OpenKnowledge

FP6-027253

# OpenKnowledge Manual

David Dupplaw[1], Paolo Besana[2], Dave Robertson[2]

[1] IAM Group, School of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK.

[2] Informatics, University of Edinburgh, Edinburgh, EH8 9LE, UK.

Report Version: final
Report Preparation Date: 24th January 2008
Classification: deliverable 9.2
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

# Contents

**1 Introduction** **5**

**2 The OpenKnowledge System in a Nutshell** **7**

**3 Writing OpenKnowledge Interactions in LCC** **11**
3.1 A Basic Example . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.2 LCC Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2.1 Variables, Constants, Terms, IDs and Roles . . . . . . . . 14
3.2.2 Messages . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.3 Constraints . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.4 Comments . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2.5 Sequence and Choice . . . . . . . . . . . . . . . . . . . . 17
3.3 Design Patterns . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**4 How the OpenKnowledge Kernel Operates** **21**
4.1 The Subscribe-Bootstrap-Run Cycle . . . . . . . . . . . . . . . . . 21
4.1.1 Subscription to an Interaction Model . . . . . . . . . . . . 21
4.1.2 Bootstrapping an Interaction . . . . . . . . . . . . . . . . 22
4.1.3 Running an Interaction . . . . . . . . . . . . . . . . . . . 22
4.2 Mapping Constraints to Methods . . . . . . . . . . . . . . . . . . 24
4.2.1 Adaptors . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.3 Access to Peer State . . . . . . . . . . . . . . . . . . . . . . . . 26

**5 Programming for the OpenKnowledge Kernel** **29**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2 Creating Interaction Models . . . . . . . . . . . . . . . . . . . . 29
5.2.1 Publishing Interaction Models . . . . . . . . . . . . . . . . 35
5.3 Creating Components . . . . . . . . . . . . . . . . . . . . . . . . 38
5.3.1 Programming New Components . . . . . . . . . . . . . . . 38
5.3.2 Creating Component JAR Files . . . . . . . . . . . . . . . 39
5.3.3 Loading in Local Components . . . . . . . . . . . . . . . . 40
5.3.4 Publishing Components . . . . . . . . . . . . . . . . . . . 40
5.3.5 Providing Alternate User Interfaces . . . . . . . . . . . . 41

# Chapter 1

# Introduction

The purpose of this manual is to introduce you to programming for the OpenKnowledge kernel. There are several ways in which you might be interested in OpenKnowledge programming:

- You are not a programmer but you want to know what it would take to write programs for OpenKnowledge. Your best option is to read Chapter 2 and quickly skim through the remainder of the manual.
- You want to describe interactions in OpenKnowledge but you are not interested in writing specialised components to do new things as part of those interactions. Read up to the end of Chapter 3 and skim the rest.
- You are intending to take a good look at the open source code for the OpenKnowledge system and want an introduction to the main operational mechanism before you come to grips with the code. Read up to the end of Chapter 4.
- You want to write your own components to fire off applications within an interaction so you need to know how those are connected before looking at examples on `www.openk.org`. Read the whole manual, perhaps skimming over Chapter 4.

Much of the discussion in this manual relates to the OpenKnowledge interaction modelling language (LCC). Several interpreters for LCC have been written in different languages (Prolog, Lisp, Java) but the OpenKnowledge kernel (which you can download from `www.openk.org`) is the first attempt to produce a kernel system that combines LCC with peer to peer query routing, ontology matching and visualisation. It has been released as Java open source, so you can develop your own system based on it or contribute to our version.

Many of the ideas contained in this manual are covered in the (basic and advanced) video tutorials available from the tutorial area of the `www.openk.org`). You may find those helpful as an alternative presentation of the same concepts.

This manual will change as our work on the OpenKnowledge project proceeds. If you have suggestions for improvements to this manual then please mail them to Dave Dupplaw (`dpd@ecs.soton.ac.uk`).

# Chapter 2

# The OpenKnowledge System in a Nutshell

Suppose you want to get something complex done using components (such as Web services) that are available on the Internet. You could write your own Web service, that calls out to those other components, and host that on your own system but that will only work for you alone. What if you think others might benefit from coordinating in a similar way, or if you want to avoid always having your system perform the coordination? That's where OpenKnowledge comes in. It provides you with a compact language for describing coordination and, if you wish to do so, a means of sharing coordination with others.

The "world view" taken by OpenKnowledge is illustrated in the picture below, where the different coloured arrows represent people (or automated systems) participating in different interactions. Each interaction is coordinated by a model of the interaction, discussed later. An individual gains knowledge of how to interact with other individuals through interacting with those who he or she already knows. For example, in the illustration in Figure 2.1 the individual on the left might initially know about only two other individuals (the two interacting via the light green arrows) but those two individuals know about other interactions (in dark blue and in lime green) so can communicate them to the individual on the left.

Interaction models are can be shared and used in many different ways but the standard way to use them is by downloading the OpenKnowledge kernel system from `www.openk.org`. The kernel is a compact program that automatically finds interaction models that you might want to use; allows you to subscribe to interactions that interest you; and interprets the interaction models in which you actually become involved. In the illustration of Figure 2.2, the red dots are copies of the kernel system (loaded from the supplier) being run on individual peers.

