Science of Computer Programming 77 (2012) 4–28

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A bargaining-specific architecture for supporting automated service agreement negotiation systems

Manuel Resinas *, Pablo Fernández, Rafael Corchuelo

Universidad de Sevilla, ETSI Informática, Avda. Reina Mercedes, s/n, E-41012 Sevilla, Spain

ARTICLE INFO

Article history:
Received 25 February 2009
Received in revised form 3 September 2010
Accepted 7 September 2010
Available online 1 November 2010

Keywords:
Software architecture
Automated negotiation
Service agreements

ABSTRACT

The provision of services is often regulated by means of agreements that must be negotiated beforehand. Automating such negotiations is appealing insofar as it overcomes one of the most often cited shortcomings of human negotiation: slowness. Our analysis of the requirements of automated negotiation systems in open environments suggests that some of them cannot be tackled in a protocol-independent manner, which motivates the need for a protocol-specific architecture. However, current state-of-the-art bargaining architectures fail to address all of these requirements together. Our key contribution is a bargaining architecture that addresses all of the requirements we have identified. The definition of the architecture includes a logical view that identifies the key architectural elements and their interactions, a process view that identifies how the architectural elements can be grouped together into processes, a development view that includes a software framework that provides a reference implementation developers can use to build their own negotiation systems, and a scenarios view by means of which the architecture is illustrated and validated.

1. Introduction

Agreements play a major role to regulate both functional and non-functional properties, as well as guarantees regarding the provisioning of a service [1,8,23,26]. Many authors have focused on automating the negotiation of such agreements as a means to improve efficiency and benefit from the many opportunities that electronic businesses bring [25,37]. Developing an automated negotiation system requires selecting and integrating the most appropriate negotiation protocol, decision-making algorithm, and data model. This selection depends on the context of the negotiation, and it has usually been accomplished in ad hoc manners [16].

Our work focuses on open environments in which change is the major driving force. This makes ad hoc solutions of little interest because of the development costs incurred. The reason for changes in open environments is manifold: on the one hand, parties may appear or disappear unexpectedly, it is not usually possible to have complete information about them, they may implement a variety of negotiation protocols, and they may have diverging behaviours during a negotiation; on the other hand, users' requirements are subject to continuous changes due to new business rules and regulations, new types of business-related events, or new operations [29].

In Ref. [32], we identified a number of objective requirements that allowed us to compare current proposals regarding service agreement negotiation in open environments [2,3,6,17,14,19,24,30,33,38,40]. The conclusion was that none of the eleven proposals surveyed was adequate to negotiate service agreements in open environments. To overcome this issue, the NegoFAST-Core architecture was developed; it defines the most abstract architectural elements of an automated negotiation system from a protocol-independent and technology-agnostic point of view. Unfortunately, although some of

* Corresponding author.
E-mail address: resinas@us.es (M. Resinas).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.09.008

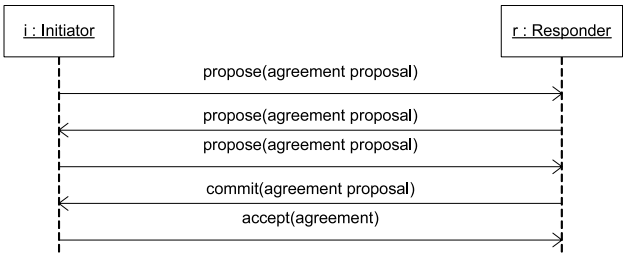

Fig. 1. Sequence diagram of a typical interaction in a bargaining protocol.

the requirements can be satisfied using protocol-independent architectural elements, e.g., acquiring information from other parties, modelling the world, or negotiating a protocol, a few requirements are protocol-dependent and require specific architectural elements.

The contribution that we present in this article is an architecture called NegoFAST-Bargaining that refines and complements the NegoFAST-Core architecture by defining protocol-specific architectural elements to deal with bargaining protocols.

On the one hand, having a software architecture [22] for a problem domain is appealing insofar it provides an abstract, reusable, easy-to-maintain, and easy-to-adapt design that provides the harness developers need to face the development of a system without incurring the cost of developing it from scratch or in ad hoc manners. Furthermore, a software architecture provides a common vocabulary that bridges the gap between the many existing terminologies, which, therefore, helps ease the communication amongst developers. Note that we use the term 'architecture' in the same sense as in Kruchten's 's seminar article [22], according to which an architecture is composed of $4 + 1$ views, namely: a logical view, which details the functional view of the system; a process view, which describes how the architectural elements can be grouped together into processes; a development view, which describes the organisation of the software modules in the software development environment; a scenarios view in which the architecture is illustrated; and a physical view, which describes how a system must be deployed.

On the other hand, bargaining protocols are in widespread use [5,16,36]. Such protocols have the following features: they are bilateral, which means that they are carried out between two parties that play the roles of initiator and responder; they are sequential, which means that the same party cannot send two messages in a row, except for negotiation messages that include the rejectNegotiation or the withdraw performatives, which can be sent at any time; finally, messages are proposal-based, which means that they contain one performative and one or more proposals, but they do not include any arguments to convince the other party about accepting a proposal. Fig. 1 shows a sequence diagram that illustrates a typical execution of a bargaining protocol; note that both parties exchange several proposals using performative propose until one of them decides to send a binding proposal by means of performative commit, which is accepted by the other party by means of performative accept.

The rest of the article is organised as follows: in Section 2, we summarise our requirements, with an emphasis on the subset of protocol-specific ones, and prove that none of the state-of-the-art proposals that we have surveyed address all of these requirements together; then, the Kruchten architectural views [22] of our proposal are presented in Sections 3 (logical view), 4 (process view), 5 (development view), and 6 (scenarios view). (Note that we do not provide a physical view as such, since our proposal is not intended to be the architecture of a specific system, but a reusable architecture that can be used to devise and implement a variety of actual negotiation systems.) We report on our conclusions and future research paths in Section 7. Finally, Appendix provides an overview of NegoFAST-Core.

2. Motivation

In this section, we first report on a number of key problems and requirements with which a service agreement negotiator must deal in open environments; we then analyse these requirements and conclude that some of them cannot be tackled in a protocol-independent manner; finally, we survey current related proposals and conclude that none of them addresses all of the protocol-specific requirements we have identified, which motivated us to work on a new proposal.

2.1. Key problems and requirements

Automated negotiation systems must cope with four problems when facing automated negotiations of service agreements in open environments. These problems led to the formulation of a number of requirements that are summarised in Table 1 [32], namely:

Negotiations are multi-term. The negotiations of a typical service agreement involves terms like availability, response time, security, or price, to name a few. This has an influence on the negotiation protocol and the agreement preferences that the automated negotiation system must support. On the one hand, it makes it desirable for an automated negotiation system to support multi-term negotiation protocols (REQ 1.1), such as most (if not every) bargaining

Table 1
Requirements in automated negotiation systems.

Negotiations are multi-term	
REQ	Description
1.1	Support multi-term negotiation protocols.
1.2	Manage expressive agreement preferences.
Parties are heterogenous	
REQ	Description
2.1	Support multiple negotiation protocols.
2.2	Negotiate the negotiation protocol in a pre-negotiation phase in which a negotiation protocol is agreed.
2.3	Support multiple decision-making algorithms to face the different behaviours of the other parties during the negotiation.
2.4	Allow user preferences about the negotiation process to be changed over time.
Partial information about parties	
REQ	Description
3.1	Manage different types of knowledge about the other parties.
3.2	Support diverse query capabilities.
3.3	Build analysis-based models of parties.
Markets are dynamic	
REQ	Description
4.1	Support several negotiations with different parties at the same time.
4.2	Select decision-making algorithms dynamically according to the evolution of the simultaneous negotiations.
4.3	Build market models.
4.4	Support decommitment from previously established agreements.
4.5	Supervised creation of agreements to avoid committing to agreements that cannot be satisfied.

protocol [16]; on the contrary, most auctioning protocols allow us to negotiate on a unique term only, usually the price [15], except for the proposal by Bichler [7]. On the other hand, user preferences can be expressed in a variety of ways, e.g., utility functions [11], combinations of attributes [9], or fuzzy constraints [25], but multi-term negotiations require the management of expressive agreement preferences regarding multiple terms (REQ 1.2) so that they capture the relationships between terms and, hence, enable making trade-offs during negotiations.

Parties are heterogenous. In an open environment, the parties that may get involved in a negotiation process are obviously expected to be heterogeneous, not only regarding their technology, but also their semantics and behaviours. It is then desirable for an automated negotiation system to support multiple negotiation protocols (REQ 2.1), to be able to negotiate the negotiation protocol (REQ 2.2) in cases in which a party supports several of them, support multiple decision-making algorithms (REQ 2.3) to adapt to different behaviours of the other parties, and to allow user preferences about the negotiation process (REQ 2.4), e.g., deadline or eagerness.

Partial information about parties. Having as much information as possible about the other parties is important to strengthen one's negotiation capabilities [43]. Unfortunately, more often than not, automated negotiation systems have only partial information about their context [25]. As a conclusion, it is important that such systems can manage different types of knowledge about the other parties (REQ 3.1), e.g., whether they tend to concede [28], their negotiation deadline, their reputation or geographical location; it is also important to be able to support a variety of query capabilities (REQ 3.2), e.g., a reputation provider, an ad hoc API, or a WS-Agreement template [1]; finally, it must be able to build analysis-based models of parties (REQ 3.3) building on the previous information and on previous negotiation processes, since this shall definitely help make better decisions in the future [43].

Markets are dynamic. Services are not storable, which means that resources not used yesterday are worthless today [14]. This is the reason why service markets tend to be extremely dynamic. As a consequence, it would be convenient for an automated negotiation system to support several negotiations with different parties at the same time (REQ 4.1), to select decision-making algorithms dynamically (REQ 4.2), to support decommitment from previously established agreements (REQ 4.3), to supervise the creation of agreements (REQ 4.4), and to build market models (REQ 4.5). These requirements allow negotiation systems to choose the party with which the most profitable agreement can be made, to change its behaviour with regard to specific parties according to how the negotiation is going on globally, to revoke an agreement in case another that is more profitable is found [35], and not to commit to an agreement that cannot be satisfied or that might not be satisfactory.

2.2. Protocol-specific requirements

Negotiation protocols govern how negotiations are performed. They have a strong influence on the decision-making techniques used by the parties and the final outcome of the process.

Although there are a variety of negotiation protocols, they all may be analysed in terms of five fundamental aspects, namely: parties, which are commonly a consumer and a provider, although some negotiation protocols require a mediator to facilitate the negotiation process [19,20]; performatives, which reflect the intention of a message, e.g., propose, accept, or commit; rules, which express restrictions regarding how proposals are built or when a party can send a proposal; information exchanged, which is the type of information exchanged during a negotiation; and agreement terms negotiability, which is the set of terms that are negotiable.

Negotiation protocols can be grouped into a number of families, namely: bargaining protocols [10], which involve exchanging proposals and counter-proposals between the parties; auction protocols [15], which involve one or more parties called auctioneers, who start the auction, and other parties called bidders that bid following a protocol that may allow one or several rounds; and argumentation-based protocols [31], which involve exchanging arguments to persuade the other party to accept an offer.

This variability in negotiation protocols causes that some of the requirements described in the previous section must be dealt with in a protocol-specific manner, namely:

- Support multi-term negotiation protocols (REQ 1.1) and support multiple negotiation protocols (REQ 2.1). Although most negotiation protocols involve the exchange of proposals between parties, this exchange may be carried out without restrictions on the contents of the proposal and counter-proposal [10] or with restrictions on the order in which terms are negotiated [12] or on the terms of the proposals [18]. This variability must be dealt with in a protocol-specific manner.
- Support several decision-making algorithms (REQ 2.3) and select them dynamically (REQ 4.2). The decisions that must be made by an automated negotiation system can be divided into two groups. First, it has to handle the commitment to new agreements, i.e., deciding if an agreement is acceptable and convenient to commit to, and the decommitment from previously created agreements [28,35]. Second, it has to generate responses to the other parties in the negotiation by implementing an appropriate algorithm [10,11,21,25]. The first decision is protocol-independent. However, depending on the negotiation protocol, the way decisions are made regarding the generation of responses can be significantly different. For instance, in a bargaining protocol, the response has to be requested for each negotiation message received, whereas in an auction protocol, bids can be placed at any moment. As a consequence, supporting decision-making algorithms must be designed in a protocol-specific manner.
- Support several negotiations simultaneously (REQ 4.1) and allow user preferences about the negotiation process (REQ 2.4). On the one hand, it is necessary to have a global view of all negotiations to support them properly; on the other hand, allowing user preferences about the negotiation process involves changing the negotiation behaviour based on the user's preferences. Consequently, the behaviour of each negotiation must be guided based on how well the other simultaneous negotiations are performing and on the user preferences. To this end, it is necessary to have an understanding of how the decision-making algorithms work. Therefore, the changes in the behaviour must be designed in a protocol-specific manner.

