SPICA's Multi-party Negotiation Protocol: Implementation using YAWL

E. Bacarin[*] E.R.M. Madeira[†] C.B. Medeiros [†] W.M.P van der Aalst[‡]

November 28, 2009

Abstract

A supply chain comprises several different kind of actors that interact either in an ad hoc fashion (e.g., an eventual deal) or in a previously well planned way. In the latter case, how the interactions develop is described in contracts that are agreed on before the interactions start. This agreement may involve several partners, thus a multi-party contract is better suited than a set of bi-lateral contracts. If one is willing to negotiate automatically such kind of contracts, an appropriate negotiation protocol should be at hand. However, the ones for bi-lateral contracts are not suitable for multi-party contracts, e.g., the way of achieving consensus when only two negotiators are haggling over some issue is quite different if there are several negotiators involved. In the first case, a simple bargain would suffice, but in the latter a ballot process is needed. This paper presents a negotiation protocol for electronic multi-party contracts which seamlessly combines several negotiation styles. It also elaborates on the main negotiation patterns the protocol allows for: bargain (for peer-to-peer negotiation), auction (when there is competition among the negotiators) and ballot (when the negotiation aims at consensus). Finally, it describes an implementation of this protocol based on Web services, and built on the YAWL Workflow Management System.

1 Introduction

Face-to-face negotiations are being increasingly replaced by electronic negotiations (e-negotiation) in all situations in which interactions among partners are dynamic and must follow the "just-in-time" principle. Such a negotiation process gives rise to electronic contracts (e-contract) that are enacted by a suitable supporting system and produce data that can be used to assess in which extension their partners are fulfilling the contract's provisions, allowing for early corrective actions. Electronic negotiations, either assisting human negotiators or automating the whole process, facilitate the interaction among parties and speed up contract construction.

[*]Department of Computer Science - UEL - CP 6001 86051-990 Londrina,PR Brazil. bacarin@dc.uel.br

[†]Institute of Computing - UNICAMP - CP 6176 13081-970 Campinas,SP Brazil. {edmundo,cmbm}@ic.unicamp.br

[‡]Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands. w.m.p.v.d.aalst@tue.nl

eBay is a prime example of such a situation, but it presents just a facet of the multiple challenges to be faced. One can extend this kind of scenario to a situation where multiple kinds of enterprises need to interact and negotiate distinct issues, at different levels. One typical example of such a complex situation is contract negotiation within supply chains.

A supply chain is a network of retailers, distributors, transporters, storage facilities and suppliers that participate in the sale, delivery and production of a particular product [MZ02]. It is composed of distributed, heterogeneous and autonomous elements, whose relationships are dynamic.

Efficiency and profitability within a supply chain depends on several factors, including the speed and flexibility with which the participants arrange their relationship in terms of goals and commitments, and their capability of assessing how these goals and commitments have been reached or fulfilled.

A contract negotiation differs from an usual single item (or single bundle) negotiation. In the latter, partners haggle over the item's price and configuration and, if they agree, two actions follow: the buyer hands over money to the seller, and the seller hands over the item to the buyer.

Conversely, a contract states rights and duties and mainly comprises a set of actions or intended effects that, if accomplished, would result in the fulfillment of this duty or right. In this perspective, a negotiation comprises taking into consideration this set of actions and haggling over specific attributes that qualify them (e.g., configuration, price and quality constraints). In a nutshell, a contract is a future plan and contains a number of statements of intention.

Multi-party contracts are a special kind of contracts. They are signed by more than two parties. In this case, the rights and duties are distributed over the parties. The negotiation process must be able to express to whom each duty or right is applicable to.

e-Negotiation and contract management are subject to intensive research. However, most papers concentrate on one single kind of negotiation protocol (e.g., auction, bargain). Moreover, implementation issues are handled either at the protocol (communications) level or concern negotiation logics. In [BMM08] we discuss SPICA Negotiation Protocol (SPICA, for short), a new kind of contract e-negotiation in which each clause may be established according to a different negotiation protocol. This provides the flexibility needed in multi-party negotiation in which partners change at each situation (e.g., typical of supply chains). In [BMM09] we present how SPICA's protocols can be combined to build virtual organizations. This paper details our implementation of SPICA which shows innovative ways of solving the challenges in implementing combined negotiation styles for e-contracts:

1. we present how multi-protocol issues can be attacked by means of specific negotiation patterns, which can then be instantiated at will, according to patterns decisions at any given time.

2. we discuss how to take advantage of a workflow engine – YAWL – to implement the middleware that orchestrates interactions among partners. The negotiation protocol is seen as a workflow in which each interaction (e.g., an offer) is considered a task the partner negotiator must carry out (e.g., decide whether to accept the offer or not). In

addition, YAWL's workflow engine is extended with a tailor-made so-called custom service which takes care of the information exchanged among negotiators.

The main contributions of this paper are: (a) it presents a negotiation protocol for electronic contracts which combines three basic negotiation styles: bargain (for peer-to-peer negotiation), auction (when there is competition among the negotiators) and ballot (when the negotiation aims at consensus); (b) it depicts the main negotiation patterns the protocol allows for; (c) it describes an implementation built on a WfMS (YAWL).

The paper is organized as follows. Section 2 presents an example that motivates and illustrates the protocol usage. Section 3 overviews YAWL. Section 4 describes briefly SPICA's contracts and the SPICA negotiation protocol. Section 5 presents the main negotiation patterns supported by the protocol. Section 6 discusses the protocol's implementation and shows a brief example of a negotiation execution. Section 7 reviews related work. Finally, section 8 concludes the paper.

2 Running Example

This section presents a running real life example that will be used throughout the paper. It shows the complex contract interactions that are typical of supply chains.

Consider the scenario in which a set of small farms export coffee, but their individual productions are not sufficient to allow independent handling. Thus, they get together and select one exporter that will handle brokerage and exportation procedures. Coffee is first processed in the farms, put in bags and then transported and stored at the docks to be shipped. This involves a number of producers, transportation companies, storage companies and an exporter, all of which undergo a complex negotiation process, creating an ad hoc temporary association that will last for one season. The following negotiation issues are handled:

- The farms must choose the best transportation companies among several elegible ones. They choose the one that offers the best price.

- Storage costs at the the docks are shared between farms and exporter. Here, farms and exporter must agree on storage provider for two reasons: (a) it influences the cost of the first two partners; (b) the exporter may veto a specific storage facility due to, e.g., bad past commercial interactions.

Contract obligations may be summed up as follows: (a) the farms are (indirectly) responsible for delivering coffee bags to be stored at the docks; (b) the transportation company is (directly) responsible for collecting coffee at each farm and delivering them at the docks; (c) the storage provider is responsible for making empty space available at specific temperature and humidity levels; (d) the exporter is responsible for brokering the coffee, sending it from docs to its destination (how the exporter does this is out of the scope of the contract), receiving the money, paying the storage costs and the farms (after subtracting its commission and half of the storage cost); (e) the farms are also responsible for paying the transportation company.

3 YAWL Overview

In this paper, we use YAWL (Yet Another Workflow Language) to realize and support the SPICA protocol. YAWL was developed after a rigorous analysis of existing workflow management systems and related standards using a comprehensive set of workflow patterns [AHKB03]. YAWL is both a language and a system supporting this language [HAAR10]. There are three main reasons for using YAWL. First of all, the language is simple yet much more expressive than most other languages. Since a wide range of workflow patterns are directly supported, it is easy to quickly realize complex workflows. Second, the YAWL system has rigorously adopted a service-oriented architecture. This makes it easy to use YAWL in a distributed setting where multiple parties and also software from multiple vendors need to cooperate. Finally, YAWL is well-grounded, i.e., right from the start formal semantics and analysis techniques were provided. (Unlike most other languages where semantics and analysis are more of an afterthought, rather than a first priority.)

This section introduces the YAWL language, the supporting system, and some of the more advanced capabilities (some of which are already used in the SPICA realization).

3.1 YAWL: A Language Based on Patterns

In the area of workflow one is confronted with a plethora of products (commercial, free and open source) supporting languages that differ significantly in terms of concepts, constructs, and their semantics. One of the contributing factors to this problem is the lack of a commonly agreed upon formal foundation for workflow languages. The workflow patterns initiative [AHKB03] aims at establishing a more structured approach to the issue of the specification of control flow dependencies in workflow languages. Based on an analysis of existing workflow management systems and applications, this initiative identified a collection of patterns corresponding to typical control flow dependencies encountered in workflow specifications, and documented ways of capturing these dependencies in existing workflow languages. These patterns have been used as a benchmark for comparing process definition languages and in tendering processes for evaluating workflow offerings. See `http://www.workflowpatterns.com` for extensive documentation, flash animations of each pattern, and evaluations of standards and systems.

While workflow patterns provide a pragmatic approach to control flow specification in workflows, Petri nets provide a more theoretical approach. Petri nets form a model for concurrency with a formal foundation, an associated graphical representation, and a collection of analysis techniques. These features, together with their direct support for the notion of state (required in some of the workflow patterns), make them attractive as a foundation for control flow specification in workflows. However, even though Petri nets support a number of the identified patterns, they do not provide direct support for the cancellation patterns (in particular the cancellation of a whole case or a region), the synchronizing merge pattern (where all active threads need to be merged, and branches which cannot become active need to be ignored), and patterns dealing with multiple active instances of the same activity in the same case. This realization motivated the development of YAWL [AH05] (Yet Another Workflow Language) which combines the insights gained from the workflow patterns with the benefits of Petri nets. It should be noted though that YAWL is not simply a set of macros defined on top of Petri nets as the expressiveness is increased considerably (see [AHKB03, AH05, HAAR10] for discussions on this.

Before describing the architecture and implementation of the YAWL system, we introduce the distinguishing features of YAWL. As indicated in the introduction, YAWL is based on Petri nets. However, to overcome the limitations of Petri nets, YAWL has been extended with features to facilitate patterns involving multiple instances, advanced synchronization patterns, and cancellation patterns. Moreover, YAWL allows for hierarchical decomposition and handles arbitrarily complex data.

Figure 1 shows the modeling elements of YAWL. At the syntactic level, YAWL extends the class of workflow nets described in [Aal98] with multiple instances, composite tasks, OR-joins, removal of tokens, and directly connected transitions. YAWL, although being inspired by Petri nets, is a completely new language with its own semantics and specifically designed for workflow specification.

A *workflow specification* in YAWL is a set of *process definitions* which form a hierarchy. *Tasks*[1] are either *atomic tasks* or *composite tasks*. Each composite task refers to a process definition at a lower level in the hierarchy (also referred to as its decomposition). Atomic tasks form the leaves of the graph-like structure. There is one process definition without a composite task referring to it. This process definition is named the *top level workflow* and forms the root of the graph-like structure representing the hierarchy of process definitions.

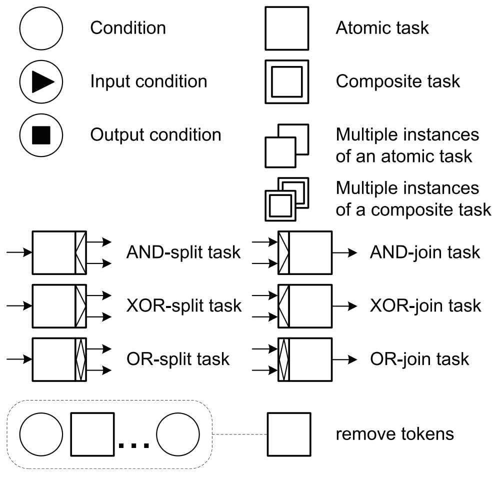


Figure 1: Symbols used in YAWL [AH05].

Each process definition consists of *tasks* (whether composite or atomic) and *conditions* which can be interpreted as places. Each process definition has one unique *input condition* and one unique *output condition* (see Figure 1). In contrast to Petri nets, it is possible to connect 'transition-like objects' like composite and atomic tasks directly to each other without using a 'place-like object' (i.e., conditions) in-between. For the semantics this

[1]We use the term *task* rather than *activity* to remain consistent with earlier work on workflow nets [Aal98].

construct can be interpreted as a hidden condition, i.e., an implicit condition is added for every direct connection.

Both composite tasks and atomic tasks can have multiple instances as indicated in Figure 1. We adopt the notation described in [Aal98] for AND/XOR-splits/joins as also shown in Figure 1. Moreover, we introduce OR-splits and OR-joins corresponding respectively to Pattern 6 (Multi choice) and Pattern 7 (Synchronizing merge) defined in [AHKB03]. Finally, Figure 1 shows that YAWL provides a notation for removing tokens from a specified region denoted by dashed rounded rectangles and lines. The enabling of the task that will perform the cancellation may or may not depend on the tokens within the region to be "canceled". In any case, the moment this task completes, all tokens in this region are removed. This notation allows for various cancellation patterns.

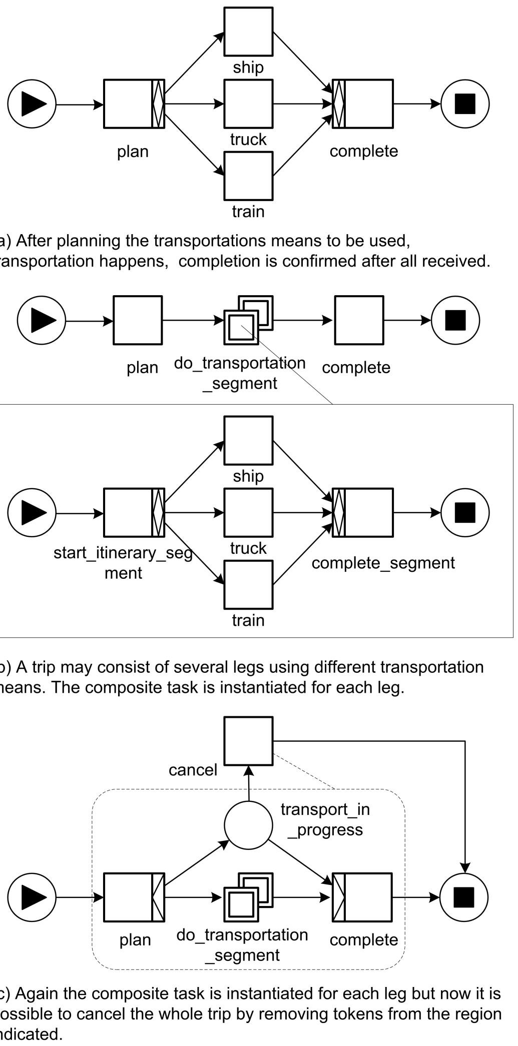

ship
truck
train
plan
complete
(a) After planning the transportations means to be used, transportation happens, completion is confirmed after all received.
plan
do_transportation_segment
complete
ship
truck
train
start_itinerary_segment
complete_segment
(b) A trip may consist of several legs using different transportation means. The composite task is instantiated for each leg.
cancel
transport_in_progress
plan
do_transportation_segment
complete
(c) Again the composite task is instantiated for each leg but now it is possible to cancel the whole trip by removing tokens from the region indicated.

Figure 2: Three YAWL specifications (adapted from [AADH04]).

To illustrate YAWL we use the three examples shown in Figure 2. The first example (a) illustrates that YAWL allows for the modeling of advanced synchronization patterns. Consider that, within an agricultural supply chain, a given load of coffee, packed in bags, is to be transported between two warehouses in two different cities. This amount can be divided in up to three partial loads and transported by different means of transportation. Task *plan* is an 'OR-split' (Pattern 6: Multi-choice) and task *complete* is an 'OR-join' (Pattern 7: Synchronizing merge). This implies that every planning step is followed by a set of transportation tasks *ship*, *truck*, and/or *train*. It is possible that all three transportation tasks are executed, but it is also possible that only one or two tasks are performed. The YAWL OR-join synchronizes only if necessary, i.e., it will synchronize only the transportation tasks that were actually selected, signaling that the full amount of coffee bags were received at the destination.

