OpenKnowledge

FP6-027253

Plug-in Components

Coordinator: Paolo Besana[1],

with contributions from
David Dupplaw[2], Adrian Perreau de Pinnick[3]

[1] University of Edinburgh
[2] University of Southampton
[3]IIIA (CSIC) Barcelona

Report Version: final
Report Preparation Date: 21.12.2007
Classification: deliverable D1.3
Contract Start Date: 1.1.2006 Duration: 36 months
Project Co-ordinator: University of Edinburgh (David Robertson)
Partners: IIIA(CSIC) Barcelona
Vrije Universiteit Amsterdam
University of Edinburgh
KMI, Open University
University of Southampton
University of Trento

Abstract

Constraints in LCC protocol are preconditions on message sending, and postconditions (possibly changing the peer internal state) on message receiving.

The constraints do not specify how they must be solved: peers solve constraints through plug-in components, called OKCs. An OKC exposes methods that are mapped to constraints in protocols: the activity of choosing to what interaction subscribe - with the aim of acting in the interaction in a particular role - requires the comparison between the constraints of the roles in an interaction model and the methods in its repository of OKCs. The comparison process results in an evaluation of the similarity between the protocol and the peers components, and if possible in a set of adaptors between the constraints and the methods. Similarly, adaptors are used by the OKC to access methods exposed by the peer (for subscribing to other interactions, for example).

1 Introduction

The main objective of the OpenKnowledge project is to study an open, distributed system for sharing knowledge about processes. The development of the kernel is an attempt of providing a working framework able to answer these requirements.

The peers in the OpenKnowledge system share the descriptions of processes in the form of *Interaction Models*: an Interaction Model is defined in Lightweight Coordination Calculus [2], and contains a set of clauses for the various roles that the peers can perform during an interaction. The peers exchange messages, and a message can have constraints upon sending and reception.

The constraints are used to bridge between the interaction and the peers' knowledge, either querying for values or changing the local state of the peer. The success or the failure of a constraint is used to choose among different possible paths in the interaction. In the fragment of interaction in figure 1, the peer's knowledge is first accessed to obtain what he wants, and then to verify if the peer accepts the offers. Depending on the result of the last constraint, the interaction can take two different paths, one in which an acceptance message is sent, and another where a rejection is sent.

We need a technique to link constraints in Interaction Models to peers' knowledge. Moreover, peers should be able to take part in many different interactions, some of which may not have been considered when the peer was developed. Finally, as the system is open, it is impossible to assume that knowledge is represented uniformly in interaction models and in peers.

2 Plug-in architecture

The first engineering decision taken is to link constraints to *methods* in the peer code body. When a constraint needs to be solved, a method in the peer

$$
\begin{array}{l}
a(buyer, B) :: \\
\quad ask(X) \Rightarrow a(seller, S) \leftarrow want(X) \\
\quad then \\
\quad offer(X, P) \Leftarrow a(seller, S) \\
\quad then \\
\quad \left(\begin{array}{l} accept \Rightarrow a(seller, S) \leftarrow affordable(X, P) \\ or \\ reject \Rightarrow a(seller, S) \end{array} \right)
\end{array}
$$

Figure 1: The buyer clause from a simple purchase interaction model

is called. A constraint in LCC is a predicate, where some of its arguments may be instantiated and others still to unify. The task of the method is to instantiate arguments if possible, and to return true or false to indicate whether the constraint was successfully satisfied. The method solving the constraint $want(X)$ will instantiate X with the product desired by the peer, while the method solving $affordable(X, P)$ will either return true or false depending on some local utility function. Constraints are solved by methods defined as:

$boolean\ methodName(Argument\ A_1, ..., Argument\ A_n)$

The link between the constraint and the method is provided by *adaptors*, as we will see in Section 4.

The second engineering decision taken is to use a *plug-in* architecture for the methods: initially peers do not have the methods in their code used for solving constraints in interactions. The methods are in components, called OKCs (OpenKnowledge Components), that can be added to the peer repository of components. An OKC is a Java Archive (*jar*) file containing a class extending the superclass `OKCFacadeImpl`, that provides the methods for solving constraints. The jar file is used to contain auxiliary resources used by the methods. Figure 2 shows the UML class diagram of an OKCFacade class.

