FINE-GRAINED CONTRACT NEGOTIATION FOR HIERARCHICAL SOFTWARE COMPONENTS

Hervé Chang, Philippe Collet

Projet OCL

Rapport de recherche
ISRN I3S/RR–2005-08–FR

Mars 2005
RéSUMÉ :

MOTS CLÉS :

ABSTRACT:
Component-based development addresses the complexity of large applications by building software systems from reusable software components. To support reuse and successfully combine units of software, the contractual approach turns out to be well-suited to specify and verify components and their interactions. However, as contracts must support functional and extra-functional properties, they are frequently challenged by fluctuations in extra-functional aspects and dynamic reconfigurations of components. In this paper, we propose a negotiation model in which components have clearly identified roles and interact in order to automatically restore the validity of contracts. The negotiation model currently supports a concession-based negotiation policy and is well-suited to behavioral contracts based on executable assertions. This model is integrated into ConFract, a contracting system for the Fractal hierarchical component model.

KEY WORDS:
component-based software engineering, negotiation, contract, hierarchical components, Contract-Net Protocol, ConFract, Fractal
Fine-grained Contract Negotiation for Hierarchical Software Components

Hervé Chang and Philippe Collet
University of Nice Sophia Antipolis, I3S Laboratory
Sophia Antipolis, France

Abstract

Component-based development addresses the complexity of large applications by building software systems from reusable software components. To support reuse and successfully combine units of software, the contractual approach turns out to be well-suited to specify and to verify components and their interactions. However, as contracts must support functional and extra-functional aspects, they are frequently challenged by fluctuations in extra-functional properties and dynamic reconfigurations of components. In this paper, we propose a negotiation model in which components have clearly identified roles and interact in order to automatically restore the validity of contracts. The negotiation model currently supports a concession-based negotiation policy and is well-suited to behavioral contracts based on executable assertions. This model is integrated into ConFract, a contracting system for the Fractal hierarchical component model.

1. Introduction

Component-based software engineering (CBSE) has received much attention from both academia and industry in recent years. Applying such an approach enables developers to manage software development complexity, in particular by separating interface from implementation and by explicitly describing the architecture of the application. Moreover, software components usually expose only required and provided interfaces and basic software contracts are defined during component assembly. A crucial issue in contracting components is then the ability to finely address functional properties as well as extra-functional requirements that concern both the assembly and the execution of components [1, 20]. This becomes essential with hierarchal software components, as contracts must concern not only component interfaces but also component themselves and compositions of components. The challenge is thus to express and to contractualise properties over compositions of components.

To tackle this problem, we proposed ConFract [6], a contracting system for hierarchical components, based on the Fractal [4] platform, which provides reflective capabilities and supports dynamic reconfigurations of components. The aim of the ConFract system is to specify and to verify both functional and extra-functional properties through interface contracts established on the interface bindings, and composition contracts located on components to supervise their use and their content. However, as extra-functional properties can be highly fluctuating, contracts can be frequently challenged and violated. The resulting exceptions and handy code are then multiplied over the architecture.

We are thus concerned about automatically restoring, in most cases, the validity of contracts.

In this paper, we propose negotiation mechanisms, inspired from those conceived in multi-agent systems, which make it possible to adapt components or contracts at assembly and run times. In this negotiation model, contract violations are handled by activating an atomic negotiation for each violated provision of a contract. The structure of the atomic negotiation is inspired from the Contract-Net Protocol [19] and involves clearly identified components with their responsibilities in the contract. Different negotiation policies are intended to be integrated in the model, and we describe in this paper a concession-based negotiation policy, which is well-suited to behavioral contracts based on executable assertions. Moreover, as components carry their own negotiation ability, the negotiation model supports dynamic reconfigurations of components.

The rest of the paper is organized as follows. In the next section, an overview of the ConFract contracting system is given and the problem is stated through an example. In section 3, the negotiation model is described. Section 4 summarizes the integration of this negotiation model in the ConFract system. In section 5, related work are discussed and finally section 6 concludes this paper with some indications on future work.

2. The ConFract System

The purpose of the ConFract system is to specify and to verify functional and extra-functional properties on Fractal software components. This system dynamically builds contracts from specifications at assembly time and updates...
them according to the dynamic reconfigurations of components. Consequently, these contracts are first class entities during assembly and run times.