Figure 2(b) illustrates another YAWL specification of the same stage of the supply chain. In contrast to the first example, a transportation trip may include multiple legs, i.e., an itinerary between the two warehouses may include multiple segments. Typically, transportation between two consecutive legs is performed by one transportation means. However, the load that arrives at one point may be divided and transported by different transportation means to the next point in the itinerary, where again the load can be rearranged for the next leg. For example, the trip between the warehouses may entail three itinerary segments with distinct characteristics. In the first segment, coffee bags are transported from a farm warehouse to a cooperative by truck. The cooperative collects bags from several farms, and the load is then transported to the docks by train. Finally, the coffee is transported to a final warehouse overseas by ship. Figure 2(b) shows that multiple segments are modeled by multiple instances of the composite task *do_transportation_segment*. This composite task is linked to the process definition also shown in Figure 2(b). In the case of multiple instances, it is possible to specify upper and lower bounds for the number of instances. It is also possible to specify a threshold for completion that is lower than the actual number of instances, i.e., the construct completes before all of its instances complete. The example shows that YAWL supports the patterns dealing with multiple instances (Patterns 12-15). Only few systems support multiple instances.

Finally we consider the YAWL specification illustrated in Figure 2(c). Again composite task *do_transportation_segment* is decomposed into the process definition shown in Figure 2(b). Now it is however possible to withdraw planned segments by executing task *cancel*. Task *cancel* is enabled if there is a token in *transport_in_progress*. If the environment decides to execute cancel, everything inside the region indicated by the dashed rectangle will be removed. In this way, YAWL provides direct support for the cancellation patterns (Patterns 19 and 20). Note that support for these patterns is typically missing or very limited in existing systems.

In this section we illustrated some of the features of the YAWL language while focusing mainly on the control-flow. We did not discuss the data aspects – each workflow execution involves several variables (over 10 variables in the example of coffee transportation) – because they are not needed to understand the rest of the paper, even though variable specification and instantiation is an essential aspect in workflow execution. It is important to note that YAWL also supports many resource patterns [RAHE05], data patterns

[RHEA05], exception patterns, flexibility patterns, service interaction patterns, etc.

3.2 YAWL System

The YAWL language is supported by a full-fledged workflow management system. Figure 3 shows the architecture of YAWL. Like any workflow management system, YAWL has an *Engine* and *Process Designer*. The *Process Designer* is used to construct YAWL specification using the notations described before. These can be verified and once they contain no error, they are runable and the engine can instantiate instances of such models. The engine also records events in so called event logs and persists data beloning to running instances. The *Resource Service* is used to offer or push work to human resources. This is just one example of a service that can be invoked from YAWL. There are many other *YAWL Services* and Figure 3 only shows a fraction of the available services. The main design principle of the YAWL System is that the *Engine* should be completely agnostic with regards to the services interacting with it. The *Engine* is unaware of the inside behavior of services, i.e., services can be seen as a black box that subcontract work. Moreover, YAWL itself can be seen as a service, e.g., one YAWL engine can subcontract work to another engine. This makes YAWL truly service-oriented system and highly suitable for supporting the SPICA protocol.

For more details, we refer to [HAAR10].

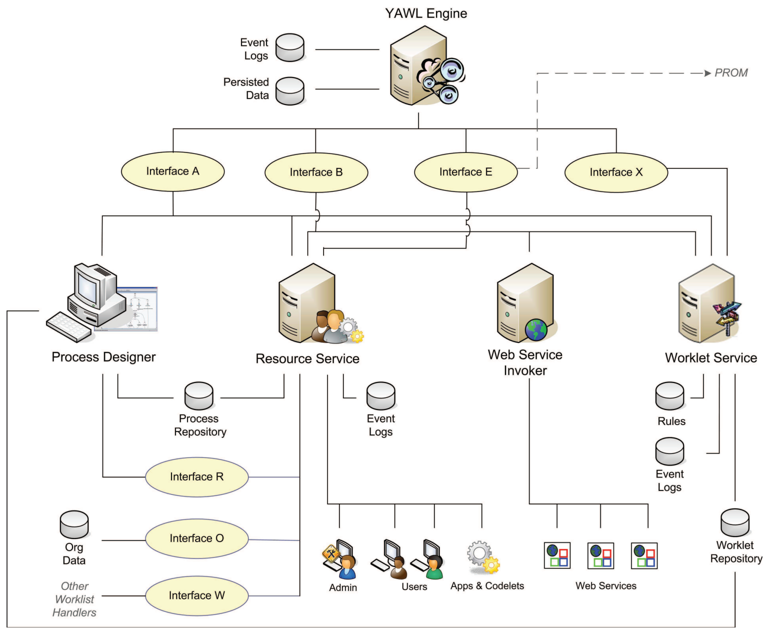


Figure 3: The architecture of YAWL [HAAR10].

3.3 Analysis Capabilities

YAWL has had formal semantics right from the start. A lesson than can be learned from standardization efforts and commercial systems, is that it is difficult to add formal semantics afterwards. Moreover, having a well-grounded language enables all kinds of analysis. For example, in [WVA$^+$09] it is shown that YAWL supports all kinds of verification questions. While designing a process, the correctness can be checked at various levels. This way deadlocks, livelocks, and other anomalies can be detected in an early stage. As Figure 3 shows, YAWL can be connected to ProM (`www.processmining.org`) for various types of process mining. This allows for the discovery of patterns, case prediction, conformance checking, social network analysis, etc. For example, in [RWA$^+$08] it is shown how the connection between YAWL and ProM can be used for operational decision support.

4 SPICA's contracts and negotiation protocol: an overview

This section highlights some basics about SPICA's contracts and negotiation protocol. For simplicity's sake some details were omitted. The complete description is presented in [BMM08].

Usually, a negotiation process consists of determining the price for an item. Our negotiation process is more general, i.e.: (a) the items to negotiate may involve values other than prices or numbers. Thus, we prefer to think of the "best value", instead of the "highest" ("lowest") value. Sometimes, we use the typical negotiation jargon in the broad sense, e.g., the phrase "pay more" would mean to offer a better value (in a negotiator's perspective); (b) the negotiation process may concern something other than a physical item (e.g., a requirement or a quality criterion to be met). Thus, we use the term "goods" in this broader sense.

This section is organized as follows. Section 4.1 reviews briefly the format of SPICA's contracts and the roles run by the negotiation partners. Section 4.2 overviews the main data exchanged by the negotiators within a negotiation process. Section 4.3 presents the negotiation messages that convey such data and shape the interactions among the negotiators. Section 4.4 describes the overall approach we adopted to implement SPICA'S negotiation framework on top of YAWL system.

4.1 The Contract and the Actors

Figure 4 depicts a class diagram for our contracts. A contract is an instance of a contract model. A contract model is composed of clauses. A clause may have one or more properties. A property is an attribute to be negotiated. It has a name (which is unique in the contract) and may appear in more than one clause. The negotiation process aims at assigning values to properties.

A property is negotiated once. If it appears in more than one clause, once negotiated, its value holds for all occurrences. There are two kinds of properties: simple properties and compound properties. A *simple property* holds a scalar value. A *compound property* is a vector of scalar values and each entry corresponds to a different partner.

The partners in a contract are identified by unique names. Each clause may have two sets of partner names: the *authorized* partners and the *obliged* partners. An obliged partner must perform some action to produce the intended result. Authorized partners have the right to receive such a result.

Consider, for instance, the clause below. Two parties – a farm and a storage provider – engage a negotiation process over this clause. They will haggle over the storage space in cubic meters (property `QC`) and the price (property `PC`) of each cubic meter. The properties `OBLIGED` and `AUTHORIZED` are assigned with the names of the parties who agreed on the values for `QC` and `PC`. Note that, in this example, there is only one obliged negotiator (the storage provider) and only one authorized negotiator (the farm). Note also that the names of these (simple) properties are preceded by $ and properties that refer to partner names are preceded by @.

```
The party @OBLIGED agrees to make available $QC cubic meters at a cost
of $PC per cubic meter.
```

The next example is quite similar to the previous one. Now, several storage firms (St 1, St 2, etc) promise to offer space to the farms. Each firm will make available a different amount of space at different prices. Note that all properties in this example are compound and, thus, are preceded either by `@@` or `$$`.

```
The party @@OBLIGED agrees to make available $$QC cubic meters at a
cost of $$PC per cubic meter.
```

Conversely, in case space has the same price for all firms, but each firm provides a different amount of space, the clause would have been written like (note that there is only one `$` before `PC`):

```
The party @@OBLIGED agrees to make available $$QC cubic meters at a
cost of $PC per cubic meter.
```

The actors in a negotiation setup are the *negotiators* and the *notary*. There is a special kind of negotiator – the *leader* – who orchestrates the negotiation. Every negotiation setup has at least two negotiators (the leader is always present). More negotiators are possible. Unlike other contract models, SPICA supports several kinds of negotiation styles for the negotiation of a single contract. This is made possible by defining typical *negotiation patterns* — e.g., auction and ballot. The notary is a trustworthy third-party. It mediates some negotiation patterns (e.g., ballots) and is responsible for building the contract instance after a successful negotiation.

Some negotiators may be *proxy negotiators*, i.e., they represent a group of negotiators. The negotiators in such a group do not take part directly in the negotiation process, but they will always be represented by the proxy negotiator. However, the proxy negotiator cannot sign the contract (only the negotiators it represented can sign).

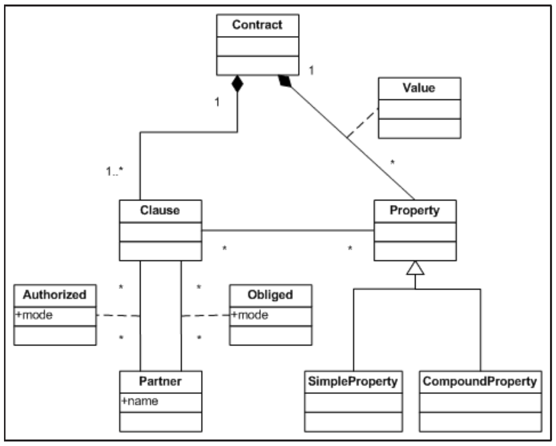


Figure 4: *Contract.*

4.2 Main Data Types

The negotiation process begins with a setup phase. In this phase, all necessary setup operations are performed, such as: the contract model to be negotiated is determined and a new negotiation instance is created, the negotiators register in. Its main outcome is twofold. First, it creates a new negotiation instance that is distinguished by an identifier. This identifier is named `nid` throughout the paper and used in virtually all messages exchanged within that negotiation process. Second, the negotiators and their roles are determined. Negotiators have distinct names and credentials. The name identifies the negotiator and is used to address messages. The *credential* states its capabilities in the negotiation process, e.g., some negotiators may have veto power in a ballot.

The negotiation of a contract's properties may involve several negotiation rounds. Each round runs one of the possible negotiation styles (bargain, ballot, auction), aims at assigning values to the properties of a particular clause, and establishes who are the obliged and the authorized partners regarding such a clause. The actual negotiation style, the obliged and benefited partners and other attributes that configure the negotiation round are described by means of *negotiation descriptions*, which are represented by `nd` throughout this paper.

The properties to be negotiated – i.e., the subject of the negotiation – are not included in a negotiation description, rather, they are described by means of an RFP (request for proposal), an offer, an RFI (request for information) or an Info (information).

An RFP is an invitation. A negotiator A sends an RFP to a negotiator B asking for a value for one or more properties. It may state restrictions over the expected answer, e.g., a maximum value for a given property. RFPs are represented by either `rfp` or `r( )`throughout the paper.

An offer is a promise. Typically, it is an answer for an RFP. A negotiator answers an RFP by sending back an offer that confirms the values of predefined properties and proposes values for the desired properties. The offer must comply with the restrictions indicated in the RFP. The negotiator that issued an offer is committed to it. Offers are represented by `offer` or `o( )` throughout the paper.

An RFI is similar to an RFP: it requests values for properties. In addition, it also requests lower and upper bounds for them. An RFI is answered by an Info. An Info is similar to an offer; however, the negotiator who issues an Info is not committed to it. An RFI is represented by `rfi` and an Info by `info`.

RFPs, offers, RFIs and Infos have also other data elements, such as, unique identifiers respectively, `rid` (for RFPs), `oid` (for offers), `riid` (for RFIs), and `iid` (for Infos); the credential of the negotiator who has created it, and the list of names of the receiver negotiators. The credential for a particular negotiator, say `n1`, is represented by `!n1` in the figures presented in the paper.

4.3 Message types

Table 1 presents the types of messages that are provided by SPICA Negotiation Protocol and the attributes they convey. They are used in the negotiation patterns described in Section 5. There are also a few extra messages needed by the framework's implementation. They are called *control messages* and will be presented later.

Most of these messages convey a few common parameters, namely: the sender name (`from`) that can be used to address a response message; the identification of the negotiation instance (`nid`) the message corresponds to, and the negotiation description (`nd`).

As an introductory example consider Figure 5, suppose that a farm (`F`) would bargain over the freight fee (`ff`) with a transportation company (`TC`) to deliver coffee bags to the docks. In this scenario, `F` will send an RFP to TC by means of a `Rp` message asking `TC` how much it would charge the transportation (i.e., it asks a value for `ff`). This message conveys four attributes. The first holds the name the sender used to register in the negotiation. The receiver uses this name to address a reply message. The second (`nid`) is the negotiation instance identification. The third attribute describes the details of the current negotiation style (a bargain, in this example). Each negotiation style has specific attributes within this description, but all of them share a common subset. These three attributes are present in most of the negotiation messages. The last attribute is an RFP that asks a value for `ff`.

`TC` receives this message, analyses the received RFP and answers it by sending back an offer that assigns a value to `ff`. This offer is conveyed by means of an `Ra` message.

