Negotiation as a Generic Component Coordination Primitive

Jean Marc Andreoli and Stefania Castellani

Xerox Research Centre Europe
Grenoble, France
Firstname.Lastname@xrce.xerox.com

Abstract. In this paper, we claim that negotiation is a powerful abstract notion for the coordination of distributed autonomous components, and is therefore a suitable candidate for the definition of a generic coordination middleware tool, at the same level as transactions or messaging. Although specific negotiation mechanisms have been proposed in various application contexts, there is still a need to define a truely generic concept of negotiation, suitable for a middleware layer. This paper provides some elements towards the definition of such a concept. A salient feature of our proposal is that it introduces a rich representation of the state of a negotiation, inspired by proof-nets in Linear Logic and their game semantics, well beyond the traditional state-transition graphs. Furthermore, this representation is entirely decoupled from the dynamics of the negotiation processes that may use it, and hence avoids to rely on any specific "rule of the game" as to how a negotiation should proceed. It can thus adapt to any such rule, the definition of which is delegated to the negotiating components themselves.

Keywords: Negotiation, components, coordination, middleware, protocols

1 Introduction

Negotiation is a pervasive aspect of everyday life and it is not surprising that various approaches have been proposed to use computer software to support some forms of negotiation processes in various applications. In particular, the literature on multi-agent systems is rich with proposals to support negotiations in meeting scheduling, electronic trading, service matching and many other collaborative applications. More generic forms of negotiation also exist in service discovery mechanisms, advanced transaction models, quality of service selection etc. While most of these proposals make sense in the context of the applications for which they have been designed, they are hardly transportable across applications, and across architectural layers within an application. In other words, there is no satisfactory *generic* model of negotiation which could provide the basis of a middleware tool that any distributed application could rely on at multiple levels.

Middleware infrastructures, such as Corba [1], Jini [2], but also the more recent WebServices [3], are gaining momentum, following the success of the Internet and private networks, trying to address recurrent needs of distributed application development, esp. in the domain of e-commerce [4] and trading [5]. From "glue" between various components of a distributed program, middleware is evolving towards the role of "integration tool" coordinating users and applications. Existing middleware products provide some form of support for network communication, coordination, reliability, scalability, and heterogeneity [6]. In the present paper, we propose negotiation as a new concept to enrich the set of traditional coordination facilities offered by middleware systems, such as transactions, messaging, discovery. The challenge here is to devise a notion of negotiation which, on the one hand, goes beyond simple arbitrary interaction, and on the other hand, avoids adopting a specific rule of the game (like Contract Nets or any of the multitude of variants of Auctions studied in the literature) that would only make sense in specific contexts. This is achieved through a game theoretic approach, where the negotiation behaviour of a component is understood in terms of moves in an abstract game "component vs rest-of-the-world".

In order to illustrate our model, called **Xplore**, we make use of a sample case in the domain of collaborative commerce, namely an alliance of printshops in which each partner has the ability to negotiate the outsourcing of print jobs, possibly split into multiple slots, to the other partners in the alliance. This application scenario has been detailed in [7]. Of course, the scope of the proposed model and methods goes well beyond this specific example, and aims at applying to any form of negotiation (commercial or not) in any application involving software objects capable of making autonomous decisions.

Section 2 introduces the basic building blocks of our negotiation model: components, services and data-items. The model itself, based on the notion of negotiation graphs and their partial mirroring, is presented in Section 3 together with an algorithm exploiting this model to implement a fully generic negotiation process. Section 4 provides a detailed example of use of our model in the e-commerce application mentioned above. Sections 5 and 6 conclude the paper.

2 Negotiation Concepts

2.1 Components, Services, Data-Items

From a generic viewpoint, the goal of a negotiation is to have a set of components, behaving as autonomous decision makers, reach an agreement as to a set of actions to be executed by each of them. Once a rule of the game is adopted, the participants to a negotiation may have assigned roles, differentiating their possibilities of interaction, but *a priori*, a negotiation process is essentially *peer-to-peer*. On the other hand, negotiations do not arise out of the blue, and need to be initiated by some components inviting other components. This invitation mechanism is, by nature, *client-server*: the client invites the server and the two roles (host/guest) are essentially distinct. Hence, negotiations offer a perfect framework in which to resolve the classical antithesis between the client-server and peer-to-peer component interaction models.