3 Lifecycle of an OKC

3.1 Developing and Obtaining OKCs

The creation of the OKC jar file for a specific role in an Interaction Model can be done through the OpenKnowledge GUI: once an interaction has been chosen and a role has been selected, by clicking on "Create New OKC for Role" it is possible to select the files to include in the OKC. The application automatically generates the jar, with the correct `manifest` and `okcinfo.xml` files. Another useful tool for generating OKCs is the WSDL2OKC application for accessing web services: it receives the URL of a WSDL file and it automatically generates the OKC able to call the web service. The operations listed in the WSDL file are converted into methods in the OKC.

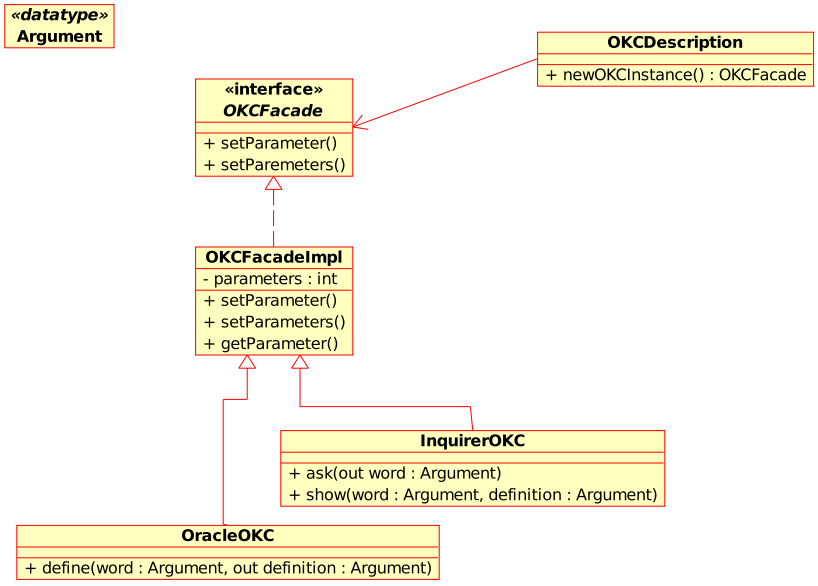

Figure 2: UML Class diagram for two example OKCs

Once an OKCs has been developed, it can either be kept private and installed only on a restricted number of machines, or it can be published and shared on the Discovery Service. A published OKC is described by a set of keywords that other peers can use to find them querying the Discovery Service.

The `.jar` files are stored on the peer's hard disk, and the peer keeps a record of them as `OKCDescriptions` in its local OKC repository.

3.2 Subscription to an Interaction Model

When a peer needs to perform a task it asks the Discovery Service [1] for a list of Interaction Models matching the description of the task. Then, for each received Interaction Model, the peer compares the methods in its OKCs with the constraints in the entry role it is interested in. If the peer finds an Interaction Model whose constraints (in the role the peer needs to perform) are covered by the methods in its OKCs, then the peer can subscribe to that Interaction Model in the Discovery Service. The subscription is handled by a subscription negotiator and can be interpreted as an intention to participate in the interaction. The subscription, through a subscription adaptor, binds the Interaction Model to a set of methods in the OKCs in peer. A subscription can endure for only a single interaction run or for many, possibly unlimited, interaction runs: a buyer will likely subscribe to run a purchase interaction once, while a vendor may want to keep selling its products or services.

An additional functionality for subscription, not yet included in system, is to allow an interaction to specify the OKCs it requires, and let the peer download

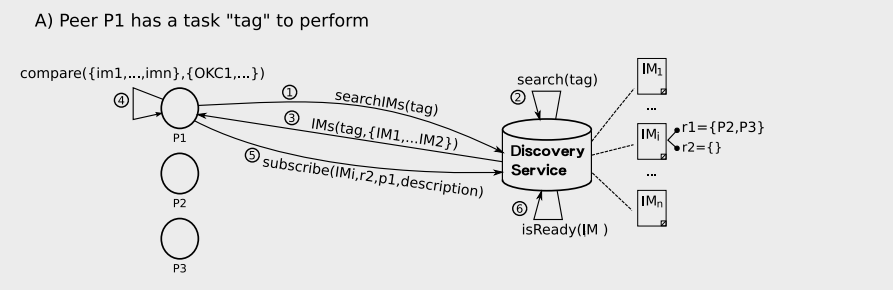

Figure 3: Exchange of messages between peers and DS for subscription

them (given permission by the user and some threshold on the trust level for the IMs and OKCs) in order to run the interaction.