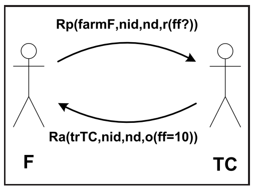


Figure 5: *A negotiation message and an answer.*

Request Proposal. This message communicates an RFP (`rfp`). This message is within the context of a particular negotiation style (described by means of (`nd`))	`Rp(from,nid,nd,rfp)`
Request Information. It communicates an RFI (`rfi`).	`Ri(from,nid,nd,rfi)`
Request Agreement. It communicates an offer.	`Ra(from,nid,nd,offer)`

Request Auction Step. The leader asks the notary to conduct an auction step. An English auction step is described by an RFP (`rfp`) and a Dutch auction step is described by means of an offer (`offer`).	`Ras(from,nid,nd,rfp)`
	`Ras(from,nid,nd,offer)`

Request Ballot. The leader asks the notary to conduct a ballot process. The issue to be voted is described by (`ballot`)	`Rb(from,nid,nd,ballot)`
Request Vote Agreement. The notary uses this message to request a vote from a negotiator when the expected answer is “agree” or “disagree”. (`bid`) identifies a specific ballot instance.	`Rva(from,nid,nd,bid,ballot)`
Request Vote Preference. The notary uses this message to request a vote from a negotiator when the expected answer is the preferred value for a property.	`Rvp(from,nid,nd,bid,ballot)`

Answer for Request Proposal. An `Rp` message can be answered by two mutually exclusive messages. The first – `Ra` – sends an offer in response to the previous RFP. The second – `Ino` (Inform No Offer) – declines the invitation.	`Ra(nid,nd,offer)`
	`Ino(from,nid,rid)`

Answer for Request Information. An `Ri` message can be answered by two mutually exclusive messages. The first – `Ari` – sends an Info in response to the previous RFI. The second – `Ini` (Inform No Info) – informs that the negotiator will not provide the asked information.	`Ari(from,nid,info)`
	`Ini(from,nid,riid)`

Answer for Request Agreement. An `Ra` message can be answered by three exclusive messages. The first – `Aa` – agrees on the proposed offer. The second – `Ad` – disagrees on the offer. The `Ra` message can also be answered by a counter-offer by means of another `Ra` message.	`Aa(from,nid,nd,offer)`
	`Ad(nid,nd,offer)`
	`Ra(from,nid,nd,offer)`

Answer for Request Auction Step. There are two consecutive messages in response to a Request Auction Step (`Ras`) message. First, the notary sends a `Aas` message to the leader to inform the leader that the requested auction step will be performed (conversely, `Nas` to refuse doing so). At the end of the auction step, notary sends an `Ica` message to the leader containing all received bids.	`Aas(nid,aucid,rid)`
	`Aas(nid,aucid,oid)`
	`Ica(nid,aucid, offer_lst)`
	`Nas(from,nid,rid)`
	`Nas(from,nid,oid)`

Answer for Request Ballot. Notary returns two consecutive messages in response to a `Rb` (Request Ballot) message. First, the message `Ab` acknowledges the leader that it will conduct the requested ballot (conversely, `Nb` to refuse it). At the the end of the ballot, the notary returns to the leader the ballot’s result by means of an `Ibr` message.	`Ab(from,nid,bid,rid)`
	`Ab(from,nid,bid,oid)`
	`Ibr(from,nid,bid,bresult)`
	`Nb(from,nid,rid)`
	`Nb(from,nid,oid)`

Answer for Request Vote Preference. It answers a previous `Rvp` message. The voter sends the notary its vote, abstention, or veto (if applicable). This message also conveys the voter’s credential (`crd`), allowing the notary checking, for instance, if the voter has veto power.	`Av(from,nid,bid,crd,vote)`
Answer for Request Vote Agreement. The same `Av` message, but the only allowed alternatives are: abstention, `ok` (i.e., agree), `nok` (i.e., not agree), and veto (if applicable).	`Av(from,nid,bid,crd,vote)`

Table 1: Negotiation messages.

4.4 Implementation Overview

The participants are Web services that use the messages presented in the previous section (Section 4.3) to develop the intended negotiation (see [BMM08]). Thus, sending a negotiation message to a participant means invoking a particular operation of a service. Thus, in principle, each participant (web service) can make sure it complies with the protocol rules and the negotiation can proceed without any external help. However, there are a few chores that would be better undertaken by a tailor-made middleware, providing a clear separation between the participant's negotiation strategy implementation and message transportation. A participant's implementation, should not be aware of details like: dispatching messages to the right receivers (i.e., invoking web services operations), verifying (and handling) whether a message has aged, to name just a few.

We use the YAWL WfMS as such a middleware. It is used to model and to execute the protocol. In this model, a workflow task typically represents a message received or to be sent. Thus, when a message is received by the middleware, the corresponding task is enabled. This causes the execution engine to invoke a specific routine (i.e., a Web service operation) to handle such a message. Handling a message means analysing it and sending back an answer for it. Thus, there must be another already enabled task responsible for receiving such a response. For this purpose, these tasks are, in general, arranged in pairs. See Figure 6. The first task (`T1d`) receives a negotiation message (e.g., `Rp`). The underlying workflow system instrumented with additional software validates the message, dispatches it to the intended negotiator (`Negotiator 1`) and, simultaneously, enable the second task by sending it a an internal *control message* (ACK, in this case). The second task (`T1f`) is responsible for receiving the negotiator's answer (e.g., `Ra`) and forwarding it to the next task (e.g., `T2d`). Thus, tasks like the first one are *dispatch-like tasks* and like the second one are *forward-like tasks*. Tasks `T2d` and `T2f` have similar arrangement. This kind of arrangement occurs frequently in the patterns presented in Section 5.

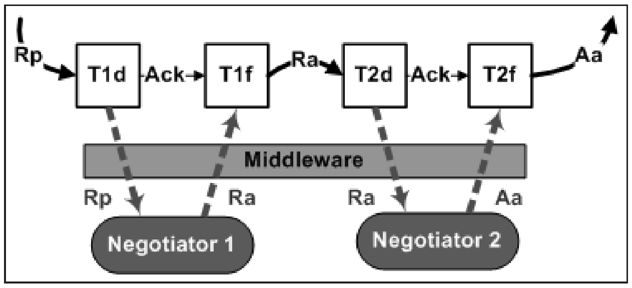


Figure 6: *Tasks arranged in pairs.*

It is worth having a more accurate view on how data is transfered between tasks. A task exists within a net and has a set of input and output variables. A net may have local variables. When a task gets enabled, its input variables are assigned with values contained in such local variables by means of XQuery expressions, i.e., an XQuery expression (that may refer to local variables) is executed and the result value is assigned to an input variable. The task's execution uses such input values, performs the intended computation and its results are (internally) assigned to the task's output variables. Finally, the output values may be assigned to local variables by means of other XQuery expressions, being available to the next enabled tasks. There is no direct data transfer between an output variable to an input variable of the next task.

Although Figure 6 may suggest that an arc between two tasks conveys data from one to the other that is not accurate. In a nutshell, for those familiar to Petri nets, an arc only conveys black tokens; when a task is enabled, it gets its inputs by executing XQuery expressions.

5 Description of Patterns

This section presents several negotiation patterns supported by SPICA protocol.

Each pattern is composed of several sections (Figure 7). *Pattern name* and *Known as* are used to identify the pattern. The *Motivation* part elaborates on the goal and presents the context of the pattern. This part uses common concepts and does not use SPICA protocol's parlance. The *Problem description* part presents the problem to be handled by the pattern at in terms of SPICA's concepts. *Problem Solution* details a possible solution to the problem, i.e., it shows how the message types (described in Section 4.3) can be combined to achieve the expected results and how the participants interact. *Implementation of Solution* shows the pattern realization in YAWL and addresses some specific setups or additional elements needed for the presented solution.

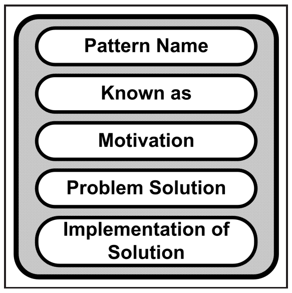


Figure 7: *Sections of a pattern.*

5.1 Pattern name: *Bargain* Known as: -

Motivation: The simplest style of negotiation is a bargain. Two negotiators haggle over a value to be settled (the subject of the negotiation). They usually exchange counter-offers – e.g., between set of farms and the transportation company
Problem Solution:

Bargains are characterized by the following pattern. A leader starts a bargain requesting a proposal from the other negotiator. The leader and the negotiator exchange offers and counter-offers until the negotiation comes to a conclusion (either by an acceptance or final disagreement message). Figure 8 shows such a pattern, where the offer and counter-offers appear in messages 2,3 and 4. In more detail, the figure shows five messages exchanged between the leader (`ld`) and a negotiator (`n1`). It starts **(1)** when the leader (`ld`) sends a request proposal message (`Rp`). The negotiator responds **(2)** with either (a) an offer conveyed by a request agreement message (`Ra`), or (b) a non-offer message (`Ino`) declining to take part of this bargain. If the negotiator has decided to take part, it sends an `Ra` message with an initial proposal. The negotiation carries on. **(3)** The leader does not agree with the proposal and sends a counter-offer (another `Ra` message). The negotiator (4) sends another counter-offer. This situation is repeated until one of the participants (in this case, the leader) comes to a conclusion finishes the process by either (a) agreeing by means of an agreement message (`Aa`), or (b) disagreement message (`Ad`).

Let us now examine the parameters in each message.

(1) When the leader requests a proposal from Negotiator `n1` (message `Rp`), the parameter (`nid`) identifies the negotiation instance, created during the negotiation setup (not shown in pattern). The second parameter (tuple `d`) is negotiation description. It informs that this negotiation concerns a bargain (`BG`), the clause being negotiated (`cj`),the obliged party list (`ol`), and the authorized party list (`al`). The third parameter describes the proposal requested by means of an RFP (tuple `r`). Note that the first and second parameters happen in all pattern's messages.

The RFP's parameters are the following. The value `r1` identifies the RFP. An RFP has two set of properties: one pre-assigned by the originator (`pas`) and one set of asked properties (`aps`). Finally, `rt` describes the restrictions (i.e., a boolean expression) on the expected answer. An RFP also has the identification of the originator and of the receiver(s); they are implied by the arrows in all patterns presented in the paper. Here, the originator is the leader `ld` and the receiver is negotiator `n1`.

In the counter-offer cycle started at **(2)**, the partners use `Ra` messages to exchange offers (tuple `o`, in the third parameter). The offer, identified by `oi1`, correlates to the starting `Rp` message (`r1`). The offer must also mention and assign all the properties of the corresponding RFP. Thus, it contains the same assignment in the previous RFP (`pas`) and assigns values for all the asked properties (`aps`). This set of assignments is represented by `assgn1` in the figure. The offer also repeats the RFP's restrictions (`rt`). The last but one parameter means that the negotiator `n1` agrees with this offer (because it has created the offer itself), and the last parameter means that the offer has not been evaluated by the receiver. Like an RFP, an offer also identifies the originator and the receivers, implied by the arrow as well. **(3)** The leader does not agree with the proposal and sends a counter-offer (another Request Agreement message). It is similar to the previous one with different assignment for the asked properties. Note that the counter-offer correlates to the previous offer (`oi1`), instead of the starting RFP (`r1`). **(5)** (a) When a negotiator (the leader in this example) agrees on an offer, the agreement message (`Aa`) contains exactly the same offer (`assgn3`) the leader has received (which it agrees upon); however the last but one offer's parameter contains the identification of both the leader and the negotiator (since both agree upon `assgn3`) and the last parameter (`A`) states that the offer was agreed upon. Conversely, (b) a disagreement message (`Ad`) to the other negotiator is similar to the `Aa` message, but the last parameter that is set to `D`.

Implementation of Solution:

This pattern's implementation in YAWL is shown in Figure 9. In this figure, arrows correspond to messages and boxes to tasks. Tasks labeled with `L` inside a pentagon represents tasks performed by the leader and the ones labeled with a `Ng` inside a circle are performed

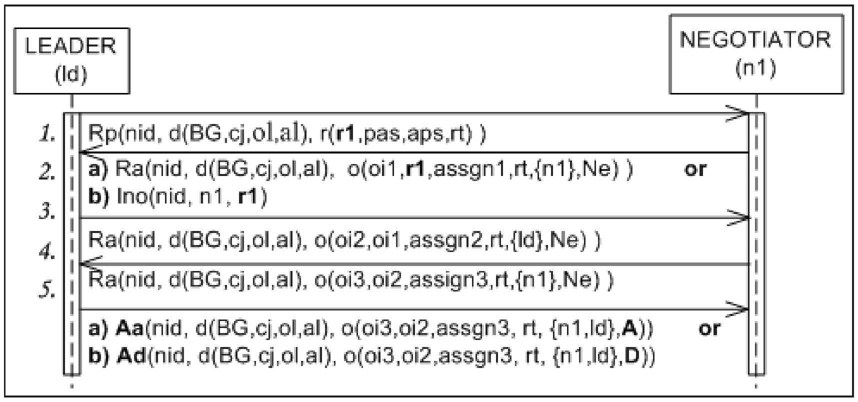

LEADER
(ld)
NEGOTIATOR
(n1)
1.
Rp(nid, d(BG,cj,ol,al), r(r1,pas,aps,rt))
2.
a) Ra(nid, d(BG,cj,ol,al), o(oi1,r1,assgn1,rt,{n1},Ne)) or
b) Ino(nid, n1, r1)
3.
Ra(nid, d(BG,cj,ol,al), o(oi2,oi1,assgn2,rt,{ld},Ne))
4.
Ra(nid, d(BG,cj,ol,al), o(oi3,oi2,assign3,rt,{n1},Ne))
5.
a) Aa(nid, d(BG,cj,ol,al), o(oi3,oi2,assgn3, rt, {n1,ld},A)) or
b) Ad(nid, d(BG,cj,ol,al), o(oi3,oi2,assgn3, rt, {n1,ld},D))

Figure 8: *Bargain interaction pattern.*

by a negotiator (other than the leader). This process starts at the topmost circle (with a triangle inside) and finishes ate the bottommost circle (with a square inside).

An offer and an RFP have a valid lifespan. A late offer is only discarded. Note that the tasks on the left are related to the leader and the ones on the right relate to the negotiator.

An `Rp` arrives at the task `Negotiator Received RFP`. The task is accomplished by *dispatching* the `Rp` message to the intended negotiator. The task's result is an `ACK` message sent to the task `Negotiator Creates Offer` (enabling it). The negotiator uploads an answer via the infrastructure. The infrastructure executes this task by matching the `ACK` message and the answer provided by the negotiator and *forwarding* the answer to one of `RFP Declined` or `Received Negotiator Offer` or `Received Notification from Negotiator`, depending on the negotiator's reply.

This two-task arrangement appears on most of the interactions in this net. Dispatch-like tasks are identified by a little triangle on the bottom-leftmost corner of its label (e.g., `Negotiator Received RFP`) and the forward-like ones have a small mark on its top-rightmost corner (`Negotiator Creates Offer`). However, some sent messages do not demand a response (e.g., `Aa`. In this case, there is only the dispatch-like task (e.g., `RFP Declined`) and its resulting ACK message is just discarded.

5.2 Pattern name: *English Auction* Known as: *Ascending Auction*

Motivation: The subject of the negotiation is in dispute by several negotiators – e.g., the several coffee farms. Supposedly the negotiator who is more willing to get the item will offer a better value for it.

Problem Solution: In Figure 10, there are a leader (`ld`) and a notary (`nt`). There are several bidders that are represented by `n+`. One individual bidder is represented by `nk`. An auction is characterized by the following pattern.

The auction is controlled by the leader and helped by the notary. It develops in auction steps. In one auction step (a) the leader requests the notary to broadcast the subject of the auction to the bidders and to accept the corresponding bids. The auction's subject defines

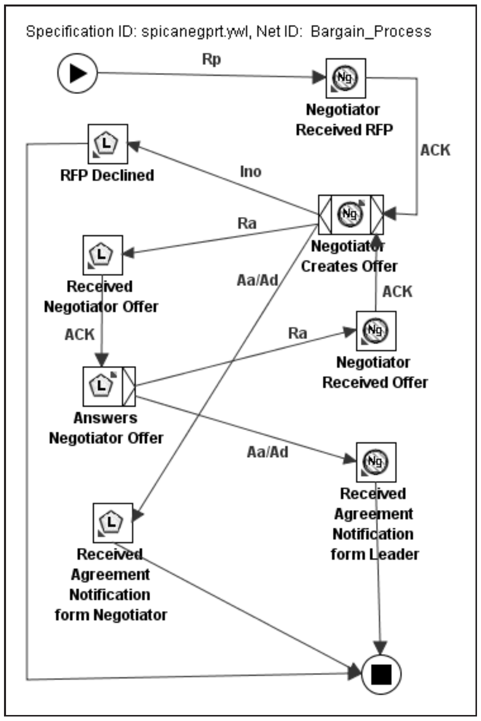

Specification ID: spicanegprt.ywl, Net ID: Bargain_Process
Rp
Negotiator Received RFP
ACK
RFP Declined
Ino
Ra
Negotiator Creates Offer
Received Negotiator Offer
Aa/Ad
ACK
ACK
Ra
Negotiator Received Offer
Answers Negotiator Offer
Aa/Ad
Received Agreement Notification form Leader
Received Agreement Notification form Negotiator

Figure 9: *Bargain process (YAWL)*

restrictions over the bids (e.g., a minimum price). (b) the bidders send bids to the notary. (c) The notary collects a few bids and sends them all to the leader, finishing the auction step. The leader may either agree on one or more bids received in the last or any previous steps or request the notary to run another step. In the latter case, the leader may increase the restrictions over the expected bids (e.g., increase the mininum price).