Without committing to any computational model, we assume that a component is an encapsulated piece of software executing on one or more machines, and publishing to the outside world an interface composed of a list of services. The notion of interface here is similar to that in object-oriented systems, except that the mechanism of invocation of a service in our system (described below) is fundamentally different from that of a method in the interface of a traditional object, be it synchronous (client-server) or asynchronous (peer-to-peer).

A service declaration in the interface simply consists of a name and a list of formal parameters. We are not concerned here with typing issues, so we assume that the parameters are not explicitly typed. Also, parameters have no attached mode (input or output), since that would not make sense in our invocation mechanism. Invoking a service results in some actions being triggered within the component. They may not be visible outside the component but they may be long-lived and may themselves result in other invocations.

Consider, for example, the sample application mentioned in the introduction, supporting the interactions within an alliance of printshops in the process of outsourcing or insourcing some of their print jobs. We may assume that each printshop A is represented by a software agent performing actions related to outsourcing and insourcing of print jobs (possibly under direct human control), realised through the following services:

– `outsrc(job)`: denotes an action by A of outsourcing a print job named `job` (as a whole or in slots); `outsrc` is the service name and `job` is the (here unique) formal parameter.
– `split(job,job1,job2)`: denotes an action by A of splitting a job named `job` into two slots named `job1` and `job2` respectively.
– `insrc(job)`: denotes an action by A of accepting a job named `job`.

Components manage sets of data-items over which negotiations can be engaged. To support heterogeneity and to achieve genericity, the infrastructure makes no assumption whatsoever as to how the components store these data-items and in what format. The only assumption is that each data-item can be described by its properties. Each property is supposed to describe an "aspect" of the item by a "term" (constraint). In our example, a print job is a data-item that can be described by various aspects such as `cost`, `size`, `deadline`, etc., and the term `cost<20` denotes a property that pertains to the aspect `cost`. From the point of view of the negotiation infrastructure, aspects and terms are uninterpreted.

2.2 Bilateral Negotiations

A service invocation is the most basic form of negotiation, between two components with distinguished roles (client and server). It consists of three phases (detailed in Section 3).

– **Invitation:** client $\rightarrow$ server
The client initiates the negotiation by specifying a service from the interface of the server, together with a mapping between the formal parameters of the service declaration in the server and local names of data-items in the

client. These data-items constitute the objects of the negotiation. For example `split(job=J,job1=J1, job2=J2)` denotes an invocation of a service `split` in the server, with formal parameters `job,job1,job2` mapped onto data-items named `J,J1,J2` in the client. Hence, there are three negotiation objects in that negotiation, and agreement must be reached on all of them.
- **Unwinding:** peer-to-peer
  At the time of the invitation, the client may already have established properties about the negotiation objects. Once the invitation is completed, both the client and the server may iteratively refine these properties. During this phase, the roles of client and server are completely blurred, at least from the infra-structural point of view (of course, under a specific rule of the game, the client and the server may take asymmetric roles).
- **Agreement:** client ← server
  Finally, the negotiation ends when the server is satisfied with the properties of the data-items attached to its parameters. The client may then choose to enact or not the proposed agreement.

2.3 Multi-party Negotiations

Service invocations, which are basic bilateral negotiations, can easily be combined to achieve arbitrary multi-party negotiations. The key here is to have a coordinator component launch multiple bilateral negotiation (ie. service invocations) with each participant, and play with each of them as a copycat representative of the others. The inter-dependencies between the bilateral negotiations are simply achieved by *sharing* of the negotiation objects.

For example, consider a multi-party negotiation in which a printshop A_0 wishes to outsource a job split into two slots allocated to, respectively, printshops A_1 and A_2. Here, the participants are A_0, A_1, A_2. We introduce an additional participant C, whose sole role is to coordinate the negotiation in order to achieve the stated result. C just needs to perform the following invocations in parallel (parallelism is denoted by `@`), lauching corresponding bilateral negotiations:

A_0:`outsrc(job=J) @` A_0:`split(job=J,job1=J1,job2=J2) @`
A_1:`insrc(job=J1) @` A_2:`insrc(job=J2)`

