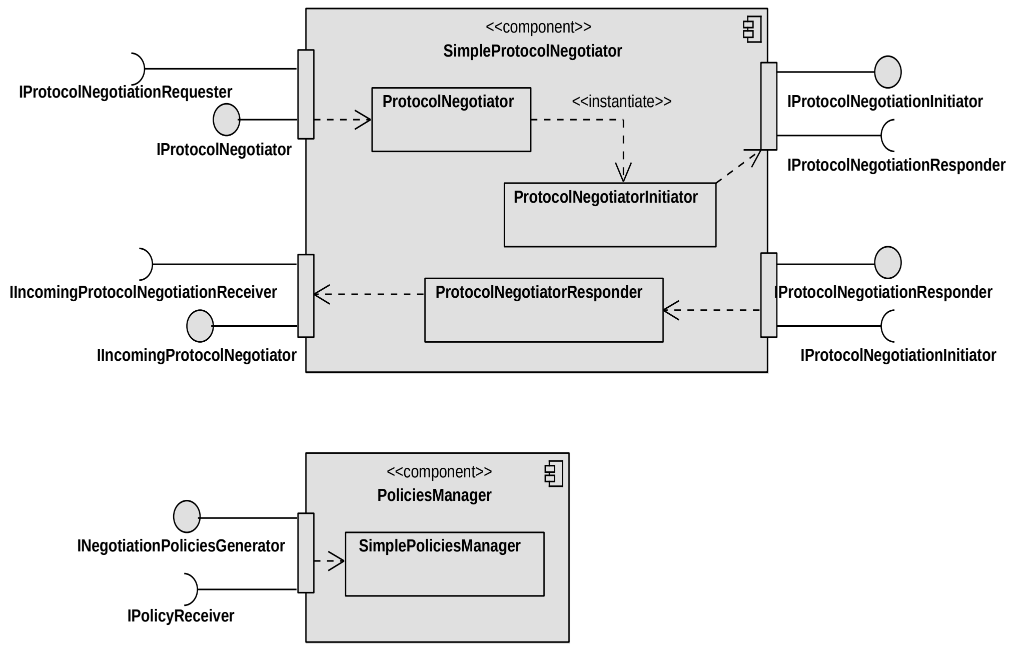
A.5 Optional roles

There are other optional roles (ProtocolNegotiator, Inquirer, Informant, WorldModeller and PoliciesManager) that are not strictly necessary to implement an automated negotiation system, but provide additional features that may be appropriate for automated negotiation systems as detailed in Section §2.2. Furthermore, decision-making roles may require some estimators in the WorldModeller.

The decision of whether implementing these roles depends on the particular scenario we are tackling with the automated negotiation systems. If it is a scenario with heterogeneous parties, it may be convenient to allow a protocol negotiation prior to the negotiation to agree on which negotiation protocol to use. If the scenario involves negotiating with parties of which the automated negotiation system has no previous knowledge, it may be necessary to provide an information exchange mechanism by means of the Inquirer and the Informant, and to build analysis-based models of the other parties by means of the different estimators that are part of the WorldModeller. If the scenario is dynamic, it may be convenient to develop a PoliciesManager that helps in the coordination of concurrent negotiations (*cf.* Section §8.1). Regarding the implementation itself, the ProtocolNegotiator just implements a standard protocol for negotiating negotiation protocols; Inquirer and Informant implement a protocol[†1] for exchanging information such as WS-MetadataExchange [7]; and the estimators of the WorldModeller implement some modelling algorithms such as those described in Section §4.3.

In our case, we consider that we have heterogeneous parties and that our scenario is dynamic, but that we do not require a vast amount of previous knowledge of the other parties because we are just going to establish very short-term relationships with them (*i.e.* the submissions of only one job). Therefore, we just provide the ProtocolNegotiator and a simple PoliciesManager. The ProtocolNegotiator (*cf.* Figure §A.5) implements a simple protocol negotiation protocol based on the SNego protocol[63]. This protocol involves an initiator, implemented by class `ProtocolNegotiatorInitiator`, that sends a set of accepted negotiation protocol instances to the other party, and a responder, implemented by class `ProtocolNegotiatorResponder`, that either accepts one of the negotiation protocol instances or rejects the protocol negotiation. The PoliciesManager (*cf.* Figure §A.5), implemented by class `SimplePoliciesManager`, is very simple and just sets several policies

[†1]Note that it is not necessary to implement a full protocol. For instance, Inquirer and Informant may implement the exchange of proposals that conform the first steps of the WS-Agreement specification [4].



<<component>>
SimpleProtocolNegotiator
IProtocolNegotiationRequester
IProtocolNegotiator
ProtocolNegotiator
<<instantiate>>
ProtocolNegotiatorInitiator
IProtocolNegotiationInitiator
IProtocolNegotiationResponder
IIncomingProtocolNegotiationReceiver
IIncomingProtocolNegotiator
ProtocolNegotiatorResponder
IProtocolNegotiationResponder
IProtocolNegotiationInitiator
<<component>>
PoliciesManager
INegotiationPoliciesGenerator
IPolicyReceiver
SimplePoliciesManager

Figure A.5: *Implementation of optional roles.*

(`threshold`, `deadline`, `agreementsNumber` and `eagerness`) based on the preferences given by the user and gives them default values if the user does not specify them in his or her preferences.

A.6 Summary

In this chapter, we have presented one use case of an automated negotiation system, and we have detailed an instantiation of the NegoFAST framework to support it.

Appendix B

Use case: Hosting computing jobs

The goal of this chapter is to show some of the reusability capabilities of the NegoFAST framework. To this end, we describe the changes that have to be made to the implementation of the automated negotiation system described in Appendix §A to use a different decision-making model. It is organised as follows: in Section §B.1, we review the automated negotiation system of Appendix §A and overview the changes to be made in this appendix; Section §B.2, Section §B.3, Section §B.4 and Section §B.5 detail the steps that are necessary to build the automated negotiation system; finally, Section §B.6 summarises the ideas of this chapter.

B.1 Introduction

The main design goal of the NegoFAST framework is to enhance the reusability of the elements of the system so that they can be interchangeable. To illustrate this reusability, in this chapter we take the automated negotiation system described in Appendix §A as a starting point and we analyse the additional development it involves to change it to negotiate on behalf of the job hosting service instead of the computing job submitter. This involves, at least, providing new implementations for some decision-making roles (*i.e.* PerformativeSelector, BuilderManager and ProposalBuilders). It may also involve changes in the way of expressing preferences or agreements, and it may require some estimators to provide it with models of the other parties or the market. The new decision-making model is inspired on the Nash bargaining solution [95]. Section §B.4 details this decision-making model. In addition, the automated negotiation system must be provided with an estimator of the utility function of the other party.

B.2 Preferences, agreements and proposals

Currently, the model used to express agreements, proposals and preferences is an usual one, for which there are a variety of algorithms available. Specifically, it expresses terms of agreements and proposals as name-value pairs and preferences statements are expressed as name-value pairs for negotiation guidelines and weighted utility functions for requirements and features.

The new decision-making model only requires an utility function that expresses the preferences of the user. Therefore, the model does not need to be changed to be used with the new decision-making model and, hence, the implementation can be fully reused.

B.3 Negotiation protocols

The protocol implemented in Appendix §A is a usual negotiation protocol described by Sierra and others, *cf.* Section §3.4. It is a simple bargaining negotiation protocol in which both parties exchange binding proposals until an agreement is reached or one party decides to finish the negotiation.

Since the automated negotiation system is going to negotiate in the same scenario and the change in the decision-making model does not have any influences on the negotiation protocol, the implementation of the BargainingProtocolHandler can be reused.

B.4 Decision-making roles

The NegoFAST framework divides the decision-making model into four decisions, each implemented by a different role. These decisions are: selecting a performative, which is implemented by the PerformativeSelector; selecting the proposal creator, implemented by the BuilderManager; creating a proposal, implemented by the ProposalBuilder; and approving the sending of a binding proposal, implemented by the CommitHandler. Now, we have to analyse the differences between the new decision-making model and that implemented in Appendix §A in order to determine which role implementations can be reused and which roles must be implemented again.

As we mention in the introduction of this appendix, the new decision-making model is inspired on the Nash bargaining solution [95]. In this article, Nash describes the so-called Nash Bargaining Game, which is a simple two-player game used to model bargaining interactions. In the Nash Bargaining Game two players demand a portion of some good (usually some amount of money). If the two proposals (x and y) sum to no more than the total good (*i.e.* $x + y \leq z$), then both players get their demand. Otherwise, both get d, which represents the disagreement point.