Let us examine the dynamics of the auction pattern presented in Figure 10. **(1)** The leader asks the notary to announce the auction (message `Ras`).

The first two parameters, as usual, are the negotiation identification and the negotiation description. The negotiation description for an auction is similar to the one of a bargain, but conveys a few extra attributes: the list of the negotiators that take part in the auction ({`n1,n2,...`}), the number of expected bids (`max_answr`)), and the time interval the notary should wait for the expected bids (`tmout`). These last two parameters indicate that each auction step has a predefined lifespan and may receive many bids (typically, it is only one). Aged or excess bids are only discarded. The third parameter conveys the RFP that describes the property(ies) to be auctioned. Typically, such an RFP imposes a restriction on the asked properties (e.g., a minimum acceptable price). **(2)** The notary signals the leader that it will control the auction (message `Aas`) and informs the identifier for this auction step (`a1`) and the RFP's identification (`r1`). **(3)** The notary broadcasts the negotiation description and the RFP to all negotiators. The broadcast is represented by a stair-shaped arrow. **(4)** Each negotiator answers this request by (a) sending an offer to the notary (message `Ra`), or (b) informing the notary that it is not interested in the current auction step (message `Ino`). The offer has a suitable assignment for each asked property (not shown). The last two parameters are shown. They state that the negotiator `nk` agrees with this offer and that the offer was not evaluated by the counter-party (`Ne`). **(5)** The notary collects these offers and sends them back to the leader by means of an `Ica` message. The last parameter is the list of received offers. It does not include the `Ino` messages sent in step 4b (if any). **(6)** The leader chooses the best offer(s) (the *step winner(s)*) according to its own criterion, and starts another auction step with a more restrictive RFP (e.g., a higher minimum acceptable price). This cycle repeats and **(7)** eventually no negotiator proposes an offer (the leader receives an empty list of offers). **(8)** The leader agrees with the best offer(s) of the previous round (identified by `oy`) and **(9)** may disagree with all defeated offers (or simply let them age).

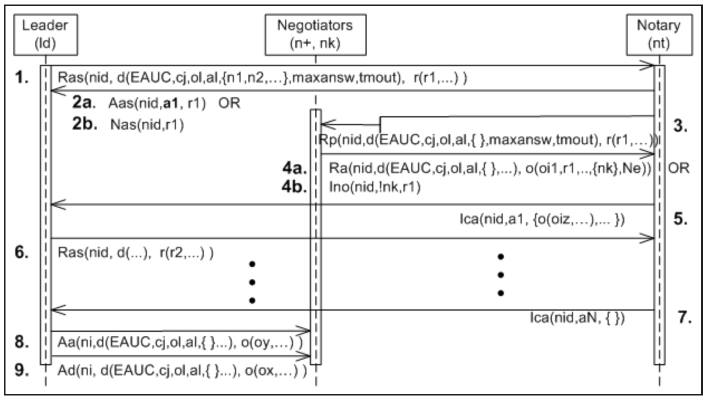

Leader (ld)
Negotiators (n+, nk)
Notary (nt)
1. Ras(nid, d(EAUC,cj,ol,al,{n1,n2,...},maxansw,tmout), r(r1,...))
2a. Aas(nid,**a1**, r1) OR
2b. Nas(nid,r1)
3. Rp(nid,d(EAUC,cj,ol,al,{ },maxansw,tmout), r(r1,...))
4a. Ra(nid,d(EAUC,cj,ol,al,{ },...), o(oi1,r1,..,{nk},Ne)) OR
4b. Ino(nid,!nk,r1)
5. Ica(nid,a1, {o(oiz,...),... })
6. Ras(nid, d(...), r(r2,...))
7. Ica(nid,aN, { })
8. Aa(ni,d(EAUC,cj,ol,al,{ }...), o(oy,...))
9. Ad(ni, d(EAUC,cj,ol,al,{ }...), o(ox,...))

Figure 10: *English auction interaction pattern.*

Implementation of Solution:

Consider Figure 11. It models both English and Dutch auctions. Dutch auction is described in Section 5.6. The leader's tasks are on the left, the notary's are in the middle, and the negotiator's (bidder's) are on the right. A new auction request (`Ras`) arrives at task `Requested Auction Step`. It is dispatched to the notary. The notary may either refuse or accept conducting this auction step. In the first case, it answers an `Nas` message to leader. In the second case, the notary starts an internal timer to control the lifespan of the initiating auction step and answers two simultaneous messages: `Aas` to the leader accepting the job and `Rp` to the bidders announcing the new auction step. To model such a behaviour and due to the fact that `Aas` and `Rp` will not be uploaded by the notary exactly at the same point in time, but in any order, `Requested Auction Step` has a AND-split that sends an `ACK` to `Notary Decides I` and `Notary Decides II` enabling them to receive all those three messages. Let us discuss what happens in either cases.

Recall that upon receiving the `ACK` message, both tasks `Notary Decides I` and `Notary Decides II` are enabled. In the first case (the refusal), the `Nas` message may reach either of them and, as a result, control is diverted to `Notary has Refused`, which dispatches the message to the leader.

Conversely, the notary might have accepted. In this case, control heads for `Start New Auction Step`. It is just a synchronization task (AND-join) that waits for the arrival of both messages `Aas` and `Rp`. It directs the message `Aas` to task `Notary has Acknowledge` and message `Rp` to task `New English Auction`. It relies on non-trivial XQueries to direct the received messages to the right tasks. It also enables task `Collected Bids` by sending the same `ACK` as of `Requested Auction Step`. Recall that Section 4.4 observed that an arc itself does not convey any data. This enabling is a clear example: task `Collected Bids` gets enabled by the arc originated from `Start New Auction Step`, but obtains its input data of a local variable assigned at the completion of task `Requested Auction Step`.

Task `New English Auction` dispatches the `Rp` message to the bidders and sends an `ACK` to task `Waiting Bids`. Whenever `Waiting Bids` receives a bid, it forwards the bid to task `Receive Bids`. This task dispatches the bid to the notary and re-enables `Waiting Bids` allowing it to receive other bids from other negotiators. The notary stores the received bids in its private database.

When the auction step's lifespan has elapsed, the notary collects the bids it has received, creates an `Ica` message containing the received bids and uploads it to the engine by means of the `Collected Bids` task. This message is forwarded to task `Leader Waiting Bids`, which dispatches it to the leader. Finally, the leader can start a new auction step (`Ras`) or agree with a few bids (message `Aa`) and disagree with the defeated ones (message `Ad`). These messages are dispatched to the corresponding negotiators by task `Receive (Dis)Agreement`. The bottommost arc is used when there was no bid at all (auction failed) or it was a Dutch auction. Note that tasks `Notary has Acknowledged` and `Receive Bids` do not have their forward-like counter-parts. This happens because the received message do not demand an answer.

It is worth having a closer look at task `Collected Bids` and its surroundings. Note that this task should be enabled to receive the upcoming `Ica` message, even though there should be no bids at all. Its input is the very same `ACK` delivered by task `Requested Auction Step`. Thus, this `ACK` helps both `Aas`, `Rp`, and `Ica` to be correlated correctly with the previous (parent) `Ras` message. Consider now task `Receive Bids` and the arcs coming from `Collected Bids` and going to `Leader Waiting Bids`. Strictly speaking, these arcs are not necessary and are somehow a bit odd: `Receive Bids` will be execute always before `Collected Bids` (The latter will pack the bids received by the former). However, those arcs were added due to two facts: (a) YAWL enforces the *soundness property* ([Aal98]) and (b) `Receive Bids` may receive no bids at YAWL preventing it to be enabled by its previous task (`Waiting Bids`).

Lets start with (a) the soundness property. Note that `Receive Bids` is reachable from the net's input condition. Thus, there must be a path from this task to the output condition, otherwise, the soundness property would be violated. So, the arc from `Receive Bids` to `Leader Waiting Bids` is added. However, (b) Receive Bids may receive no bids, not getting enabled. If it does not get enabled, `Leader Waiting Bids` will not be enabled as well, deadlocking the auction net. The solution is this arc from `Collected Bids` to `Receive Bids`. Now, `Receive Bids` may be enabled to receive (and dispatch) a bid or enabled by `Collected Bids`. In the first case, `Receive Bids` should handle the incoming bid in the usual way. In the second, it should do nothing but enabling task `Leader Waiting Bids`. To avoid mingling both situations, `Collected Bids` sends a `NULL` message to `Receive Bids`. This kind of control message is not really delivered, but discard by the middleware.

This model uses two cancellation regions: for tasks `Notary has Refused` and `Collect Bids`. The first region comprises tasks `Notary Decides I` and `Notary Decides II` and their outgoing arc. Recall that both tasks were simultaneously enabled, but `Nas` message reaches one of them. This cancellation region makes sure the other task is canceled. The second region comprises all tasks and arcs on the right of task `Collected Bids`, disabling the reception of late bids.

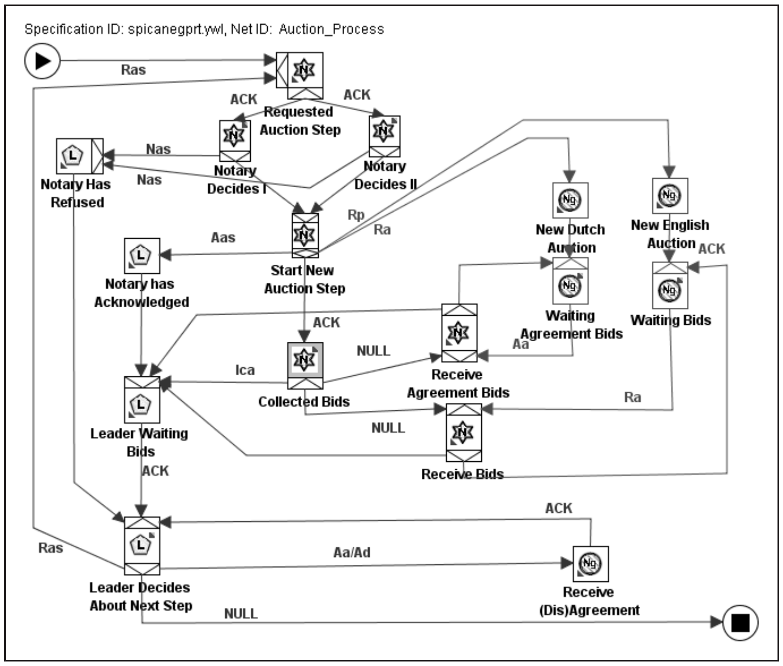


Figure 11: *Auction process (YAWL)*

5.3 Pattern name: *Open Ballot* Known as: -

Motivation: Some negotiation issues comprise a set of options to be chosen by the negotiators. The goal of the negotiation is to determine the most preferred option.

Problem Solution:

This pattern, shown in Figure 12. The ballot's issue may include several properties. Most of the properties' values are assigned in advance. There is exactly one property not assigned and a predefined list of possible values for this property. The ballot's goal is to assign the most preferred value to this property. There are a leader (`ld`) helped by the notary (`nt`) and several negotiators, all of them represented by `n+`. An individual negotiator is represented by `nk`. **(1)** The leader asks the notary to conduct the ballot process (`Rb` message). This message conveys three parameters: the negotiation instance identification (`nid`), the negotiation description (tuple `d`) and the ballot description (tuple `b`). The negotiation description informs that this negotiation concerns a ballot (`BLT`). Besides the common attributes, it also conveys the list of voters ({`n1,...`}). The leader is not always a voter: if so, it will be included in this list. The negotiation description includes other parameters that specify the dynamics of the ballot, all represented by `xp`, such as: how much time to wait for votes, the number of votes needed to approve the issue, the minimum number of votes to an alternative be elected, how to handle ties (e.g., by considering the leader's vote twice, considering a tie as an approval, etc).

The ballot description comprises an RFP (tuple `r`, showing only the RFP identifier) and the set of alternatives ({`a1,...,az`}). This RFP has exactly one unbound property. Thus, voting means choosing one among the alternatives. **(2)** (a) The notary acknowledges the leader that it will conduct the ballot (message `Ab`) and informs the ballot identifier (`bid`). (b) Conversely, the notary refuses this job (message `Nb`) and there is no further interactions. **(3)** If the notary agreed, it broadcasts the ballot's subject to all negotiators (`Rvp`). If the leader is a voter, it also receives this request for voting. The dashed line indicates that it is not always the case. **(4)** A voter sends its vote to the notary (message `Av`): (a) Each vote may choose one alternative; or (b) may be an *abstention*, or (c) a veto (if applicable). A non-authorized veto is considered as an abstention. Note that parameters `nid` and `bid` correlate the vote to the the negotiation instance and to the ballot. The vote also conveys the voter's credential (`!nk`). **(5)** The notary counts the votes (*xi* is the number of votes the alternative *ai* has received) and broadcasts the result. The result may be approved, not approved, or vetoed. The result of a ballot is disclosed by means of `br` data type, written as:

```
br(blst,{ch1:nv,...},ofr)
```

The first parameter is the ballot status, i.e., whether the ballot was approved (`A`), not approved (`Na`) or vetoed (`V`). The second parameter lists how many votes (`nv`) each choice (`chI`) has received. The approved choice is in the beginning of this list. In case of an approved ballot, the last parameter (ofr) conveys an offer that assigns the chosen value to the respective property. Such an offer is agreed by the leader and the notary (the notary is a kind of a proxy for the voters).

Implementation of Solution:

Figure 13 shows the ballot's implementation. It implements both the open ballot pattern and the close ballot pattern (Section 5.6). This net follows the same rationale of the auction's net (Section 5.2). However, it is a bit simpler. Note that the auction's net also implements two patterns (English and Dutch auction). The bids for each pattern are semantically and structurally different. Thus, each kind of bids is received in different parts of the auction's net. Conversely, there is no substantial difference between votes of each pattern and they can be handled equally. Another difference is that, whereas the auction step's result is only sent to the leader the ballot's result is broadcasted to all negotiators, including the leader.