Here, C manipulates three data-items (print jobs), locally named, respectively, `J,J1, J2`. Interdependence between the bilateral negotiations launched by C is simply achieved by sharing the negotiation objects, picked among these data-items. Thus, the two bilateral negotiations launched by the invocations A_0:`split` and A_1:`insrc`, between C and, respectively, A_0 and A_1, are interdependent simply because they share a common negotiation object, named `J1` in C and, respectively, `job1` and `job` in the corresponding servers. As far as `J1` is concerned, in the negotiation, C plays with A_0:`split` exactly as A_1:`insrc` does with C and, vice-versa, C plays with A_1:`insrc` exactly as A_0:`split` does with C. This is an instance of the "copycat" strategy, a quite classical concept of game theory. Note however that here: (i) the copycat may involve more than two players and (ii)

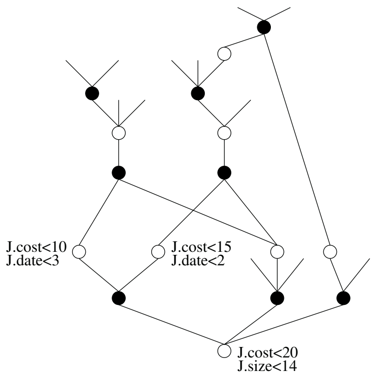

Fig. 1. An example of negotiation graph.

the moves of the players are asynchronous (in fact, the former property is made possible by the latter). Thus, if a data-item named `X` in C is shared in three bilateral negotiations launched by C as service invocations s_1, s_2, s_3, then, as far as `X` is concerned, C plays with its partner in s_i as both of its partners play with it in s_j and s_k (where i, j, k is any permutation of $1, 2, 3$).

Note that, although in the example above the set of servers invited by C is statically defined at its initialisation, this need not always be the case. It may very well happen that C starts a bilateral negotiation with a first server, and then, depending on its evolution, C decides to invite other servers. This dynamicity is both conceptually and practically essential. Conceptually, it means there is no boundary to the complexity of the patterns for combining bilateral negotiations. Practically, for example, it is usual to start e-commerce negotiations by selecting a set of potential partners, where the selection process is itself a negotiation, eg. with a yellow-page service such as UDDI [3].

3 Negotiation Model

We focus here on bilateral negotiations obtained by service invocation, since more complex ones can be built from them. In **Xplore**, a component acting as server in a service invocation stores the state of the negotiation as a data structure called a *negotiation graph* which is manipulated via a *negotiation protocol*.

3.1 Negotiation Graphs

It is assumed that a participant to a negotiation, be it client or server, is capable of making decisions, so that: (i) during the negotiation, the component may explore several alternatives in a decision, leading to negotiation contexts characterised by different combinations of choices for these alternatives; and (ii) each decision by the component, whether it involves alternatives or not, is made on the basis of previous negotiation contexts. Consequently, the overall state of a negotiation for a given component can be captured in a "bi-colored" graph: white nodes represent negotiation contexts and black nodes represent decision points with alternatives. Figure 1 gives an example of such a negotiation graph. The graph must be directed and acyclic, and its edges have the following meaning:

- A white node N has at most one parent node, which must (if any) be black, and represents one alternative in the decision expressed by its parent (black) node. A white node without parent represents a context that does not result of a decision (typically, the initial context of a negotiation).
- A black node N has at least one parent node, which must (all) be white. The context in which the decision represented by N is taken is given by the fusion of all its parent (white) nodes.

To make decisions in an informed way, the participants must have access to the information available at each negotiation context (ie. white node) about the state of the negotiation in that context. Such information consists of pairs composed of a local name of a data-item, and a constraint (term) on that data-item. In our example, such a pair could be `J.cost<20` (the name and the term constraining its value are on each side of the dot), meaning that the cost of the job `J` should not exceed a certain threshold `20`. Consequently, each white node is decorated with specific information about the negotiation at that node in the form of name-term pairs. By inheritance, the overall information available at a white node is given by the set of such pairs attached to that node and to all its ancestor white nodes.

3.2 The Xplore Protocol

The **Xplore** protocol is a set of primitives allowing a component to express negotiation decisions through manipulations of a negotiation graph.