A Nash bargaining solution is a (Pareto efficient) solution to a Nash Bargaining Game with the following constraints: invariant to equivalent utility representations; Pareto optimality (*cf.* Section §3.1); independence of irrelevant alternatives, which means that if A is preferred to B, then introducing a third option X, must not make B preferred to A; and symmetric for all players in the game (*i.e.* parties in the negotiation). Under this circumstances, rational agents will choose the Nash bargaining solution, which is the pair of proposals (x, y) that maximises: $|u_1(x) - u_1(d)||u_2(y) - u_2(d)|$, where $u_1(x)$ is the utility function for player 1 (*e.g.* the consumer), $u_2(x)$ is the utility function for player 2 (*e.g.* the provider).

Following this idea, the decision-making model of a player involves the following steps. First, it starts with a proposal that maximises its utility. Then, in the following proposals, it concedes by trying to maximise the product of both utility functions, assuming that $u_1(d) = u_2(d) = 0$.

The concession between two consecutive proposals (x_n and x_{n+1}) is de-

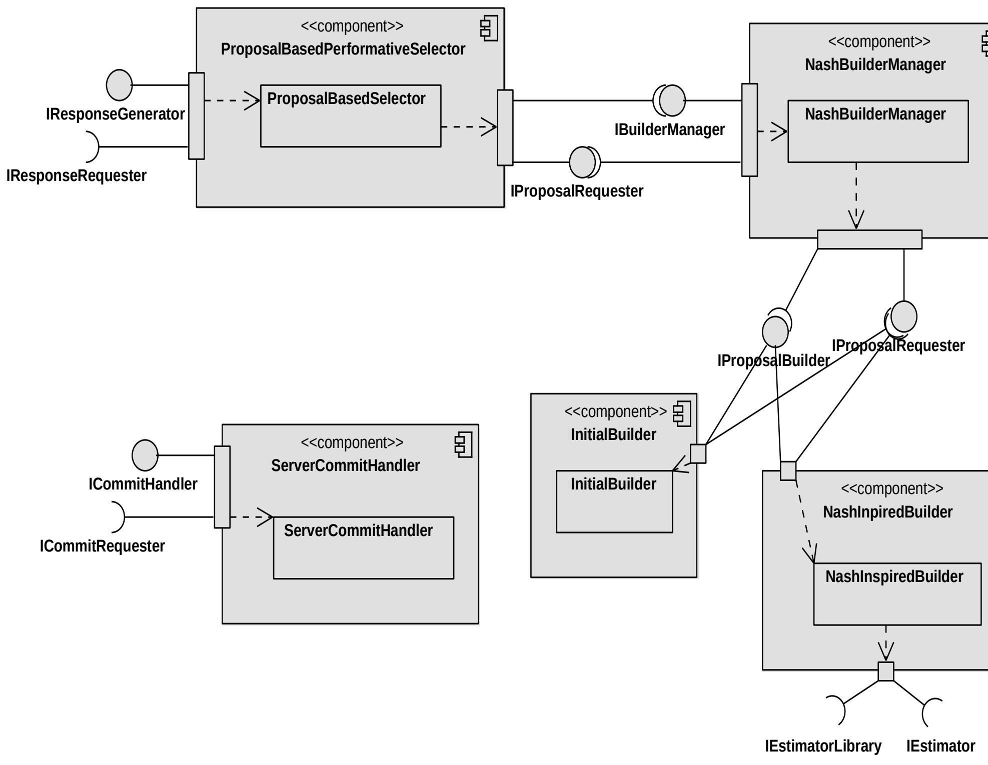


Figure B.1: *Implementation of decision-making roles.*

fined as follows: $u_1(x_{n+1}) = \delta(t)u_1(x_n)$, where $\delta : \mathbb{R} \to [0, 1]$ determines the amount of concession between two consecutive proposal and depends on the time elapsed since the beginning of the negotiation.

Performative `accept` is selected instead of sending a counter proposal when the utility of the received proposal (y_n) is higher than the utility of its potential response (x_{n+1}): $u_1(y_n) \geq u_1(x_{n+1})$.

Regarding the approval of binding negotiation messages, since we take the perspective of a job hosting service, there is no restriction about the number of agreements that we can reach provided that we do not exceed the server capacity.

After detailing the new decision-making model, we can extract the following conclusions about the changes we have to make with the decision-making roles:

- The implementation of the PerformativeSelector must change because the criterion to select performative `accept` is not an user-defined threshold, but depends on the proposal received and its potential response.

- There are two implementation of ProposalBuilders. One is the `InitialBuilder`, which creates the initial proposal by selecting the values that maximise the utility and was described in Appendix §A. The other has to be developed and implements the above-described algorithm inspired on the Nash bargaining solution.

- The implementation of the BuilderManager is almost the same as in Appendix §A, but it uses the Nash-inspired builder instead of the negotiation decision functions builder.

- The implementation of the CommitHandler must change because it does not control any restriction about the number of agreements that can be reached and it is not decision-point based (*cf.* Section §A.4).

Figure §B.1 depicts the structure of the implementation of decision-making roles, in which the PerformativeSelector is implemented by class `ProposalPerformativeSelector`; the BuilderManager is implemented by class `SimpleBuilderManager`; the ProposalBuilders are implemented by class `InitialBuilder` and class `NashInspiredBuilder`, and the CommitHandler is implemented by class `ServerCommitHandler`. In addition, note that class `NashInspiredBuilder` requires an estimator of the utility function of the other party that is described in the next section.

B.5 Optional roles

The automated negotiation system described in Appendix §A implements two optional roles: ProtocolNegotiator and PoliciesManager (*cf.* Section §A.5). The former implements a simple protocol negotiation based on the SNego protocol, whereas the latter sets several policies (`threshold`, `deadline`, `agreementsNumber` and `eagerness`) based on the preferences given by the user and gives them default values if the user does not specify them in his or her preferences. In our case, we can reuse them both. The only consideration is that the `agreementsNumber` policy, if set, shall be ignored by the CommitHandler.

However, as mentioned above, an estimator of the utility function of the other party is required to implement the Nash-inspired builder. This is implemented by means of class `UtilityFunctionEstimator`, which implements interface `IEstimator` as depicted in Figure §B.2. The implementation is based

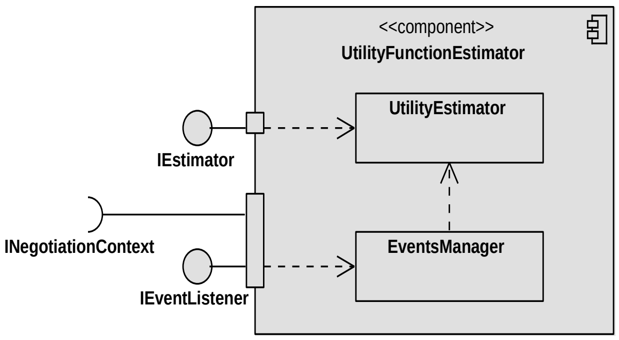


Figure B.2: *Implementation of utility function estimator.*

on the mechanism described in Reference [115] to deduce the importance the other party gives to the negotiation terms from the other party's proposals history. This mechanism is based on the idea that the most important attributes for the other party are the ones with less variations between proposals. For example, if we are not interested in delivery time, it makes no difference to us to change it as the other party demands it. But, in the case of the price, it is important to us to try to keep it as stable as possible with small variations. Building on this idea, Ros and Sierra [115] define the variability of a term in a window of size m of the proposals history as follows:

$$f(j) = \frac{\sum_{i=0}^{m-1} |y_{t-i}[j] - y_{t-(i+1)}[j]|}{(m-1)\Delta_j}$$

where $y_t[j]$ is the value of term j in the last proposal made by the other party, t is the current time, $m > 1$ and Δ_j is the difference between the maximum and the minimum value that j can take. This variability measure allows to calculate the weight the other party gives to each attribute in its utility function. The utility function of the other party for each attribute is estimated as the opposite as our utility function. Note that the estimator uses the event mechanism provided by the environmental resources to get updates when a new negotiation message has been sent or received and, hence keep up to date the estimation.

B.6 Summary

In this appendix, we have modified the decision-making model of the automated negotiation system described in Appendix §A and have shown how the implementation of the model to express preferences, agreements and proposals; the implementations of the protocol management roles; the implementations of ProtocolNegotiator and PoliciesManager; and even the implementations of some decision-making roles can be fully reused. Moreover, we have described a Nash-inspired decision-making model.