In the case of an open ballot, the notary, via task `Start Ballot`, forwards the message `Rvp` to the negotiators via task `Request Vote`. Later, the notary receives the votes enclosed within message `Av`

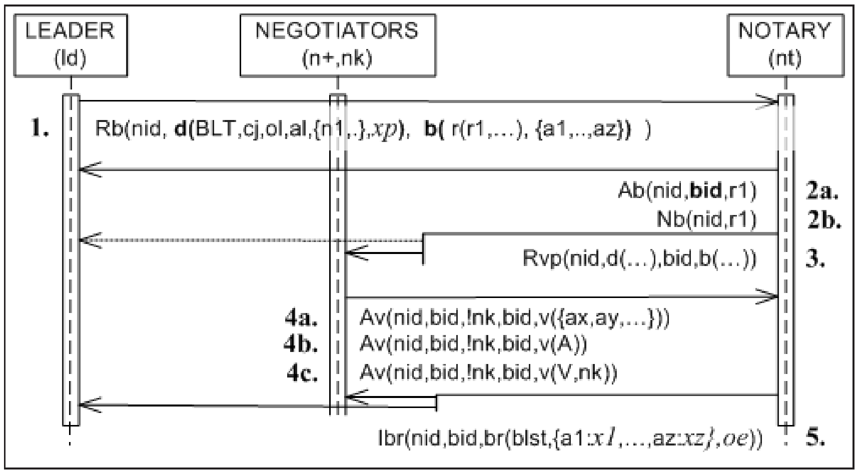

LEADER
(ld)
NEGOTIATORS
(n+,nk)
NOTARY
(nt)
1.
Rb(nid, d(BLT,cj,ol,al,{n1,.},xp), b(r(r1,...), {a1,..,az}))
Ab(nid,bid,r1)
2a.
Nb(nid,r1)
2b.
Rvp(nid,d(...),bid,b(...))
3.
4a.
Av(nid,bid,!nk,bid,v({ax,ay,...}))
4b.
Av(nid,bid,!nk,bid,v(A))
4c.
Av(nid,bid,!nk,bid,v(V,nk))
Ibr(nid,bid,br(blst,{a1:x1,...,az:xz},oe))
5.

Figure 12: *Open Ballot interaction pattern.*

5.4 Pattern name: *Request for Information* Known as: -

Motivation: When preparing a proposal, a negotiator knows what is convenient to itself and it tries to guess what should be acceptable by the counter-party(ies). It is specially important when the proposal must be accepted by several (or all) parties. Thus, in order to make an educated guess, the negotiator may first ask the other parties to provide upper-bounds and lower-bounds for a few properties. After, the negotiator analyzes the answers and prepare its proposal.

Problem Solution:

All the presented patterns can be augmented by a number of requests for information (RFIs) sent by the leader before starting the proper negotiation.

An RFI is very similar to an RFP. It is written as:

```
ri(riid,pas,iaps).
```

The first parameter (`riid`) is the RFI's identification. The `pas` parameter is exactly the same of an RFP, i.e., it has a few property assignments. Parameter `iaps` is slightly different from RFP's `aps`. It contains properties with operators. The operators indicate what kind of information is asked for a specific property. The operator `?` asks for a value; `<`, for a lower bound the property, and `>`, the upper bound. For instance, recall the motivating example. Suppose that the exporter is about to negotiate a contract with a buyer overseas. It might ask the farms about their capacity. The following RFI:

```
ri(ri1,{prod:  coffee, typ:  arabica},{q:<, q:>, p:?})
```

tells that the originator is interested in buying `coffee` of type (`arabica`) and asks a negotiator the coffee's price (`p`), and the minimum and maximum amount it is interested in supplying at that price.

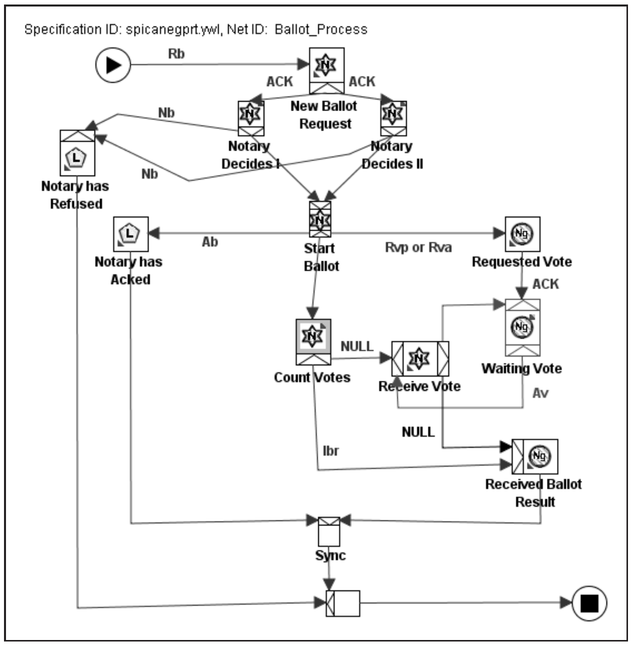


Figure 13: *Ballot process.*

The corresponding Info is written as:

```
i(iid,riid,ias).
```

An Info has an identification (`iid`) and correlates to a previous RFI (`riid`). The last parameter (`ias`) is an *information assignment*. It is a triple `op:P:v`, such that, `op` is one of the same operator used in the RFI or the operator `=`, `P` is a property name, `v` is the information provided. Thus, the following example informs that negotiator `n1` is willing to sell `coffee` of type `arabica` at `150` dollars a bag, if the receiver bought more than `300` bags. It also informs that the seller can provide no more than `500` bags. Note that the operator `=` is used for property `typ` because it had been pre-assigned by the respective RFI.

```
i(i1,ri1,{=:typ:arabic, =:prod:coffee, ?:p:150, <:q:300, >:q:500}).
```

Like an RFP or an offer, an RFI and an Info also contain the respective originators and receivers. They are implied by the arrows depicted in the figures.

See Figure 14. **(1)** The leader sends an RFI to a negotiator asking information about some properties. The first parameter is the negotiation instance identification, the second is the negotiation description, and the last parameter is the RFI. Note that the negotiation description (`nd`) may disclose the details of the negotiation in which the provided information will be used for, since the answer of a specific negotiator may depend on it. **(2)** The partner negotiator answers the request either (a) sending the asked information or (b) informing it will not answer the request.

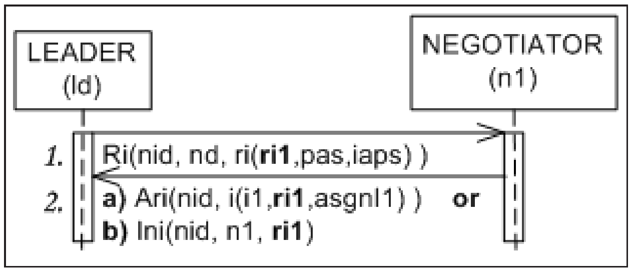

LEADER
(ld)
NEGOTIATOR
(n1)
1.
Ri(nid, nd, ri(ri1,pas,iaps))
2.
a) Ari(nid, i(i1,ri1,asgnl1)) or
b) Ini(nid, n1, ri1)

Figure 14: *Request for Information pattern.*

Implementation of Solution: Figure 15 depicts the YAWL model that implements this pattern. It is similar to a bargain with only one step.

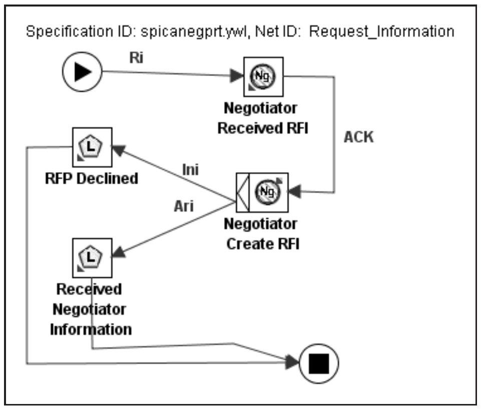

Specification ID: spicanegprt.ywl, Net ID: Request_Information
Ri
Negotiator Received RFI
ACK
RFP Declined
Ini
Ari
Negotiator Create RFI
Received Negotiator Information

Figure 15: *Request for Information implementation.*

5.5 Pattern name: *Quota minimizing/maximizing a property* Known as: -

Motivation: The distribution of quotas may impact the value of some value related to the negotiation issue (e.g., cost). Thus, the negotiation process aims at distributing wisely such quotas in a way that this value is minimized (or maximized). For instance, an industry wants to buy a great amount of coffee. There is no single supplier that can delivery the whole amount. Each supplier charges different prices. The price per kilogram may vary according the asked amount. Thus, the industry tries to buy different amounts of coffee from some suppliers such that the total cost is minimized.
Problem Solution: The remaining of this pattern only discusses the minimizing scenario. The maximizing scenario is symmetrical. There is a property `Qt` that denotes a total amount of a specific resource, good, etc that must be fulfilled or enjoyed by several parties. Each party will provide a share of this total amount at a certain cost each measurement unit (e.g., Kg). The cost (per unit) for different amounts may vary. The goal is to assign different amounts to different parties such that the total cost is minimized. Thus, there are two

properties to be negotiated together: an amount `Q` and a cost `C`. `Q` and `C` are compound properties, i.e., `Qk` and `Ck` refer to the amount (quota) assigned to and the cost per unit asked by negotiator `k`. These properties are in the same clause. The property `Qt` may be in a different clause. The sum total of the entries in `Q` must be exactly `Qt`.

For each negotiator, the leader has to determine its *fulfillment capacity* and its *cost function*. The fulfillment capacity refers to how much of the total amount the negotiator is willing (or is able) to fulfill. The cost function determines how much the negotiator will charge (or ask) for a specific amount. Some negotiators may not disclose their actual cost function, thus the leader will try to infer it.

This pattern comprises four main phases: a) determining the negotiator's capacity, b) determining the negotiator's cost function, c) based on each negotiator's cost function, determining the (near) optimal amount to be asked to each negotiator, d) negotiating with each negotiator to buy that amount.

The pattern's steps are detailed in Figure 16. The first phase is quite simple, **(1)** the leader broadcasts an RFI and asks the negotiators about the minimum and maximum amount they are interested in fulfilling. This is done by means of an Request Information (`Ri`) message. The first parameter of this message is the negotiation instance (`nid`). The second parameter is the negotiation description. In this pattern, the negotiation style is a bargain. The negotiation description also identifies the clause (`cj`) in which the leader intends to negotiate such a property, and lists the parties that will be the obliged (`ol`) and authorized (`al`) ones. The third parameter is the RFI itself. It has an identification (`ri1`), the sender's credential is shown (`!ld`) and the receivers are omitted, but inferred by the direction of the arrow, the last but one parameter specifies the value for `Qt`, and other assignments may be added if convenient. The last parameter asks for the minimum and maximum for `Q`. Each RFI has a time interval it holds valid. **(2)** Each negotiator either (a) sends back the asked information (`Ari`) or (b) refuses to provide it (`Ini`). In the first case, the message's parameters are: the negotiation instance identification (`nid`), the negotiation description (which is the same of the previous message), and the information. The first field of the information identifies the information (`i1`) and the second correlates the information to the previous RFI (`ri1`). The sender's credential (`!nk`) is also shown. The receiver is omitted, but can be inferred by the arrow. Finally, it provides the information asked: the minimum and maximum value for `Q`. The leader can check if the negotiators can fulfill the whole amount `Qt`. If not, the negotiation does not progress.

In the second phase, the leader broadcasts several RFIs asking about the cost for different amounts. **(3)** The leader asks the cost (per unit) if the amount (`Q`) to be ordered were `q`. **(4)** Each negotiator provides this information (message `Ari`) or may refuse to inform it by means of an `Ini` message. The leader believes that the cost for bigger amounts would be cheaper. Thus, it repeats step (3) and (4) several times with different values for `q`. The answers are used to infer the cost function of each negotiator.

In the third phase, **(5)** the leader uses some optimization technique to determine the amount `Qk` to be asked to negotiator `nk` that minimizes the total cost `CT`. Finally, in the fourth phase, the leader negotiates the calculated amount `Qk` with each negotiator `nk`. **(6)** The leader sends an RFP for each negotiator `nk` comprising the calculated amount `Qk`. The RFP also asks for the cost `C`, but restricts it to the price informed in step 4 proportionally to `Qk`. **(7)** The negotiators send back offers. Presumably, all negotiators will send offers (message `Ra`), but they can also decline the invitation by replying `Ino` messages (not shown). **(8)** If all negotiators send back offers, (a) The leader agrees on all offers; otherwise, (b) the leader disagrees on all offers (it will not be able to fulfill the full amount `Qt`). The offers in the messages `Aa` and `Ad` are the same of the one in the previous step, but the list of negotiators that agree on (or disagree) is properly updated.

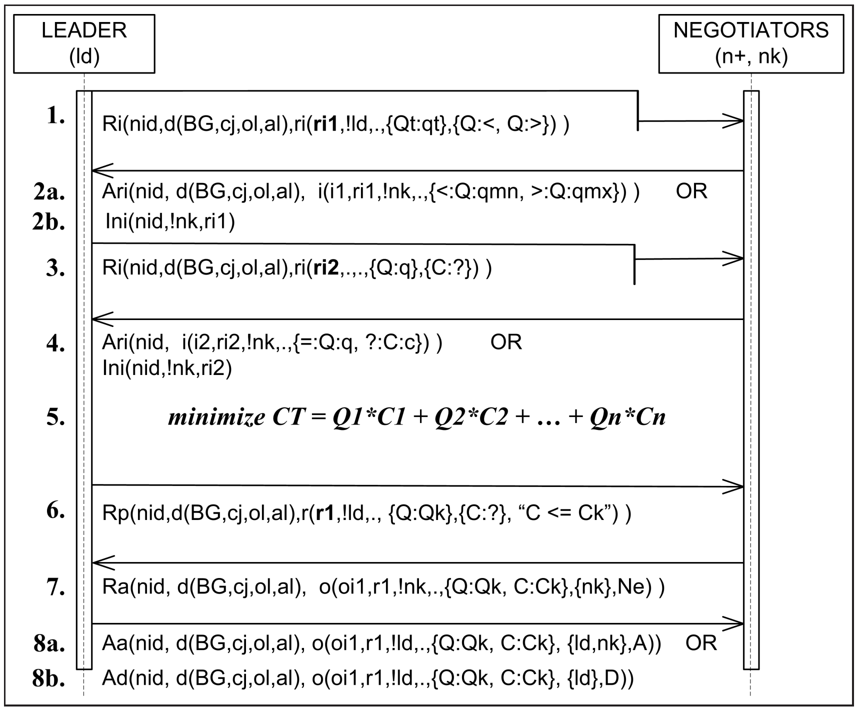

LEADER (ld)
NEGOTIATORS (n+, nk)
1. Ri(nid,d(BG,cj,ol,al),ri(ri1,!ld,.,{Qt:qt},{Q:<, Q:>}))
2a. Ari(nid, d(BG,cj,ol,al), i(i1,ri1,!nk,.,{<:Q:qmn, >:Q:qmx})) OR
2b. Ini(nid,!nk,ri1)
3. Ri(nid,d(BG,cj,ol,al),ri(ri2,.,.,{Q:q},{C:?}))
4. Ari(nid, i(i2,ri2,!nk,.,{=:Q:q, ?:C:c})) OR
Ini(nid,!nk,ri2)
5. minimize CT = Q1*C1 + Q2*C2 + … + Qn*Cn
6. Rp(nid,d(BG,cj,ol,al),r(r1,!ld,., {Q:Qk},{C:?}, "C <= Ck"))
7. Ra(nid, d(BG,cj,ol,al), o(oi1,r1,!nk,.,{Q:Qk, C:Ck},{nk},Ne))
8a. Aa(nid, d(BG,cj,ol,al), o(oi1,r1,!ld,.,{Q:Qk, C:Ck}, {ld,nk},A)) OR
8b. Ad(nid, d(BG,cj,ol,al), o(oi1,r1,!ld,.,{Q:Qk, C:Ck}, {ld},D))

Figure 16: *Quota - minimizing/maximizing a property interaction pattern.*

Implementation of Solution: This pattern uses the RFI process (Figure 15) followed by a bargain process (Figure 9).