3.3 Interaction bootstrap

When all the roles in the Interaction Model have subscriptions, the Discovery Service selects a random peer as a coordinator. The coordinator bootstraps and runs the interaction. The bootstrap involves first asking the peers who they want to interact with, among all the peers that have subscribed to the various roles, then creating a team of mutually compatible peers and finally - if possible - asking the selected group of peers to commit to the interaction. Figure 4 shows the exchange of messages between the peers, the coordinator and the Discovery Service.

For a peer, committing to an interaction, implies the creation of an `InteractionRunContext`, that receives the `SubscriptionAdaptor` from the `SubscriptionNegotiator` as in Figure 5.

3.4 Interaction run

The run of the interaction is handled by the coordinator and the `InteractionRunContext` of the involved peers. The coordinator peer runs the interaction locally: the messages are exchanged between local proxies of the peers. However, when the coordinator encounters a constraint in a role clause, it sends the message `solveConstraintMessage` to the `InteractionRunContext` in the peer performing the role. The message contains the constraint to be solved. The `InteractionRunContext` asks the `SubscriptionAdaptor` the corresponding method - found during the comparison at subscription time. The OKCs are instantiated lazily: if the OKC that contains the method corresponding to the constraint has not been instantiated yet within the context of the interaction, the class is instantiated, and stored in the context. If the instance exists in the context, the corresponding method is called dynamically. The method will use the adaptor to access the elements of the arguments. The peer then sends back the message

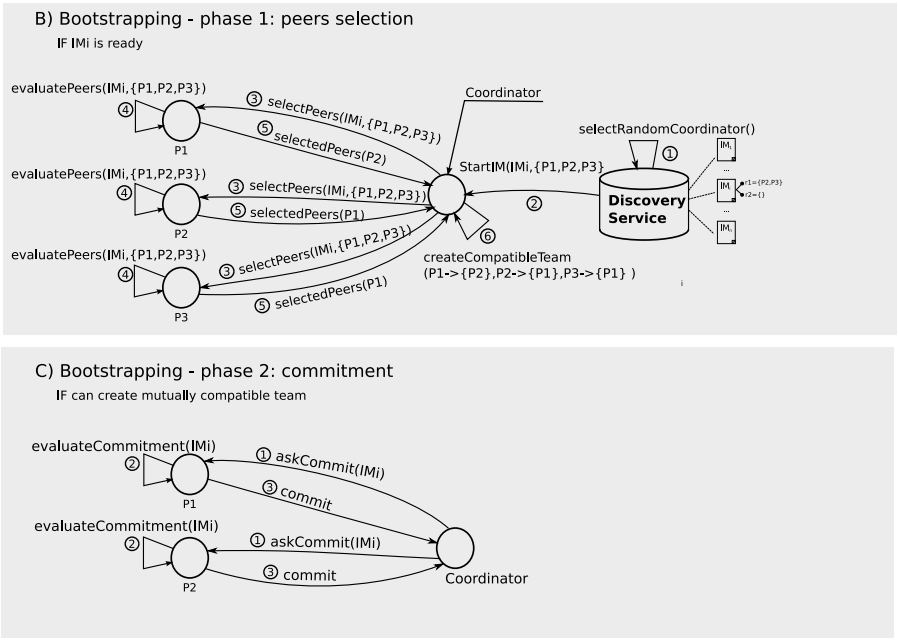

Figure 4: Bootstrap of interaction: exchange of messages for the selection of peers and commitment