2.3. Analysis of the related work

The key problems described above provide a number of objective requirements that can be used to compare current state-of-the-art automated negotiation architectures. Table 2 summarises our comparison. An '✓' in a cell means that the corresponding proposal provides explicit support for the corresponding requirement; a '∼' indicates that it is supported partially; a '✗' indicates that the feature is not supported; 'NA' means that there is no information available. (Note that we do not take into account proposals like the ones described in [21,39] because they are specific-purpose negotiation systems, not software architectures.)

A quick look at Table 2 reveals that none of the state-of-the-art proposals we have surveyed is complete with regard to the protocol-dependent requirements we have identified, which justifies the need for research regarding an architecture that addresses them all. Next we provide additional details on the comparison. For the sake of homogeneity, we have grouped them into protocol-oriented and intelligence-oriented proposals.

Protocol-oriented proposals focus on providing support to deal with the negotiation protocol and low-level interoperability issues. However, they do not provide any kind of support to implement decision making algorithms. Protocol-oriented proposals can be divided into two categories: some of them define a negotiation host or marketplace that acts as a mediator between the negotiating parties [19,33,38]; contrarily, others just describe the elements that are required to manage negotiation protocols properly [3]. Kim and Segev [19] presented a proposal that describes a web services-enabled marketplace architecture that defines executable negotiation protocols in BPEL. Rinderle and Benyoucef [33] followed a similar approach, but negotiation protocols are specified as statecharts and later mapped onto BPEL. Similarly, Silkroad [38] relies on a meta-model called roadmap that is intended to capture the characteristics of a negotiation process, and an application framework called skeleton that provides several modular and configurable negotiation service components. Contrary to these proposals, Bartolini et al. [3] presented a taxonomy of rules to capture a variety of negotiation mechanisms and a simple interaction protocol based on FIPA specifications that is used together with the rules to define negotiation protocols; they also defined a set of roles and an OWL-based language to express negotiation proposals and agreements.

Intelligence-oriented proposals try to make the development of automated negotiation systems easier by giving a common grounding for developing decision making algorithms and improving reusability. Furthermore, some of these

Table 2
Comparison of automated negotiation architectures.

Protocol-oriented architectures						
Proposal	Requirement					
	1.1	2.1	2.3	2.4	4.1	4.2
Kim and Segev [19]	✓	✓	✗	✗	✗	✗
Rinderle and Benyoucef [33]	✓	✓	✗	✗	✗	✗
Bartolini et al. [3]	✓	✓	✗	✗	✗	✗
Silkroad [38]	✓	✓	✗	✗	✗	✗
Intelligence-oriented architectures						
Proposal	Requirement					
	1.1	2.1	2.3	2.4	4.1	4.2
Ashri et al. [2]	✓	✓	NA	NA	NA	NA
Ludwig et al. [24]	✓	✓	✓	✗	✗	✗
PANDA [14]	✓	✓	✓	✓	∼	✗
DynamiCS [40]	✓	✓	✓	NA	✗	✗
Benyoucef and Verrons [6]	NA	✓	✓	NA	✗	✓
Jonker et al. [17]	✓	NA	✓	✓	✗	✗
Paurobally et al. [30]	✓	✗	∼	✗	∼	✗

1.1 Support multi-term negotiation protocols.
2.1 Multiple protocol support.
2.3 Multiple decision-making algorithms.
2.4 Allow user preferences about negotiation processes.
4.1 Support several negotiations simultaneously.
4.2 Select decision-making algorithms dynamically.

proposals [2,14,40] also deal with protocol management. However, instead of dealing with the interactions amongst automated negotiation systems like protocol-oriented proposals, they focus on decoupling the protocol and the intelligence algorithms to make the latter compatible with a number of different protocols. Ashri et al. [2] described an agent-oriented architecture at a very high level of abstraction, which means that they left many aspects related to the decision-making algorithms unspecified. Ludwig et al. [24] presented an architecture for service agreement negotiation in service grids. It builds on WS-Agreement and provides a protocol service provider and a decision making service provider to deal with the negotiation process. Unfortunately, it does not support several simultaneous negotiations or changing the decision-making algorithms dynamically. PANDA [14] is an architecture that mixes utility functions and rules to carry out the decision-making process. The decision-making component relies on rules, utility functions and an object pool with several estimation libraries, the negotiation history and the current offer. However, it does not allow us to select decision-making algorithms dynamically and, although it claims to support several simultaneous negotiations, the coordination mechanism amongst them is not described. DynamiCS is an actor-based architecture that was devised by Tu et al. [40]. It 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. However, it does not support several simultaneous negotiations or provide support to change the decision-making algorithms at run time. Benyoucef and Verrons [6] presented an architecture that is based on the separation of protocols and strategies within a service-oriented architecture that makes the deployment and integration with current infrastructures easier. However, the services that can be composed into the system are vaguely defined. In addition, it does not provide any mechanisms to allow several simultaneous negotiations. Jonker et al. [17] described a component-based generic agent architecture for multi-attribute negotiation. It copes with multi-term negotiations successfully. However, it is not clear if it supports several negotiation protocols and it does not provide a mechanism to coordinate several simultaneous negotiations or to change the decision-making algorithm at run time. Paurobally et al. [30] described an architecture for web service negotiation in grids. It deals with the expressiveness of service agreements successfully, and it allows agreements to be created in a supervised manner by means of the so-called WS-DAIOnt service; unfortunately, it only supports a negotiation protocol, and, although the architecture seems to support several simultaneous negotiations, the authors do not delve into these details.

3. The NegoFAST-Bargaining logical view

According to Kruchten [22], the logical view of an architecture provides an insight into the functional view of the system, which includes both the static and dynamic models of its architectural elements. We used the Gaia methodology and its organisational metaphor to work on this view [42]. The Gaia methodology decomposes the logical view of an architecture into organisations, roles, interactions, and environmental resources. 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 amongst two or more roles; the environment determines the resources that roles can use, control, or consume to achieve an organisational goal.

As mentioned in the introduction, NegoFAST-Core [32] defines the most abstract architectural elements of an automated negotiation system from a protocol-independent and technology-agnostic point of view, whereas NegoFAST-Bargaining

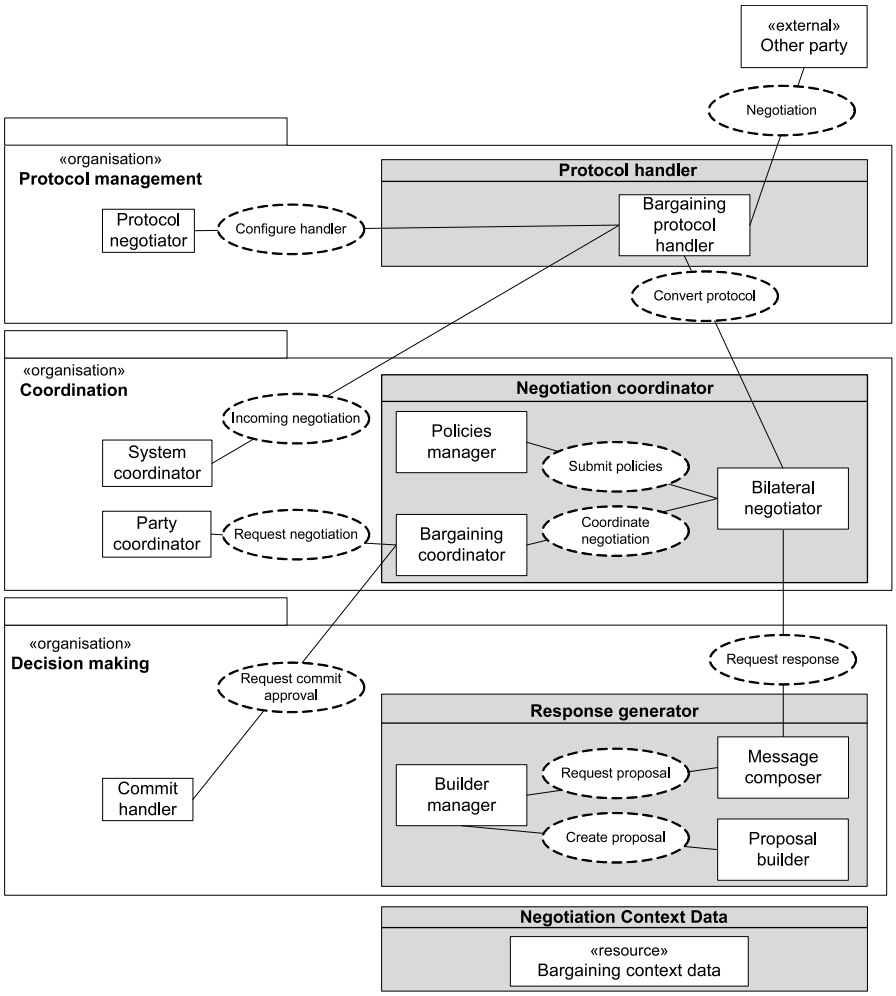

Fig. 2. Logical view of NegoFAST-Bargaining.

refines and complements it by means of bargaining-specific elements. (Cf. Appendix for a short introduction to NegoFAST-Core.) Consequently, the logical view of NegoFAST-Bargaining is organised into those protocol-specific roles of architectural elements, namely: ProtocolHandler, which provides support for several multi-term negotiation protocols, NegotiationCoordinator, which coordinates the ProtocolHandler and the ResponseGenerator to handle several negotiations simultaneously, ResponseGenerator, which decides the most appropriate message to answer an incoming negotiation message, and NegotiationContextData, which stores information related to the negotiations that are being carried out by the automated negotiation system.

NegoFAST-Bargaining refines the previous architectural elements into the following bargaining-specific roles, cf. Fig. 2:

- Although different, all negotiation protocols supported by one protocol-specific architecture have many commonalities, e.g., all bargaining protocols involve the sequential exchange of proposals between two parties. As a consequence, the support for multi-term negotiation protocols (REQ 1.1) and several bargaining negotiation protocols (REQ 2.1) is dealt with by defining a generic bargaining protocol and specialising the ProtocolHandler in a BargainingProtocolHandler, which converts specific bargaining protocols into the generic one.
- Since all negotiation protocols supported by a protocol-specific architecture use a common set of performatives and the same type of message contents, e.g., a proposal composed of a set of terms, the support for multiple decision-making algorithms (REQ 2.3) and the dynamic selection of such algorithms (REQ 4.2) is tackled by dividing the ResponseGenerator into the MessageComposer, which selects the performative that is going to be used and composes the message, and the BuilderManager and ProposalBuilder, which deal with the creation of a proposal.
- The support for several simultaneous bargaining negotiations (REQ 4.1) and for allowing user preferences about the negotiation process (REQ 2.4) is implemented by introducing the concept of negotiation policy to guide the generation of responses based on how well the other negotiations are performing and the user preferences about the negotiation pro-

cess. To this end, the NegotiationCoordinator is divided into a role that is responsible for those policies (PoliciesManager), and two roles (BilateralNegotiator and BargainingCoordinator) that deal with the coordination task.

These elements must interact with several NegoFAST-Core protocol-independent roles that are depicted on the left side of Fig. 2, namely: the ProtocolNegotiator, which decides the concrete negotiation protocol that shall be used during a negotiation and configures the ProtocolHandler according to it; the SystemCoordinator and the PartyCoordinator, which coordinate when new protocol-specific negotiations must be started, and the CommitHandler, which must approve the submission of all binding negotiation messages, i.e., those messages that involve a firm commitment with the other party. Furthermore, the generation of responses must be provided with information stored in several different protocol-independent environmental resources, namely: the user's preferences, the world model, the negotiation history, and other aspects related to the current state of the automated negotiation system.

The way in which the protocol-specific requirements are tackled shape the logical view of NegoFAST-Bargaining, which follows a hybrid architectural style [4]. On the one hand, it uses a centralised control structure in which the BargainingCoordinator and the BilateralNegotiator play the main roles: the former focuses on the interaction with protocol-independent elements of an automated negotiation system, whereas the latter focuses on the interaction with the elements of the protocol-specific architecture. On the other hand, it uses the repository style, in which the NegotiationContextData stores and provides information generated by the other elements of the architecture for each negotiation. The NegotiationContextData also uses the publisher–subscriber pattern to act as an active repository and notify the other elements of the architecture about the occurrence of a number of events. This is particularly useful to the PoliciesManager because its behaviour depends on how the concurrent negotiations are evolving. Finally, the plug-in architectural pattern [13] is used to allow several BargainingProtocolHandlers and ProposalBuilders at the same time to enable the support of several bargaining negotiation protocols and several decision-making algorithms to create proposals. The BuilderManager allows the selection of the ProposalBuilder that better suits the current negotiation scenario, whereas the selected BargainingProtocolHandler depends on the specific bargaining negotiation protocol that is being executed with the other party.