5.6 Other Patterns

This section briefly presents a few other patterns the negotiation protocol supports. Patterns Dutch Auction (Section 5.6.1) and Closed Ballot (Section 5.6.2) are already implemented within patterns English Auction (Section 5.2) and Open Ballot (Section 5.3), respectively. The other patterns combine different patterns.

5.6.1 Dutch auction

In a Dutch auction, one or more properties are in dispute by several negotiators (bidders). The auctioneer announces successive (typically, decreasing) assignments for such property(ies). There is a predefined time span between successive announcements. The negotiators that first agree upon the current value wins the auction. Recall that an auction step may receive several bids.

This pattern is quite similar to the English auction pattern (Section 5.2). However, each auction step is described by an offer conveyed by message `Ra` (see Figure 11) and the bidder agrees with such an offer by means of message `Aa`.

5.6.2 Closed ballot

A closed ballot is similar to the open one (Section 5.3). However, the ballot issue is described via an offer and the voters may only agree or disagree on it (abstention and veto are also possible). The ballot's issue is forwarded by the notary via an `Rva` message and the votes are conveyed by `Av` messages.

5.6.3 Sealed bid

A sealed bid is a kind of auction with only one auction step. The auctioneer asks for bids and chooses the best bid. The implementation of this pattern is quite similar to the bargain pattern: the auctioneer broadcasts an RFP (message `Rp`) to all bidders and waits for the bids during a predefined time span. The bidders send offers, via `Ra` messages, in response. Then, the auctioneer agrees on the best offer.

5.6.4 Consensus

There are scenarios in which the negotiation process aims at finding a specific value for a negotiation subject (one or more properties). However, there is not a list of predefined alternatives. The negotiator who conducts the negotiation have to find wisely an alternative that should be agreed by most negotiators.

Such a sought value may be determined by means of a sequence of closed ballots. The leader chooses the values for the properties, creates an offer and submits this offer to a closed ballot. If the issue is not approved, the leader chooses another assignment to the properties (creating a new offer) and runs another ballot. This process repeats until there is a successful assignment or the leader gives up.

The leader might submit several RFIs before running the ballot sequence intending to infer the negotiator's preferences. Thus, the leader may have an educated guess when choosing values for the successive ballots.

5.6.5 Quota - Distribution

There is a partitionable "resource" or "chore" to be distributed to the negotiators. The negotiation aims at assigning different partial "amounts" (i.e., quotas) of it to different negotiators. For instance, 50 farms commit themselves to provide weekly a cooperative with 500 coffee bags. Each farm will provide a part of it. There are several flavours of this pattern concerning: (a) if every negotiator must receive its share on the total amount or a few negotiators might have no share of it; (b) if the total amount must be wholly distributed or some leftover is acceptable; (c) if the negotiators must receive equal quotas or each negotiator might receive a different quota.

The above scenarios are detailed ahead. They comprise a few properties, namely:

- `T`: total amount to be distributed;
- `Q`: the quota;
- `Ps`: other properties. In general, determining a quota depends on other properties. For instance, price and amount are generally related (a larger amount implies in a cheaper value for each unit). Thus, a bunch of properties are negotiated together.

There also other negotiation parameters needed in a few scenarios. They are represented by means of the following negotiation properties:

- `qmin`: minimum quote;
- `default`: there might be a default assignment to the properties `Ps`.

Thus, the negotiation process comprises `n` negotiators and aims at assigning values for properties `Q` and `Ps` for a given value of `T`. In a few scenarios, `Q` and `Ps` are compound properties. The negotiation process may also use negotiation properties to enforce some desired effect.

The quota distribution scenarios are depicted in tables 2 and 3. Table 2 presents the scenarios where every negotiator should receive a share of the distributed amount. Table 3 is similar, but a few negotiator may not have its own share.

	All Distributed	**Leftover**
Equal Share	A1	A2
Different Share	A3	A4

Table 2: Each negotiator receives a share.

	All	**Leftover**
Equal Share	NA1	NA2
Different Share	NA3	NA4

Table 3: A negotiator may receive no share.

Let us describe briefly how the mentioned scenarios could be handled by SPICA protocol:

- **A1**: In this scenario the whole amount should be evenly distributed to all negotiators. The negotiation leader may run the negotiation in two phases. In the first phase, the leader proposes $Q = T/n$ by means of a ballot. This guarantees equal quota and complete distribution. If accepted, the second phase takes place: the leader follows the Consensus Pattern (Section 5.6.4) to determine values for properties `Ps`. Conversely, the leader might also try to bargain properties `Ps` with each negotiator individually.

However, since the quota is the same for all negotiators, they would not have any motivation to haggle over the other properties. These two phases might be combined into a single one, thus `Q` and `Ps` would be negotiated together following the Consensus Pattern, provided that Q is restricted to T/n. Both `Q` and `Ps` are simple properties.

- **A2**: In this scenario all negotiators receive an equal quota, but there would be some leftover. It is similar to scenario A1. However, in the first phase the leader may follow the Consensus Pattern to establish a value for `Q` and $Q \leq T/n$. Both `Q` and `Ps` are compound properties.

- **A3**: In this scenario, the whole amount is distributed to all negotiators, but they may receive different quotas. The leader may run the negotiation in several phases. The first two phases would follow the Consensus Pattern to determine (a) `qmin` and (b) the `default` values for `Ps`. The third phase comprises a sequence of English Auctions in which the negotiators compete to assign values for their properties `Q` and `Ps`. Thus, the auction winner gets its quota and assign the values it proposed to properties `Ps`. It does not take part in the next auctions. The leader should restrict the acceptable values for `Q` in each auction such that the remaining negotiators would get, at least, `qmin` as their quota. Finally, the defeated negotiator receives the remaining quota and its properties `Ps` are assigned with the `default` values.

- **A4**: In this scenario, (uneven) quotas are distributed to all negotiators. Leftover is possible. This is similar to scenario A3. However, `qmin` is set to "1 unit" enforcing that each negotiator will receive its share.

- **NA1**: In this scenario, to whole amount is distributed in equal quotas, however it may happen that a few negotiators do not receive a share. The leader runs this pattern in two phases. First, it runs an auction to determine values for properties `Ps`. Recall that the SPICA protocol does not specify how the winner bid(s) is (are) chosen. Thus, the leader may choose several winner negotiators. Then, the total amount is distributed evenly among the winner negotiators. To do so, the leader sends (counter) offers to the winner negotiators assigning an equal value for property `Q` and keeping unchanged the other values the negotiator had proposed in its bid. Both `Q` and `Ps` are simple properties.

- **NA2**: In this scenario, quotas are distributed evenly among a few negotiators, but there may be some leftover and a few negotiators might not receive their share. The leader runs a few phases. First, it follows the Consensus Pattern to determine a value for `Q`. Note that, (a) if $Q * n > T$, a few negotiators will be left without their share; otherwise, (b) if $Q * n < T$ there will be leftover. Then, a subset of negotiators is chosen. This may be done in two ways. If $Q * n > T$, there is competition among the negotiators. Thus, a sequence of auctions may be run (like in A3) in which the negotiators compete to assign values for their properties `Ps`. In such a setup, `Q` is a simple property and `Ps` is compound. Conversely, if $Q*n < T$, there is no competition among the negotiators. Thus, the leader should follow the Consensus Pattern to assign values for properties `Ps`. Thus, both `Q` and `Ps` are simple properties.

- **NA3**: In this scenario, the whole amount is distributed into uneven quotas to a few negotiators. The leader would run such a negotiation in a few phases. In the first phase, a subset of the negotiators is chosen. These negotiators commit themselves to fulfill the total amount `T`. To do so, the leader would run a Dutch Auction. The offer conveyed in this auction assigns a value for `T`, `qmin`, and `default` values for `Ps`. Recall that several bids may be received during an auction step. The negotiators that sent their bids are the sought subset. If this auction fails, the leader may choose another assignment for `T`, `qmin` and `default` and start the first phase anew. The next phases are similar to those of scenario A3: the negotiators compete to assign values for their properties `Q` and `Ps`, by means of English Auctions. The defeated negotiator will receive the remaining quota and its properties `Ps` are assigned with the `default` values. Both `Q` and `Ps` are compound properties.

- **NA4**: In this scenario, `T` is partially distributed into uneven quotas to a few negotiators. The leader will try to assign values for `Q` and `Ps` by running a sequence of auctions, like in scenario A3. However, the only restriction is that the sum of the already distributed quotas are no greater than `T`. Both `Q` and `Ps` are compound properties.

5.6.6 Priority Quota

There is a right or obligation that must be completely distributed or fulfilled by some parties. Sometimes, when dividing this right or obligation, some parties should have precedence over others to choose their share of it. This precedence is determined by a predefined criteria, such as, specific party's characteristic (e.g., age), the value it offers by the share, to name a few. This can be illustrated by an example. There is a long road to be build and a deadline to be meet. The road is divided into "pieces" that are assigned to different Engineering companies. There are a number of Engineering companies interested in building this road. Each company submits a bid. The bid comprises the price per kilometer and the length it can build before the deadline. The Engineering company that proposed the cheapest price per kilometer, chooses first the part of the road it will build. The second best price, chooses the second part, and so on.

5.6.7 Vickrey Auction

The winner of an auction may be, in fact, the unfortunate party. It might have proposed a value much higher than the values proposed by the other negotiators and even higher than the market price. A solution for such a disparity would be the winner bider paid the price proposed by the second one.

An strategy to accomplish this scenario would be as follows: the leader broadcasts an RFP asking the value for a(group of) property(ies); the parties send offers and the leader agrees with the best offer. However, the value of the best offer must be changed (to the value of the second best offer). To do so, the leader sends a counter-offer to the winner party with the changed value. The partner will always be satisfied with the new value, thus it agrees on it.

5.6.8 Double Auction

There are negotiators willing to sell a few items (of the same kind or not). Other negotiators are willing to buy the same kinds of items. However, potential partners must meet each other before developing negotiation.

This problem is described by means of an example. Undergraduates must develop some training activities in a company before getting their diplomas. Several companies have internship programs and are willing to accept undergraduates. However, there are several legal issues concerning the admission of undergraduates. Thus, some agencies help this process: the students fill in forms describing their abilities and the companies inform the positions they have available. The agency matches both needs.

Both student and company announce RFPs. The student's RFP describe his abilities and asks for a company. The company's RFP describes the open position and asks for a student. The auctioneer (matchmaking agency) matches both kinds of RFPs and sends the student RFP to the potential company. Then the company sends an offer to the student. Note that several variants are possible: (a) the offer is sent directly to the student. The student may only accept or refuse it; (b) the offer is sent directly to the student and both negotiators may develop a bargain; (c e d) a and b mediated by the auctioneer.

5.6.9 Hierarchical Negotiation - low level consensus

The motivation for this pattern is explained by means of an example. There are a cooperative, several farms and a nationwide supermarket network. The farms will provide some organic products to this supermarket network. The negotiation will be conducted by the cooperative on farms' behalf, but the farms will be the signatories of the contract. However, the cooperative's decisions must be agreed on by the farms beforehand. Thus, there are two simultaneous and dependent negotiation being carry out: among the farms (the cooperative is the leader), and between the cooperative and the supermarket. Before taking any decision at the higher level, the cooperative leads a negotiation among the farms on the same subject and the farms' decision is taken by the cooperative in the higher negotiation. At the end, the farms sign the contract. The cooperative is the so-called *proxy negotiator*. The supermarket network could also be a proxy negotiator on behalf of its branches. This is a symmetrical setup.

The negotiation style developed in both negotiation levels may be different. The high level may virtually be of any style. The low level negotiation aims at consensus. Thus, it will run at least a bare ballot or will follow the Consensus Pattern.

6 Middleware Implementation

The previous sections explained SPICA's negotiation messages (Section 4) and how they can be combined into different negotiation styles which are modeled as workflows (Section 5). Such messages are transported by means of a middleware (recall Figure 6) and delivered at specific interfaces of a few Web services (negotiator's and notary's). This middleware comprises YAWL's workflow engine which we instrumented with tailor-made software (here-after called NS Service). The engine executes the workflow and invokes NS to handle the transported messages. Sections 3 presented YAWL system and Section 4.4 pointed out a few implementation issues needed to understand design decisions. This section details the protocol's implementation.

SPICA's negotiation protocol is modeled as a YAWL model comprising several subnets. Some of them were presented in Section 5. Each subnet has a number of tasks that are responsible for handling the messages exchanged among the negotiators. A task has a *decomposition*. The decomposition defines its *input* and *output variables* as well as a so-called *YAWL service*. This is a Web service and can be seen as a procedure called by the workflow engine to process the input variables and return a set of results (the outputs). Thus, the engine can forward these results to subsequent tasks. The format of the messages are described in Section 6.1. SPICA's specific decompositions are described in Section 6.2. Section 6.3 presents the tailor-made custom service (NS Service).

6.1 Message Format

Section 4.3 presented the negotiation messages exchanged within a negotiation process. Messages are XML files sent between tasks. This section shows their layout and also describes a few extra messages, called *control messages*, needed by the infrastructure.

Figure 17 shows an excerpt of a message. All messages have a common header. The first piece of information in the header is the message's type (`tp`). It is an enumeration of short strings that typically coincides with the message names presented in Section 4.3 (e.g., `Ra`, `Rp`, etc). The second piece of information (`posted`) records when the message was posted to the middleware, while the third one (`expires`) determines its lifespan, i.e., how long the sender will wait for an answer. The fourth piece of information (`expectansw`) is a boolean value that instructs the SPICA's infrastructure whether it must expect a reply for such a message. Finally, `fromname` and `receivers` record the message's originator and recipients. Following this header there is a long `choice` element with an entry for each possible message type. Only the first two appear in Figure 17.

Such elements encode the message's parameters (presented in Section 4.3)

Figure 18 depicts the encoding of the parameters of an `Rp` message. Note that the elements `nid`, `nd`, and `rfp` are exactly the same as the `Rp` message's parameters shown in Section 4.3. Other messages follow similar arrangement.

Besides the negotiation messages, a few so-called control messages were introduced and are used internally by the middleware (e.g., to activate a forward-like task). They are:

- `ACK`. This message is used between a dispatch-like and a forward-like task to correlate the dispatched message with the corresponding answer.

- `NULL`. This is an empty message used as a default message. It is simply discarded by the NS service.

```
<complexType name=‘‘MessageType’’>

   <sequence>

      <element name=‘‘tp’’ type=‘‘MsgType’’/>
      <element name=‘‘mid’’ type=‘‘MsgIDType’’/>
      <element name=‘‘pid’’ type=‘‘MsgIDType’’/>
      <element name=‘‘posted’’ type=‘‘dateTime’’/>
      <element name=‘‘expires’’ type=‘‘dateTime’’/>
      <element name=‘‘expectansw’’ type=‘‘YesNoType’’/>
      <element name=‘‘fromName’’ type=‘‘string’’/>
      <element name=‘‘receivers’’ type=‘‘ToType’’/>
      <choice>
         <element name=‘‘rp’’ type=‘‘MsgReqProposalType’’/>
         <element name=‘‘ra’’ type=‘‘MsgReqAgreementType’’/>
          Similar for other types of messages...
      </choice>
   </sequence>
</complexType>
```

Figure 17: *Negotiation Message Format.*

Variable	**Mode**	
RecM	In & Out	The received message. The received message can be a negotiation message or a control message.
ExpectAnsw	In Only	The types of the expected reply messages, i.e., the outgoing messages in RespM1.
RespM1	Out Only	A task can generate an outgoing message

Table 4: NSTask decomposition.