Appendix C

Use case: Strategies equilibrium

In this appendix we face the problem of finding the equilibrium between a number of strategies. We show how to integrate the NegoFAST framework with a Java framework for genetic algorithms (JGAP) in order to apply a known evolutive approach to calculate the equilibrium amongst strategies. It is organised as follows: in Section §C.1, we introduce concepts about strategies equilibrium and describe the genetic algorithm we use to calculate evolutionarily stable strategies; Section §C.2 details the steps that are necessary to build an automated negotiation system that represents the individuals during the execution of the evolutive algorithm; Section §C.3 describes the integration of such automated negotiation system with JGAP; finally, Section §C.4 summarises the ideas of this chapter.

C.1 Introduction

One of the most important elements of the theoretical study of a negotiation model is its equilibrium. The earliest concept of equilibrium was the Nash equilibrium [96]. Informally, a set of strategies are in Nash equilibrium if each party has chosen a strategy and no party can benefit by changing his or her strategy while the other players keep theirs unchanged.

A consequence of the Nash equilibrium is that, assuming the parties in the negotiation are rational and that they have full knowledge of the preferences of the other parties, each party will select an equilibrium strategy when choosing independently because it is the more profitable for them. Therefore, the Nash equilibrium provides valuable information to predict the outcome of a negotiation assuming, first, that every party takes the most profitable action given his expectations of what the others will do, and second, everyone is correct in the expectations of what the others will do, or equivalently every one ends up doing what is expected of him or her [89]. However, Nash equilibrium is hard to calculate in the general case [25], and it often needs to be considered in an ad-hoc manner such as in Reference [46].

An alternative approach to calculate equilibrium strategies is the so-called evolutionarily stable strategies (ESS). ESS were defined and introduced by Smith and Price [124] and were later developed by Smith [125]. Here, it is assumed that, instead of a single set of players, the game is played by large populations of players. Players are individuals with biologically encoded, heritable strategies. The individuals reproduce themselves and are subject to the forces of natural selection (with the payoffs of the game representing biological fitness). It is assumed that the alternative strategies of the game occasionally occur, via a process like mutation, and for a strategy to be an ESS, it must be resistant to these mutations.

In this appendix, we are interested in building a system that uses genetic algorithms to calculate ESS. In particular, we show how to integrate the NegoFAST framework with JGAP [1], a Java framework for genetic algorithms, in order to develop such a system.

In the literature, there are several approaches to apply evolutionary methods, usually genetic algorithms, to bargaining [40, 45, 90, 137]. In this use case, we follow an approach similar to Fatima and others [45]. In this paper, the authors argue that, since bargaining involves two parties with different utility functions (consumer and provider), the population should be divided into two subpopulations: one representing the consumer and the other representing the provider. In such asymmetric configurations, the evolution of strategies in each subpopulation affects the evolution of strategies in the other subpopulation (*i.e.*, the strategies co-evolve). Thus, it is studied the competitive co-evolution in which the fitness of an individual in one population is based on direct competition with individuals of the other population.

The chromosome of an individual configures its negotiating behaviour, *i.e.* its decision-making model. Furthermore, in many approaches [45, 90], chromosomes also encode the requirements and features of the individual in terms of utility functions. Therefore, chromosomes is an expression of individuals' preferences, in which the negotiation behaviour is expressed as negotiation guidelines. For instance, if the negotiation behaviour is based on negotiation decision functions [44] (*cf.* Section §4.2 and Section §A.4), the chromosome encodes the tactic to create the proposals (time-based, resource-based or behaviour-based), and the parameters that configure each tactic [90].

The fitness function of an individual in both populations is determined by competition between the individuals in the two populations. Each individual competes against either a subset or all of the other individuals in the other population. The average utility obtained by the individual in these negotiations is the individual's fitness value. In the next stage, a new generation of individuals is created for each population using selection, crossover and mutation [45].

Next, we detail how to build an automated negotiation system, based on the NegoFAST framework, that can be used to implement the individuals of both populations. Then, we describe how to integrate that automated negotiation system with a genetic algorithms Java framework in order to develop a system that calculates ESS.

C.2 The automated negotiation system

We are interested in building an automated negotiation system that represents the individuals of both populations during the execution of the genetic algorithm. This entails that we must create a lightweight automated negotiation system because many of them shall be instantiated during each stage of the execution of the genetic algorithm, one for each individual in both populations. Furthermore, in this use case we are just interested in analysing the behaviour of different negotiation decision functions in bilateral negotiations. Therefore, other parts of the automated negotiation system must be implemented so that they interfere as little as possible in the negotiation result.

These requirements determine many of the decisions that are necessary to make while developing an automated negotiation system, which are the concrete model in which preferences, agreements and proposals are expressed; an implementation of the roles that implement negotiation protocols; an implementation of the decision making roles; and implementations of additional roles that provide advanced features to the automated negotiation system to cope with heterogeneous parties in dynamic scenarios. Next, we detail each of these decisions for our automated negotiation system.

C.2.1 Preferences, agreements and proposals

The information stored in an individual's chromosome is mapped onto the preferences of the individual's automated negotiation system. Preferences contain two kinds of information. On the one hand, they keep information regarding the requirements and features of the individual, which should be expressed in a way that can be interpreted by the strategies that are being studied. In this case, the strategies are negotiation decision functions that are designed to work with weighted utility functions. On the other hand, preferences keep negotiation guidelines for the automated negotiation system. These negotiation guidelines can be used to configure the negotiation behaviour of the automated negotiation system. In this case, that means that we shall use one negotiation guideline for each gene that expresses the negotiation behaviour in the chromosome (*i.e.* the tactic that shall be used to create proposals and its configuration). Therefore, negotiation guidelines can be expressed as name-value pairs. Regarding agreements and proposals, the model chosen is imposed by negotiation decision functions, *i.e.* the terms of agreements and proposals are name-value pairs. Note that this model for expressing preferences, agreements and proposals is the same as the one described in Section §A.2.

C.2.2 Negotiation protocols

We choose to use the negotiation protocol described by Sierra and others [119] and detailed in Section §3.4. It is a simple bargaining negotiation protocol in which both parties exchange binding proposals until an agreement is reached or one party decides to finish the negotiation.

We have chosen this protocol for two reasons: first, because it is simple and, hence, it helps to build a lightweight automated negotiation system; second, because it was the protocol used in the paper in which negotiation decision functions were first described [44] and, hence, it helps to minimise the impact that the negotiation protocol may have in the negotiation result. This protocol was already implemented in Appendix §A, therefore we can reuse its implementation.

C.2.3 Decision-making roles

The decision-making roles are heavily influenced by the strategies under consideration, in this case negotiation decision functions, and the fact that we want to minimise the influence of other parts of the automated negotiation system in the negotiation result. This has the following implications:

- The implementation of the PerformativeSelector must follow the behaviour described in Reference [44], which involves selecting performative `accept` when the utility of the received proposal (y_n) is higher than the utility of its potential response (x_{n+1}): $u_1(y_n) \geq u_1(x_{n+1})$ and selecting performative `commit` otherwise. This implementation of the PerformativeSelector has been already described in Section §B.4.

- There is only one implementation of the ProposalBuilders. It is the `NegotiationDecisionFunctionsBuilder`, which implements negotiation decision functions and is described in Section §A.4.

- The implementation of the BuilderManager is simple and its only task is to configure the `NegotiationDecisionFunctionsBuilder` with the values specified by the negotiation guidelines.

- Regarding the CommitHandler, its implementation must not interfere with the decisions made by the other decision-making roles in order to minimise its influence. Therefore, the best alternative is to implement a CommitHandler that approves all binding negotiation messages. Furthermore, this has an interesting side-effect: it allows the automated negotiation system to carry out n simultaneous but independent negotiations, *i.e.* the negotiations with all individuals of the other parties shall be carried out in parallel.

Summarising, only two trivial new implementation of roles must be provided: an implementation of the BuilderManager that configures the `NegotiationDecisionFunctionsBuilder` according to the negotiation guidelines and an implementation of the CommitHandler that approves all binding negotiation messages.

C.2.4 Optional roles

Decision-making roles do not require any additional elements such as an estimator to work properly. Therefore, we are not going to use any optional roles in this automated negotiation system to keep the system as simple as possible.