- **Invitation:** During any negotiation, a component can launch and become client of a new negotiation by invoking a service from another (or the same) component (in server mode). Invitation is a negotiation decision of its own, and hence is attached to a negotiation context (white node).
  - **Connect**(n:*nodeId*, m:*mapping*, c:*component*): invites a component service, specified by c, to join the negotiation from node n onward. This results in the creation of a new graph, initially reduced to the single node n, in the server component. The mapping m specifies the translation table between the formal parameters of server c and the local names of data-items in the client.
- **Unwinding:** At any stage, a component (whether client or server) participating in a negotiation can further refine its graph for that negotiation:
  - **Open**($n, n_1, \ldots, n_p$:*nodeId*): creates a node n (which must not already exist) with parent nodes $n_1, \ldots, n_p$ (which must exist). All the parent nodes $n_1, \ldots, n_p$ (if any) must be of the same colour, and n is then of the opposite colour. If $p = 0$, then n is white (creation of a negotiation root context) and if $p \geq 2$, then n is black, hence $n_1, \ldots, n_p$ must (all) be white (fusion of negotiation contexts). In the latter case, the parent nodes must be pairwise compatible. Two nodes are said to be compatible

when they do not appear on branches of the negotiation graph which diverge at a black node, since that would mean that the two nodes are alternatives and hence cannot be part of any final agreement.
  - **Assert**(n:*nodeId*, v:*name*, a:*aspect*, t:*term*): expresses the decision that, in the negotiation context represented by node n in the graph, the value of the data-item named v must have the property expressed by term t pertaining to aspect a. Node n must exist and be white. This is the way to populate context nodes with information about the negotiation state at these nodes, for the other concerned participants to see (and eventually react).
  - **Request**(n:*nodeId*, v:*name*, a:*aspect*): expresses that, to proceed with the negotiation, the component is interested in information, obtained through assertions by the other concerned participants, about a particular aspect a of a data-item named v at node n (which must exist and be white).
  - **Quit**(n:*nodeId*): expresses that the negotiation will never succeed at node n (which must exist and be white), so the other concerned participants need not elaborate further.
- **Agreement:** A component in server mode in a negotiation may end the negotiation in one branch of the graph
  - **Ready**($n_1, \ldots, n_p$:*nodeId*): expresses that the server is satisfied with the state of the negotiation at nodes $n_1, \ldots, n_p$ (which must exist and be white). In other words, the component has seen enough information and is ready to finalise the negotiation in the state represented by the fusion of these nodes, which must be pairwise compatible.

When a server expresses its readiness, the client must decide whether or not to enact the agreement. This can be obtained by a classical transactional two-phase commit protocol, where the client first **Reserve**s the agreement delivered by the server, then **Confirm**s or **Cancel**s the reservation. This allows real synchronisation of agreements between multiple servers (needed since all the other operations, including the **Ready**, are asynchronous). We do not discuss here further the transactional management of agreements, as this has been widely investigated in the literature. For example, the component coordination infrastructure CLF [8], which had a deep influence on the design of Xplore, proposes a form of light-weight transaction management which is directly applicable here.

3.3 The Xplore Infrastructure

An application in **Xplore** is seen as one big negotiation, with sub-negotiations nested at an arbitrary level by the invitation mechanism (as with nested transactions). Hence, from the negotiation infrastructure point of view, it is possible to represent the state of the application as a tree (also called the invocation network), the vertices of which are labeled by negotiation graphs and the edges of which are the service invocations. The negotiation infrastructure operates on that network, augmented each time the **Connect** primitive is executed. Its main roles are: (i) at each edge of the network, to synchronise back and forth the *relevant* decisions taken in the two graphs linked by that edge; and (ii) at each vertex of the network, to detect *consensus* at that vertex, ie. agreement decisions at the end vertex of its out-going edges. These two functions form the software core of the **Xplore** infrastructure, which is entirely distributed across the vertices of the invocation network (each vertex lives in a component). The main difficulty arises from the fact that each negotiation operation at a vertex of the network is contextualised by a node in the corresponding negotiation graph, and the topology of the graph must be taken into account to process each operation. For example, the synchronisation algorithm has the following behaviour.