Although our interaction models are portable and could be used by different systems, there is a social advantage in having many peers running the same

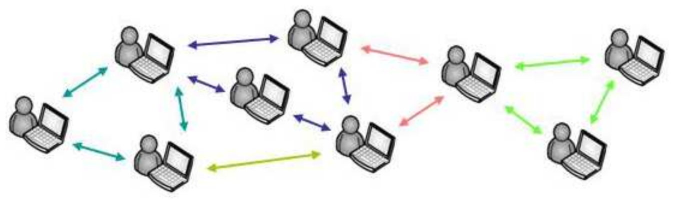


Figure 2.1: People share knowledge through shared interactions

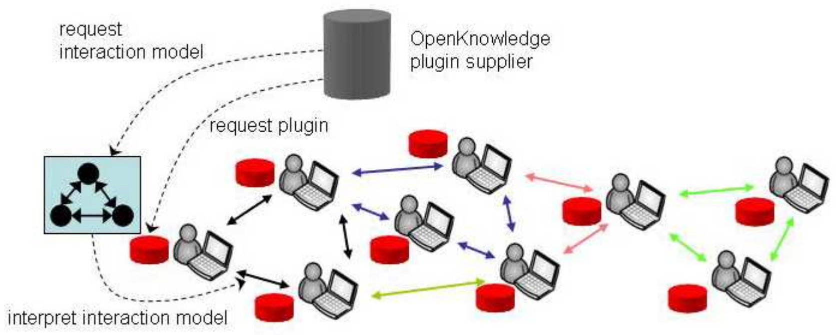


Figure 2.2: Interactions are "injected" into per groups from service areas, like www.openk.org

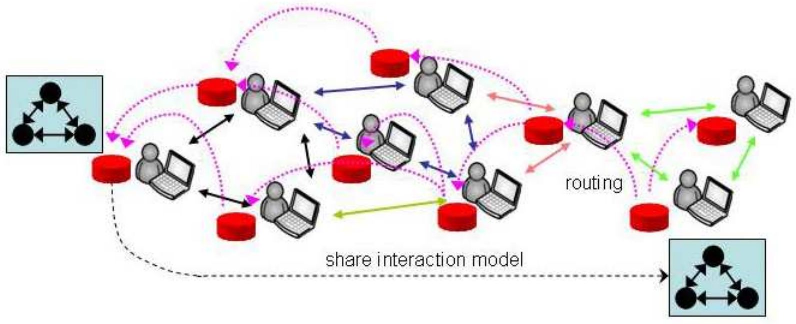


Figure 2.3: Interactions are shared by routing through the peer network.

OpenKnowledge kernel. The social advantage comes from query routing, which works roughly as illustrated in Figure 2.3. Suppose that the peer at bottom right of the picture wants to undertake an interaction but does not have an appropriate interaction model. That peer would describe the sort of interaction he or she is seeking, using a sequence of keywords (in a similar way to the way you search for Web pages in traditional Web browsers). This query then is routed through the peer network until matching interaction models are found (in our picture the interaction model in black matches the query) and are relayed back to the peer. When the peer receives the interaction model it receives not only the interaction but, through it, may also access other peers in the network with which it may not previously have interacted. In this way, sharing interaction models extends and reinforces social networks.

If you want simply to use interaction models then you do not need to understand any technical detail of the underlying system because using an interaction model is analogous to using a program - if the interaction model is well crafted then it will be easy for an appropriate group of people to use without them knowing how it is built. You may, however, want to write your own interaction models or adapt those you find on the network. This is the topic of the next chapter.

# Chapter 3

# Writing OpenKnowledge Interactions in LCC

The language used in OpenKnowledge is the Lightweight Coordination Calculus (LCC). This chapter explains how to write interaction models in LCC, which you can then use and share with others. There is more to constructing interaction models than just the specification of the interaction itself; there is also the issue of connecting interactions to services and other computational components. We get to this in Chapter 5 but our task in the current section is to introduce you simply to writing interactions. We begin with a basic example (Section 3.1) then describe LCC syntax in detail (Section 3.2); then describe some essential design patterns (Section 3.3).

## 3.1 A Basic Example

The diagram in Figure 3.1 shows an interaction between three peers: $p1$, $p2$ and $p3$. Each peer knows different things:

- $p1$ knows that queries asking about $p(Y)$ can be sent to $p2$ and that queries asking about $q(Z)$ can be sent to $p3$. We write this as $query\_from(p(Y), p2)$ and $query\_from(q(Z), p3)$.
- $p2$ knows that $p(a)$ is true. We write this as $know(p2, p(a))$.
- $p3$ knows that $q(b)$ is true. We write this as $know(p3, q(b))$.

The interaction we require is depicted by the numbered messages in the diagram:

- $p1$ sends a message $ask(p(Y))$ to $p2$.
- $p2$ sends a message $ask(p(a))$ to $p1$.

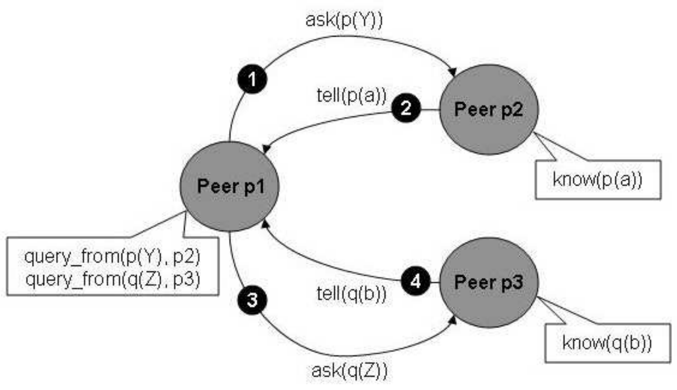


Figure 3.1: Basic interaction example

- $p1$ sends a message $ask(q(Z))$ to $p3$.
- $p3$ sends a message $ask(q(b))$ to $p1$.