C.3 Integration with JGAP

After defining the automated negotiation system that implements the individuals of both populations (consumer and provider), it is necessary to develop an implementation of genetic algorithms to carry out the evolution of the populations. To avoid implementing a genetic algorithm from scratch, we use JGAP [1], which is a Genetic Algorithms and Genetic Programming component provided as a Java framework. It provides basic genetic mechanisms that can be easily used to apply evolutionary principles to problem solutions.

The implementation of a genetic algorithm using JGAP is simple and involves the following steps:

- Instantiation of a sample chromosome (class `Chromosome`).
- Definition of a fitness function by extending class `FitnessFunction` or `BulkFitnessFunction`. The former is used to evaluate each individual separately, whereas the latter is used to evaluate the whole population.
- Creation of a configuration object to set up the execution.
- Create the population and evolve it.

Figure §C.1 depicts the integration between the automated negotiation system and JGAP implemented by `ANSFitnessFunction`. This integration is carried out, on the one hand, by implementing abstract class `BulkFitnessFunction` and, on the other hand, by implementing interface `IUser`. Therefore `ANSFitnessFunction` plays role User in the automated negotiation system.

The whole system works as follows. First, component `Initialiser` (*cf.* Figure §C.1) instantiates a sample chromosome, sets up the execution of the algorithm by creating two configuration objects (one for each population), creates two populations and co-evolves them.

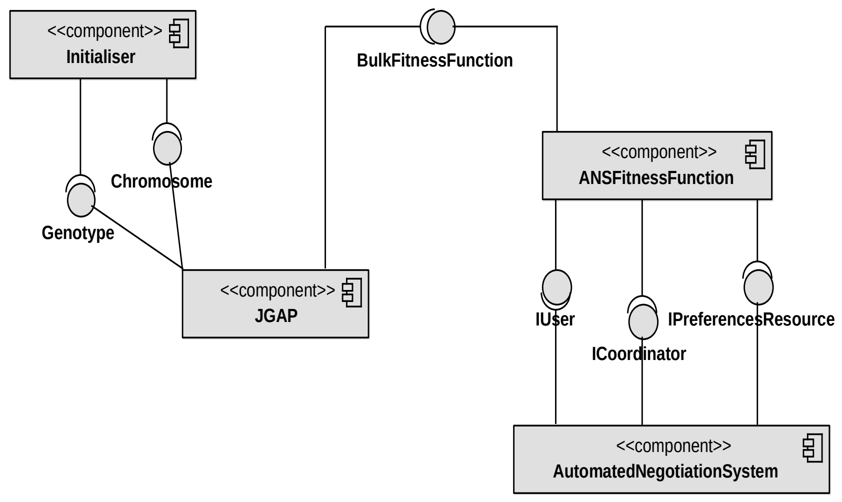


Figure C.1: *Integration of the automated negotiation system with JGAP.*

The first step of an evolution cycle is calculating the fitness function for each individual. In this step, JGAP invokes the implementation of `BulkFitnessFunction` provided by `ANSFitnessFunction`. To calculate the fitness function, `ANSFitnessFunction` first maps the chromosomes of each individual onto preferences that are understood by the automated negotiation system and, then, it uses the preferences to initialise the automated negotiation system (one per individual). Next, it sends to all automated negotiation system of one population, say the consumers population, the references of all automated negotiation system of the other population in order to start the negotiations. Note that due to the implementation of the CommitHandler, all negotiations are independent from each other (*cf.* Section §C.2.3). When an agreement is reached between a consumer and a provider, the automated negotiation system sends it to `ANSFitnessFunction`. Finally, when all negotiations finish, `ANSFitnessFunction` uses the *PreferencesResource* to evaluate the utility of all agreements that have been reached. The fitness value is the mean of all those utilities.

With the fitness value calculated for all individuals of both populations, the genetic algorithm continues its execution. This evolution cycle is repeated a number of times until the chromosomes of the individuals of both populations stabilise. These chromosomes define the evolutionarily stable strategies of the negotiation we are considering.

C.4 Summary

In this appendix, we are interested in building a system that calculates equilibrium strategies by means of genetic algorithms. Particularly, we describe how to use the NegoFAST framework to build a lightweight automated negotiation system that represents the individuals of both populations during the execution of the genetic algorithm and how to integrate it with JGAP, a genetic algorithm Java framework.

Appendix D
Auctions

Auctions are a popular and effective method for procuring goods and services [5]. There are many types of auctions depending on a variety of criteria and there have been several attempts to create a taxonomy that covers them all [8, 127]. Below, we describe the main characteristics of auctions following the high-level aspects of negotiation protocols described in Section §3.4.

Parties: Depending on the parties, auctions may be regular auctions, reverse auctions and double auctions. A regular auction is an auction in which there are many consumers of a good or service and only one provider. In a reverse auction, the configuration is the opposite, i.e., there is a unique consumer and many providers. This type of auction is also known as procurement auction. Finally, in a double auction, there are many consumers and providers selling and buying goods or services at a time.

Performatives: The basic performatives in an auction are `bid`, `accept`, `reject` and `inform` (information sent by the auctioneer to all bidders related to the state of the auction). Some auctions also include a `withdraw` performative like in those defined in the Trading Agent Competition [2].

Rules: Rules determines characteristics such as the number of rounds, which bids can be submitted, when the auction finishes, who the winner is, or which the final agreement is. These rules can be significantly different from one type of auction to another.

Information exchanged: In auctions, the most basic type of information exchanged is a bid. However, it is also common to receive information about the other parties' bids. In particular, depending on the information revealed, auctions may be either sealed or public. In a sealed auction, a party does not know the bids submitted by the others; in a public auction, all parties know the bids the other parties have submitted.

Agreement terms negotiability: Depending on the terms under negotiation, auctions can be classified into:

- One type of item, one attribute: This is the most typical configuration: there is only one type of good or service auctioned and the only attribute under negotiation is the price. However, in some auctions, the number of goods or services that are being sold or bought can also be a part of the bids.
- One type of item, multiple attributes: It is an extension of one-attribute auctions to support negotiations with terms other than the price. In References [15, 131], it is argued that this type of auctions are specially useful in reverse auctions, *i.e.*, in procurement scenarios. In these cases, it is common that the consumer is interested in more terms than price. For instance, Bichler [15] present the case of large food retailers, where consumers are interested in product quality, price, terms of payment and delivery. Multi-attribute auctions usually follow the same rules as single-attribute auctions. The only difference is that, whereas in common auctions the improvement of the bid is clear for the bidder (*i.e.* the higher the price, the higher the utility), in multi-attribute auctions, it is necessary to guide the bidder in that process [14]. There are several approaches to that, namely: in the Auction Maker Controlled Bid Mechanism [130], the auction maker discretises the attributes under negotiation and represents a preference path for the bidders. An alternative is presented in Reference [14] following the articles by Che [23] and Branco [17]; in this case, the auction maker reveals a scoring function to the bidders based on its utility function.
- Combinations of items, one attribute: In these auctions, several types of items are auctioned at the same time and bidders can bid on combinations of these types of items. The advantage of these auctions over one-type-of-item auctions is when bidders have non-additive values for the goods or services that are being auctioned. For instance, the value of a estate for a bidder may be significantly increased if it also obtains and adjacent estate [73].
- Combinations of items, multiple attributes: Although these auctions are theoretically possible, in practice they are not used because of their complexity.

Although concrete implementations of auctions may have specific details that make them slightly different, the most significant types of auctions are as follows (summarised in Table §D.1):

Auction	Parties	Rules	Information	Negotiability
English	Regular or reverse	Starts with the minimum acceptable price and parties must raise their bids	Public	One type of item, one/several attributes
Dutch	Regular or reverse	Starts with the maximum price and the auctioneer lowers it while no party bids	Public	One type of item, one attribute
First-price sealed	Regular or reverse	Parties submit their bids, winner is highest bid, price is highest bid	Sealed	One type of item, one/several attributes
Vickrey	Regular or reverse	Parties submit their bids, winner is highest bid, price is second highest bid	Sealed	One type of item, one/several attributes
Continuous double	Double	Bids and asks are submitted continuously, when a bid matches an ask, a trade is executed	Public	One type of item, one attribute
Combinatorial	Regular or reverse	Bids expressing preferences about combinations of goods, the auctioneer decides the winners	Public/sealed	Combinations of items, one attribute

Table D.1: *Summary of auction protocols.*

English auctions: They are always regular or reverse auctions. There is only one good or service auctioned at the same time and there may be either only one or several attributes under negotiation. An English auction starts with the minimum acceptable price and parties must raise their bids until either nobody bids higher during a certain time or a time limit is reached. The higher bid wins the auction and all bids are public.