In ConFract, contracting mechanisms are clearly separated from the definition of specifications. Specifications are currently written in the CCL-J language (Component Constraint Language for Java) which is inspired by OCL [11] and enhanced to be adapted to the Fractal model. Currently, classic categories of specifications like preconditions, post-conditions and invariants are supported. In the rest of the paper, the specifications given as examples are expressed in CCL-J. ConFract supports several types of contracts which take into account the specificities of the Fractal model. The Fractal component model basically enables software developers to hierarchically organize an application. Fractal components can be nested and can be bound through several required (client) and provided (server) interfaces to enable components communication. These interfaces are composed of a name and a signature, which is equivalent to a Java interface. Internally, a Fractal component is formed out of two parts: a content and a membrane. The content of a composite component consists of a finite set of other components called subcomponents, which are under the control of the enclosing component. The membrane embodies the control behavior and is typically composed of several controller and interceptor objects provided by the Fractal platform. It can have external interfaces which are accessible from outside the component and internal interfaces which are only accessible from its subcomponents. Basic Fractal controllers are dedicated to manage life cycle (Lifecycle-Controller), component bindings (BindingController) and component content (ContentController), which are respectively depicted as LC, BC and CC in figure 1. More details about ConFract and Fractal can be respectively found in [6] and [4].

In the rest of this paper, we use, as a working example, a basic multimedia player developed with the Sun Java Media Framework API1. The architecture of the multimedia player is shown on figure 1 and presents a FractalPlayer component containing four subcomponents: Player which exclusively provides the playing service through its start method and manages some of its functioning parameters through attributes, GuiLauncher which manages the GUI part, VideoConfigurator which provides services to optimize the playing service (the canPlay method evaluates the ability to entirely play a video in its specific display size, according to battery and memory levels) and finally Logger which manages a history of played videos (the lastUrl method allows one to get the url of the most recently played video).

2.1. Types of Contract

The ConFract system distinguishes several types of contracts according to the specifications given by the designers.

- Interface contracts are established on the connection point between each pair of client and server interfaces and the retained specifications only refer to methods and entities in the interface scope. As an example, a CCL-J specification could be defined in the scope of the MultimediaPlayer interface to express a precondition concerning the start method (upper left part of figure 1).

- External and Internal composition contracts are defined on a component membrane. External composition contracts are located on the external side of each component membrane and consist of specifications which refer only to external interfaces of the component. Internal composition contracts are located on the internal side of a composite component membrane and consist, in the same way, of specifications which only refer to internal interfaces of the enclosing component or to external interfaces of its subcomponents. Consequently, external composition contracts express usage and external behavior rules of components, whereas internal composition contracts express the assembly and internal behavior rules of the implementation of composite components.

As an example, the specification referring to <pl> in figure 1 will be added to the external composition contract of <pl>. This specification defines both a precondition and a postcondition for the start method of the Fractal interface named mpl. The precondition also refers to another external interface of <pl>, the required interface named c of type Configurator, to express acceptable conditions to play the video. As for the postcondition, it refers to the required interface named h of type History and specifies that the last entry of the history matches the played video. In the remainder of the paper, we focus on this external composition contract to illustrate our contribution.

During the reification of a contract, specifications are brought together given their category and each specification becomes a provision of a contract with clearly identified responsibilities. The ConFract metamodel describes each category of specification (e.g a precondition of an external composition contract) and its associated responsibilities among the set of participating components. These responsibilities can be either (i) guarantor which acts to ensure the provision and must be notified in case of violation of the provision, (ii) beneficiaries which can rely on the provision or (iii) possible contributors which are needed to check the provision. The following table describes the re-

---

1 http://java.sun.com/products/java-media/jmf/
sponsibilities² of guarantor and beneficiaries for the external composition contract of the component \(<pl>\):

<table>
<thead>
<tr>
<th>Interface role</th>
<th>Construct</th>
<th>Guaran tor</th>
<th>Beneficiaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>server (mpl)</td>
<td>post</td>
<td>(&lt;fp&gt;)</td>
<td>(&lt;gl&gt;)</td>
</tr>
<tr>
<td>client (c)</td>
<td>pre</td>
<td>(&lt;pl&gt;)</td>
<td>(&lt;fp&gt;, &lt;gl&gt;)</td>
</tr>
</tbody>
</table>

For example, on the component \(<pl>\), for the postcondition of a method on its server interface mpl, the guarantor is the component itself as it implements the method and provides the interface, and the beneficiaries are \(<fp>\), which contains \(<pl>\), and \(<gl>\), which is connected to the interface \(<mpl>\). As the external composition contract represents the usage rules of the component \(<pl>\), it is logical to attribute the responsibility of this provision to the component \(<fp>\), which contains \(<pl>\), as it is to \(<fp>\) to handle its internal assembly.