Let us first define an interaction model that does exactly the message passing defined above. There are two roles that agents take in this model: the role of a requester (which asks for information) and the role of an informer (which supplies information). We define a LCC clause for each role as shown below. For the requester ($p1$) we have simply given the sequence of four messages corresponding to those above. Then we have defined a clause for the role of informer that defines the behaviour expected of $p2$ and $p3$.

$$\begin{array}{l}
\mathbf{a}(\text{requester}, A) :: \\
\quad ask(X1) \Rightarrow \mathbf{a}(\text{informer}, p2) \leftarrow query\_from(X1, p2)\ then \\
\quad tell(X1) \Leftarrow \mathbf{a}(\text{informer}, p2)\ then \\
\quad ask(X2) \Rightarrow \mathbf{a}(\text{informer}, p3) \leftarrow query\_from(X2, p3)\ then \\
\quad tell(X2) \Leftarrow \mathbf{a}(\text{informer}, p3) \\
\\
\mathbf{a}(\text{informer}, B) :: \\
\quad ask(X) \Leftarrow \mathbf{a}(\text{requester}, B)\ then \\
\quad tell(X) \Rightarrow \mathbf{a}(\text{requester}, B) \leftarrow know(X)
\end{array} \tag{3.1}$$

The LCC definition above covers the example but suppose we want a more general type of requester that takes a list, $L$, of the form $[q(Query, Peer), \ldots]$, where $Query$ is the query we want to make and $Peer$ is an identifier for the peer to which we want to send the query. We want the requester to send an $ask(Query)$ message to the appropriate $Peer$ for each query and receive a $tell(Query)$ reply each time. A standard way to do this is by giving $L$ as a parameter to the requester role (so it becomes $requester(L)$) and making the

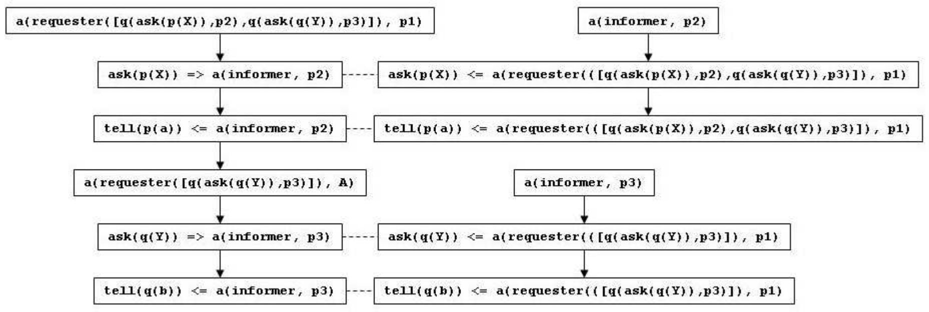

a(requester([q(ask(p(X)),p2),q(ask(q(Y)),p3)]), p1)
a(informer, p2)
ask(p(X)) => a(informer, p2)
ask(p(X)) <= a(requester(([q(ask(p(X)),p2),q(ask(q(Y)),p3)]), p1)
tell(p(a)) <= a(informer, p2)
tell(p(a)) <= a(requester(([q(ask(p(X)),p2),q(ask(q(Y)),p3)]), p1)
a(requester([q(ask(q(Y)),p3)]), A)
a(informer, p3)
ask(q(Y)) => a(informer, p3)
ask(q(Y)) <= a(requester(([q(ask(q(Y)),p3)]), p1)
tell(q(b)) <= a(informer, p3)
tell(q(b)) <= a(requester(([q(ask(q(Y)),p3)]), p1)

Figure 3.2: Event sequences for the example

definition of this role recursive, taking the first element of $L$ and then applying the same definition to the remainder of the list, $Lr$, as shown below.

$$\begin{array}{l}
a(requester(L), A) :: \\
\quad (ask(Query) \ \Rightarrow \ a(informer, Peer) \leftarrow L = [q(Query, Peer)|Lr] \ then \\
\quad tell(Query) \ \Leftarrow \ a(informer, Peer) \ then \\
\quad a(requester(Lr), A)) \\
\quad or \\
\quad null \leftarrow L = [] \\
\\
a(informer, B) :: \\
\quad ask(X) \ \Leftarrow \ a(requester(_), B) \ then \\
\quad tell(X) \ \Rightarrow \ a(requester(_), B) \leftarrow know(X)
\end{array} \tag{3.2}$$

If we were to run the interaction model shown above, starting with the role of requester for the list of queries $[q(ask(p(X)), p2), q(ask(q(Y)), p3)]$, then we get the message sequences shown in Figure 3.2. On the left is the sequence for $a(requester([q(ask(p(X)), p2), q(ask(q(Y)), p3)]), p1)$. On the right are the sequences for $a(informer, p2)$ and $a(informer, p3)$ which are the roles undertaken by $p2$ and $p3$ in response to $p1$. The dashed lines indicate synchronisation via message passing between peers.

In our earlier example (definition 1 above) we made some of the message passing events contingent on constraints. For example sending the message $ask(X1) \ \Rightarrow \ \mathbf{a}(\text{informer}, p2)$ was contingent on satisfying the constraint $query\_from(X1, p2)$. These constraints are satisfied by connecting them to methods for computing the constraint. Although some basic methods (such as for basic forms of visualisation) are pre-supplied by OpenKnowledge we expect that most methods will be specific to application domains and so will need to be written or re-used by interaction model developers. To make it possible to share these methods, we allow appropriately packaged methods to be shared, so that peers can accumulate

$$
\begin{array}{rcl}
Clause & := & Role :: Def \\
Role & := & \mathbf{a}(\text{Type}, Id) \\
Def & := & Role \mid Message \mid Def\ then\ Def \mid Def\ or\ Def \\
Message & := & M \Rightarrow Role \mid M \Rightarrow Role \leftarrow C \mid M \Leftarrow Role \mid C \leftarrow M \Leftarrow Role \\
C & := & Constant \mid P(Term, \ldots) \mid \neg C \mid C \wedge C \mid C \vee C \\
Type & := & Term \\
Id & := & Constant \mid Variable \\
M & := & Term \\
Term & := & Constant \mid Variable \mid P(Term, \ldots) \\
Constant & := & \text{lower case character sequence or number} \\
Variable & := & \text{upper case character sequence or number}
\end{array}
$$

Figure 3.3: LCC syntax

repositories of methods that they find useful. We call these OpenKnowledge Components (OKCs). A more detailed description of OKCs appears in Section 4.3 and Chapter 5.