- A call to the **Open** primitive never needs to be mirrored immediately, because it always creates a fresh node with no information attached to it, so it is irrelevant to any neighbour in the invocation network.
- A call of the form **Assert**(n,v,a,t) in one vertex need only be replicated onto those neighbours in the invocation network which have previously made calls of the form **Request**(n',v,a) where nodes n, n' are compatible, ie. do not lie on alternative branches (diverging at a black node). In the replicated call, the local name v must be replaced by its image in the conversion table attached to the concerned neighbour. Furthermore, the replicated call may need to be preceded by calls to the **Open** primitive to make sure that node n is mirrored on the concerned neighbours.
- A call of the form **Request**(n,v,a) in one vertex need only be replicated onto those neighbours in the invocation network which have a local name v' corresponding to v in the conversion table attached to them. The replicated call must specify v' in place of v. As above, the replicated call may require prior node mirroring using **Open**. Furthermore, all the calls of the form **Assert**(n',v,a,t) at any node n' related to n must be replicated on the originator of the **Request** call.

The consensus detection algorithm at a vertex k_o is even more involved as it needs to detect combinations of calls of the form **Ready**(N_k) at a set K of vertices k linked to k_o in the invocation network, ie. at servers invited by k_o as client. Not all the servers invited by k_o need to be included in K, but there must be at least one of them, and if one server k is included in K, then so should all those invited at nodes which are ancestors of the nodes of N_k. Furthermore, all the nodes in $N = \bigcup_{k \in K} N_k$ must be pairwise compatible, ie. not lie on alternative branches (diverging at a black node). In that case, consensus is detected and the primitive **Ready**(N) is called by k_o as server.

An implementation along these lines has been realised, where the components process the calls to the **XPlore** protocol in an asynchronous, actor-like fashion [9].

4 An Example

We now illustrate the negotiation framework presented above on the sample negotiation introduced in Section 2.3:

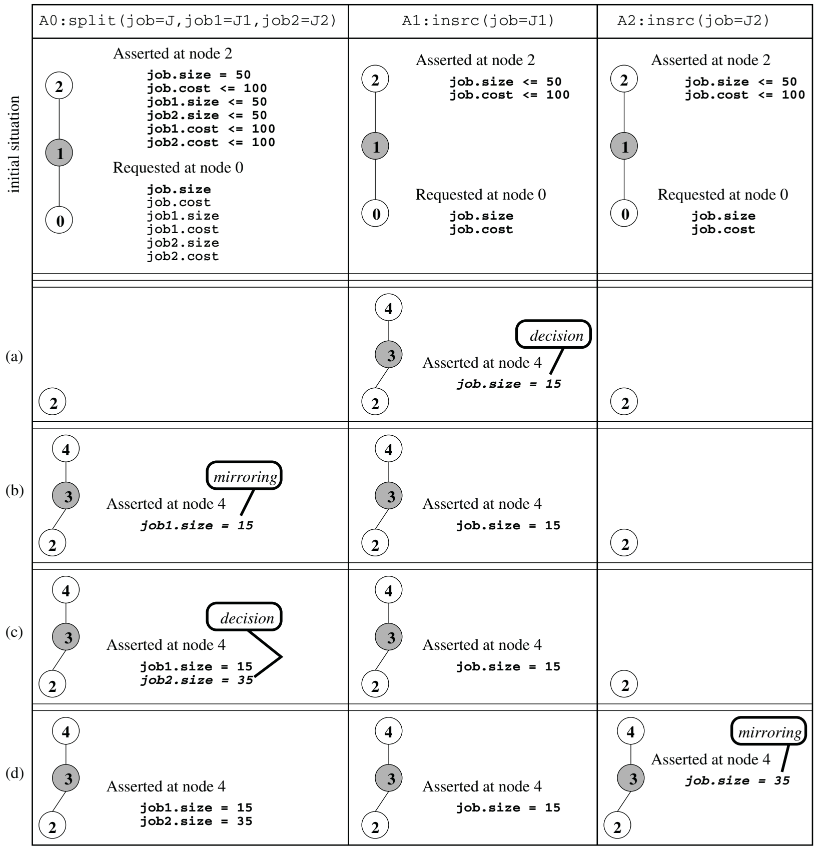

Fig. 2. An example of negotiation graph mirroring.