### 2.2. Contract Life Cycle

In the ConFract system, the various contracts are managed by contract controllers (named CTC in figure 1) which are located on the membrane of every component. As sub-components are under the control of the enclosing component, every contract controller of a composite component manages the life cycle and the evaluation of the contracts that refer to its subcomponents and their bindings. Consequently, each contract controller manages the life cycle and the evaluation of (i) the internal composition contract of the composite component on which it is located, (ii) the external composition contract of each subcomponent in its content and (iii) the interface contract of every interface bindings in its content. Contract controllers interact with other Fractal basic controllers (life cycle, binding and content controllers) in order to monitor configuration changes (e.g. adding or removing components, binding interfaces) and to dynamically react by creating and updating their contracts.

As for checking of CCL-J based contracts, contract provisions that specify component invariants are evaluated at the end of assembly time. Other provisions are dynamically checked. When a method is called on a Fractal interface, the provisions of the different contracts that refer to this method are checked with the following rules. Preconditions from the interface contract are first checked, then preconditions from the external composition contract of the callee are checked, and finally preconditions from the internal composition contract are checked. Similar checks are done with postconditions and method invariants when the method returns.

---

² We only present here responsibilities regarding pre and postconditions.
2.3. Motivations to Support Contract Negotiations

In our example given in figure 1, the precondition of the external composition contract defines acceptable conditions before playing a video. The VideoConfigurator component evaluates, through the canPlay method, the Player ability to entirely play the given video, considering different parameters such as system parameters (e.g. battery level) and the video source (taking into account the decoding complexity of the video). This specification shows the expressiveness of the CCL-J language, which allows designers to combine both functional and extra-functional aspects within a specification and to parameterize specifications with application-level elements (component, interface, method input parameters). Once built, contracts may be challenged by the fluctuations in extra-functional properties, such as variations in available resources, changing requirements and environment, or by the dynamic reconfigurations of components. Currently, the ConFracT system handles contract violations by notifying the guarantor component with the violation context. In an ad hoc approach, it is then possible to manage these violations by adding specific handling code throughout the architecture.

In our approach, as the responsibilities of each provision are precisely determined, all responsible components are involved in a negotiation process which aims at restoring the validity of contracts. This negotiation can occur either at assembly time or at run time. At assembly time, such negotiation assists the system integrator to detect and to solve incompatible constraints before run time, and to leverage the qualities of assemblies while taking into account each component specification. At execution time, this negotiation contributes more to increase the autonomy of the application by handling contract violations in an automatic way. In our example, the negotiation could then lead to lower battery consumption by reducing the video display size, or to completely withdraw the constraint, thus implying that the video might be stopped if battery gets weak.

3. Negotiation Model

In the ConFracT system, contract provisions both specify functional and extra-functional aspects. The negotiation of functional aspects is more appropriate during testing. On the other hand, as extra-functional aspects address configuration and run time qualities of components (usually classified under the larger concept of Quality Of Service) and their relationship with the environment (deployment constraints), we mainly focus on negotiating provisions which specify such aspects. As contracts are composed of provisions, our negotiation model thus tries to restore the validity of a whole contract by activating an atomic negotiation for each violated provision of the contract. This negotiation can occur at assembly time if a provision can be statically checked, or dynamically at run time if it requires an execution context. Moreover, since the ConFracT system identifies, in a fine-grained way, responsible components for each provision among the set of its participating components, our negotiation model relies on the responsibilities of guarantor, beneficiary or contributor to deal with the provision. Accordingly, given each participating component role, it is possible to develop different negotiation policies which define the participants reasoning and drive the negotiation.