Dutch auctions: These auctions are always regular or reverse auctions as well, and, like English auctions, only one good or service is auctioned at the same time. However, in Dutch auctions there must be only one attribute under negotiation, which is usually the price. Dutch auctions start with the maximum price and the auctioneer lowers it while no party bids, the first one that bids, wins the auction. Like English auctions, Dutch auctions are public.

First-price sealed auctions: They are always regular or reverse auctions, in which only one type of time with either one or multiple attributes are auctioned. Unlike English and Dutch auctions, these auctions are sealed, which means bids are not public. In these auctions, the parties submit their bids and the winner of the auction is the party with the highest bid. The price paid is the price of the highest bid.

Vickrey auctions: They are exactly like first-price sealed auctions except that, whereas in first-price sealed auctions, the price paid is the price of the highest bid; in second-price sealed auctions, the price is the second highest bid.

Continuous double auctions: A continuous double auction has a fixed-duration time period during which buy orders (bids) and sell orders (asks) are submitted continuously without any restrictions. Usually, continuous double auctions are limited to price and quantity. When a buy order matches in price and quantity with a sell order, a trade is executed immediately. An example of continuous double auctions is a stock exchange market.

Combinatorial auctions: These auctions are usually regular auctions or reverse auctions. There is only one attribute under negotiation, which is the price, but the bidders can bid on combinations of goods or services. The problem of these auctions is that the number of available combinations poses problems on the rules and the information exchanged. Regarding the negotiation rules, each bidder sends its bids expressing their preferences about several combinations of goods or services. After receiving all bids, the auctioneer must decide who the winner is. However, unlike other types of auctions, in combinatorial auctions, determining who the winner is is a complex problem [117]. De Vries and Vohra [35] formulates the problem using integer constraint programming, whereas Sandholm [117] describes several approaches based on optimisation-problems algorithms such as greedy algorithms or dynamic programming. Some commercial software such as Quotes [112] also provides mechanisms to assess the winner in combinatorial auctions. The other problem combinatorial auctions must face is how to transmit bidder's preferences in a succinct way to the auctioneer [35], *i.e.* which information is exchanged. One approach is to send preferences in the form of restrictions [99]; an alternative is to rely on an oracle, which is a program that given a bidder and a combination of goods or services, computes the bid for it [117].

Bibliography

[1] Java genetic algorithms package (JGAP), 2008. Available at http://jgap.sourceforge.net

[2] Trading agent competition. http://www.sics.se/tac/, 2008

[3] L. Amgoud, S. Belabbes, and H. Prade. Towards a formal framework for the search of a consensus between autonomous agents. In *Proceedings of the 4th international joint conference on Autonomous agents and multiagent systems*, pages 537–543. ACM Press, 2005. DOI: 10.1145/1082473.1082555

[4] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. WS-Agreement Specification, 2007. Available at http://www.ogf.org/documents/GFD.107.pdf

[5] P. Anthony and N. R. Jennings. Developing a Bidding Agent For Multiple Heterogeneous Auctions. *ACM Transactions Internet Technology*, 3 (3):185–217, 2003. DOI: 10.1145/857166.857167.

[6] R. Ashri, I. Rahwan, and M. Luck. Architectures for negotiating agents. In *Multi-Agent Systems and Applications III: 3rd International Central and Eastern European Conference on Multi-Agent Systems*, volume 2691 of *Lecture Notes in Computer Science*, pages 136–146, 2003. DOI: 10.1007/3-540-45023-8_14.

[7] K. Ballinger, B. Bissett, D. Box, F. Curbera, D. Ferguson, S. Graham, C. K. Liu, F. Leymann, B. Lovering, R. McCollum, A. Nadalin, D. Orchard, S. Parastatidis, C. von Riegen, J. Schlimmer, J. Shewchuk, B. Smith, G. Truty, A. Vedamuthu, S. Weerawarana, K. Wilson, and P. Yendluri. WS-MetadataExchange Specification, 2004. Available at http://www.ibm.com/developerworks/library/specification/ws-mex/

[8] C. Bartolini, C. Preist, and N. R. Jennings. Architecting For Reuse: A Software Framework For Automated Negotiation. In F. Giunchiglia,

J. Odell, and G. Weis, editors, *Agent-Oriented Software Engineering III: Third International Workshop*, volume 2585 of *Lecture Notes in Computer Science*, pages 88–100. Springer-Verlag, 2003. DOI: 10.1007/3-540-36540-0_7

[9] C. Bartolini, C. Preist, and N. R. Jennings. A Software Framework For Automated Negotiation. In R. Choren, A. Garcia, C. Lucena, and A. Ramonovsky, editors, *Software Engineering For Multi-Agent Systems III: Research Issues and Practical Applications*, volume 3390 of *Lecture Notes in Computer Science*, pages 213–235. Springer Verlag, 2005

[10] C. Bartolini, C. Preist, and N. R. Jennings. A Software Framework For Automated Negotiation. Technical report HPL-2006-33, HP Labs, 2006. Available at http://www.hpl.hp.com/techreports/2006/HPL-2006-33.html.

[11] M. Benyoucef, H. Alj, K. Levy, and R. K. Keller. A Rule-Driven Approach for Defining the Behaviour of Negotiating Software Agents. In *DCW '02: Revised Papers from the 4th International Workshop on Distributed Communities on the Web*, pages 165–181. Springer-Verlag, 2002. DOI: 10.1007/3-540-36261-4_16.

[12] M. Benyoucef and M.-H. Verrons. Configurable e-negotiation systems for large scale and transparent decision making. *Group Decision and Negotiation*, 17(4):211–224, 2007. DOI: 10.1007/s10726-007-9073-y.

[13] T. Berners-Lee, R. Fielding, and L. Masinter. RFC3986: Uniform Resource Identifier (URI): Generic syntax, 2005. Available at http://www.ietf.org/rfc/rfc3986.txt

[14] M. Bichler and H. Werthner. A classification framework of multidimensional, multi-unit procurement negotiations. In *Proceedings 11th International Workshop on Database and Expert Systems Applications.*, pages 1003–1009, 2000. DOI: 10.1109/DEXA.2000.875149.

[15] M. Bichler. An experimental analysis of multi-attribute auctions. *Decision Support Systems*, 29(3):249–268, 2000. DOI: 10.1016/S0167-9236(00)00075-0.

[16] N. Bieberstein, S. Bose, L. Walker, and A. Lynch. Impact of service-oriented architecture on enterprise systems, organizational structures, and individuals. *IBM Systems Journal*, 44(4):691–708, 2005

[17] F. Branco. The design of multidimensional auctions. *The RAND Journal of Economics*, 28(1):63–81, 1997

[18] P. Brereton. The software customer/supplier relationship. *Communications of the ACM*, 47(2):77–81, 2004. DOI: 10.1145/966389.966394

[19] A. Brown and G. Grant. Framing the Frameworks: A Review of IT Governance Research. *Communications of the Association for Information Systems*, 15(712):696–712, 2005

[20] A. Byde, M. Yearworth, K.-Y. Chen, and C. Bartolini. AutONA: a system for automated multiple 1-1 negotiation. In *IEEE International Conference on E-Commerce*, pages 59–67, 2003. Available at http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1210234.

[21] J. J. Castro-Schez, N. R. Jennings, X. Luo, and N. R. Shadbolt. Acquiring domain knowledge for negotiating agents: a case of study. *International Journal of Human-Computer Studies*, 61(1):3–31, 2004. DOI: 10.1016/j.ijhcs.2003.09.006

[22] M. K. Chang and C. C. Woo. A speech-act-based negotiation protocol: design, implementation, and test use. *ACM Transactions Information Systems*, 12(4):360–382, 1994. DOI: 10.1145/185462.185477

[23] Y.-K. Che. Design competition through multidimensional auctions. *The RAND Journal of Economics*, 24(4):668–680, 1993. Available at http://www.jstor.org/stable/2555752.

[24] H.-M. Chen, R. Kazman, and A. Garg. Bitam: an engineering-principled method for managing misalignments between business and it architectures. *Science of Computer Programming*, 57(1):5–26, 2005. DOI: 10.1016/j.scico.2004.10.002

[25] X. Chen and X. Deng. Recent development in computational complexity characterization of nash equilibrium. *Computer Science Review*, 1(2): 88–99, 2007. DOI: 10.1016/j.cosrev.2007.09.002.