6.2 Decompositions

Most of the tasks related to negotiation interactions have a common decomposition called `NSTask`. The parameters of such a decomposition (represented by input and output variables) are summarized in Table 4:

The first two variables provide input for the task and the last is an outgoing message. `RecM` is also an output variable, which just outputs the received message. This is necessary in some scenarios. `RecM` may be a negotiation message or a control message. The YAWL service associated to a task of such a decomposition is the NS service. Next, we show how tasks that use such a decomposition are arranged and single out some modeling decisions.

Recall the dual task arrangement (i.e., the dispatch-like and forward-like tasks arrangement) presented in Section 4.4 and the auction subnet presented in Figure 11. This subnet is zoomed in in Figure 19. A request for a new auction step (`Ras`) message arrives at `Requested Auction Step` via its `RecM` input variable. This task is a dispatch-like task. The `Ras` message is dispatched to the notary and an `ACK` message (control message) is returned via `RespM1` output variable and assigned to a local variable (named `NewAucAck`). This `ACK` message contains information that will relate the received `Ras` message to the cor-

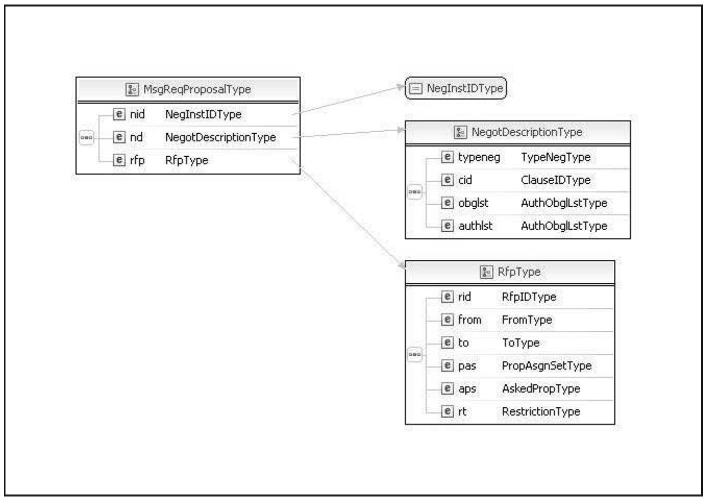


Figure 18: *Parameters of an RP message: MsgReqProposalType.*

Task	**RecM**	**ExpectAnsw**	**RespM1**
Requested Auction Step	Ras	ACK	ACK
Notary Decides I	ACK	Aas, Rp, Ra	Aas

Table 5: Examples of assignment of tasks' variables.

responding answers to it. The `ACK` message is received at `Notary Decides I` (it might be `Notary Decides II` instead) via its `RecM` input variable. This task is a forward-like task. In response to the `Ras` message, the notary has to send three different messages: **(1)** an `Aas` message to the leader; **(2)** either an `Rp` (English auction) or an `Ra` message (Dutch auction) to the negotiators, and **(3)** an `Ica` message to the leader at the end of the auction step. Thus, the notary sends these messages (only `Aas` is shown in the picture) to NS service which: a) verifies if the incoming responses are within the list of `ExpectAnsw` input variable (offending messages are just discarded), b) correlates them to the previous `ACK` message and c) checks them into the workflow engine. The output message will be assigned to output variable `RespM1`. For instance, Table 5 shows the values for these parameters for the first two tasks in Figure 11. Note that the values assigned to `RespM1` must be in `ExpectAnsw` input variable.

6.3 NS Custom Service

Tasks are processed by means of associated YAWL services. The YAWL workflow engine implements a few predefined services and provides support for the implementation of user-tailored services, the so-called *YAWL custom services.* A custom service is a Java class that complies with specific interfaces. The negotiation framework implements a custom service called NS (for Negotiation custom Service) that handles the negotiation messages.

Figure 20 shows (a) the rationale behind a generic custom service and (b) how such a

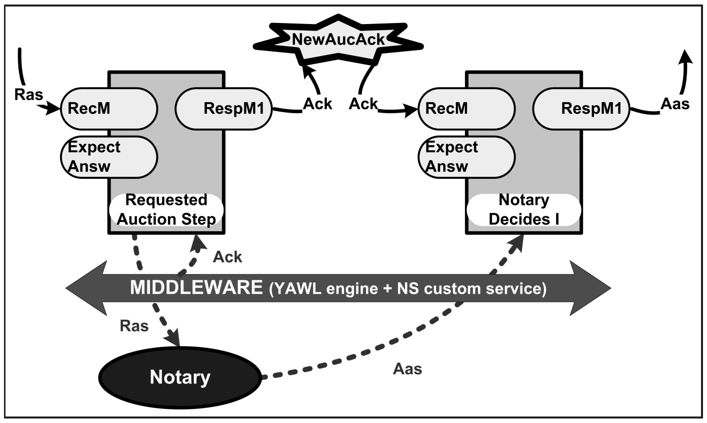


Figure 19: *Tasks with decomposition detailed.*

rationale is applied in our solution. Consider Figure 20.a. When **(1)** a task is enabled and its input parameters (IP) are available (typically from a local variable), **(2)** the engine invokes the associated custom service (Csrv). This service (2a) performs the so-called "check out" operation via a specific interface provided by the engine. This causes the task to be in the "executing" state. The service obtains the input data, process it and (2b) returns the result to the engine by means of the so-called "check in" operation. This causes the task to be completed and **(3)** the result to be assigned the output parameters (OP). In the end, other tasks may be enabled and the results may be available for further processing. To sum up, it is a synchronous arrangement.

Conversely, SPICA's messages are asynchronous. The negotiators are web services that implement some specific interfaces and are not aware of YAWL's engine. Let us examine Figure 20.b. **(1)** A negotiation message (IM) is sent to a negotiator (e.g., an Rp message). **(2)** This message enables task T' and is transfered to its input variable

`RecM`. The engine invokes NS that (2a) checks out the incoming message, (2b) immediately checks in a specific ACK control message, (2c) enabling the subsequent task (T"). At the same time, (2d) NS dispatches the incoming message to the intended negotiator. **(3)** The negotiator analyzes the received message and (3a) sends an answer to NS via the NS' `checkInIF` interface. Meanwhile, the ACK message has arrived to task T". NS is invoked by the engine and (3b) checks out the ACK message. (c) NS matches the ACK and answer messages (see Section 6.2), perform a few validation procedures and checks in the messages which are assigned to the output variable `RespM1`. This outgoing message (OM) is available for processing by the subsequent tasks.

Figure 21 goes into details about the interaction between the YAWL engine, the NS service and a negotiator from the latter's perspective.[2] It shows that NS provides two interfaces: one to receive requests from the YAWL engine (interface `B'`) and other to receive requests form a negotiator via the *checkInIF* interface (`Ci`). There is also a setup interface (interface `S`) that is out of the scope of this paper. Every negotiator (NM) implements a

[2]The rationale is the same for the notary.

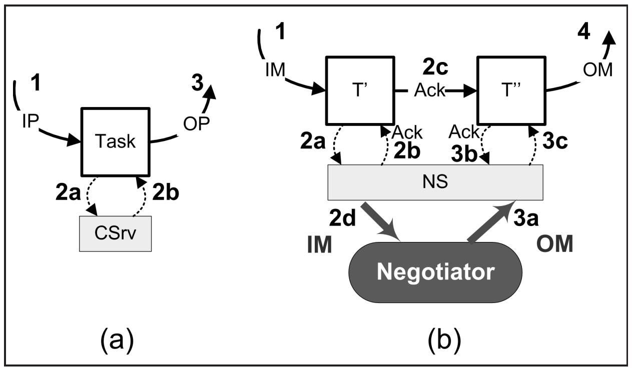


Figure 20: *Dispatch of negotiation messages.*

set of specific interfaces (described in [BMM08]) depicted as gray circles. Operations of these interfaces are activated when the corresponding negotiation message is dispatched to the negotiator. There is an extra interface (black circle) for exceptions (out of the scope of this paper). The negotiator also uses a *communication adaptor* (COM ADP) to upload an answer message.

A YAWL custom service extends the so-called `InterfaceBWebSideController`

abstract class (interface `B'` in Figure 21) and implements a method named `handle-EnabledWorkItemEvent`. When a task is enabled, this method is invoked by the YAWL engine. A custom service interacts (i.e., checks out) with the engine through the so-called interface `B`. Using this interface, the custom service gets the received message, uploads the corresponding ACK message, and uploads the corresponding answer message.

Let us examine Figure 21 in detail. When a task whose decomposition is NSTask is enabled, **(1)** the engine calls the method `handleEnabledWorkItemEvent` within the interface `B'`. This method receives a so-called *work item* as a parameter. Such work item keeps quite a few pieces of information about the enabled task. **(2)** NS service interacts with the engine and gets the message received by the task, i.e., the content of input variable `RecM`, validates the message and **(3)** checks in an `ACK` message. **(4)** According to the type of the received message, NS service calls an appropriate operation of one of these interfaces, i.e., it dispatches the negotiation message. **(5)** The negotiator interacts with NS via the `Ci` interface. In response to a negotiation message, the negotiator uploads the answer messages to NS. Recall that the negotiator is not aware about NS service or about YAWL engine. The message uploading is intermediated by means of a communication adaptor (COM ADP). This adaptor mimics a negotiator by implementing the same interfaces. When a negotiator calls one of its operations, it creates a negotiation message in the format described in Section 6.1 and forwards it to NS. If another middleware was used, only the adaptor should be replaced. Finally, **(6)** NS correlates the uploaded answers with the corresponding ACK message and checks in the corresponding task. To do so, NS uses a few data contained within the received work item. The uploaded answer is assigned to this task's output variable (RespM1). Details about the validation and correlation processes are presented next.

The algorithm used to dispatch messages uses two data structures: a table of dispatched

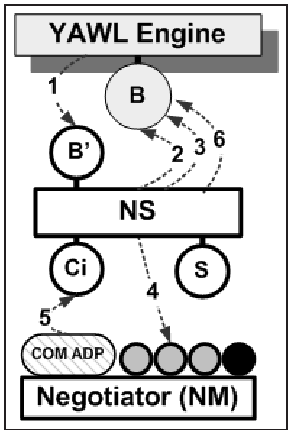


Figure 21: *NS service from the perspective of one negotiator.*

messages (TDM) and a queue of not processed incoming messages (NpMsgs). There is an instance of them to each different negotiation instance. The table TDM has 8 columns:

- mid: it is the identification of the dispatched message M.
- pid: M's parent message identification
- nansw: it keeps the number of answers the message has received so far (-1 means the message has not been dispatched; 0, the message was dispatched, but no answers have arrived so far; other positive numbers stand for the number of received answers),
- ndscd: it counts the discarded answers for a specific message
- expires: it indicates the message's expiration date
- posted: Date when the message was posted into NS service.
- msg: it keeps the message itself for auditing purposes.
- ackLst: recall that when message M is dispatched, ACKs are sent to subsequent tasks enabling them to receive answers for M. Thus,ackLst is a list that contains the work items of the tasks enabled by such ACKs.

For convenience, we use a few writing convention, for instance: TDM[nid,mid].expires means the expiration date of message mid in the negotiation nid; M.mid() means the message identifier for message M.

An incoming message, viz., a message received via the NS' checkInIF interface, can only be processed if its respective enabling ACK has already been checked out. If it is not true, the incoming message is inserted in the NpMsgs queue and processed upon the respective ACK arrival.

Figures 22, 23 and 24 depict algorithms used by NS to handle incoming messages. Figure 22 shows the actions executed by NS when a negotiation message (i.e., not an `ACK` message) is received via the `RecM` input variable (Figure 21.b, step 1). Note the created `ACK`: it keeps the identification of its respective message (`ACK.pid`, step b3) that will be used to correlate with the expected answer (to arrive).

```
a.  Check out negotiation message M (Figure 20.b, step 2a)

b.  Create an new ACK message:
     i.ACK.nid:= M.nid();
     ii.ACK.mid:= new_identifier();
     iii.ACK.pid:= M.mid();
c.  TDM[M.nid(),M.mid()].nasw:= -1:   ,
TDM[M.nid(),M.mid()].ndscd:=-1;
   TDM[M.nid(),M.mid()].pid:= M.pid() ; TDM[M.nid(),M.mid()].msg:=
M;
   TDM[M.nid(),M.mid()].expires:= M.expires();
d.  Check in the ACK message (Figure 20.b, step 2b)
e.  Dispatch message M to the appropriate negotiator's interface.
(Figure 20.b, step 2d)
```

Figure 22: *Negotiation message received via RecM input variable*

Conversely, if an `ACK` message is received via `RecM`, the algorithm presented in Figure 23 is executed. The received `ACK` is used to match a dispatched message (algorithm Figure 22) with a future answer. However, it may happen that the answer itself arrives before its `ACK`. In this case, the early answer will have been put in the queue of not processed messages (algorithm Figure 24). Thus, step (d) checks if this is the case.

```
a.  Check out the ACK message (A). (Figure 20.b, step 3b)
b.  TDM[A.nid(),A.pid()].answ = 0; TDM[A.nid(),A.pid()].ndscd=0
c.  Insert A to TDM[A.nid(),A.mid()].ackLst
d.  Check NpMsgs.  If there is already a response (R) related
to this ACK message (i.e., R.pid()=A.pid()), take the response
off the queue and execute steps (c) and (d) of the algorithm in
Figure 24
```

Figure 23: *ACK message received via RecM input variable*

Finally, Figure 24 shows the actions executed when a reply message is received via NS' `checkInIF` interface. Such reply message might have arrived earlier that its respective `ACK`. It is checked in step (b). If so, the incoming message is added to the list of not processed messages. Recall that a message may have several answers. For instance, the message `Ras` at the beginning of an auction (Figure11) may receive two answers when the auction step starts (namely, `Aas` and `Rp`) and another message at its end (an `Ica`). In this case, the `Ras`'s `ackLst` will contain three work items respective to tasks `Notary Decides I`, `Notary Decides II` and `Collected Bids` that are responsible for receiving those answers. Note that all such answers have the same parent. Thus, it must be determined which is a suitable work item for the upcoming message: this suitable work item is the one whose list of expected answers contains the type of the upcoming answer. Note that step (c) validates the upcoming message before it is check in within the context of the suitable work item.