3.1. Atomic Negotiation Parties and Protocol

An atomic negotiation involves negotiating parties and follows a negotiation protocol partly inspired from the extended Contract-Net Protocol (CNP) [19]. This protocol, commonly applied in multi-agent systems for decentralized tasks allocation, basically organizes the interactions between a manager and contractors following three steps: announcing, bidding and rewarding. In our model, we retrieve this organization by defining as negotiating parties (i) the contract controller in the role of the negotiation initiator, which controls the negotiation process, as it manages contract life cycle and operates contract checking, (ii) participants, which are composed of the participants of the provision, and of an external contributor which helps representing interests from a ‘third party’ with deeper decision ability. For example, this external contributor could be the system integrator willing to inject higher level constraints into the system (e.g. deployment constraints) or various data to parameterize the negotiation process (e.g. the negotiability of the provision 3 given a specific deployment context, negotiation timeout, propagation of negotiation information to lower hierarchy levels). In our example that refers to the external composition contract, the negotiating parties of the precondition on the start method are the contract controller of <fp> as initiator, <fp> itself as guarantor and <pl> as beneficiary (cf. figure 2). For the postcondition, the parties are the contract controller of <fp> as initiator, <pl> as guarantor, and <fp> and <gl> as beneficiaries.

The negotiation protocol splits up in three steps. The initiator requests proposals from the negotiating parties to restore the validity of the violated provision and those parties propose modifications. Afterwards, the initiator performs the proposed modifications and checks the provision validity. Those steps are iterated over all participants proposals. If a satisfactory solution is found, the negotiation process is finalized and terminates. Otherwise, if none of the proposals restore the validity of the provision or the negotiation timeout is reached, the negotiation fails. These rules governs the interactions and describes the parties actions, but

---

3 A provision is negotiable if the negotiating parties agree to negotiate it.
to complete our negotiation model it is necessary to define policies for the negotiation behavior.

3.2. Atomic Negotiation Policy

Concession-based Policy Principles As a first kind of negotiation behavior, we present a concession-based policy in which the negotiation initiator and beneficiaries interacts by a process of concession making. The negotiation initiator requests concessions from the beneficiaries by asking them either to rely on an under-constrained provision or to reconfigure some of their functioning parameters. In this way, such concession proposal can lead to:

- change the provision in the current execution context,
- or reconfigure parameters by changing some key attributes of beneficiaries components, while retaining the same provision.

This policy is sufficiently general as it can be applied as long as the specification formalism used allows the Con-Struct system to divide an expression into several unitary provisions. In section 6, some ideas to develop other policies are suggested.

In order to completely define the concession-based policy, the beneficiary role is refined by introducing a principal beneficiary and a secondary beneficiary. Principal beneficiaries are directly concerned with a provision and, in fact, have the ability to act during the negotiation process. On the opposite, secondary beneficiaries are more passive and cannot make the negotiation progress. In our example concerning the postcondition of the start method of the mpl interface, <fp> is the principal beneficiary because it is responsible of the correct usage of the contract (played video history is correct) and can act on its subcomponents during the negotiation. <g1> is the secondary beneficiary because it only appears as a simple client of the video player service and does not have knowledge of the other interfaces referenced in the contract.

Negotiation Process When the verification of a provision fails, the concession-based negotiation process is decomposed into three steps, as described in figure 3.

1. in step 1, the initiator requests the negotiability of the violated provision from the beneficiaries and it evaluates the overall negotiability by computing a weighted linear additive scoring function.

2. in step 2, if the provision is negotiable, the initiator requests concession proposals from principal beneficiaries and for each proposal, it performs changes on the provision or on parameters and re-checks the provision. If a proposal re-validates the provision, the atomic negotiation is successfully completed and changes are committed.

3. in step 3, if proposed changes are not satisfactory or the withdrawal of the provision has been issued in the process fails, the concession-based negotiation process is decomposed.

Reasoning of the Beneficiaries To successfully act during the negotiation process, the decision model of the principal beneficiaries is based on sets of alternatives which express their preferences. Thus, a component named C may define the following set of alternatives \( A_{\#p,C} := \{ A_{\#p,C}^1, A_{\#p,C}^2, ..., A_{\#p,C}^n, \text{STOP or RELEASE} \} \) to negotiate the provision identified as \(#p\). For every concession proposal requested by the initiator, the component C will successively propose its preferred alternative among this set. In this policy, an alternative \( A_{\#p,C}^i \) corresponds either to provision or parameters changes, and STOP or RELEASE are used to notify the end of the concession process while retaining the provision or withdrawing it.