## 3.2 LCC Syntax

Each LCC interaction model is defined by a set of clauses where each clause has the syntax shown in Figure 3.3. Each clause is a self contained definition of a role, with message passing being the only means of transferring information between roles. Message passing is also the only means of synchronisation between roles.

In the sections which follow we explain what each element of LCC syntax means from a programming point of view.

### 3.2.1 Variables, Constants, Terms, IDs and Roles

Variables must start with an upper case letter. The scope of a variable is local to a clause (in other words, if you use the same variable name in different clauses then these names refer to different variables). When it is unnecessary to give a specific name for a variable (because it is not used elsewhere in a clause) you can use an underscore (_) for the variable name. Constants must start with a lower case letter. Numbers also are constants. Terms are tree-structured - that is, they are either a constant or are of the form $F(A_1, \ldots, A_n)$ where $F$ is a non-numerical constant and each $A_i$ is a term. IDs are unique identifiers for peers which must be non-numerical constants. Roles are terms that describe the type of role played by a peer in a given interaction.

### 3.2.2 Messages

There are two types of messages:

**Incoming messages** : are of the form $Term \Leftarrow \mathbf{a}(\text{Role}, ID)$, where $Term$ is the content of the message. When using ASCII, the symbol $\Leftarrow$ is written using <=.

**Outgoing messages** : are of the form $Term \Rightarrow \mathbf{a}(\text{Role}, ID)$, where $Term$ is the content of the message. When using ASCII, the symbol $\Rightarrow$ is written using =>.

Constraints can be attached to both incoming and outgoing messages (see below).

### 3.2.3 Constraints

Constraints associate message passing events with conditions established by the peer.

$$Message \leftarrow constraint(Arg1, ...ArgN) \tag{3.3}$$

Constraints also may be associated with the special *null* event which represents an event that is not associated with a specific message. This frequently is used in recursive role definition where terminating the role depends on a parameter to the role, rather than a specific message passing event.

When using ASCII the constraint operator ($\leftarrow$) should be written using <-.

#### Visual Constraints

A constraint can have a mapping made available to it using the **visual**(, ) operator. The visual operator maps a constraint to a visual term. Visual terms provide an abstract representation for a particular type of user interaction.

$$\mathbf{visual}(\mathbf{constraint}(Arg1, ...ArgN), visualTerm(vArg1, ...vArgN)) \tag{3.4}$$

The OpenKnowledge kernel has a number of built-in visual term implementations, listed below:

$msg(M\langle, T\rangle)$ Display a message $M$ to the user, with the optional title $T$.

$text(\langle T, \rangle M)$ Display a large amount of text in $M$ to the user, with the optional title $T$.

$input(\langle Q, \rangle V)$ Ask the user to input some value into $V$, providing optional question text in $Q$.

**List Operations**

List operations are a common basis of the recursion techniques available when writing LCC. List operations make use of the bar | operation that delineates the head ($H$, first element) of a list from the rest of the list ($T$, the tail); that is, $L = [H|T]$.

In the case that $H$ has some value, you can append this value to the head of the list using the following constraint:

$$\ldots \leftarrow L = [H|T] \tag{3.5}$$

For example, if before the operation $H$ contained the value 5, and the list, $L$, contained $[6, 7, 8]$, after the operation the list would contain the values $[5, 6, 7, 8]$.

In the case that $H$ is not set, the following LCC will extract into $H$ the value of the head of the list.

$$\ldots \leftarrow L = [H|T] \tag{3.6}$$

Notice that $T$ is itself a list, so that the head of $T$ will be the second element in the list. This allows for recursion on $T$. If the list is empty, and no value for $H$ can be determined, the constraint will fail. For example, if before the operation $H$ was unset and the list, $L$, contained $[6, 7, 8]$, after the operation $H$ would have the value 6 and the $T$ would have the value $[7, 8]$.

To test whether a list is empty, use the following LCC:

$$\ldots \leftarrow L = [] \tag{3.7}$$

This constraint will fail if $L$ is not empty.

**Logical Operators**

Constraints can be connected by the logical operators *and* and *or*:

- $C_1$ *and* $C_2$ succeeds if both constraints $C_1$ and $C_2$ succeed, with $C_1$ being attempted first.
- $C_1$ *and* $C_2$ succeeds if one of the constraints $C_1$ and $C_2$ succeeds, with $C_1$ being attempted first.

### 3.2.4 Comments

To comment your LCC you can use the C-like comments $//\ldots$ or $/*\ldots*/$. The double-slash comment form will make the interpreter ignore the rest of the line. The slash-star comment form will ignore everything until the next star-slash. The following are valid comments:

```
// A valid single line comment
// Another single line comment

/*
   A valid
   multi-line
   comment
*/
```

Listing 3.1: Valid comment forms in LCC

### 3.2.5 Sequence and Choice

The basic operations used in LCC to determine the sequence of messages in a clause are sequence and choice, as defined below (where $E_1$ and $E_2$ are sequence expressions or message passing events):

**Sequence** : is written as $E_1$ *then* $E_2$. This sequence is completed if both $E_1$ or $E_2$ is completed, with $E_1$ being completed first.

**Choice** : is written as $E_1$ *or* $E_2$. This sequence is completed if either $E_1$ or $E_2$ is completed, with $E_1$ being attempted first. $E_2$ will only be attempted if $E_1$ fails. If $E_1$ succeeds then $E_2$ will not be attempted.

## 3.3 Design Patterns

Perhaps the easiest way to understand LCC programming is through design patterns. These are standard ways of structuring clauses that are used to obtain specific forms of interaction. The broad idea is similar to design patterns in more traditional languages but the good news for LCC is that you only need to know a small number of patterns, which you then combine to make more complex programs. The four key patterns are given below.