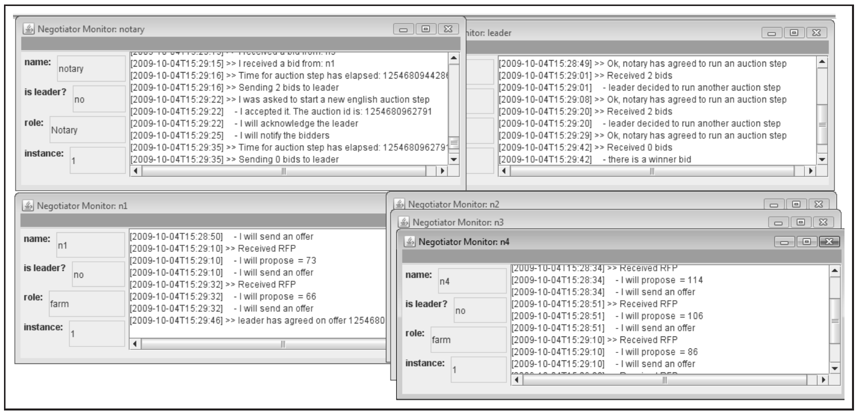
```
a. NS receives the response R for previous negotiation message P
via checkInIF interface.  It examines the response and finds the
identification of the parent message (Figure 20.b, step 3a)

    pnid:= R.nid(); pmid:= R.pid())
b. If the respective ACK has not been checkout yet (i.e.,
TDM[R.nid(),R.pid()].nansw =-1):
    i. Insert the response R into NpMsgs
    ii. Exit
c. If R is aged (TDM[R.nid(),R.pid()].expires < date_posted(R)),
or is not valid or violates some restrictions:
    i. Send exception to the sender.
    ii. Increment ndscd counter and discard the response.
d. If R is not aged:
    i. TDM[R.nid(),R.pid()].nansw++
    ii. Checks in message R. (Figure 20.b, step 3c)
```

Figure 24: *Response message (R) received via checkInIF interface.*

6.4 Example Execution

Figure 25 shows part of the output of a negotiation instance. Each participant has an window where it prints out a few remarks about the steps and decisions it has undertaken.



name: notary
is leader? no
role: Notary
instance: 1

```
[2009-10-04T15:29:15] >> I received a bid from: n1
[2009-10-04T15:29:16] >> Time for auction step has elapsed: 125468094428
[2009-10-04T15:29:16] >> Sending 2 bids to leader
[2009-10-04T15:29:22] >> I was asked to start a new english auction step
[2009-10-04T15:29:22]    - I accepted it. The auction id is: 1254680962791
[2009-10-04T15:29:22]    - I will acknowledge the leader
[2009-10-04T15:29:25]    - I will notify the bidders
[2009-10-04T15:29:35] >> Time for auction step has elapsed: 125468096279
[2009-10-04T15:29:35] >> Sending 0 bids to leader
```

```
[2009-10-04T15:28:49] >> Ok, notary has agreed to run an auction step
[2009-10-04T15:29:01] >> Received 2 bids
[2009-10-04T15:29:01]    - leader decided to run another auction step
[2009-10-04T15:29:08] >> Ok, notary has agreed to run an auction step
[2009-10-04T15:29:20] >> Received 2 bids
[2009-10-04T15:29:20]    - leader decided to run another auction step
[2009-10-04T15:29:29] >> Ok, notary has agreed to run an auction step
[2009-10-04T15:29:42] >> Received 0 bids
[2009-10-04T15:29:42]    - there is a winner bid
```

name: n1
is leader? no
role: farm
instance: 1

```
[2009-10-04T15:28:50]    - I will send an offer
[2009-10-04T15:29:10] >> Received RFP
[2009-10-04T15:29:10]    - I will propose  = 73
[2009-10-04T15:29:10]    - I will send an offer
[2009-10-04T15:29:32] >> Received RFP
[2009-10-04T15:29:32]    - I will propose  = 66
[2009-10-04T15:29:32]    - I will send an offer
[2009-10-04T15:29:46] >> leader has agreed on offer 1254680
```

name: n4
is leader? no
role: farm
instance: 1

```
[2009-10-04T15:28:34] >> Received RFP
[2009-10-04T15:28:34]    - I will propose  = 114
[2009-10-04T15:28:34]    - I will send an offer
[2009-10-04T15:28:51] >> Received RFP
[2009-10-04T15:28:51]    - I will propose  = 106
[2009-10-04T15:28:51]    - I will send an offer
[2009-10-04T15:29:10] >> Received RFP
[2009-10-04T15:29:10]    - I will propose  = 86
[2009-10-04T15:29:10]    - I will send an offer
```

Figure 25: *An example execution.*

A complete execution is shown in another paper ([BMM09]). It presents how the main negotiation styles are combined seamlessly to build a virtual organization.

7 Related Work

Modeling. According to [AKD09], a supply chain is a set of disparate members who are dependent on each other to manage resources (such as inventory and information). The common way of describing such a management is the definition of a business process (BP), i.e., a set of activities that, if properly performed, causes the intended effect. Multiple executions of a BP happen within individual, isolated contexts and, in general, each step causes some kind of transformation, be it physical (e.g., raw material becomes manufactured goods) or logical (e.g., information is produced or updated). Workflow Management Systems (WfMS) are a natural choice for modeling and executing BPs.

Modeling a workflow may be a piece of art or a piece of engineering. In the first case, it is a result of the abilities of the modeler. In the second case, the modeler employs different techniques to achieve a proper result. One of those techniques are design patterns. Design patterns have been around since the mid 90s. Aalst and colleagues have applied them to workflows ([ABHK00, vdAtHKB03]) and Mulyar ([Mul09]) presented patterns for process-aware information systems.

This paper proposes using workflows as a means of modeling and executing a specific kind of BP: contract negotiations and presents a few negotiation patterns.

Contracts and Negotiation. There are several proposals for contract specification. Some of them are designed for specific purposes where the domain of negotiatable items is predefined, like SLAs (e.g.,[FdTdSG06]). This is not our approach. More generic contract specification approaches need an expressive language to describe the commitments agreed among the partners. Several of them use logic-based approaches, like [PVSlN$^+$08, ACG$^+$08, GP04, GDtHO01, ONP07, Xu04].

This is not our approach. It would demand highly trained computer scientists to design the contract to be negotiated. However, the negotiation environment we propose aims at real agricultural supply chains where the participants are autonomous, highly heterogeneous and geographically widespread. It would not be realistic to expect that there are enough of them available and that they also have law skills.

A contract expresses commitments among partners. However, most of the ones proposed in the literature are bi-lateral, just a few are multi-party, e.g., [Xu04].

Contracts are the outcome of some negotiation process which may be done with some level of software assistance. We proposed automatic negotiation performed by software agents and guided by contract templates [BMM08]. Other proposals also use templates, e.g., [HM03, BPJ04, CCH$^+$05]. An alternative for negotiation are matchmaking approaches like [NSDM04].

Kallel and others [KJDG08] propose a multi-agent negotiation model for a particular contract type of a specific supply chain. The negotiation model consists of a heuristic negotiation protocol and a decision-making model. The authors' approach diverge from ours. They understand a supply chain as a neighborhood: a focused company, its suppliers and its customers. We consider the whole supply chain from farm to the end consumer.

FIPA[3] proposes standards for the interoperation of heterogeneous agents. A few of

[3]IEEE's Foundation for Intelligent Physical Agents: http://www.fipa.org/

them resembles our negotiation styles, e.g., *FIPA Contract Net Interaction Protocol*[4] bears resemblance to our bargains. However, they mostly aim at finding an agent to perform a task. The agent will acknowledge that it has accomplished such a task. In contrast, our negotiation protocol aims at producing a contract that will be enacted in the future. Our protocol itself does not include the execution phase. FIPA also proposes standards for Dutch and English Auctions, but "the auctioneer seeks to find the market price of a good".[5] Our approach is more general than that: we seek to determine good values for a set of properties, not restricted to only price.

A number of authors use contracts to describe the coordination of activities of partners, e.g., [WH02, LMC$^+$04]. Others use contracts a means of monitoring the fulfillment of the contract's commitments, e.g., [Xu04].

In addition, [AMSW07] use a Petri net-base approach to discuss how a partner should implement its part of the contract complying to the contract's description. Other authors use contracts are the context of agent societies to shape the agents' behaviour (e.g., [US06, ZTS$^+$08]), i.e., the actions they might or might not be undertaken regarding to the use of shared resources. They are not business contracts indeed.

Middleware. SPICA Negotiation Protocol is defined as a set of interfaces to be implemented by the negotiators and notary. Since these participants may be distributed, there must be a communication infrastructure that allow them to exchange negotiation messages (that activate such interfaces). The current choice is an implementation based on Web services supervised by a workflow engine (YAWL). Other alternatives could have been employed, such as: an implementation of OASIS ebXML Messaging Services[6] (e.g., Hermes[7]) or JADE[8] (extended with WADE).

8 Conclusion

This paper presented the core implementation of the SPICA Negotiation Protocol. There are other features included in the protocol design, but not implemented so far. Future work comprises implementing them. Such already designed features include grouping properties in a hierarchical way and negotiating each group separately. It also will allow the negotiators to express their intentions, restrictions and reasons at different phases of the negotiation. This did not influences the protocol's control flow, but are intended to help the negotiators' decision making process.

Our protocol is more generic than others. It allows for different combination of the negotiation primitives resulting in an endless negotiation styles. The ones detailed in this paper are just the most important and suitable for illustrating the use of the protocol's primitives. As an example of this, the reader is invited to visit another paper of ours

[4]http://www.fipa.org/specs/fipa00029/SC00029H.pdf

[5]English auction: http://www.fipa.org/specs/fipa00031/XC00031F.pdf; Dutch auction: http://www.fipa.org/specs/fipa00032/XC00032F.pdf

[6]http://docs.oasis-open.org/ebxml-msg/ebms/v3.0/core/cs02/ebms_core-3.0-spec-cs-02.pdf

[7]http://www.freebxml.org/msh.htm

[8]http://jade.tilab.com/

([BMM09]) and see how two hierarchical parallel negotiations are interleaved to produce a contract.

Acknowledgment

Thanks to Nataliya Mulyar and Nick Russell for their comments on the early drafts of this report and also to Michael Adams for his help with YAWL´s engine.

This paper was partially supported by CAPES/Brazil and by CNPq (WebMAPS and AgroFlow projects) and FAPESP.

References

[AADH04] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL System. In A. Persson and J. Stirna, editors, *Advanced Information Systems Engineering, Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer-Verlag, Berlin, 2004.

[Aal98] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[ABHK00] W.M.P. van der Aalst, A.P. Barros, A.H.M. ter Hofstede, and B. Kiepuszewski. Advanced workflow patterns. In *CooplS '00: Proceedings of the 7th International Conference on Cooperative Information Systems*, pages 18–29, London, UK, 2000. Springer-Verlag.

[ACG$^+$08] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Expressing and verifying business contracts with abductive logic programming. *Intl. Journal of Electronic Commerce*, 12(4):9–38, summer 2008.

[AH05] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[AHKB03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[AKD09] Arshinder, A. Kandan, and S.G. Deshmukh. A framework for evaluation of coordination by contracts: A case of two-level supply chains. *Computer & Industrial Engineering*, 56(4):1177–1191, 2009.

[AMSW07] W.M.P.v.d Aalst, P. Massuthe, C. Stahl, and K. Wolf. Multiparty Contracts: Agreeing and Implementing Interorganizational Processes. Technical report, Humboldt-Universität zu Berlin, 2007. Informatik-Berichte 213.

[BMM08] E. Bacarin, E.R.M. Madeira, and C.B. Medeiros. Contract e-negotiation in agricultural supply chains. *Intl. Journal of Electronic Commerce*, 12(4):71–97, summer 2008.

[BMM09] E. Bacarin, E.R.M. Madeira, and C.B. Medeiros. Assembling and managing virtual organizations out of multi-party contracts. In J. Filipe and J. Cordeiro, editors, *ICEIS*, volume 24 of *Lecture Notes in Business Information Processing*, pages 758–769. Springer, 2009.

[BPJ04] C. Bartolini, C. Preist, and N.R. Jennings. A software framework for automated negotiation. In *SELMAS*, pages 213–235, 2004.

[CCH$^+$05] D.K.W. Chiu, S.C. Cheung, P.C.K. Hung, S.Y.Y. Chiu, and A.K.K. Chung. Developing e-negotiation support with a meta-modeling approach in a web services environment. *Decision Support Systems*, 40(1):51–69, July 2005.

[FdTdSG06] M. Fantinato, M. B. F. de Toledo, and I. M. de S. Gimenes. A feature-based approach to electronic contracts. In *CEC/EEE'06*, pages 34–41, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[GDtHO01] G. Governatori, M. Dumas, A.H.M. ter Hofstede, and P. Oaks. A formal approach to protocols and strategies for (legal) negotiation. In *ICAIL*, pages 168–177, 2001.

[GP04] B.N. Grosof and T.C. Poon. SweetDeal: Representing Agent Contracts with Exceptions Using Semantic Web Rules, Ontologies, and Process Descriptions. *Intl. Journal of Electronic Commerce*, 8(4):61–97, 2004.

[HAAR10] A.H.M. ter Hofstede, W.M.P. van der Aalst, M. Adams, and N. Russell. *Modern Business Process Automation: YAWL and its Support Environment*. Springer-Verlag, Berlin, 2010.

[HM03] J.E. Hanson and Z. Milosevic. Conversation-oriented protocols for contract negotiations. *EDOC*, 00:40–49, 2003.

[KJDG08] O. Kallel, I.B. Jaâfar, L. Dupont, and K. Ghédira. Multi-agent negotiation in a supply chain - case of the wholsale price contract. In J. Cordeiro and J. Filipe, editors, *ICEIS (4)*, pages 305–314, 2008.

[LMC$^+$04] P.F. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal. A unified behavioural model and a contract language for extended enterprise. *Data & Knowledge Engineering*, 51(1):5–29, 2004.

[Mul09] N.A. Mulyar. *Patterns for Process-Aware Information Systems: An Approach Based on Colored Petri Nets*. PhD thesis, Technische Universiteit Eindhoven, 2009.

[MZ02] H. Min and G. Zhou. Supply chain modeling: past, present and future. *Computer & Industrial Engineering*, 43:231–249, July 2002.

[NSDM04] T. Noia, E. Sciascio, F.M. Donini, and M. Mongiello. A system for principled matchmaking in electronic marketplace. *Intl. Journal of Electronic Commerce*, 8:9–37, Summer 2004.

[ONP07] N. Oren, T.J. Norman, and A.D. Preece. Argumentation based contract monitoring in uncertain domains. In Manuela M. Veloso, editor, *IJCAI*, pages 1434–1439, 2007.

[PVSlN$^+$08] S. Panagiotidi, J. Vázquez-Salceda, S. Álvarez Napagao, S. Ortega-Martorell, , S. Willmott, R. Confalonieri, and P. Storms. Intelligent contracting agents language. In *Proceedings of the Symposium on Behaviour Regulation in Multi-Agent Systems -BRMAS'08. Aberdeen, UK*, pages 49–54, April 2008.

[RAHE05] N. Russell, W.M.P.van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In O. Pastor and J. Falcao e Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 216—-232. Springer-Verlag, Berlin, 2005.

[RHEA05] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In L. Delcambre, C. Kop, H.C. Mayr, J. Mylopoulos, and O. Pastor, editors, *24nd International Conference on Conceptual Modeling (ER 2005)*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368. Springer-Verlag, Berlin, 2005.

[RWA$^+$08] A. Rozinat, M.T. Wynn, W.M.P. van der Aalst, A.H.M. ter Hofstede, and C. Fidge. Workflow Simulation for Operational Decision Support Using Design, Historic and State Information. In M. Dumas, M. Reichert, and M.C. Shan, editors, *International Conference on Business Process Management (BPM 2008)*, volume 5240 of *Lecture Notes in Computer Science*, pages 196–211. Springer-Verlag, Berlin, 2008.

[US06] Y.B. Udupi and M.P. Singh. Contract enactment in virtual organizations: A commitment-based approach. In *AAAI*. AAAI Press, 2006.

[vdAtHKB03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[WH02] H. Weigand and W. Heuvel. Cross-organizational workflow integration using contracts. *Decision Support Systems*, 33(3):247–265, July 2002.

[WVA$^+$09] M.T. Wynn, H.M.W. Verbeek, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Business Process Verification: Finally a Reality! *Business Process Management Journal*, 15(1):74–92, 2009.

[Xu04] L. Xu. A multi-party contract model. *SIGecom Exch.*, 5(1):13–23, 2004.

[ZTS+08] M. Zuzek, M. Talik, T. Swierczynski, C. Wisniewski, B. Kryza, L. Dutka, and J. Kitowski. Formal model for contract negotiation in knowledge-based virtual organizations. In *ICCS 2008, Part III, LNCS 5103*, pages 409–418, Berlin, 2008. Springer-Verlag.