3.1. Protocol handler

In NegoFAST-Bargaining, the ProtocolHandler remains as one role called BargainingProtocolHandler that deals with the interaction with the other parties following a concrete bargaining protocol. It is configured to manage a bargaining protocol by means of interaction ConfigureHandler, but adapts it to the generic 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 and sending them by means of interaction ConvertProtocol, 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 errors, e.g., missing messages, messages that arrive too late, or unexpected messages. In addition, it may also receive requests from other parties to start negotiations, in which case it forwards them to the SystemCoordinator by means of interaction IncomingNegotiation.

Below, we report on the key features of our design:

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 in our generic protocol are cfp (call for proposals), propose, commit, rejectNegotiation, rejectProposal, withdraw and accept. Note that, depending on the performative, negotiation messages can be classified as binding negotiation messages, which involve a firm commitment with the other party, e.g., performative commit or accept, and non-binding negotiation messages, which do not involve such a firm commitment, e.g., performative propose.

Rules. The rules imposed by the generic 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 rejectNegotiation or withdraw performatives that can be sent at any time during the negotiation. These rules are formalised as the protocol state-machine in Fig. 3.

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 generic protocol does not impose any restriction on the negotiability of agreement terms. Therefore, as is the case for most bargaining protocols, every term can be negotiated.

3.2. Negotiation coordinator

The NegotiationCoordinator coordinates the negotiations by linking the ProtocolHandler with the ResponseGenerator and the CommitHandler, cf. Fig. 2. Note that, although the CommitHandler is a protocol-independent role, the

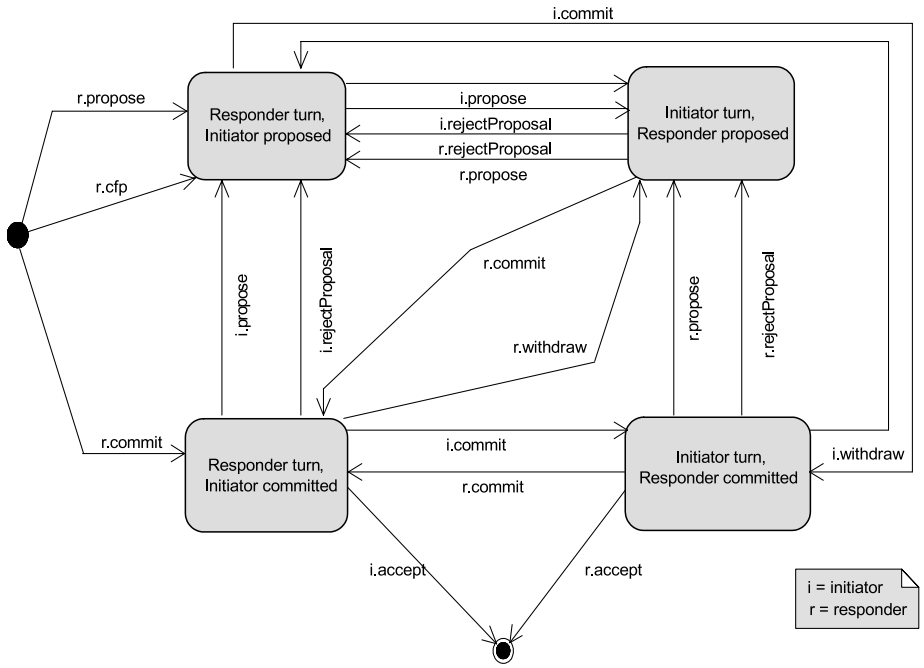

Fig. 3. State machine of the generic bargaining negotiation protocol.

NegotiationCoordinator must interact with it in order to carry out a negotiation. The reason is that the decision to send a binding negotiation message is made by the CommitHandler. As a consequence, the NegotiationCoordinator may send as many non-binding negotiation messages as necessary; but, it needs the approval of the CommitHandler before sending a binding negotiation message.

To handle several simultaneous negotiations, the NegotiationCoordinator uses negotiation contexts. Each ongoing negotiation defines one negotiation context, which corresponds to the execution of one negotiation protocol. However, to give full support to several simultaneous negotiations, not only must the NegotiationCoordinator coordinate each negotiation context independently, but also must have a global view of all negotiation contexts. Therefore, 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 concede less. Furthermore, negotiation policies are also used for guiding the negotiation according to the user preferences about the negotiation process.

Note that the use of policies helps decouple the PoliciesManager from ProposalBuilders since the communication amongst them takes place just by means of policies whose semantics are well-defined. This enables the PoliciesManager to be built independently from the ProposalBuilders, which is important because the implementation of role ProposalBuilders is a variation point of the automated negotiation system, i.e., several ProposalBuilders must usually co-exist.

In NegoFAST-Bargaining, the NegotiationCoordinator is divided into several roles to support several simultaneous 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 and interacts with the CommitHandler. To add support for negotiation policies, the PoliciesManager, which chooses negotiation policies for each negotiation to guide their behaviour, is introduced.

Next, we provide additional details about the previous roles.

Role BilateralNegotiator. Its goal is to carry out a single bilateral negotiation by orchestrating the BargainingProtocolHandler and the MessageComposer. Furthermore, it communicates with the BargainingCoordinator to ask for approval before sending a binding negotiation message, and it receives negotiation policies from the PoliciesManager. (Note that the BargainingCoordinator delegates the decision on whether to approve a binding negotiation message to the CommitHandler.)

This role implements, for each negotiation context, the state machine of a negotiation context for a bargaining protocol. Fig. 4 presents the state machine and Fig. 5 presents a sequence diagram that illustrates how it works in a typical scenario. 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 ConvertProtocol, 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

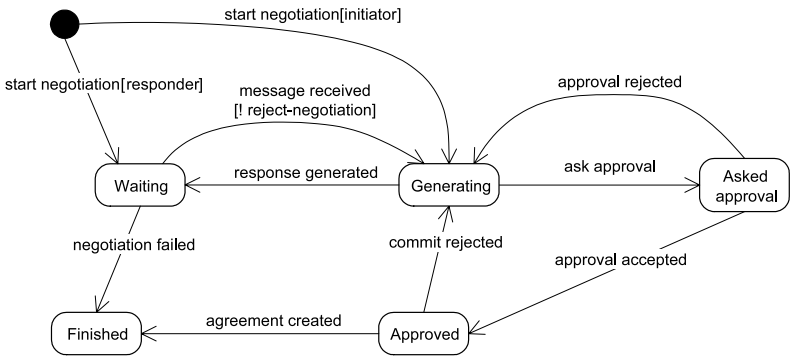

Fig. 4. State machine of a negotiation context for a bargaining protocol.

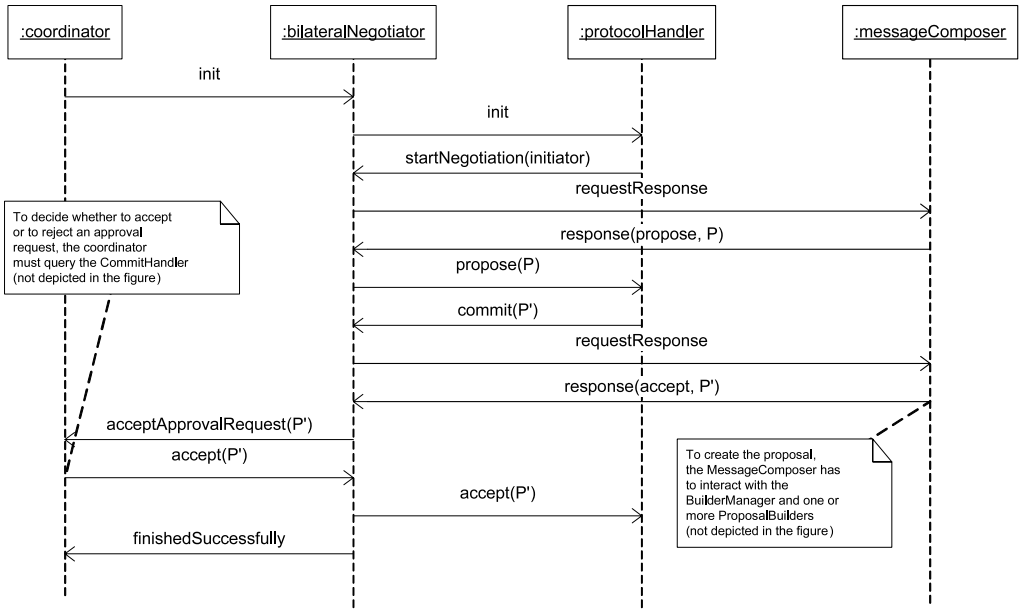

Fig. 5. Sequence diagram of a negotiation context for a bargaining protocol.

interaction: if it is initiator, it enters state generating and starts the negotiation; otherwise, it enters state waiting and waits for a message.

Next, we describe each of the states in detail:

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 ConvertProtocol, and the state machine enters state generating to reply the message that was received. However, if the negotiation message received is a rejectNegotiation message or there is an error in the protocol or in the communication link with the other party, it enters state finished and notifies that the negotiation has failed to the BargainingCoordinator.

Generating. In this state, the BilateralNegotiator interacts with the MessageComposer by means of interaction RequestResponse to obtain the negotiation message to 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; otherwise, 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., it contains performative accept, it enters state finished. Otherwise, it waits in this state until the response of the other party is received. If the other party rejects the binding negotiation message, it moves back to state generating; otherwise, it enters state finished.

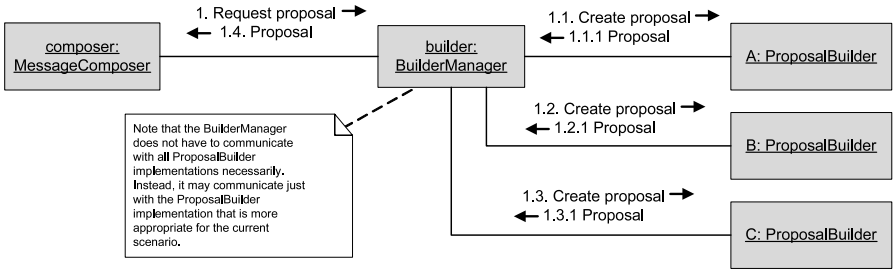

Fig. 6. Communication diagram of the response generator.

Finished. This is the final state in which the negotiation protocol is finished and no more messages shall be accepted.

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

Role BargainingCoordinator. A BargainingCoordinator is intended to act as a message dispatcher amongst PartyCoordinator, BilateralNegotiator and CommitHandler. Specifically:

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

Furthermore, it also stores the current state of the concurrent negotiations in resource BargainingContextData.

Role PoliciesManager. The PoliciesManager sets the policies that are sent to the BilateralNegotiators based on the current status of all negotiations and on the preferences set by the user. It sends the negotiation policies to the BilateralNegotiator by means of interaction SubmitPolicies.

3.3. Response generator

The goal of the ResponseGenerator is to decide on the negotiation messages to be sent to the other parties. In a bargaining protocol, this involves selecting the performative to be used and creating the proposal to be sent if necessary [16].

In NegoFAST-Bargaining, the ResponseGenerator is divided into the following roles: the MessageComposer, which composes the message by choosing performative to be used and delegating to the other roles the creation of the proposal, the BuilderManager, which selects the most appropriate ProposalBuilder and invokes it, and the ProposalBuilders, which are implementations of different algorithms to build proposals. Fig. 6 illustrates this behaviour by means of a communication diagram that includes three different implementations of ProposalBuilders that are available to the BuilderManager.

Next, we provide additional details about the previous roles.

Role MessageComposer. The goal of the MessageComposer is to generate a negotiation message by choosing the appropriate performative and requesting the accompanying proposal to the BuilderManager, if necessary. Its behaviour depends on whether the performative depends on the proposal obtained from the BuilderManager or not. If it depends on the proposal, it first uses interaction RequestProposal to obtain a proposal from the BuilderManager and then decides on the performative. Otherwise, the MessageComposer 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 MessageComposer may also be able to cancel its operation and return a valid yet not optimal negotiation message.

Role BuilderManager. The goal of the BuilderManager is threefold, namely: selecting a ProposalBuilder, using it to obtain a proposal, and sending this proposal back to the MessageComposer. First, it receives a request to build a proposal from the MessageComposer 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 set by the PoliciesManager. 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 MessageComposer. 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.