#### Pattern 1: Interaction

The simplest thing we can do with LCC is to specify a message being sent from one peer to another. To do this we decide the role ($r1$) being taken by the sender; then write $M \Rightarrow a(r2, Y) \leftarrow C$ to describe the message, $M$, being sent out to the recipient, $Y$, which is expected to receive it in role $r2$. The constraint $C1$ is used to determine whether this message can be sent by the sender, and it often is used also to determine values for any variables that appear in $M$. In the specification of the recipient's role we write $C2 \leftarrow M \Leftarrow a(r2, X)$ to describe the message, $M$, being received, with $C2$ giving a constraint that should hold as a consequence of receiving it.

$$\begin{array}{l} a(r1,X) :: \\ \quad \ldots \\ \quad M \;\Rightarrow\; a(r2,Y) \leftarrow C1 \\ \quad \ldots \\ \\ a(r2,Y) :: \\ \quad \ldots \\ \quad C2 \leftarrow M \;\Leftarrow\; a(r2,X) \\ \quad \ldots \end{array} \tag{3.8}$$

An example of using this pattern is an interaction that sends a message, $M$, to a recipient, $Y$, where the choice on $M$ is made by the constraint $message(M)$ and the choice of recipient is made by the constraint $recipient(Y)$. Acceptance of the message by the recipient is determined by the constraint $accept(M)$.

$$\begin{array}{l} a(sender,X) :: \\ \quad M \;\Rightarrow\; a(recipient,Y) \leftarrow message(M)\ and\ recipient(Y) \\ \\ a(recipient,Y) :: \\ \quad accept(M) \leftarrow M \;\Leftarrow\; a(sender,X) \end{array} \tag{3.9}$$

**Pattern 2: Sequence**

Usually we want to put an ordering on the sequence of events that can occur as part of a role. To do this we use the "then" operator to say that the earlier event, $E1$, comes before the later event, $E2$.

$$\begin{array}{l} a(r,X) :: \\ \quad \ldots \\ \quad E1\ then \\ \quad E2 \\ \quad \ldots \end{array} \tag{3.10}$$

An example that uses this pattern twice is when the recipient of the message returns a message to the sender, where $response(M1,M2)$ is a constraint determining the recipient's response message, $M2$, from the sender's message, $M1$.

$$\begin{array}{l} a(sender,X) :: \\ \quad M1 \;\Rightarrow\; a(recipient,Y) \leftarrow message(M1)andrecipient(Y)\ then \\ \quad accept(M2) \leftarrow M2 \;\Leftarrow\; a(recipient,Y) \\ \\ a(recipient,Y) :: \\ \quad accept(M1) \leftarrow M1 \;\Leftarrow\; a(sender,X)\ then \\ \quad M2 \;\Rightarrow\; a(sender,X) \leftarrow response(M1,M2) \end{array} \tag{3.11}$$

**Pattern 3: Choice**

We may want a peer taking some role, $r$, in an interaction to make a choice about the course of its interaction with other peers. This is done by writing $E1 \leftarrow C1 or E2 \leftarrow C2$ to say that the interaction described by $E1$ should be done under the conditions stipulated by constraint $C1$ or the interaction described by $E2$ should be done under the conditions stipulated by constraint $C2$. The choice we are making here is a committed choice, meaning that if $C1$ is satisfied then the alternative choice ($E2 \leftarrow C2$) will not be attempted.

$$\begin{array}{l} a(r, X) :: \\ \quad E1 \leftarrow C1 \\ \quad or \\ \quad E2 \leftarrow C2 \\ \quad \ldots \end{array} \tag{3.12}$$

An example of this pattern is when a buyer wants to send a message to a seller accepting some $Offer$ (received earlier in the definition of the buyer role) if it is acceptable or otherwise it sends a message to the seller rejecting that $Offer$ if it is unacceptable.

$$\begin{array}{l} a(buyer, X) :: \\ \quad \ldots \\ \quad accept(Offer) \;\Rightarrow\; a(seller, Y) \leftarrow acceptable(Offer) \\ \quad or \\ \quad reject(Offer) \;\Rightarrow\; a(seller, Y) \leftarrow unacceptable(Offer) \end{array} \tag{3.13}$$

Since LCC makes committed choices, we know in this example that if $acceptable(Offer)$ is satisfied then the second option (in which the peer attempts to satisfy $unacceptable(Offer)$) will not be attempted, so if testing unacceptability is not important then we might shorten this example to:

$$\begin{array}{l} a(buyer, X) :: \\ \quad \ldots \\ \quad accept(Offer) \;\Rightarrow\; a(seller, Y) \leftarrow acceptable(Offer) \\ \quad or \\ \quad reject(Offer) \;\Rightarrow\; a(seller, Y) \end{array} \tag{3.14}$$

**Pattern 4: Recursion**

Often we want an interaction to be controlled by some data structure, for example we might want to have a similar sub-interaction for each of the elements in a list (as in the basic example above). The pattern below describes this. In the pattern $r(A)$ is a role, $r$, with the data structure as its argument, $A$. Somewhere within the definition of the of the role appears a constraint, $R(A, Ar)$, that reduces $A$ to some "smaller" structure, $Ar$. Then the role recurses as $r(Ar)$. Normally there also is an alternative choice for when the data structure does not reduce any further but meets some test, $P(A)$, that it is has reached some terminating state.

$$\begin{array}{l} a(r(A), X) :: \\ \quad (\ldots\ R(A, Ar)\ \ldots \\ \quad a(r(Ar), X)) \\ \quad or \\ \quad (\ldots\ P(A)\ \ldots) \end{array} \tag{3.15}$$

One example of using this pattern is an interaction that sends as a message each element, $M$, from a list $[M1, ...]$ to peer $p2$ in role $r2$.

$$\begin{array}{l} a(r(A), X) :: \\ \quad (M\ \Rightarrow\ a(r2, p2) \leftarrow A = [M|Ar]\ then \\ \quad a(r(Ar), X)) \\ \quad or \\ \quad (null \leftarrow A = []) \end{array} \tag{3.16}$$

A second example is an interaction that sends $N$ messages to peer $p2$, each with the same content, $M$. Here, $N$ and $M$ are parameters to the role, $r$.

$$\begin{array}{l} a(r(N, M), X) :: \\ \quad (M\ \Rightarrow\ a(r2, p2) \leftarrow N > 0\ and\ N1\ is\ N - 1\ then \\ \quad a(r(N1, M), X)) \\ \quad or \\ \quad (null \leftarrow N =< 0) \end{array} \tag{3.17}$$

Many examples of LCC in use for specifying interactions can be found at `www.openk.org`, either as example interaction models, in the models area, or as case studies, in the publications area.