A_0:`outsrc(job=J) @` A_0:`split(job=J,job1=J1,job2=J2) @`
A_1:`insrc(job=J1) @` A_2:`insrc(job=J2)`

We have here four components: C, A_0, A_1, A_2, where C is a coordinator, and A_0, A_1, A_2 are printshops (their interfaces are described in Section 2.3). At initialisation, the coordinator C creates a graph with a single root node, then, from this graph, invites the participants above in separate bilateral negotiations, using calls to the **Connect** primitives at the root node. In the end, five graphs are created together with four links in the invocation network, between the graph in C and each of the graphs corresponding to the invocations performed by C (ie. two for A_0 and one for each of A_1, A_2). The participants then start to work on their negotiation graphs using the **Xplore** primitives, automatically mirrored in a timely way by the infrastructure. Each path in the graph can be viewed as a dialog between the participants consisting of successive refinements of the terms of the agreement. Having multiple paths in the graph allows several interwoven alternatives to be explored in parallel. For example, the following fragment of a conversation among the three printshops through the coordinator corresponds to a phase of the negotiation where A_1 makes a proposal to A_0 on one slot of a job. Figure 2 shows the negotiation graph mirroring corresponding to this dialog.

– A_1:`insrc`.{Open(3,2),Open(4,3),Assert(4,job.size=15)}
  A_1:`insrc` decides to explore one alternative in which the size of the job it would accept is 15. By creating black node `3`, it leaves the possibility of exploring other alternatives. At white node `4`, A_1 may locally attach resouces which it anticipates will be needed to fulfil its commitment in that alternative, but this need not be made public. These decisions are taken according to its own specific semantics, characterising, here, its strategy in accepting jobs. The resulting situation is depicted in Figure 2(a).
– A_1:`insrc`$\mapsto C$.{Open(3,2),Open(4,3),Assert(4,J1.size=15)}
  We assume that C had previously informed A_1:`insrc` that the size aspect of its `job` parameter was requested in the negotiation in context `2` (or one of its ancestors). The infrastructure therefore mirrors into C the information it has just asserted on that aspect in context `4`. Since C does not know this context yet, the infrastructure mirrors the missing bit of the graph (nodes `3` and `4`) into C. Note that the formal parameter name `job` known by A_1:`insrc` is converted into the corresponding name `J1` known by C.
– $C \mapsto A_0$:`split`.{Open(3,2),Open(4,3),Assert(4,job1.size=15)}
  The mirroring process continues from C, which now looks for all the participants which had requested information on the size of `J1` in the negotiation; here, only A_0:`split`. The information is therefore passed on to it. Again, node `4` not being heard of yet by A_0:`split`, the missing bit of the graph is first mirrored. And again, the data-item name `J1` is converted into the corresponding parameter name `job1`. The resulting situation is depicted in Figure 2(b).
– A_0:`split`.Assert(4,job2.size=35)
  A_0:`split` had previously been informed that the total size of its `job` is constrained to be equal to 50 in context `2` (or one of its ancestors). The constraint therefore holds in context `4` since it is a descendent of `2`. Furthermore, knowing that in context `4` the size of slot `job1` is constrained to be equal to 15, A_0:`split` infers that the size of the other slot `job2` is constrained to be equal to 35 in that context. This constraint propagation, which is here a characteristic of the semantics of service A_0:`split`, could be performed automatically by a constraint solver. The resulting situation is depicted in Figure 2(c).
– A_0:`split`$\mapsto C$.Assert(4,J2.size=35)
  Mirroring is re-activated from A_0:`split` to C, which had previously requested information on the size of `job2`. Again, the parameter name `job2` is converted into its corresponding name `J2` in C.
– $C \mapsto A_2$:`insrc`.Assert(4,job.size=35)
  The mirroring process continues in C, which passes on the information it has just received on the size of `J2` to A_2:`insrc` which had initially requested it. The name `J2` is converted again into the parameter name `job`. The resulting situation is depicted in Figure 2(d).

5 Discussion

A distinctive feature of distributed applications is that, even when their functionality is simple, they tend to involve a multitude of generic issues such as consistency, awareness, authorisation, etc. Each of these aspects have been studied separately, each leading to a different class of models and solutions. Thus, transactions deal with consistency issues, messaging and discovery with awareness issues, cryptography and security protocols with authorisation issues, etc. Software engineering techniques such as Aspect-Oriented Programming [10] help specify and maintain these aspects separately, but they do not provide a unified model in which these aspects would appear as facets of the same mechanism. We claim here that many aspects, at least those pertaining to component coordination, can be unified into a single model, around a generic notion of *negotiation*.