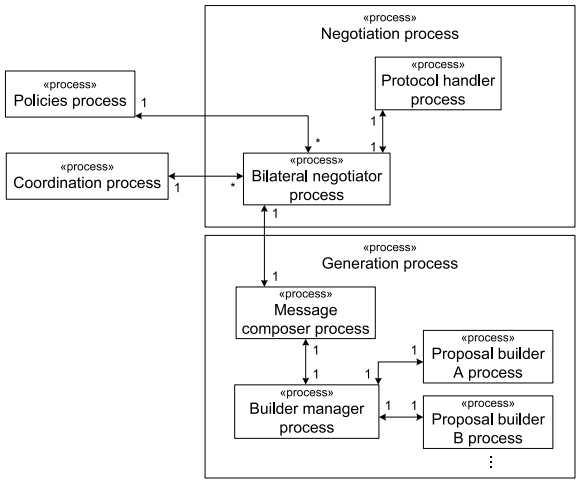

Fig. 7. Process view of NegoFAST-Bargaining.

Role ProposalBuilder. The goal of the ProposalBuilder is to create proposals to be sent to the other party by means of algorithms such as the ones described in Refs. [10,11,21,25]. Some ProposalBuilders are cancellable, which means that they may be interrupted, in which case they return a non-optimal, but acceptable proposal; this is particularly the case of proposals builders that rely on evolutionary algorithms.

3.4. Negotiation context data

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. The BargainingContextData extends this information to support the bargaining-specific states of the negotiation context, cf. Fig. 4, and to provide additional information such as the best utility value of all current negotiation parties.

4. The NegoFAST-Bargaining process view

According to Kruchten [22], the process view of an architecture provides an insight into the processes of which a system that adheres to this architecture is composed; in this context, a process is a group of tasks that form an executable unit.

The process view of the NegoFAST-Bargaining architecture is depicted in Fig. 7. The design of the processes is heavily influenced by the need to support several concurrent negotiations. Next, we provide additional details on these processes.

Coordination process. In the NegoFAST-Bargaining architecture, all negotiations are coordinated by one single process that executes the tasks related to role BargainingCoordinator. It orchestrates the active BilateralNegotiators and the CommitHandler, and it also stores the current state of the system in the BargainingContextData. This process starts when the automated negotiation system starts running, and it keeps running until the automated negotiation system stops.

Policies process. There is also one single process that executes the PoliciesManager to set the policies for each negotiation based on how the current negotiations evolve. Like the coordination process, it starts when the automated negotiation system starts running, and it does not stop until the automated negotiation system stops completely.

Negotiation process. Negotiation processes are associated with a specific bilateral negotiation. Consequently, there are several negotiation processes being executed at the same time: one for each concurrent negotiation. A negotiation process starts when a new bargaining negotiation initiates. The negotiation process stops when the bilateral negotiation finishes, be it successful or not. Each negotiation process can be decomposed into several subprocesses. One subprocess executes the BargainingProtocolHandler subprocess in order to manage the negotiation protocol with the negotiating party and to adapt the negotiation messages between the other party and the BilateralNegotiator. The other subprocess executes the BilateralNegotiator that communicates with the processes that execute the BargainingCoordinator and the ResponseGenerator in order to coordinate the negotiation. To this end, it follows the state machine of a negotiation context for a bargaining protocol that was described in the previous section.

Generation process. The generation process involves the creation of a response for a specific bargaining negotiation. Consequently, the generation process has a 1:1 relationship with a negotiation process. A generation process starts when the BilateralNegotiator requests a response to be sent to the other party and stops when this response has been generated. The response generation process may be either divided into subprocesses or not. The decision depends on the characteristics of the response generation algorithms. On the one hand, the division into subprocesses is convenient if the algorithm

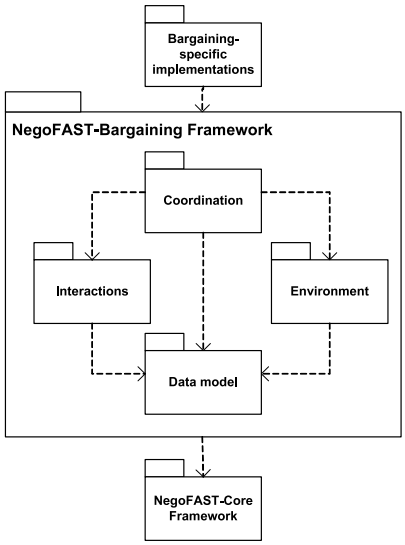

Fig. 8. Development view of NegoFAST-Bargaining.

implemented by the ProposalBuilders involves a long creation process that depends on external information, e.g., a capacity planner, or if it can be done as successive refinements that can be stopped at any time to obtain a valid yet not optimal solution, e.g., using an evolutionary algorithm. In addition, the division into subprocesses makes it possible to generate several possible responses simultaneously and, then, select the most appealing as the actual response. On the other hand, this division makes it necessary to deal with communication and synchronisation issues and requires more processes running at the same time. If a division into subprocesses is to be made, then there must be one subprocess that executes each ProposalBuilder that is available in the automated negotiation system. Regarding the MessageComposer and the BuilderManager, they can be either executed in the same subprocess or they may be executed in different subprocess. Fig. 7 depicts the alternative in which the generation process is divided into several subprocesses and the MessageComposer and BuilderManager are executed in different subprocesses.

Note that the processes detailed in this section are considered from a conceptual point of view. This means that the architecture does not impose any restriction about how these concurrent processes can be implemented. For instance, a possible implementation of the architecture could use a pool of threads in which the aforementioned processes are executed concurrently.

5. The NegoFAST-Bargaining development view

According to Kruchten [22], the development view of an architecture focuses on the actual software module organisation. A part of this view is usually materialised as a software framework that provides an implementation of the architecture so that developers can use it as a starting point for their own systems.

The development view of the NegoFAST-Bargaining architecture is depicted in Fig. 8. It is organised following a layered style. Each layer has a well-defined responsibility and a component in a certain layer only depends on other components on the same layer or in layers below. This design has been chosen to promote the reusability of each layer and to be flexible enough to adapt to the models, algorithms and protocols that are best suited for each negotiation scenario. The four layers are as follows: the data model layer, the environmental resources and interactions layer, the coordination layer, and the bargaining-specific implementations layer.

The NegoFAST-Bargaining software framework is an implementation of the first three layers so that it helps reuse the common part of bargaining-based automated negotiation systems. This is very appealing from the point of view of the development because it allows developers to focus on the parts that are specific to the negotiation scenario in which each automated negotiation system has to deal, i.e., the concrete negotiation protocols and decision-making algorithms.

These layers may be placed on top of another layer, the NegoFAST-Core software framework. It provides a protocol-independent foundation for software engineers to develop automated negotiation systems. It defines the data model and the interfaces of the protocol-independent roles that are necessary in an automated negotiation system, e.g., CommitHandler, Informant, or Inquirer [32].

5.1. Data model

This layer provides a data model for all of the bargaining-specific concepts that are used by the remaining layers. It specifies concepts such as negotiation message, performatives, message contents, the state of a bargaining negotiation, or the negotiation policies that are used to guide the behaviour of the response generation algorithms.

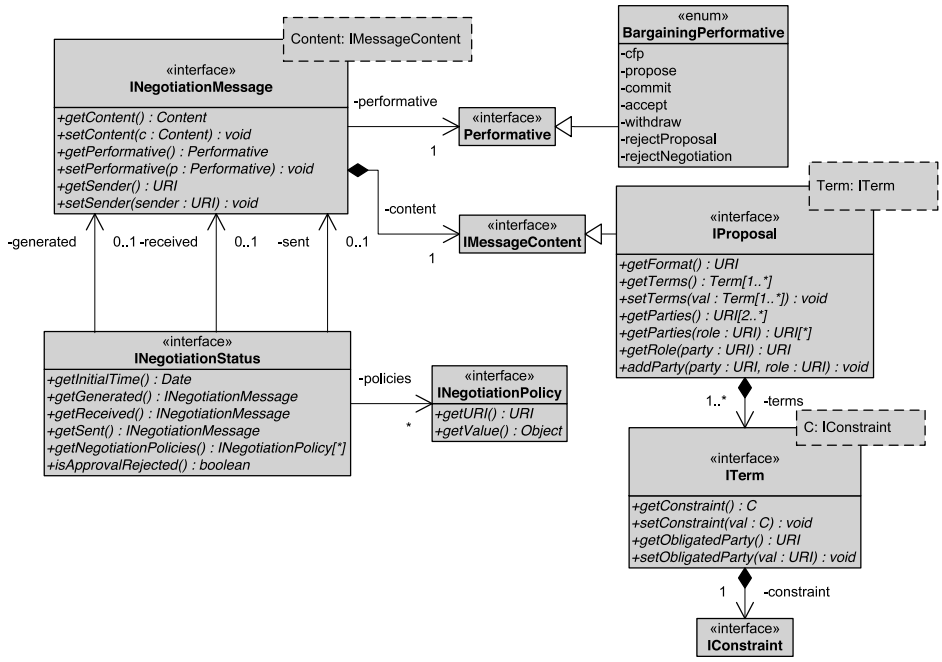

Fig. 9. Data model of NegoFAST-Bargaining.

Negotiation messages are composed of a URI that identifies the sender, a performative that expresses the intention of the sender about the message, and the contents of the message. Performatives are modelled by means of tagging interface Performative, which is in turn extended by enumeration BargainingPerformative, cf. Fig. 9; this enumeration includes the performatives used in the generic bargaining protocol. The contents of a message are modelled by means of interface IMessageContent. In our framework, we do not allow for contents other than proposals, cf. IProposal, which are parameterised by the type of terms they contain. Terms specify constraints over some agreement-related features with which a party must comply, cf. interface ITerm, and they are parameterised by the type of constraint they enclose, e.g., equality, constraints over one attribute, constraints over several attributes, or fuzzy constraints.

Interface INegotiationStatus represents the status of the negotiation in terms of the last negotiation messages sent and received, which is the basis to build new negotiation messages. It includes the time when the negotiation started, the last negotiation message that has been generated, received and sent to the other party, the set of negotiation policies for this negotiation context, and information on whether the CommitHandler did not approve the last negotiation message that was generated.

The negotiation policies are modelled by means of interface INegotiationPolicy, which stores a URI that identifies a negotiation policy and its value.

5.2. Environmental resources

Environmental resource BargainingContextData stores information related to the negotiation contexts, including bargaining-specific information. A negotiation context has one negotiation protocol, one state, and one party with which a negotiation is being carried out. In addition, BargainingContextData stores all negotiation messages that have been exchanged as part of the negotiation.

Consequently, interface IBargainingContextData includes methods to query which the current negotiation contexts are and methods to get information from each negotiation context, namely: its state, the date when the negotiation started, the party with which the system is negotiating, and the negotiation messages that have been exchanged during the negotiation, cf. Fig. 10.

The state of the negotiation is modelled by means of enumeration BargainingState, which includes states waiting, generating, askedForApproval, approved, as well as finished.

5.3. Interactions

This layer provides the definition of the interfaces for the environmental resources and the interactions amongst the bargaining-specific roles. In addition, it defines the generic bargaining protocol that regulates the interaction with the other negotiating parties.

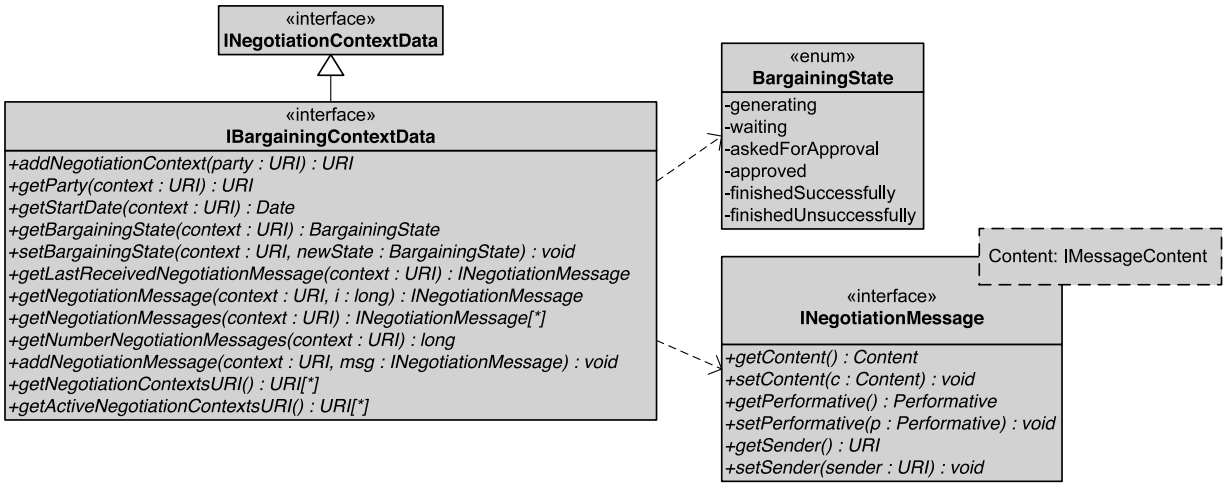

Fig. 10. Interface of BargainingContextData.