# Chapter 4

# How the OpenKnowledge Kernel Operates

This chapter explains how the OpenKnowledge works, from the point of view of its main functional elements: the subscribe-bootstrap-run cycle through which interactions are deployed (Section 4.1); the mapping between constraints and methods through which interaction models are connected to application components (Section 4.2); and the ways in which it is possible to access peer internal state (Section 4.3).

## 4.1 The Subscribe-Bootstrap-Run Cycle

Interactions in OpenKnowldge take place via a cycle of subscription (when peers say they want to take part in interactions); bootstrapping (to initiate a fully subscribed interaction) and running (to perform the bootstrapped interaction). Each of these is described below with the aid of four pictures (labelled A, B, C, D) in Figures 4.1, reffig:Bootstrap-of-interaction and 4.4. Taking these pictures together gives a picture of the whole cycle.

### 4.1.1 Subscription to an Interaction Model

The process of subscription is depicted in picture A of Figure 4.1. When a peer needs to perform a task it asks the Discovery Service for a list of Interaction Models matching the description of the task (steps 1, 2 and 3 in picture A). 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 (step 4 in picture A). 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 (step 5 in picture A). The subscription is handled by a subscription negotiator and can be interpreted as an intention to participate in the interaction. The subscription,

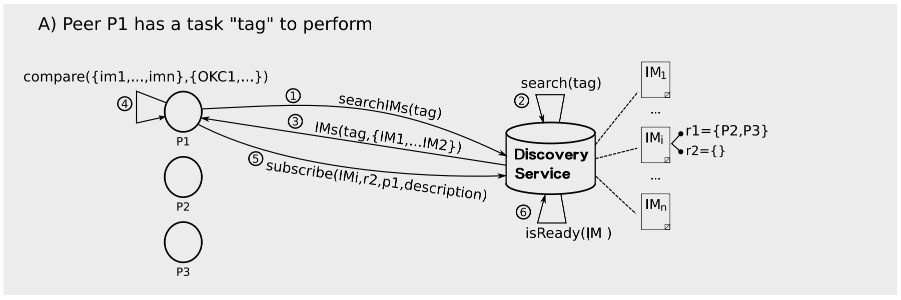


Figure 4.1: Exchange of messages between peers and discovery service for subscription

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.

### 4.1.2 Bootstrapping an Interaction

The process of bootstrapping is depicted in pictures B and C of Figure 4.2. When all the roles in the Interaction Model have subscriptions, the Discovery Service selects a random peer as a coordinator (steps 1 and 2 of picture B). The coordinator then 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 (steps 3, 4 and 5 of picture B), then creating a team of mutually compatible peers (step 6 of picture B) and finally - if possible - asking the selected group of peers to commit to the interaction (picture C).

For a peer, committing to an interaction, implies the creation of an `InteractionRunContext`, that receives the `SubscriptionAdaptor` from the `SubscriptionNegotiator` as in Figure 4.3.

### 4.1.3 Running an Interaction

The process of bootstrapping is depicted in picture D of Figure 4.4. This part of the cycle 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 (step 1 in picture D). The message contains the constraint to be solved. The `InteractionRunContext` asks the `SubscriptionAdaptor` the corresponding method - found during the comparison at subscription time (step 2