[26] D. Chiu, S. Cheung, P. Hung, S. Chiu, and A. Chung. Developing e-Negotiation support with a meta-modeling approach in a Web services environment. *Decision Support Systems*, 40(1):51–69, 2005. DOI: 10.1016/j.dss.2004.04.004.

[27] S. P. M. Choi, J. Liu, and S.-P. Chan. A genetic agent-based negotiation system. *Computer Networks*, 37(2):195–204, 2001. DOI: 10.1016/S1389-1286(01)00215-8.

[28] R. M. Coehoorn and N. R. Jennings. Learning on Opponents Preferences to Make Effective Multi-Issue Negotiation Trade-Offs. In *Proceedings of the 6th International Conference On Electronic Commerce*, pages 59–68. ACM Press, 2004

[29] M. Comuzzi, C. Francalanci, and P. Giacomazzi. Trade-off based negotiation of traffic conditioning and service level agreements in diffserv networks. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications*, pages 189–194. IEEE Computer Society, 2005. DOI: 10.1109/AINA.2005.330

[30] C. Crawford, G. Bate, L. Cherbakov, K. Holley, and C. Tsocanos. Toward an on demand service-oriented architecture. *IBM Systems Journal*, 44 (1):81–107, 2005

[31] K. Czajkowski, I. T. Foster, C. Kesselman, V. Sander, and S. Tuecke. SNAP: A Protocol For Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies For Parallel Processing, 8th International Workshop*, volume 2537 of *Lecture Notes in Computer Science*, pages 153–183. Springer, 2002

[32] Q. Dai and R. J. Kauffman. Business models for internet-based b2b electronic markets. *International Journal of Electronic Commerce*, 6(4):41–73, 2002.

[33] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web Services On Demand: WSLA-Driven Automated Management. *IBM Systems Journal*, 43(1): 136–158, 2004.

[34] J. Dang and M. Huhns. Concurrent multiple-issue negotiation for internet-based services. *Internet Computing, IEEE*, 10(6):42–49, 2006. DOI: 10.1109/MIC.2006.118.

[35] S. De Vries and R. Vohra. Combinatorial auctions: A survey. *INFORMS Journal on Computing*, 15(3):284–309, 2003. DOI: 10.1287/ijoc.15.3.284.16077

[36] M. Dean and G. Schreiber. OWL web ontology language reference. http://www.w3.org/TR/owl-ref/, 2004

[37] J. Dujmovic. A Method for Evaluation and Selection of Complex Hardware and Software Systems. *The 22nd Intl Conference for the Resource Management and Performance Evaluation of Enterprise CS. CMG*, 96: 368–378, 1996

[38] A. Elfatatry and P. Layzell. Negotiating in Service-Oriented Environments. *Communications of the ACM*, 47(8):103–108, 2004. DOI: 10.1145/1012037.1012044

[39] A. Elfatatry and P. J. Layzell. A negotiation description language. *Software, Practice and Experience*, 35(4):323–343, 2005. DOI: 10.1002/spe.638

[40] T. Ellingsen. The Evolution of Bargaining Behavior. *The Quarterly Journal of Economics*, 112(2):581–602, 1997

[41] M. Esteva, J. A. Rodríguez-Aguilar, C. Sierra, P. Garcia, and J. L. Arcos. On the formal specifications of electronic institutions. In *Agent Mediated Electronic Commerce, The European AgentLink Perspective*, volume 1991 of *Lecture Notes in Computer Science*, pages 126–147. Springer, 2001

[42] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003. DOI: 10.1145/857076.857078

[43] P. Faratin, C. Sierra, and N. R. Jennings. Using Similarity Criteria to Make Trade-Offs in Automated Negotiations. *Artificial Intelligence*, 142:205–237, 2002. DOI: 10.1016/S0004-3702(02)00290-4

[44] P. Faratin, C. Sierra, and N. R. Jennings. Negotiation Decision Functions For Autonomous Agents. *International Journal of Robotics and Autonomous Systems*, 24(3-4):159–182, 1998. DOI: 10.1016/S0921-8890(98)00029-3

[45] S. Fatima, M. Wooldridge, and N. Jennings. Comparing equilibria for game theoretic and evolutionary bargaining models. In *5th International Workshop on Agent-Mediated E-Commerce*, pages 70–77, 2003. Available at http://eprints.ecs.soton.ac.uk/8568/.

[46] S. S. Fatima, M. Wooldridge, and N. R. Jennings. Bargaining with incomplete information. *Annals of Mathematics and Artificial Intelligence*, 44 (3):207–232, 2005. DOI: 10.1007/s10472-005-4688-7

[47] S. S. Fatima, M. Wooldridge, and N. R. Jennings. An agenda-based framework for multi-issue negotiation. *Artificial Intelligence*, 152(1):1–45, 2004. DOI: 10.1016/S0004-3702(03)00115-2

[48] S. S. Fatima, M. Wooldridge, and N. R. Jennings. A Comparative Study of Game Theoretic and Evolutionary Models of Bargaining For Software Agents. *Artificial Intelligence Review*, 23(2):187–205, 2005. DOI: 10.1007/s10462-004-6391-1.

[49] FIPA. FIPA Dutch Auction Interaction Protocol Specification, 2001. Available at http://standards.computer.org/fipa/specs/fipa00032/XC00032F.pdf

[50] FIPA. FIPA English Auction Interaction Protocol Specification, 2001. Available at http://standards.computer.org/fipa/specs/fipa00031/XC00031F.pdf

[51] FIPA. FIPA Iterated Contract Net Interaction Protocol Specification, 2001. Available at http://www.fipa.org/specs/fipa00030/XC00030F.pdf

[52] FIPA. FIPA Contract Net Interaction Protocol Specification, 2002. Available at http://www.fipa.org/specs/fipa00029/

[53] S. Frølund and J. Koistinen. Quality-of-service specification in distributed object systems. *Distributed Systems Engineering*, 5(4):179–202, 1998. Available at http://www.iop.org/EJ/article/0967-1846/5/4/005/ds8404.pdf

[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994

[55] H. Gimpel, H. Ludwig, A. Dan, and B. Kearney. PANDA: Specifying Policies For Automated Negotiations of Service Contracts. In *International Conference on Service-Oriented Computing*, Lecture Notes in Computer Science, pages 287–302. Springer-Verlag, 2003. DOI: 10.1007/b94513

[56] G. Governatori and Z. Milosevic. Dealing with contract violations: formalism and domain specific language. In *9th IEEE International EDOC Enterprise Computing Conference*, pages 46–57, 2005. DOI: 10.1109/EDOC.2005.13.

[57] G. Governatori. Representing business contracts in ruleml. *International Journal Cooperative Information Systems*, 14(2-3):181–216, 2005. DOI: 10.1142/S0218843005001092

[58] D. G. Gregg and S. Walczak. Auction advisor: an agent-based online-auction decision support system. *Decision Support Systems*, 41(2):449–471, 2006. DOI: 10.1016/j.dss.2004.07.007.

[59] B. N. Grosof and T. C. Poon. Sweetdeal: representing agent contracts with exceptions using xml rules, ontologies, and process descriptions. In *Proceedings of the 12th international conference on World Wide Web*, pages 340–349. ACM Press, 2003. DOI: 10.1145/775152.775200.

[60] Y. Guo, J. P. Muller, and C. Weinhardt. Learning user preferences for multi-attribute negotiation: An evolutionary approach. In *Multi-Agent*

Systems and Applications III: 3rd International Central and Eastern European Conference on Multi-Agent Systems. Proceedings, volume 2691 of *Lecture Notes in Artificial Intelligence*, pages 303–313, 2003. DOI: 10.1007/3-540-45023-8_29

[61] M. He, N. R. Jennings, and H.-F. Leung. On Agent-Mediated Electronic Commerce. *IEEE Transactions On Knowledge and Data Engineering*, 15 (4):985–1003, 2003

[62] P. C. K. Hung, H. Li, and J.-J. Jeng. WS-Negotiation: An Overview of Research Issues. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, 2004. Available at http://portal.acm.org/citation.cfm?id=962749.962847.