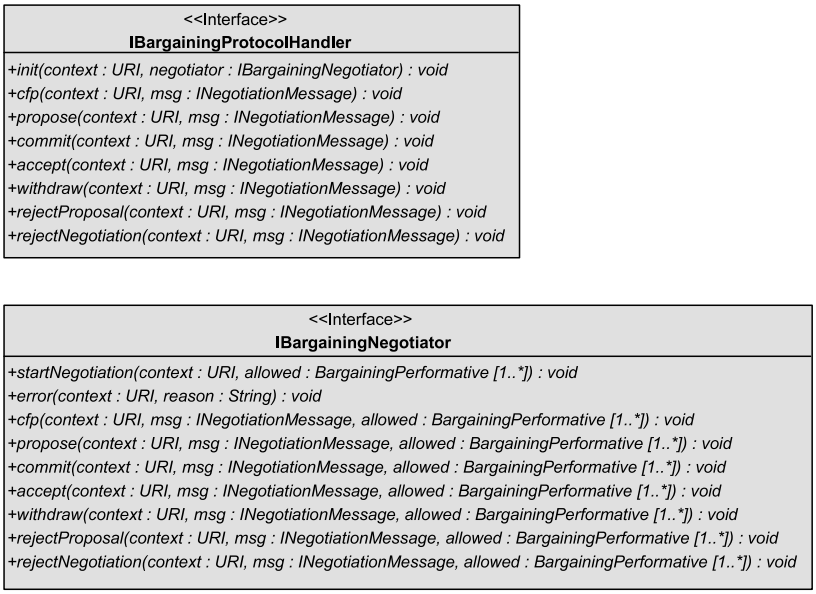

Fig. 11. Interaction ConvertProtocol.

Interaction ConvertProtocol. This interaction implements the generic bargaining protocol described above, together with some control messages. To this end, both the BargainingProtocolHandler and the BilateralNegotiator have one method in their interfaces (IBargainingProtocolHandler and IBargainingNegotiator, respectively) for each negotiation performative of the generic bargaining protocol, namely: accept, rejectProposal, rejectNegotiation, propose, commit, cfp and withdraw, cf. Fig. 11. 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 change depending on the negotiation protocol, e.g., a negotiation protocol may not allow non-binding proposals. Interface IBargainingProtocolHandler also includes method init, which is used to notify the BargainingProtocolHandler that the BilateralNegotiator is ready to start the negotiation.

Interface IBargainingNegotiator also includes additional methods to notify the BilateralNegotiator that it must start the negotiation by sending a negotiation message with one of the allowed performatives (startNegotiation) and to notify that an error has happened during the negotiation, e.g., a time-out or an unexpected message, and, hence, the negotiation must end (error).

Fig. 12 depicts a sequence diagram for this interaction. Before starting a negotiation protocol instance, a previous exchange of messages must be carried out to set up the interaction. This is necessary because the BargainingProtocolHandler does not know the BilateralNegotiator beforehand 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

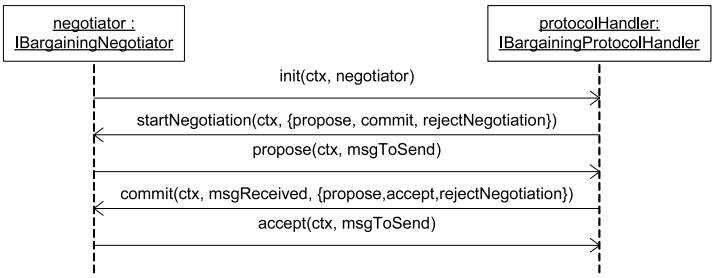

Fig. 12. Sequence diagram for interaction ConvertProtocol.

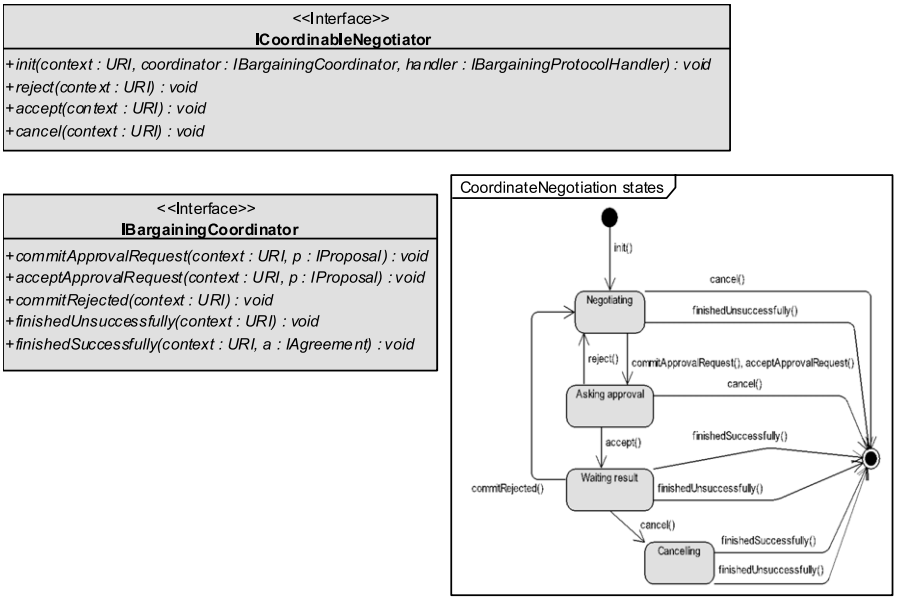

Fig. 13. Interaction CoordinateNegotiation.

BargainingProtocolHandler know which 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. After this happens, 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 of the generic bargaining protocol depicted in Fig. 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. Calling this method amounts to the unsuccessful finalisation of a negotiation process.

Interaction 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 to the BargainingCoordinator. Furthermore, it also involves the sending of initialisation and finalisation messages to and from the BilateralNegotiator.

Fig. 13 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. Now, this role may request an approval from the CommitHandler by invoking method commitApprovalRequest and the interaction moves to state commit approval request or by invoking method acceptApprovalRequest and the interaction enters state asking approval, in which case, there are two choices: the BargainingCoordinator may send message reject, i.e., the CommitHandler rejected the approval request, 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

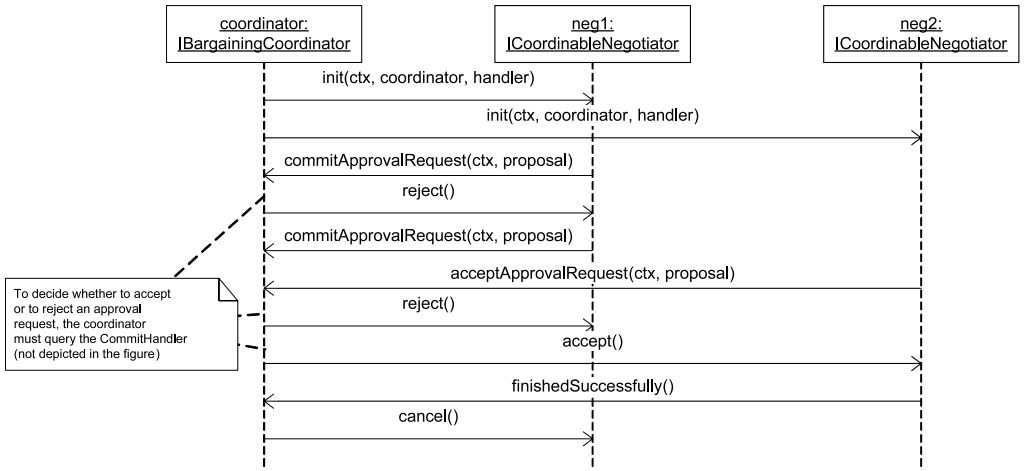

Fig. 14. Typical sequence diagram of interaction CoordinateNegotiation with two bilateral negotiators.

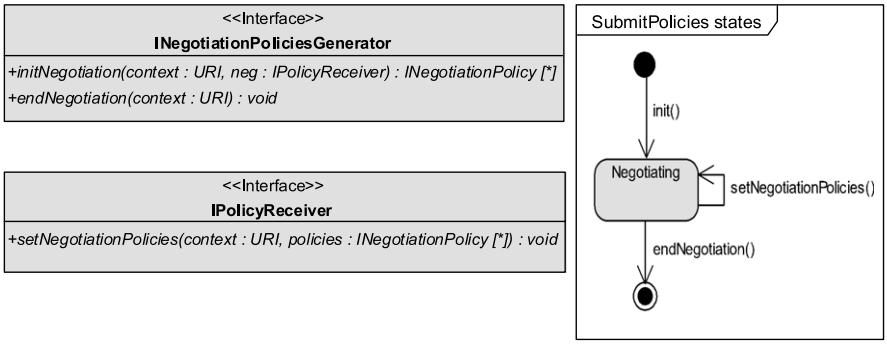

Fig. 15. Interaction SubmitPolicies.

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 interaction to finish.

Similarly, the BargainingCoordinator may decide to cancel the negotiation by invoking method cancel on the BilateralNegotiator. If this method 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 cancel is invoked in that state, the interaction must not finish immediately, but enter state cancelling; now, the BilateralNegotiator can invoke either method finishedSuccessfully or method finishedUnsuccessfully.

Fig. 14 depicts a sequence diagram of this interaction with one BargainingCoordinator (coordinator) and two BilateralNegotiators (neg1 and neg2). In this sequence diagram, after the initialisation, neg1 requests an approval, which is rejected by Coordinator after querying the CommitHandler (not shown in this figure). Next, both neg1 and neg2 request approvals. Coordinator rejects the request from neg1 and accepts the request from neg2. Finally, neg2 notifies that the negotiation finished successfully and Coordinator cancels the negotiation of neg1.

Interaction SubmitPolicies. This interaction implements the submission of negotiation policies from the PoliciesManager, which implements interface IPoliciesManager, to the BilateralNegotiator, which implements interface IPolicyReceiver.

The interaction protocol is simple, cf. Fig. 15. When a new bargaining negotiation starts, the BilateralNegotiator invokes method initNegotiation on the PoliciesManager to notify that there is a new bargaining negotiation and to receive the initial set of negotiation policies. Then, when the PoliciesManager finds it appropriate, 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.

Interaction 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 is depicted in Fig. 16. It has two participants: the BilateralNegotiator, which requests the negotiation message and implements interface IResponseRequester, and the MessageComposer, which returns the generated negotiation message and implements interface IResponseGenerator.

The interaction takes place as follows: when the BilateralNegotiator needs a negotiation message as response, it invokes method generateResponse on the MessageComposer together with information about the bargaining performatives that can be used in the negotiation message, the current status of the negotiation and a reference to the requester that shall receive

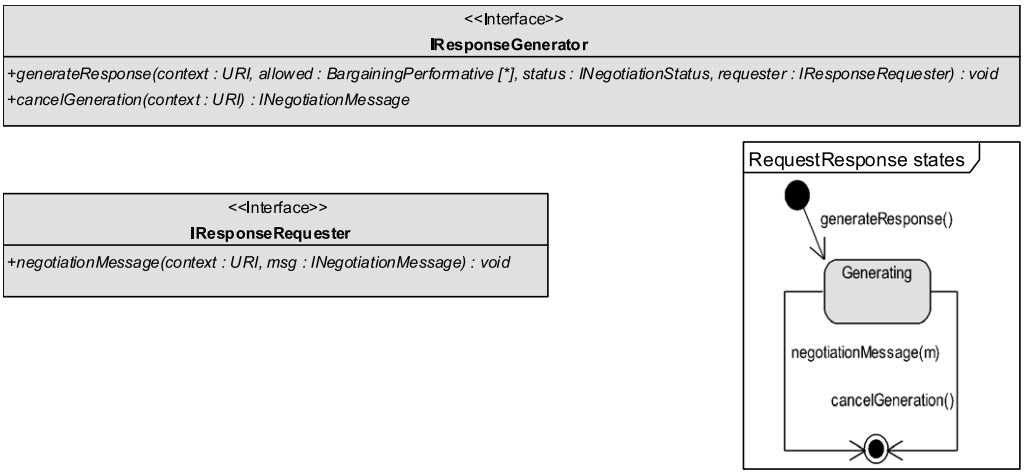

Fig. 16. Interaction RequestResponse.