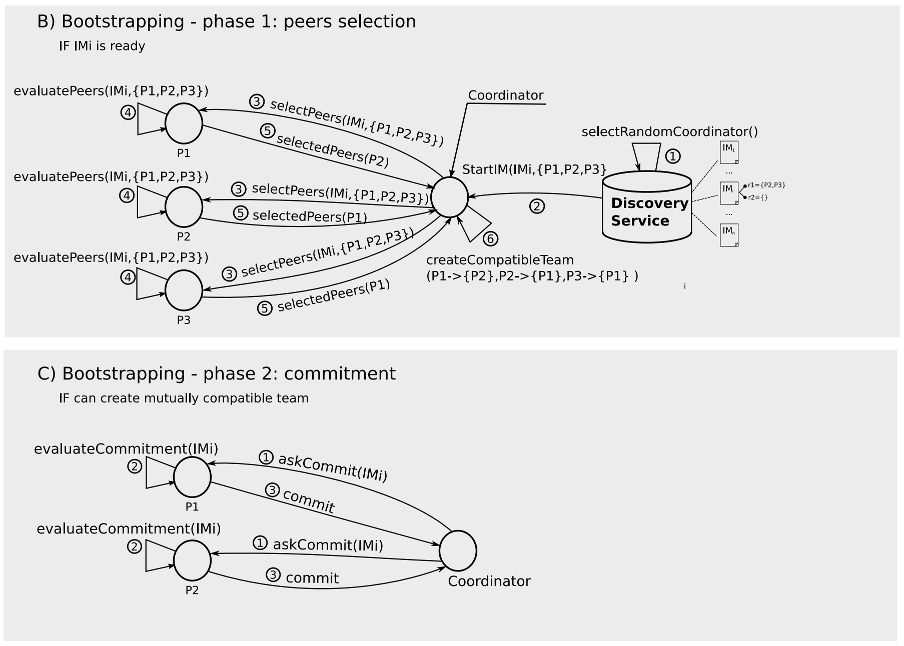

B) Bootstrapping - phase 1: peers selection
IF IMi is ready
evaluatePeers(IMi,{P1,P2,P3})
④
P1
③ selectPeers(IMi,{P1,P2,P3})
⑤ selectedPeers(P2)
Coordinator
selectRandomCoordinator()
①
StartIM(IMi,{P1,P2,P3}
evaluatePeers(IMi,{P1,P2,P3})
④
P2
③ selectPeers(IMi,{P1,P2,P3})
⑤ selectedPeers(P1)
②
Discovery Service
evaluatePeers(IMi,{P1,P2,P3})
④
P3
③ selectPeers(IMi,{P1,P2,P3})
⑤ selectedPeers(P1)
⑥
createCompatibleTeam
(P1->{P2},P2->{P1},P3->{P1} )
C) Bootstrapping - phase 2: commitment
IF can create mutually compatible team
evaluateCommitment(IMi)
②
P1
① askCommit(IMi)
③ commit
evaluateCommitment(IMi)
②
P2
① askCommit(IMi)
③ commit
Coordinator

Figure 4.2: Bootstrap of interaction: exchange of messages for the selection of peers and commitment

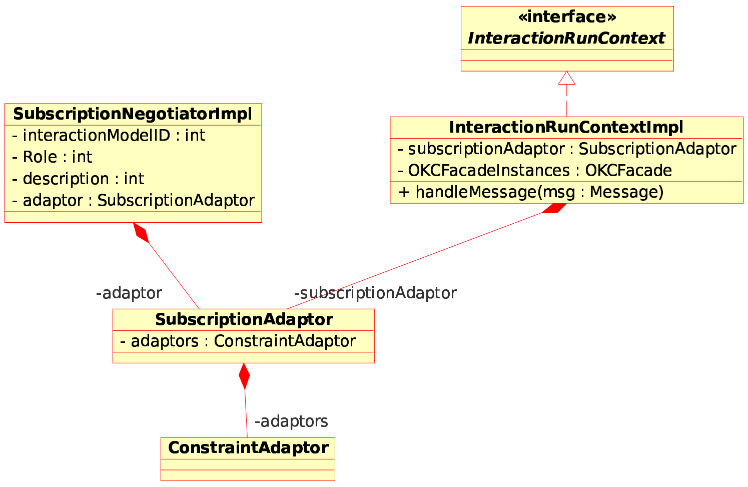

«interface»
InteractionRunContext
SubscriptionNegotiatorImpl
- interactionModelID : int
- Role : int
- description : int
- adaptor : SubscriptionAdaptor
InteractionRunContextImpl
- subscriptionAdaptor : SubscriptionAdaptor
- OKCFacadeInstances : OKCFacade
+ handleMessage(msg : Message)
-adaptor
-subscriptionAdaptor
SubscriptionAdaptor
- adaptors : ConstraintAdaptor
-adaptors
ConstraintAdaptor

Figure 4.3: UML class diagram of Subscription/ContextRun

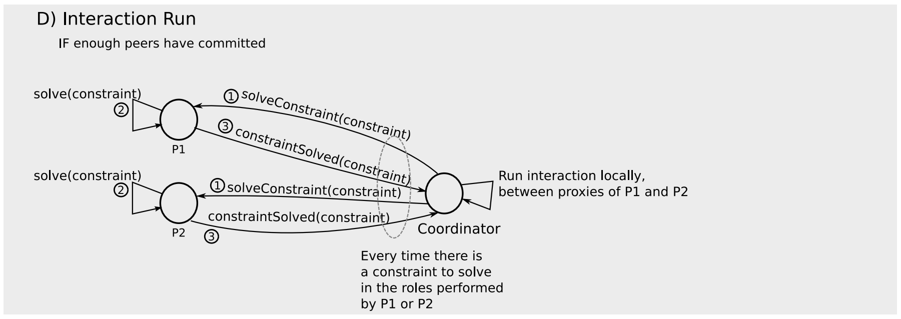

D) Interaction Run
IF enough peers have committed
solve(constraint)
② 
P1
① solveConstraint(constraint)
③ constraintSolved(constraint)
solve(constraint)
②
P2
① solveConstraint(constraint)
constraintSolved(constraint)
③
Coordinator
Run interaction locally, between proxies of P1 and P2
Every time there is a constraint to solve in the roles performed by P1 or P2

Figure 4.4: Interaction Run: exchange of messages between coordinator and peers

of picture D). 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 `SolveConstraintResponseMessage` to the coordinator with the updated values of variables and the boolean result obtained from satisfying the constraint (step 3 of picture D).

## 4.2 Mapping Constraints to Methods

The matcher (described in detail in OpenKnowledge Deliverable 3.6), 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.

### 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.

#### 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)]"})
```

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]");
}
```

### 4.2.1 Adaptors

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

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

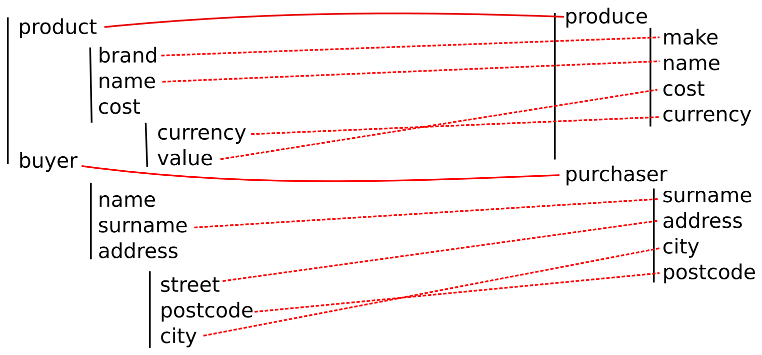


Figure 4.5: Adaptor between `register(...)` constraint and `registerPurchase(...)` method

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 4.5.

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.

## 4.3 Access to Peer State

Some interaction 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 whether 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 a 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. 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.

OKCs are given the references, at instantiation time, of the objects they need to use through the `setParameter(..)` and `getParameter(...)` methods. 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.