Negotiation is indeed a proto-typical kind of coordinated process, and the literature on e-negotiation abounds with system descriptions which specify how participants should coordinate in order to achieve certain goals. However, these proposals are often embedded into full-blown applications (eg. [11]), and rely on a plethora of so-called generic mechanisms [12] (Contract-Nets, Auctions, Match-making etc. with many variants) which it is difficult to choose from *a priori*, outside a specific application context. These proposals are therefore not suitable for a generic middleware infrastructure, where the components and the service they offer, hence their negotiation needs, can be widely diverse. Our approach, on the contrary, seeks to abstract away any specific negotiation scheme and to formalise only universal characteristics of negotiation processes: (*i*) the possibility of exploring alternative branches and (*ii*) the incremental refinement of the terms in a negotiation.

The model of negotiation we have presented here draws on ideas coming from constraint programming and constraint propagation [13], in particular distributed constraint satisfaction [14] where constraints are viewed as autonomous agents propagating "no-good" information via their shared variables, and cooperative constraint solving [15]. The other major source of inspiration is proof-theory in formal logic, in particular proof-nets in Linear Logic [16] which offer a totally desequentialised representation of logical inferences as a graph, similar to our negotiation graphs (inferences are here negotiation decisions). The game theoretic interpretation of proofs [17] has also strongly influenced our view of negotiation as games.

6 Conclusion

In this paper, it is claimed that negotiation is an appropriate abstraction for a middleware level service for the coordination of distributed components, on a par with traditional middleware concepts such as transactions, messaging and discovery. Our model provides an abstract representation of the state of a negotiation through a bi-colored graph, and exploits this model to build a fully generic negotiation process, viewed as the partially synchronised construction of such graphs.

References

1. McConnell, S.: Negotiation Facility. Technical report, OMG (1999)
2. Waldo, J.: The Jini Architecture for Network-Centric Computing. Communications of the ACM **42** (1999) 76–82
3. Newcomer, E.: Understanding WebServices: XML, WSDL, SOAP, and UDDI. Addison Wesley Professional (2002)
4. Charles, J.: Middleware Moves to the Forefront. IEEE Computer Magazine **32** (1999) 17–19
5. Marvie, R., Merle, P., Geib, J.M., Leblanc, S.: TORBA: Trading Contracts for CORBA. In: Proc. of COOTS –6th USENIX Conference on Object-Oriented Technologies and Systems, San Antonio, Texas, USA. (2001)
6. Emmerich, W.: Software Engineering and Middleware: A Roadmap. In: Proc. of ICSE 2000, The future of Software Engineering, Munich, Germany (2000)
7. Andreoli, J.M., Castellani, S., Munier, M.: AllianceNet: Information Sharing, Negotiation and Decision-Making for Distributed Organizations. In: Proc. of EcWeb2000, Greenwich, U.K. (2000)
8. Andreoli, J.M., Arregui, D., Pacull, F., Riviere, M., Vion-Dury, J.Y., Willamowski, J.: Clf/mekano: a framework for building virtual-enterprise applications. In: Proc. of EDOC'99, Manheim, Germany (1999)
9. Agha, G., Mason, I., Smith, S., Talcott, C.: A foundation for actor computation. Journal of Functional Programming **7** (1997) 1–72
10. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proc. of ECOOP '97, Jyväskylä, Finland, Springer-Verlag (1997)
11. Chavez, A., Maes, P.: Kasbah: An agent marketplace for buying and selling goods. In: Proc. of 1st Conference on Practical Applications of Intelligent Agents and Multi-Agents, London, U.K. (1996) 75–90
12. Dignum, F., Sierra, C., eds.: Agent Mediated Electronic Commerce. LNAI 1991, Springer Verlag (2001)
13. van Hentenryck, P., Saraswat, V.: Strategic directions in constraint programming. ACM Computing Surveys **28** (1996) 701–726
14. Yokoo, M., Hirayama, K.: Algorithms for Distributed Constraint Satisfaction: A Review. Autonomous Agents and Multi-Agent Systems **3** (2000) 185–207
15. Monfroy, E., Castro, C.: Basic operators for solving constraints via collaboration of solvers. In Campbell, J., Roanes-Lozano, E., eds.: Proc. of AISC 2000, Madrid, Spain, Springer-Verlag (2001) 142–156
16. Girard, J.Y.: Linear logic. Theoretical Computer Science **50** (1987) 1–102
17. Laurent, O.: Polarized games. In: Proc. of LICS'02, Copenhagen, Denmark, IEEE Computer Society (2002) 265–275