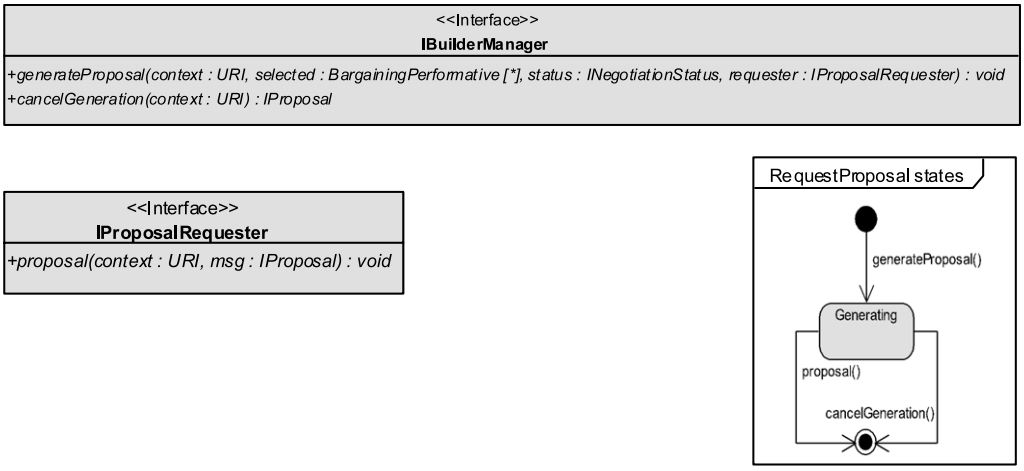

Fig. 17. Interaction RequestProposal.

the generated negotiation message. Then, the MessageComposer creates the negotiation message and 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 MessageComposer must respond synchronously with a valid negotiation message or a null value if no valid negotiation message could be generated.

Interaction RequestProposal. The goal of this interaction is to obtain a proposal that shall be sent as a part of a negotiation message to the other negotiating party. This interaction is very similar to 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, whereas the goal of the latter is just to obtain the proposal. The interaction has two participants, cf. Fig. 17: the BuilderManager, which provides the proposal and implements interface IBuilderManager, and the MessageComposer, which requests the proposal and implements interface IProposalRequester. The interaction protocol is exactly the same as the previous one.

Interaction 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 and, hence, it implements interface IProposalRequester, cf. Fig. 18.

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. The main intent of this step is to convert negotiation policies into builder-specific configuration parameters, and, hence, making independent the

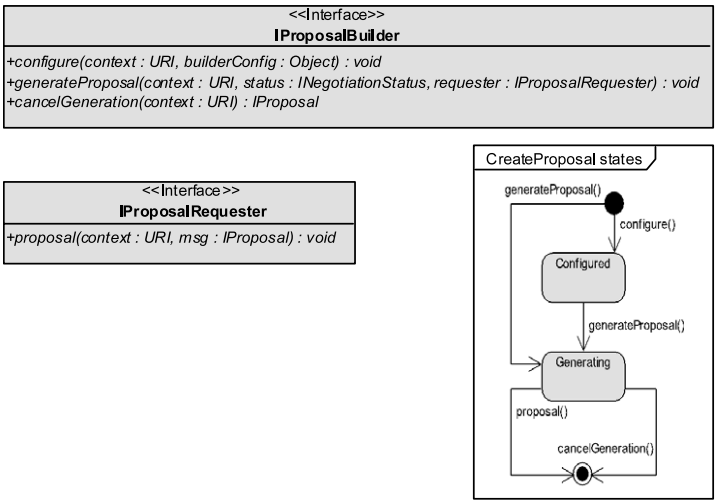

Fig. 18. Interaction CreateProposal.

ProposalBuilder of the negotiation policies defined for an automated negotiation system. However, this step is optional; if the BuilderManager does not configure the ProposalBuilder, it shall take its default configuration.

5.4. Coordination behaviour

This layer provides the specification of the state machine of the BargainingCoordinator and the BilateralNegotiator, which coordinates the other roles of the architecture. This specification defines the behaviour of the whole architecture.

In this section, we report on the behaviour of the BilateralNegotiator and the BargainingCoordinator roles, which coordinate the others.

Role 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 bargaining 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.

Fig. 19 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 MessageComposer 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 MessageComposer and waits for the response in state generating response. In addition, in state waiting initial, the BilateralNegotiator may also receive message cancel, in which case, it enters state cancelling.

The BilateralNegotiator leaves state generating response when either the BargainingProtocolHandler invokes method error, the BargainingCoordinator invokes method cancel or the MessageComposer invokes method negotiationMessage. In the first two cases, the BilateralNegotiator invokes method cancelGeneration in the MessageComposer 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 MessageComposer and moves back to state generating response to wait for another response.

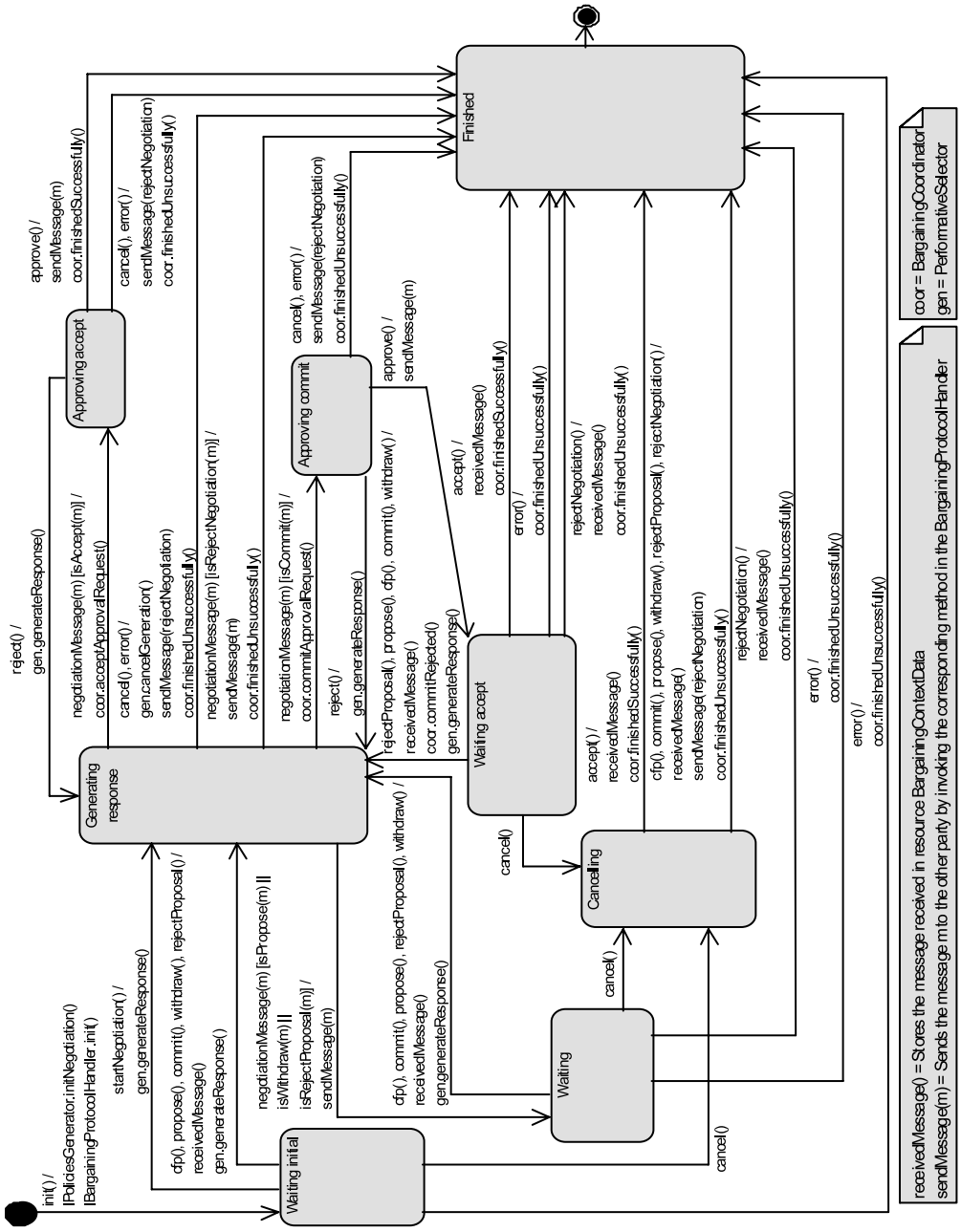

Fig. 19. State machine of the BilateralNegotiator.

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: it invokes method generateResponse and enters state generating response. In addition, the BilateralNegotiator may also receive message cancel or message error, in which case it behaves as in state approving accept.

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 mes-

Table 3
Requirements covered by the scenarios.

Scenarios	Requirement					
	1.1	2.1	2.3	2.4	4.1	4.2
SCEN1: A computing job submission system that negotiates with several IaaS providers.	✓	✓	✓	✓	✓	✓
SCEN2: An IaaS provider that negotiates with several consumers.	✓		✓	✓	✓	
SCEN3: A meeting scheduler negotiation system.		✓	✓			

1.1 Support multi-term negotiation protocols.
2.1 Multiple protocol support.
2.3 Multiple decision-making algorithms.
2.4 Allow user preferences about negotiation processes.
4.1 Support several negotiations simultaneously.
4.2 Select decision-making algorithms dynamically.

sage is either cfp, commit, propose, rejectProposal or withdraw, the BilateralNegotiator invokes method generateResponse in the MessageComposer 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 which 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 because the BilateralNegotiator was waiting for a negotiation message from the other party, i.e., message rejectNegotiation has not been sent. 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 both methods rejectNegotiation and finishedUnsuccessfully in the BargainingProtocolHandler and the BargainingCoordinator, respectively, and enters state finished.

In addition, at any moment, the PoliciesManager may send negotiation policies to the BilateralNegotiator (setNegotiationPolicy).

Role BargainingCoordinator. The BargainingCoordinator is a dispatcher that coordinates the messages exchanged amongst the BilateralNegotiator, the CommitHandler, the PartyCoordinator and the BargainingContextData. Its tasks include initialising and finalising negotiations, as well as coordinating the approval requests between the CommitHandler and the BilateralNegotiators.

Regarding the first task, whenever a 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 of 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.

6. The NegoFAST-Bargaining scenarios view

According to Kruchten [22], the scenarios view provides additional details on how an architecture is instantiated in typical cases, which serves two purposes: on the one hand, it illustrates the architecture and shows how its accompanying software framework can be used to build automated bargaining negotiation systems; on the other hand, it validates the proposal because, in a sense, the scenarios are an abstraction of the most important requirements.

We have developed three scenarios, namely: SCEN1 and SCEN2 are automated negotiation systems in an Infrastructure-as-a-Service (IaaS) context from the perspective of the consumer and provider, respectively; SCEN3 focuses on the implementation of an automated negotiation system to schedule meetings by means of the multi-agent negotiations described by Wainer et al. [41]. Table 3 summarises the requirements covered by each scenario; note, too, that these scenarios helped us check desirable non-functional properties since SCEN1 and SCEN2 were chosen to test the reusability of the architecture and SCEN3 was chosen to test its adaptability to new situations.

Implementing these scenarios required us to make three groups of related decisions, namely:

Negotiation protocol: 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 on the contents of the proposals and the performatives it allows. In addition, the negotiation protocol also has an influence on the decision-making algorithms.

Negotiation policies: The second step is to decide on the negotiation policies that are used by the automated negotiation systems. These policies are used to allow user preferences about the negotiation process and to enable advanced coordination mechanisms of concurrent bargaining negotiations.

Decision making: The last step is to decide on the algorithms to implement the decisions regarding which responses must be created during a negotiation. There are two different decisions: selecting the performative to be used and creating

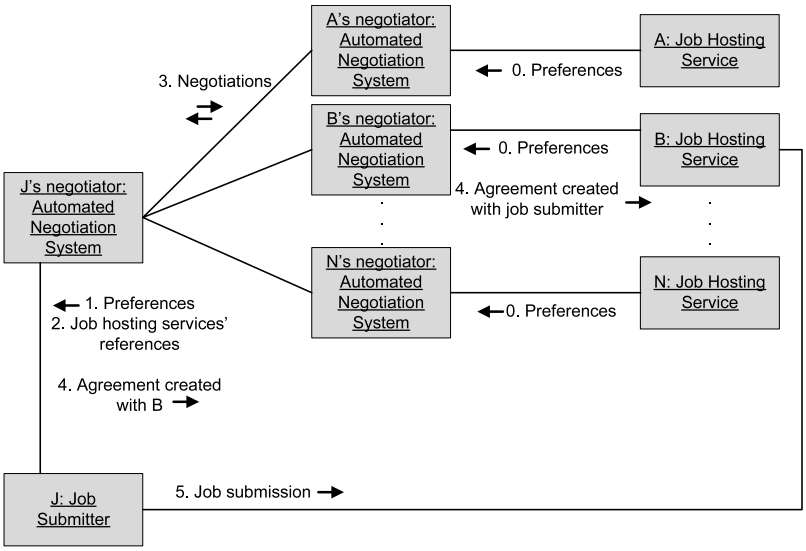

Fig. 20. Infrastructure-as-a-Service scenario.

the proposal to be sent if necessary. Both decisions are influenced by the bargaining protocol because the use of a particular algorithm may depend on the features of a concrete bargaining protocol. It is also influenced by the negotiation policies because the decision-making algorithm must be configurable enough to support the degree of control set by the negotiation policies.