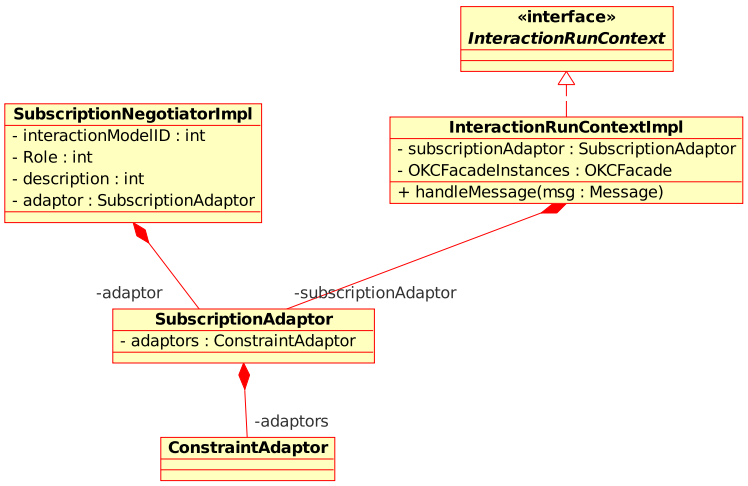

Figure 5: UML class diagram of Subscription/ContextRun

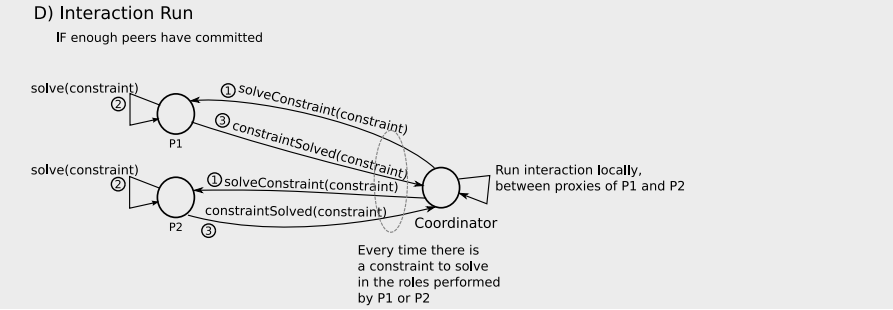

Figure 6: Interaction Run: exchange of messages between the coordinator and the peers

`SolveConstraintResponseMessage` to the coordinator with the updated values of variables and the boolean result obtained from satisfying the constraint.

4 Mapping Constraints to Methods

The matcher, described in [3], allows the OKCs and the Interaction Models to be decoupled. The peer compares the constraints in the roles in which it is interested with the methods in its OKCs and creates a set of adaptors that maps the constraint in the roles to similar methods. In order to match constraints and methods they both need to be semantically annotated.

4.0.1 Semantic Mark-up of Methods

The exchanged messages can contain complex structures. The structures can be trees or lists. The structure of the arguments is defined in the semantic annotation of the method, written using Java 5 annotations:

```
@MethodSemantic(
language=tag,
args={“product(brand, name, cost(currency, value))”,
  “buyer(name, surname, address(street, postcode, city))”}
)
public bool registerPurchase(Argument P, Argument B) {...}
```

The code inside the method can access the elements in the structure by path (similarily to XPath):

```
System.out.println(P.getValue("/product[0]/cost[0]/value[0]”)+
  ” “ +
  P.getValue("/product[0]/cost[0]/currency[0]”))
```

The nodes in the path are coupled with an index, because there might be more than one node of the same ontological type at the same depth. For example, a parameter that contains a relationship can be a expressed as tree with two identical children:

```
@MethodSemantic(
language=tag,
args={’friends(person,person)’})
public boolean add(Argument F){
...
System.out.println(F.getValue(“friends[0]/person[0]”) + “ knows ”
+F.getValue(“friends[0]/person[1]”));
...
}
```

The elements of the structure are reached independently of how they are kept in the exchanged messages: the adaptor between the constraint and the method maps the elements in the arguments of the constraint to the elements in the arguments of the method.

4.0.2 Lists

We have two possibilities: one is to only allow access to the lists through LCC operators and recursion, the other is to use the indexes of the root elements:

```
@MethodSemantic (..., args={"[move(from,to,vehicle)]"})
```

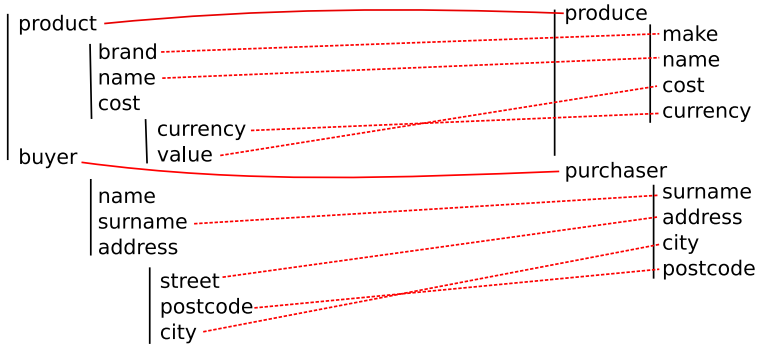

Figure 7: Adaptor between `register(...)` constraint and `registerPurchase(...)` method

represents an argument that contains a list. To access the elements in the list, the index of the root changes.

```
public boolean do(Arg A){
 System.out.println(A.getValue("/move[2]/from[0]");
}
```

4.1 Adaptors

For example, the constraint in the following snippet of a protocol:

```
register(P,B) <- bought(P,B) <= a(buyer, ID)
```

where the constraint is defined as:

```
register(produce(make,name,cost,currency),
  purchaser(surname, address, city, postcode))
```

will be mapped to the method in the OKC seen in the previous section with the adaptor in Figure 7.