[63] IETF. SNego protocol. http://tools.ietf.org/html/draft-ietf-cat-snego-08, 1998

[64] N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, M. Wooldridge, and C. Sierra. Automated Negotiation: Prospects, Methods and Challenges. *Group Decision and Negotiation*, 10:199–215, 2001. DOI: 10.1023/A:1008746126376

[65] J. Jin and K. Nahrstedt. Qos specification languages for distributed multimedia applications: a survey and taxonomy. *Multimedia, IEEE*, 11(3): 74–87, 2004. DOI: 10.1109/MMUL.2004.16

[66] C. Jonker, V. Robu, and J. Treur. An agent architecture for multi-attribute negotiation using incomplete preference information. *Autonomous Agents and Multi-Agent Systems*, 15(2):221–252, 2007. DOI: 10.1007/s10458-006-9009-y.

[67] C. Kalmbach and D. Palmer. *eCommerce and alliances: how eCommerce is affecting alliances in value chain businesses*. Accenture LLP, 2002

[68] A. H. Karp. Rules of engagement for automated negotiation. In *Proceedings of the 1st IEEE International Workshop on Electronic Contracting*, pages 32–39, 2004. DOI: 10.1109/WEC.2004.1319506.

[69] A. H. Karp. Getting Agents to Negotiate Good Deals: A Progress Report. Technical report HPL-2002-161, HP Laboratories, 2002

[70] A. H. Karp. Representing Utility for Automated Negotiation. Technical report HPL-2003-153, HP Laboratories, 2003. Available at http://www.hpl.hp.com/techreports/2003/HPL-2003-153.html.

[71] A. H. Karp. Rules of Engagement For Automated Negotiation. Technical report HPL-2003-152, HP Laboratories, 2003. Available at http://www.hpl.hp.com/techreports/2003/HPL-2003-152.html

[72] A. H. Karp, R. Wu, K.-Y. Chen, and A. Zhang. A Game Tree Strategy For Automated Negotiation. In *Proceedings of the 5th ACM conference on Electronic commerce*, pages 228–229. ACM Press, 2004. DOI: 10.1145/988772.988807

[73] F. Kelly and R. Steinberg. A Combinatorial Auction with Multiple Winners for Universal Service. *Management Science*, 46(4):586–596, 2000

[74] J. B. Kim and A. Segev. A Framework For Dynamic EBusiness Negotiation Processes. In *IEEE International Conference On Electronic Commerce*, pages 84–91. IEEE Computer Society, 2003. Available at http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=1210237.

[75] J. B. Kim and A. Segev. A web services-enabled marketplace architecture for negotiation process management. *Decision Support Systems*, 40(1): 71–87, 2005. DOI: 10.1016/j.dss.2004.04.005.

[76] M. Klein, P. Faratin, H. Sayama, and Y. Bar-Yam. Protocols For Negotiating Complex Contracts. *IEEE Intelligent Systems*, 18(6):32–38, 2003

[77] R. Kowalczyk. Fuzzy e-negotiation agents. *Soft Computing*, 6(5):337–347, 2002. DOI: 10.1007/s00500-002-0187-5.

[78] R. Kowalczyk and V. Bui. On Constraint-Based Reasoning in E-Negotiation Agents. In *Agent-Mediated Electronic Commerce III*, pages 31–46. Springer-Verlag, 2001

[79] K. Kurbel and I. Loutchko. A model for multi-lateral negotiations on an agent-based job marketplace. *Electronic Commerce Research and Applications*, 4(3):187–203, 2005. DOI: 10.1016/j.elerap.2005.01.002.

[80] C. Li, J. Giampapa, and K. Sycara. Bilateral negotiation decisions with uncertain dynamic outside options. In *Proceedings of the 1st IEEE International Workshop on Electronic Contracting*, pages 54–61, 2004. DOI: 10.1109/WEC.2004.1319509.

[81] C. Li, J. Giampapa, and K. Sycara. Bilateral negotiation decisions with uncertain dynamic outside options. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 36(1):31–44, 2006. DOI: 10.1109/TSMCC.2005.860573.

[82] P. F. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal. A unified behavioural model and a contract language for extended enterprise. *Data & Knowledge Engineering*, 51(1):5–29, 2004. DOI: 10.1016/j.datak.2004.03.005

[83] G. Lodi, F. Panzieri, D. Rossi, and E. Turrini. SLA-Driven clustering of qos-aware application servers. *IEEE Transactions on Software Engineering*, 33(3):186–197, 2007. DOI: 10.1109/TSE.2007.28.

[84] A. Ludwig, P. Braun, R. Kowalczyk, and B. Franczyk. A framework for automated negotiation of service level agreements in services grids. In *Business Process Management Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 89–101. Springer, 2006. DOI: 10.1007/11678564_9.

[85] H. Ludwig, A. Dan, and R. Kearney. Cremona: An Architecture and Library For Creation and Monitoring of WS-Agreements. In *Proceedings of the 2nd International Conference On Service Oriented Computing*. ACM Press, 2004

[86] X. Luo, N. R. Jennings, and N. Shadbolt. Acquiring tradeoff preferences for automated negotiations: A case study. In *Agent-Mediated Electronic Commerce V*, volume 3048 of *Lecture Notes in Computer Science*, pages 37–55. Springer, 2004. Available at http://www.springerlink.com/content/fbj3j7a6fmnelyry

[87] X. Luo, N. R. Jennings, N. Shadbolt, H.-F. Leung, and J. H. Lee. A fuzzy constraint based model for bilateral, multi-issue negotiations in semi-competitive environments. *Artificial Intelligence*, 148(1-2):53–102, 2003. DOI: 10.1016/S0004-3702(03)00041-9

[88] O. Martín-Díaz. *Emparejamiento Automatico de Servicios Web usando Programacion con Restricciones*. PhD thesis, Universidad de Sevilla, 2007

[89] A. Mas-Colell. Nash equilibrium and economics: Remarks. *Computer Science Review*, 1(2):100–102, 2007. DOI: 10.1016/j.cosrev.2007.10.002

[90] N. Matos, C. Sierra, and N. R. Jennings. Determining successful negotiation strategies: an evolutionary approach. In Y. Demazeau, editor, *Proceedings of the 3rd International Conference on Multi-Agent Systems*, pages 182–189. IEEE Press, 1998. Available at http://citeseer.ist.psu.edu/matos98determining.html

[91] P. McBurney, R. M. V. Eijk, S. Parsons, and L. Amgoud. A dialogue game protocol for agent purchase negotiations. *Autonomous*

Agents and Multi-Agent Systems, 7(3):235–273, 2003. DOI: 10.1023/A:1024787301515

[92] C. Molina-Jimenez, S. Shrivastava, and J. Warne. A method for specifying contract mediated interactions. In *9th IEEE International EDOC Enterprise Computing Conference*, pages 106–115, 2005. DOI: 10.1109/EDOC.2005.1

[93] C. Molina-Jimenez, J. Pruyne, and A. van Moorsel. The Role of Agreements in IT Management Software. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems III*, volume 3549 of *Lecture Notes in Computer Science*, pages 36–58. Springer-Verlag GmbH, 2005. DOI: 10.1007/11556169_2.

[94] R. Myerson and M. Satterthwaite. Efficient Mechanisms for Bilateral Trading. *Journal of Economic Theory*, 29(2):265–281, 1983

[95] J. Nash. The Bargaining Problem. *Econometrica*, 18(2):155–162, 1950

[96] J. Nash. Non-cooperative games. *The Annals of Mathematics*, 54(2): 286–295, 1951

[97] T. D. Nguyen and N. R. Jennings. Reasoning About Commitments in Multiple Concurrent Negotiations. In *Proceedings of the 6th International Conference On E-Commerce*, pages 77–84, 2004

[98] T. Nguyen and N. Jennings. Managing commitments in multiple concurrent negotiations. *Electronic Commerce Research and Applications*, 4:362–376, 2005. Available at http://www.ecs.soton.ac.uk/~nrj/download-files/ecra05.pdf

[99] N. Nisan. Bidding and allocation in combinatorial auctions. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 1–12. ACM Press, 2000. DOI: 10.1145/352871.352872

[100] N. Oldham, K. Verma, A. Sheth, and F. Hakimpour. Semantic ws-agreement partner selection. In *Proceedings of the 15th international conference on World Wide Web*, pages 697–706. ACM Press, 2006. DOI: 10.1145/1135777.1135879

[101] L. Olsina and G. Rossi. Measuring web application quality with webqem. *IEEE MultiMedia*, 9(4):20–29, 2002. DOI: 10.1109/MMUL.2002.1041945

[102] OMG. Negotiation facility specification, 2002. Available at http://www.omg.org/cgi-bin/doc?formal/2002-03-14