Next, we present further details on these scenarios and provide additional details regarding the previous decisions.

6.1. SCEN1: computing job submission

This scenario focuses on the submission of a computing job to an IaaS provider. In this context, service agreements must be created amongst job submitters and IaaS providers to set the terms under which jobs 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 scheduling requirements like job start or job completion deadlines [1]. Fig. 20 depicts a communication diagram that overviews the whole scenario: 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 a number of references to IaaS providers, it starts a bargaining negotiation with them. When an agreement is made, the automated negotiation system notifies the job submitter and sends the agreement to it. Finally, the job submitter sends the job to the IaaS provider, which executes it following the terms established in the agreement.

Next we provide additional details on the three groups of decisions made to implement this scenario:

Negotiation protocol: In this scenario, we have selected the multi-term negotiation protocol described by Faratin et al. [10] (REQ 1.1), which is a bargaining negotiation protocol in which both parties exchange binding proposals until an agreement is made or one party abandons the negotiation. The implementation of this negotiation protocol is encapsulated in component FaratinProtocolHandler, which plays role BargainingProtocolHandler to deal with particular bargaining protocols.

Negotiation policies: In this scenario, four policies are defined to control the negotiation process (REQ 2.4): negotiation deadline, number of agreements to reach, eagerness to reach an agreement, and the minimum utility threshold. These policies are managed by component PoliciesManager, which sets the values of these policies according to the user preferences and the state of current negotiations (REQ 4.1).

Decision making: Fig. 21 depicts the software artifacts that implement the decision making roles in SCEN1. The first decision, i.e., selecting the performative, is encapsulated in component MessageComposer, specifically in class PerformativeSelector. It is designed to select performative accept if the utility of the received proposal exceeds a user-defined threshold. Otherwise, it selects performative commit. The second decision is carried out by the BuilderManager, which relies on two additional components to create proposals, namely: component InitialBuilder, which creates a proposal by selecting the values that maximise the utility, and component NDFBuilder, which implements the decision-making algorithms proposed by Faratin et al. [10] based on negotiation decision functions and two tactics: time tactic and behaviour tactic. The BuilderManager selects InitialBuilder at the beginning of the negotiation and, as the negotiation goes on, it starts selecting an NDFBuilder (REQ 4.2). Furthermore, it also configures the ProposalBuilders with the policies provided by the PoliciesManager (REQ 2.4).

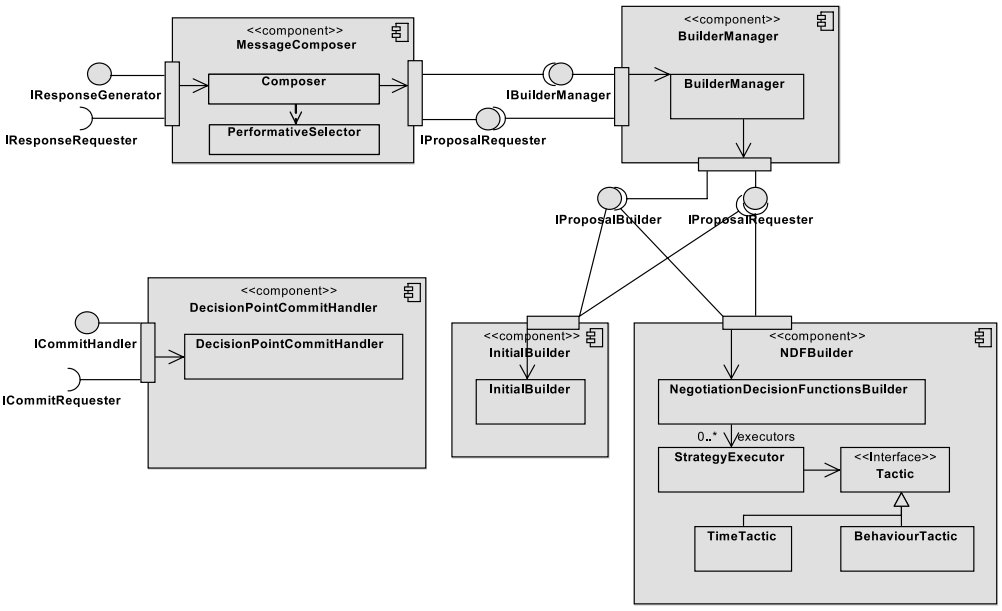

Fig. 21. Implementation of decision making roles in SCEN1.

We must also provide an implementation of role CommitHandler to approve the submission of all binding negotiation messages. In this scenario, the implementation is based on decision points. Decision points take place when a given event occurs, e.g., when the number of binding negotiation messages waiting for approval exceeds a certain threshold or the negotiation deadline is close. When a decision point takes place, some binding negotiation messages are approved, whereas others are rejected.

6.2. SCEN2: infrastructure-as-a-Service Provider

This scenario uses the same context as the previous scenario, but it focuses on the IaaS provider's perspective. Note that since the context is the same as in the previous scenario, it is not surprising that both automated negotiation systems share many components. Therefore, this scenario helps illustrate the reusability of the elements of NegoFAST-Bargaining by analysing the changes that have to be made to the automated negotiation system described in the previous scenario to adapt to the new one.

A description of the three groups of decisions made to implement this scenario follows:

Negotiation protocol: Since the automated negotiation system is going to negotiate in the same context as before, we can keep the same negotiation protocol. Therefore, the implementation of the BargainingProtocolHandler can be reused (REQ 1.1).

Negotiation policies: Regarding the negotiation policies, threshold, deadline and eagerness can be reused since they are also useful from the point of view of the IaaS provider (REQ 2.4). The only consideration is the policy regarding the number of agreements, since a provider has no restrictions on the number of agreements it can reach provided that it has enough resources to comply with them. Therefore, the PoliciesManager can be reused, except for the fact that it should update the policy regarding the number of agreements with an estimation based on the availability of the provider's resources.

Decision making: Finally, this scenario uses a new decision-making model inspired by Nash's bargaining solution [27] that does not impose any restrictions on the number of agreements that can be reached *a priori*. This causes that new implementations of several decision-making roles must be provided, namely: the implementation of the PerformativeSelector must change because the criterion to select performative accept is not a user-defined threshold, but depends on the proposal received and its potential response; a new BuilderManager must be developed to use the Nash-inspired builder instead of the NDFBuilder (REQ 2.3), and the implementation of the BuilderManager must change to use the Nash-inspired builder instead of the negotiation decision functions builder. Furthermore, a new CommitHandler must be developed so that the commitment to an agreement does not depend on a static number of agreements that can be reached, but on the availability of the provider's resources.

6.3. SCEN3: scheduling meetings

This scenario focuses on the implementation of a mechanism to schedule meetings by means of the multi-agent negotiations described by Wainer et al. [41]. The features of the automated negotiation system used in this scenario are

significantly different from the previous ones since there are multiple participants in the negotiation and negotiations are carried out one at a time. Therefore, this scenario helps validate the adaptability of NegoFAST-Bargaining to new situations.

Next we report on the three groups of decisions related to this scenario:

Negotiation protocol: The suggestions protocol by Wainer et al. [41] is a significantly different negotiation protocol (REQ 2.1). It allows the interaction of multiple participants in a negotiation, one for each person invited to a meeting. This interaction is done by means of a host that receives negotiation messages from all participants and broadcasts them to the rest. The implementation of this protocol is encapsulated in component SuggestionsProtocolHandler, which transforms it into the generic negotiation protocol detailed in Section 3 so that the concrete details of the protocol are transparent to the rest of the automated negotiation system.

Negotiation policies: In this scenario, there is only one meeting negotiation at a time. Consequently, there is no need for an advanced coordination of automated negotiations and the negotiation policies can be reduced to only one: information, which sets the amount of information about the own schedule that the system is willing to provide.

Decision making: Regarding the decision-making algorithm to create proposals, Wainer et al. [41] detailed three algorithms called egotistic, laconic, and deceiving. Each of them offers a different trade-off between the amount of information provided to the other participants about its own schedule and the best result in the negotiation. These algorithms are implemented in three different components that play role ProposalBuilder (REQ 2.3) and the BuilderManager selects one amongst them based on the policies provided by the user in his or her preferences.

7. Conclusions and future work

In this article, we have tackled the problem of building automated service agreement negotiation systems that rely on a bargaining protocol and work in open environments. In Ref. [32], we identified a number of key requirements for such systems, some of which are protocol-dependent; our focus in this paper has been on this subset of requirements. We have surveyed the current literature on automatic negotiation, and our conclusion was that none of the current state-of-the-art proposals support all of the protocol-dependent requirements together, which motivated us to work on a new proposal called NegoFAST-Bargaining. Our analysis of the related work also revealed a few weaknesses of the existing literature; for instance, there exists a plethora of proposals to create ProposalBuilders and ProtocolHandlers [10,11,21,25,39]; contrarily, there are very few proposals to create BuilderManagers [34].

Our architecture has been presented according to the guidelines that were established by Kruchten [22]. We have presented a logical view, a process view, a development view, and a scenarios view. The logical view identifies the functional architectural elements of our proposal, and helps define a vocabulary that bridges the gap between the many different terminologies used in this field. The process view identifies how the architectural elements can be grouped together into processes; we have discussed two different organisations for this view. The development view includes a software framework that provides a reference implementation of our proposal and can, thus, be seen as a starting point for developers who need to create their own negotiation systems. The scenarios view illustrates the architecture in typical cases and also helped us validate it.

Future research paths include the following: allowing for inter-dependent negotiations and allowing for protocols that are not based solely on proposals. (Note that extending NegoFAST-Core to support auctioning, for instance, is also quite interesting, but it is a research line in its own. The previous proposals are research paths that would help enhance the results presented in this article.) Note that so-called composite web services are gaining importance as languages such as BPEL are becoming more and more mainstream. The problem with such composite services is that negotiating their components may require negotiations to be inter-dependent. Although the current version of NegoFAST-Bargaining provides a little support for such negotiations by means of our policy-related roles, the problem has not been studied in its full extent. Furthermore, supporting protocols that are not based on proposals only would require new response generation elements to create this additional information and changes to the data model.

Acknowledgements

The work on which this paper reports was partially funded by the European Commission (FEDER), the Spanish Ministry of Science and Innovation, and the Andalusian Government. The work by MR and PF was supported by grants TIN2009-07366 (SETI) and P07-TIC-2533 (Isabel); the work by RC was supported by grants TIN2010-21744-C02, TIN2007-64119, P07-TIC-02602, P08-TIC-4100, and TIN2008-04718-E (IntegraWeb).

Appendix. An overview of NegoFAST-Core

This appendix provides a short introduction to NegoFAST-Core, which constitutes the context of our work. A detailed presentation of NegoFAST-Core was published in Ref. [32].