In our example, the provision that ensures the ability to entirely play a video can only be checked at run time since it requires the execution context and up-to-date resource information. If the verification of this provision fails, the concession-based negotiation process would involve the contract controller of \(<fp>\) as the initiator and \(<g1>\) as the unique and principal beneficiary. The negotiation outcomes may lead to progressively reduce the video display size in order to decrease battery consumption. If these concessions are not satisfactory, the provision could then be completely withdrawn, and in this case, since the constraint has been discarded, the video might be interrupted if the battery level becomes weak.

---

4 A proposal can consist in the withdrawal of a provision to suggest to completely remove the provision.
In this scenario, `<pl>`’s successive concessions may be driven by the following set of alternatives $A_{pre,<pl>} := \{ (width \leftarrow width, height \leftarrow height) , RELEASE \}$. With this set of alternatives, `<pl>` initially proposes an alternative describing the changes on its parameters width and height. The initiator performs the change and evaluates the provision. If the verification succeeds, the negotiation outcome is successful, otherwise the initiator cancels the changes, and requests for a new concession to which `<pl>` responds by proposing the provision withdrawal with the RELEASE alternative. The negotiation finally terminates with the provision withdrawal since `<pl>` is the only beneficiary.

In the same way, the provision concerning the correct history of played videos can only be checked at run time. The negotiating parties would be the contract controller of `<fp>` as initiator, `<pl>` as guarantor and `<fp>` and `<gl>` respectively as principal and secondary beneficiaries. During the negotiation process, `<fp>`, with the set of alternatives $A_{post,<pl>} := \{ RELEASE \}$, will propose the provision withdrawal and the initiator will consult `<fp>` and `<gl>` to definitely decide to withdraw it.

### 3.3. Other Negotiation Examples

As previously mentioned, the ConFract system also allows one to specify component internal composition. For example, one could define the following specification:

```java
param jmfMin = JMF.V2_1
on <fp>
inv <pl>.attributes.getJmfVersion().compareTo(jmfMin) >= 0
```

to constraint about the content of `<fp>` such as its subcomponent `<pl>` runs with a version of the JMF API greater than 2.1. The following internal composition contract will then be built:

```java
internal composition contract on <fp>
participants: <fp> <pl>
param jmfMin = JMF.V2_1
inv guarantor: <fp> beneficiaries: <fp> <pl>
pl.attributes.getJmfVersion().compareTo(jmfMin) >= 0
```

In this contract, the provision expresses a configuration invariant that can be verified at assembly time, after all re-configuration actions. In this way, a component `<pl>` running with a JMF version older than 2.1 would trigger an atomic negotiation which involves the contract controller of `<fp>` as initiator, and `<fp>` itself as the unique guarantor and beneficiary. Given the set of alternatives for `<fp>` $A_{inv,<fp>} := \{ jmfMin <= 2.0, STOP \}$, the constraint would be first changed to version 2.0. If this is not satisfactory, STOP would be proposed to notify the termination of the negotiation while retaining the provision.
ther actions could be considered such as updates or reconfigurations of the components $<f_p>$ or $<p_1>$.

As for interface contracts, their negotiating parties are the contract controller of the surrounding component and the client and server components as the guarantor and the beneficiary. However, as interface contracts mostly specify functional properties, the relevance of the negotiation depends on the target application and on the stage in the development process (development, testing, integration, deployment). If such negotiation occurs, a consistent policy could completely withdraw the violated provision (if the beneficiary agrees).

4. Integration

The negotiation parties and the negotiated provisions are retrieved from the ConFract system. In the implementation of our negotiation protocol, the negotiating parties communicate through method calls. Moreover, since the negotiation is considered as an important and intrinsic role of the contract life cycle, contract controllers are extended with negotiation ability through negotiation interfaces. In the concession-based policy, given the negotiation role of the component on which a contract controller is set up (initiator or principal beneficiary), the negotiation interfaces allows the contract controller to take over the control of the negotiation process of its internal composition contract and of all interface contracts in its content, and to reply to the requests of the contract controller of the surrounding component (negotiability, concession proposals, provision withdrawal) when negotiating its external composition contract. Concerning the reasoning of the beneficiaries, sets of alternatives drive each principal beneficiary reasoning and can be defined either by the designers of the specifications or by the system integrator from documented specifications.