[103] A. Ordanini. What drives market transactions in b2b exchanges? *Communications of the ACM*, 49(4):89–93, 2006. DOI: http://doi.acm.org/10.1145/1121949.1121953

[104] S. Paurobally, P. J. Turner, and N. R. Jennings. Automating Negotiation For M-Services. *IEEE Transactions On Systems, Man, and Cybernetics, Part A*, 33(6):709–724, 2003

[105] J. Pérez, M. Bravo, R. Pazos, G. Reyes, J. Frausto, V. Sosa, and M. López. Design of a shared ontology used for translating negotiation primitives. In *International Conference on Computational Science and Its Applications*, volume 3983 of *Lecture Notes in Computer Science*, pages 169–178, 2006. DOI: 10.1007/11751632_18

[106] S. Phelps, V. Tamma, M. Wooldridge, and I. Dickinson. Toward Open Negotiation. *IEEE Internet Computing*, 8(2):70–75, 2004

[107] I. Rahwan, S. D. Ramchurn, N. R. Jennings, P. McBurney, S. Parsons, and L. Sonenberg. Argumentation-Based Negotiation. *The Knowledge Engineering Review*, 18(4):343–375, 2003. DOI: 10.1017/S0269888904000098

[108] H. Raiffa. *The art and science of negotiation*. Harvard University Press, 1982

[109] S. D. Ramchurn, N. R. Jennings, and C. Sierra. Persuasive Negotiation For Autonomous Agents: A Rhetorical Approach. In *Procedings of the IJCAI Workshop On Computational Models of Natural Argument*, pages 9–17, 2003

[110] M. Resinas, P. Fernández, and R. Corchuelo. Towards automated service trading. In *Proceedings Of The International Conference On E-Business*, pages 38–45, 2006.

[111] M. Resinas, P. Fernández, and R. Corchuelo. An analysis of service trading architectures. In K. Bauknecht, B. Pröll, and H. Werthner, editors, *7th International Conference on E-Commerce and Web Technologies*, volume 4082 of *Lecture Notes in Computer Science*, pages 203–212. Springer, 2006. DOI: 10.1007/11823865_21.

[112] A. Reyes-Moro, J. Rodríguez-Aguilar, M. López-Sánchez, J. Cerquides, and D. Gutiérrez-Magallanes. Embedding decision support in e-sourcing tools: Quotes, a case study. *Group Decision and Negotiation*, V12(4):347–355, 2003. DOI: 10.1023/A:1024824005214.

[113] J. Reynolds and R. Mofazali. *The complete e-commerce book: design, build and maintain a successful web-based business*. CMP Books, 2000

[114] S. Rinderle and M. Benyoucef. Towards the automation of e-negotiation processes based on web services - a modeling approach. In *6th International Conference on Web Information Systems Engineering*, volume 3806 of *Lecture Notes in Computer Science*, pages 443–453. Springer, 2005. DOI: 10.1007/11581062_34.

[115] R. Ros and C. Sierra. A negotiation meta strategy combining trade-off and concession moves. *Autonomous Agents and Multi-Agent Systems*, 12(2):163–181, 2006. DOI: 10.1007/s10458-006-5837-z.

[116] A. Ruiz-Cortés, O. Martín-Díaz, A. Durán-Toro, and M. Toro. Improving the automatic procurement of web services using constraint programming. *International Journal Cooperative Information Systems*, 14(4): 439–468, 2005. DOI: 10.1142/S0218843005001225

[117] T. Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, 135(1-2):1–54, 2002. DOI: 10.1016/S0004-3702(01)00159-X

[118] T. Sandholm and V. Lesser. Leveled commitment contracts and strategic breach. *Games and Economic Behavior*, 35(1):212–270, 2001.

[119] C. Sierra, P. Faratin, and N. R. Jennings. A Service-Oriented Negotiation Model Between Autonomous Agents. In *Proceedings of the 8th European Workshop On Modelling Autonomous Agents in a Multi-Agent World*, pages 17–35. Springer-Verlag, 1997

[120] C. Sierra, N. R. Jennings, P. Noriega, and S. Parsons. A framework for argumentation-based negotiation. In *Proceedings of the 4th International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages*, pages 177–192. Springer-Verlag, 1998. Available at http://veracruz.lania.mx/~pablo/articles/jennings.pdf

[121] K. M. Sim and C. Y. Choi. Agents that react to changing market situations. *Systems, Man and Cybernetics, Part B, IEEE Transactions on*, 33 (2):188–201, 2003. DOI: 10.1109/TSMCB.2002.805694.

[122] K. M. Sim and S. Y. Wang. Flexible negotiation agent with relaxed decision rules. *Systems, Man and Cybernetics, Part B, IEEE Transactions on*, 34(3):1602–1608, 2004. DOI: 10.1109/TSMCB.2004.825935.

[123] K. M. Sim and E. Wong. Toward market-driven agents for electronic auction. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, 31(6):474–484, 2001. DOI: 10.1109/3468.983399.

[124] J. M. Smith and G. R. Price. The logic of animal conflict. *Nature*, 246 (5427):15–18, 1973. DOI: 10.1038/246015a0

[125] J. M. Smith. *Evolution and the Theory of Games*. Cambridge University Press, 1982

[126] R. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29 (12):1104–1113, 1980. DOI: http://doi.ieeecomputersociety.org/10.1109/TC.1980.1675516

[127] M. Ströbel and C. Weinhardt. The Montreal Taxonomy for Electronic Negotiations. *Group Decision and Negotiation*, 12(2):143–164, 2003. Available at http://www.ingentaconnect.com/content/klu/grup/2003/00000012/00000002/05119337.

[128] M. Ströbel. Design of roles and protocols for electronic negotiations. *Electronic Commerce Research*, 1(3):335–353, 2001. DOI: 10.1023/A:1011554323604.

[129] S. Y. Su, C. Huang, J. Hammer, Y. Huang, H. Li, L. Wang, Y. Liu, C. Pluempitiwiriyawej, M. Lee, and H. Lam. An internet-based negotiation server for e-commerce. *The VLDB Journal on Very Large Data Bases*, 10(1):72–90, 2001. DOI: 10.1007/s007780100051

[130] J. Teich, H. Wallenius, and J. Wallenius. Multiple-issue auction and market algorithms for the world wide web. *Decision Support Systems*, 26 (1):49–66, 1999. DOI: 10.1016/S0167-9236(99)00016-0.

[131] J. E. Teich, H. Wallenius, J. Wallenius, and A. Zaitsev. Designing electronic auctions: An internet-based hybrid procedure combining aspects of negotiations and auctions. *Electronic Commerce Research*, V1(3):301–314, 2001. DOI: 10.1023/A:1011550222695

[132] V. Tosic, B. Pagurek, K. Patel, B. Esfandiari, and W. Ma. Management applications of the web service offerings language (wsol). *Information Systems*, 30(7):564–586, 2005. DOI: 10.1016/j.is.2004.11.005.

[133] D. Trastour, C. Bartolini, and C. Preist. Semantic Web Support For the Business-to-Business E-Commerce Pre-Contractual Lifecycle. *Computing Networks*, 42(5):661–673, 2003

[134] D. Tsichritzis. *Electronic commerce*. Centre Universitaire pur laInformatique (University of Geneva), 1998

[135] M. T. Tu, F. Griffel, M. Merz, and W. Lamersdorf. A Plug-in Architecture Providing Dynamic Negotiation Capabilities For Mobile Agents. In *Mobile Agents: Second International Workshop. Proceedings.*, volume 1477 of *Lecture Notes in Computer Science*, pages 222–236. Springer-Verlag, 1998. DOI: 10.1007/BFb0057642

[136] M. Tu, C. Seebode, F. Griffel, and W. Lamersdorf. Dynamics: An actor-based framework for negotiating mobile agents. *Electronic Commerce Research*, 1(1 - 2):101–117, 2001. DOI: 10.1023/A:1011575629387

[137] H. P. Young. An evolutionary model of bargaining. *Journal of Economic Theory*, 59(1):145–168, 1993.

[138] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodologies*, 12(3):317–370, 2003. DOI: 10.1145/958961.958963

[139] D. Zeng and K. Sycara. Bayesian Learning in Negotiation. *International Journal Human-Computer Studies*, 48(1):125–141, 1998

This document was typeset on 2008/7/11 using RC–BOOK α2.11 for LaTeX$_{2\epsilon}$.
Should you want to use this document class, please send mail to
contact@tdg-seville.info.