Fig. 22 provides an overall picture of the four organisations into which a negotiation system is decomposed, namely: protocol management, coordination, world modelling, and decision making, which are represented as large white boxes

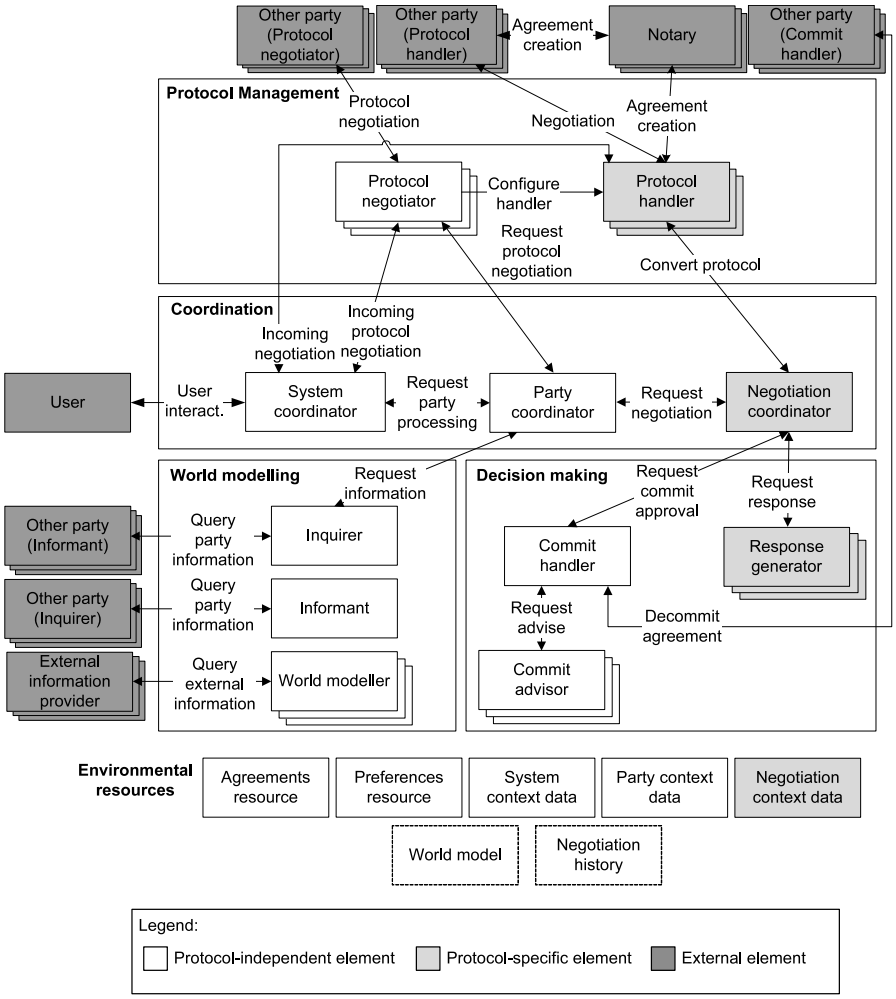

Fig. 22. The NegoFAST-Core reference architecture.

in the middle of the figure, and seven environmental resources, which are depicted as small boxes at the bottom of the figure. The external entities are depicted as dark grey boxes at the top and the left of the figure. Roles are depicted as boxes inside the organisations. They represent artefacts that model the world (WorldModeller), gather and provide information from and to other parties (Inquirer and Informant), interact with the user (SystemCoordinator), decide on the concrete negotiation protocol to be used (ProtocolNegotiator), handle the negotiation protocol (ProtocolHandler), generate negotiation messages during a negotiation (ResponseGenerator), decide whether to commit to or accept an agreement proposal (CommitHandler and CommitAdvisor), and coordinate them all (SystemCoordinator, PartyCoordinator and NegotiationCoordinator). NegoFAST-Core also includes environmental resources that store the data that are required by the previous roles during a negotiation process, namely: AgreementsResource, PreferencesResource, SystemContextData, PartyContextData, NegotiationContextData, WorldModel and NegotiationHistory.

The majority of the previous architectural elements are protocol independent, with the exception of the following, which are dealt with by NegoFAST-Bargaining: ProtocolHandlers to provide support for several multi-term negotiation protocols; NegotiationCoordinator to support several negotiations simultaneously; ResponseGenerators to decide which is the most appropriate message to answer an incoming negotiation message, and NegotiationContextData to store information related to the negotiations that are being carried out by the automated negotiation system.

In addition to these architectural elements, there is a decision that, despite being protocol-independent, has an influence on the protocol-specific architecture: the decision on sending a binding negotiation message or not. 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. This decision is made by the CommitHandler. As a consequence, the NegotiationCoordinator may send as many non-binding negotiation messages as necessary; but, it needs the approval of the CommitHandler before sending a binding negotiation message. Therefore, although the CommitHandler is a protocol-independent role, the NegotiationCoordinator must interact with it in order to carry out a negotiation. Furthermore, the generation of responses must be provided with the user's preferences, the

world model, the negotiation history, and other aspects related to the current state of the automated negotiation system. Such information is stored in the aforementioned environmental resources.

References

[1] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, M. Xu, WS-Agreement specification, 2007. URL: http://www.ogf.org/documents/GFD.107.pdf.
[2] R. Ashri, I. Rahwan, M. Luck, Architectures for negotiating agents, in: V. Marík, J.P. Müller, M. Pechoucek (Eds.), Multi-Agent Systems and Applications III: 3rd International Central and Eastern European Conference on Multi-Agent Systems, in: Lecture Notes in Computer Science, vol. 2691, Springer, 2003, pp. 136–146.
[3] C. Bartolini, C. Preist, N.R. Jennings, A software framework for automated negotiation, in: R. Choren, A. García, C. Lucena, A. Ramonovsky (Eds.), Software Engineering For Multi-Agent Systems III: Research Issues and Practical Applications, in: Lecture Notes in Computer Science, vol. 3390, Springer, 2005, pp. 213–235.
[4] L. Bass, P. Clemens, R. Katzman, Software Architecture in Practice, Addison-Wesley, 2003.
[5] D. Battré, O. Wäldrich, W. Ziegler, F. Brazier, K. Clark, M. Oey, A. Papaspyruo, P. Wieder, A proposal for WS-Agreement negotiation, in: Proceedings of 11th ACM/IEEE International Conference on Grid Computing, Grid 2010, IEEE, 2010, pp. 233–241.
[6] M. Benyoucef, M.-H. Verrons, Configurable e-negotiation systems for large scale and transparent decision making, Group Decision and Negotiation 17 (3) (2008) 211–224.
[7] M. Bichler, An experimental analysis of multi-attribute auctions, Decision Support Systems 29 (3) (2000) 249–268.
[8] A. Elfatatry, P. Layzell, Negotiating in service-oriented environments, Communications of the ACM 47 (8) (2004) 103–108.
[9] A. Elfatatry, P.J. Layzell, A negotiation description language, Software, Practice and Experience 35 (4) (2005) 323–343.
[10] P. Faratin, C. Sierra, N.R. Jennings, Negotiation decision functions for autonomous agents, International Journal of Robotics and Autonomous Systems 24 (3–4) (1998) 159–182.
[11] P. Faratin, C. Sierra, N.R. Jennings, Using similarity criteria to make trade-offs in automated negotiations, Artificial Intelligence 142 (2) (2002) 205–237.
[12] S.S. Fatima, M. Wooldridge, N.R. Jennings, An agenda-based framework for multi-issue negotiation, Artificial Intelligence 152 (1) (2004) 1–45.
[13] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Longman Publishing Co., Inc., 2002.
[14] H. Gimpel, H. Ludwig, A. Dan, B. Kearney, PANDA: Specifying policies for automated negotiations of service contracts, in: M.E. Orlowska, S. Weerawarana, M.P. Papazoglou, J. Yang (Eds.), First International Conference on Service-Oriented Computing, ICSOC 2003, in: Lecture Notes in Computer Science, vol. 2910, Springer-Verlag, 2003, pp. 287–302.
[15] M. He, N.R. Jennings, H.-F. Leung, On agent-mediated electronic commerce, IEEE Transactions on Knowledge and Data Engineering 15 (4) (2003) 985–1003.
[16] N.R. Jennings, P. Faratin, A.R. Lomuscio, S. Parsons, M. Wooldridge, C. Sierra, Automated negotiation: prospects, methods and challenges, Group Decision and Negotiation 10 (2) (2001) 199–215.
[17] C. Jonker, V. Robu, J. Treur, An agent architecture for multi-attribute negotiation using incomplete preference information, Autonomous Agents and Multi-Agent Systems 15 (2) (2007) 221–252.
[18] A.H. Karp, Rules of engagement for automated negotiation, in: B. Benatallah, C. Godart (Eds.), First IEEE International Workshop on Electronic Contracting, IEEE Computer Society, 2004, pp. 32–39.
[19] J.B. Kim, A. Segev, A web services-enabled marketplace architecture for negotiation process management, Decision Support Systems 40 (1) (2005) 71–87.
[20] M. Klein, P. Faratin, H. Sayama, Y. Bar-Yam, Protocols for negotiating complex contracts, IEEE Intelligent Systems 18 (6) (2003) 32–38.
[21] R. Kowalczyk, Fuzzy e-negotiation agents, Soft Computing 6 (5) (2002) 337–347.
[22] P. Kruchten, The view model of architecture, IEEE Software 12 (6) (1995) 42–50.
[23] G. Lodi, F. Panzieri, D. Rossi, E. Turrini, SLA-Driven clustering of qos-aware application servers, IEEE Transactions on Software Engineering 33 (3) (2007) 186–197.
[24] A. Ludwig, P. Braun, R. Kowalczyk, B. Franczyk, A framework for automated negotiation of service level agreements in services grids, in: Business Process Management Workshops, in: Lecture Notes in Computer Science, vol. 3812, Springer, 2006, pp. 89–101.
[25] X. Luo, N.R. Jennings, N. Shadbolt, H.-F. Leung, J.H. Lee, A fuzzy constraint based model for bilateral, multi-issue negotiations in semi-competitive environments, Artificial Intelligence 148 (1–2) (2003) 53–102.
[26] C. Molina-Jimenez, J. Pruyne, Aad van Moorsel, The role of agreements in it management software, in: R. de Lemos, C. Gacek, A. Romanovsky (Eds.), Architecting Dependable Systems III, in: Lecture Notes in Computer Science., vol. 3549, Springer, 2005, pp. 36–58.
[27] J.F. Nash, The bargaining problem, Econometrica 18 (2) (1950) 155–162.
[28] T.D. Nguyen, N.R. Jennings, Managing commitments in multiple concurrent negotiations, Electronic Commerce Research and Applications 4 (4) (2005) 362–376.
[29] M.P. Papazoglou, The challenges of service evolution, in: Z. Bellahsene, M. Léonard (Eds.), 20th International Conference on Advanced Information Systems Engineering, CAiSE 2008, in: Lecture Notes in Computer Science, vol. 5074, Springer, 2008, pp. 1–15.
[30] S. Paurobally, V. Tamma, M. Wooldrdige, A framework for web service negotiation, ACM Transactions on Autonomous and Adaptive Systems 2 (4) (2007) 14.
[31] I. Rahwan, S.D. Ramchurn, N. R Jennings, P. McBurney, S. Parsons, L. Sonenberg, Argumentation-based negotiation, The Knowledge Engineering Review 18 (4) (2003) 343–375.
[32] M. Resinas, P. Fernandez, R. Corchuelo, Automatic service agreement negotiators in open commerce environments, International Journal of Electronic Commerce 14 (3) (2010) 93–128.
[33] S. Rinderle, M. Benyoucef, Towards the automation of e-negotiation processes based on web services, in: A.H. Ngu, M. Kitsuregawa, E.J. Neuhold, J.-Y. Chung, Q.Z. Sheng (Eds.), Sixth International Conference on Web Information Systems Engineering, in: Lecture Notes in Computer Science, vol. 3806, Springer, 2005, pp. 443–453.
[34] R. Ros, C. Sierra, A negotiation meta strategy combining trade-off and concession moves, Autonomous Agents and Multi-Agent Systems 12 (2) (2006) 163–181.
[35] T.W. Sandholm, V.R. Lesser, Leveled commitment contracts and strategic breach, Games and Economic Behavior 35 (1–2) (2001) 212–270.
[36] K.M. Sim, Grid resource negotiation: survey and new directions, IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 40 (3) (2010) 245–257.
[37] K.M. Sim, S.Y. Wang, Flexible negotiation agent with relaxed decision rules, IEEE Transactions on Systems, Man and Cybernetics, Part B 34 (3) (2004) 1602–1608.
[38] M. Ströbel, Design of roles and protocols for electronic negotiations, Electronic Commerce Research 1 (3) (2001) 335–353.
[39] S.Y. Su, C. Huang, J. Hammer, Y. Huang, H. Li, L. Wang, Y. Liu, C. Pluempitiwiriyawej, M. Lee, H. Lam, An internet-based negotiation server for e-commerce, The International Journal on Very Large Data Bases 10 (1) (2001) 72–90.
[40] T. Tu, C. Seebode, F. Griffel, W. Lamersdorf, DynamiCS: an actor-based framework for negotiating mobile agents, Electronic Commerce Research 1 (1–2) (2001) 101–117.
[41] J. Wainer, P.R. Ferreira, E.R. Constantino, Scheduling meetings through multi-agent negotiations, Decision Support Systems 44 (1) (2007) 285–297.
[42] F. Zambonelli, N.R. Jennings, M. Wooldridge, Developing multiagent systems: the Gaia methodology, ACM Transactions on Software Engineering and Methodology 12 (3) (2003) 317–370.
[43] D. Zeng, K. Sycara, Bayesian learning in negotiation, International Journal Human–Computer Studies 48 (1) (1998) 125–141.