Allowing the code inside the method to access the elements without knowing how they are actually structured in the message, decoupling de facto the protocol from the components.

5 Access to the peer state

Some of the constraints are *functional*: they expect that the method in the component, given a set of input arguments, will always unify the non instantiated arguments with the same values, or will succeed or fail, independently of

the peer that has downloaded and executed the component. For example, the constraint `sort(List,SortedList)` for sorting a list of elements should always unify `SortedList` with the ordered version of `List`, even though different peers may have OKCs that implement different algorithms for sorting it.

Other components work as bridge between the interaction model and the peer local knowledge, and will unify non instantiated variables with values that depend on the peer in which the OKC is running. For example, a constraint `price(Product, Price)` expects that the corresponding method in the OKC unifies the variable `Price` with the price assigned to `Product` by the peer, possibly accessing the database local to the peer: different peers may have different prices for the same product. Similarly, in the emergency response scenario, the firefighters are queried about their location when they need to satisfy the constraint `at(Position)`: every firefighter will have a different location, and will set `Position` with their local position.

Moreover, the same peer can be involved in many interactions simultaneously, and the peer local knowledge (or state) is changed by one interaction and read in another. For example, a peer selling products will have the total amount of available products reduced after each successful selling interaction. For peer local knowledge, or internal state, we mean any element of information that can persist over different interactions (and possibly be altered by them): for example the database of products just described.

At the moment the OKCs are given the references, at instantiation time, of the objects they need to use through the `setParameter(..)` and `getParameter(...)` methods: for example, the peer that runs the simulator in the emergency response scenario is given the reference to the Prolog client that allows it to query the Prolog environment in which the simulator is actually run.

In the planned development, the peer will expose a set of methods that allow the OKCs to access its internal state, and the OKCs will expose the list of methods in the peer they need to use. The same matching process that takes place between the constraint and the methods in the OKC will take place between the OKC and the peer exposed method, generating a set of adaptors allowing the methods in the OKC to access the peer local knowledge.

The class implementing the `OKCFacade` interface is annotated (via Java 5.0 annotations) with the semantic descriptions of the peer's methods it needs to use. When a peer downloads an OKC, the methods required by it are matched against the methods exposed by the peer: if the matching is good enough (there might be OKCs not compatible with a particular implementation of a peer), then the result of comparison is an *adaptor,* similar to those between OKC methods and constraints, that allow the OKC methods to access the elements in the peer's methods using its internal terminology.

The peer's ontology is considered as local knowledge. The actual implementation of the ontology handler is up to the peer developers, but the OKCs - if they need it - can access it through the same procedure of calling a set of exposed methods.

5.1 Deferred Interactions

An interesting possibility is for the OKC method to ask the peer to run another interaction in order to collect further information, or to obtain something asynchronously. Starting another interaction can happen in two ways: implicitly or explicitly.

In the *implicit* case, the peer is asked for further information, and it is up to the peer to decide how to collect the information: it may already have it, and answer directly, or it may decide to proceed with a new interaction to obtain the information.

In the *explicit* case, the method can either tell the ID of the interaction to perform to the peer, together with additional filter information for the subscription and the bootstrap, or be more generic and tell the peer to search an interaction using a set of keywords, and follow the standard subscription procedure.

References

[1] S. Kotoulas and R Siebes. Deliverable 2.2: Adaptive routing in structured peer-to-peer overlays. Technical report, OpenKnowledge.

[2] D Robertson. A lightweight coordination calculus for agent systems. In *Declarative Agent Languages and Technologies*, pages 183–197, 2004.

[3] P Shvaiko, F. Giunchiglia, M. Yatskevitch, J. Pane, and P. Besana. Deliverable 3.6: Implementation of the ontology matching component. Technical report, OpenKnowledge, 2007.