5. Related Work

Negotiation is used in multi-agent systems (MAS) to coordinate actions, resolve conflicts and above all share scarce resources. Several negotiation mechanisms base their interactions model on the general Contract-Net Protocol [19]. The negotiation strategies are often specialized and dedicated to a specific application whereas, Faratin et al. defined in [8] advanced strategies with responsive and deliberative mechanisms. In our model, although we adapted the CNP protocol, the negotiation differs by dealing with the provisions of a violated contract and our strategies are defined by sets of alternatives closely related to work from Balogh et al. [2]. Furthermore, while MAS are endowed with social and environmental autonomy, our approach is more closely related to the autonomic computing concept [12], by aiming at automatically restore the validity of contracts through adaptation of components or contracts at assembly and run times.

In component-based applications, some approaches handle fluctuations in the available resources through a control architecture. It is based on a monitoring system combined with adaptation mechanisms. These mechanisms basically relies either on adaptation rules based on the event-action construct which performs component re-parameterization and structural modifications [7, 14, 5], or on setting up the optimal component configuration deduced by an exhaustive computation. Compared to those propositions, our model not only clearly separates the property which leads to the adaptation process (in our case, a violated contract provision) to recovery actions, but also it appears to be more flexible since it consults precisely involved components on the basis of a negotiation process. Moreover, as components carry their reasoning, our model supports reconfigurations of components. A newly inserted component that is involved in a negotiation process will thus be able to act during the negotiation.

Numerous works use QML (QoS Modeling Language) [9] to specify and contractualise QoS properties. In QML, extra-functional aspects are described by specifying expected quality levels. However, specifications in QML express relatively high-level constraints without explicit representation at run time. Particularly, it is not possible, contrary to CCL-J, to combine both functional and extra-functional aspects within a specification, as well as referring to component, interface or method parameters at the application level. QML is usually used to specify QoS in distributed systems [3, 13] and component models [17, 15]. In distributed systems, negotiation protocols deal with a restricted number of QoS parameters relating to network characteristics and mainly consider the negotiation as the process of reserving resources [18, 16]. In component models, most QoS negotiation protocols are dedicated to multimedia applications and the negotiation consists in controlling components admission and performing resources reservation to support the agreement. In [10], the negotiation protocol consists in selecting either an implementation or a QoS profile that fulfills the specification whereas in [15], negotiation is statically defined and consists in trying alternate services with lower quality levels. Our approach differs as we view the negotiation as a generic process of consulting and adapting responsible components or contracts to restore the validity of contracts.

6. Conclusion and Future Work

In this paper, we described mechanisms to support contract negotiation on hierarchical software components. These mechanisms were designed for ConFract, a con-
tracting system for the Fractal hierarchical component model. *ConFrac*t allows one to define contracts on both interface bindings and components. In our negotiation model, the protocol is partly inspired from the extended *Contract-Net Protocol*, frequently used in multi-agent systems. A negotiation initiator thus consults clearly identifiable responsible components in order to restore the validity of violated contracts. Given components responsibilities, several negotiation policies may be developed and we have presented in this paper a concession-based negotiation policy. This policy takes advantage of beneficiaries ability in such a way that the negotiation initiator requests for concessions from beneficiaries and asks them to rely on an under constrained provision or to reconfigure themselves. Moreover, this policy is well-suited to the *CCL-J* language, which is based on executable assertions and is currently used in *ConFrac*t.

Nevertheless, our model does not ensure the termination of the negotiation process and does not verify that modifications made on contracts or components to restore the validity of a given provision will not challenge other valid provisions. However, these problems are still open and they arise commonly in approaches which aim at adapting components in the more general way.

As future work, we plan to extend and generalize this negotiation model by developing other negotiation policies and designing further recovery actions to address contracts violations. We are particularly interested in proposing an effort-based policy in which the negotiation initiator will now turn towards guarantor and contributors to ask them to make an effort. To develop this policy and to propagate the negotiation down the hierarchy, our model needs to rely on further information expressed or deduced from provisions and to reason about the quality attributes of a composition of components given the quality attributes of involved components. With such compositional reasoning, it would also be possible to negotiate by making trade-offs between negotiation issues and to manage the reconfigurations of components by changing or updating their content.

**References**