# Chapter 5

# Programming for the OpenKnowledge Kernel

## 5.1 Introduction

This chapter is intended for programmers who intend to create interaction models as well as creating components which implement roles in these models. To make the discussion self-contained we recap some material from earlier chapters but, if you are coming to OpenKnowledge for the first time, we recommend you at least skim through those background chapters before reading this one.

Let us assume you are sitting in front of a computer and you run the OpenKnowledge software; this will create what is known as a peer on the OpenKnowledge network. In most cases, you can consider your computer to be a single peer on the network and for now we shall assume this. Your peer stores bits of code that allow it to do interesting things when it is contacted. These bits of code are called OKCs (which stands for OpenKnowledge Components). To start with, this is all you should need to know, so we will start in section 5.2 by describing how to create interaction models in the default language for OpenKnowledge interaction models: LCC.

## 5.2 Creating Interaction Models

Chapter 3 describes the LCC language. Here we use it to begin a basic programming example that bridges interaction models to OpenKnowledge components (from Chapter 4). We will examine a simple form of interaction that most of us would be familiar with: greeting someone.

During the interaction that occurs when you greet someone, you would first say “Hello” and then you might expect a reply “Good Morning” back. In this simple interaction you would be the initial ‘greeter’ while the person you are greeting might be described as a ‘responder’. So let us write that interaction in sequence:

- **Greeter:** Say "Hello"
- **Responder:** Say "Good Morning"

We can be a little more rigorous in our description, as the responder will only reply "Good Morning" after they hear the initial greeting. So let us write this as a sequence:

- **Greeter:** Say "Hello"
- **Responder:** Hears "Hello"
- **Responder:** Say "Good Morning"
- **Greeter:** Hears "Good Morning"

Now that the interaction has been fully described, it is clear which of the people in the interaction are doing what and, very importantly, when they do it. The person assuming the 'greeter' role says "Hello" and hears the reply "Good Morning". The person assuming the 'responder' role hears "Hello" so replies "Good Morning".

Let's write out the sequence of actions for Johnny, who is going to be taking the 'greeter' role in our interaction:

```
greeter : Johnny
    Say Hello to Bob
    Hear Responder Bob say Good Morning
```

Bob is going to be the responder, so let's write out his actions, which are pretty much the exact opposite:

```
responder : Bob
    Hear Greeter Johnny say Hello
    Respond to Greeter Johnny with Good Morning
```

In LCC, the act of interacting is represented by the idea of sending and receiving messages. In our simple example, saying hello is the equivalent of sending the message *hello* to the responder. In the example above Bob is taking the responder role, so we would write the first interaction like this in LCC:

$$hello \Rightarrow \mathbf{a}(\text{responder}, Bob) \tag{5.1}$$

You can read this as 'Message hello is sent to responder Bob'.

In this particular interaction, where Bob is playing responder, he will be expecting to hear hello from a greeter called Johnny, so we write this like so:

$$hello \Leftarrow \mathbf{a}(\text{greeter}, Johnny) \tag{5.2}$$

You can read this as 'Message hello is received from greeter Johnny'.

Note: You can type $\Rightarrow$ as => and $\Leftarrow$ as <=.

So, if we were to extend this to the full interaction and allow any people to play it (not just Johnny and Bob), we would end up with this LCC interaction model:

$$\begin{array}{l}
\mathbf{a}(\text{greeter}, Person1) :: \\
\quad hello \Rightarrow \mathbf{a}(\text{responder}, Person2)\ then \\
\quad goodMorning \Leftarrow \mathbf{a}(\text{responder}, Person2) \\
\\
\mathbf{a}(\text{responder}, Person2) :: \\
\quad hello \Leftarrow \mathbf{a}(\text{greeter}, Person1)\ then \\
\quad goodMorning \Rightarrow \mathbf{a}(\text{greeter}, Person1)
\end{array} \tag{5.3}$$

Notice that we define the roles of greeter and responder above the sequence of interactions that they should perform. The start of a role is defined using the notation: $\mathbf{a}(\text{role}, name) :: \ldots$. Also notice that the sequence of events is defined using the *then* operator.

LCC has the concept of variables, or placeholders, into which values can be placed and then passed in the messages. Let's extend our simple interaction model so that our greeter first asks the name of our responder, then greets them. To achieve this, the responder must send their name through a message to the greeter in a variable. For example, if $N = Bob$, 'Hello $N$' will expand to 'Hello Bob'.

The interaction is now something like:

- **Greeter:** What's your name?
- **Responder:** My name is $N$
- **Greeter:** Hello $N$
- **Responder:** Good Morning

The interaction model expands because each role now describes an interaction that will determine the responder's name. The first two lines of each role describe the interaction to ask for the responder's name.

$$
\begin{array}{l}
\mathbf{a}(\text{greeter}, Person1) :: \\
\quad askName \ \Rightarrow \ \mathbf{a}(\text{responder}, Person2) \ then \\
\quad name(N) \ \Leftarrow \ \mathbf{a}(\text{responder}, Person2) \ then \\
\quad hello(N) \ \Rightarrow \ \mathbf{a}(\text{responder}, Person2) \ then \\
\quad goodMorning \ \Leftarrow \ \mathbf{a}(\text{responder}, Person2) \\
\\
\mathbf{a}(\text{responder}, Person2) :: \\
\quad askName \ \Leftarrow \ \mathbf{a}(\text{greeter}, Person1) \ then \\
\quad name(N) \ \Rightarrow \ \mathbf{a}(\text{greeter}, Person1) \leftarrow \mathbf{getMyName}(N) \ then \\
\quad hello(N) \ \Leftarrow \ \mathbf{a}(\text{greeter}, Person1) \ then \\
\quad goodMorning \ \Rightarrow \ \mathbf{a}(\text{greeter}, Person1)
\end{array}
\tag{5.4}
$$

The LCC above introduces two new aspects. The first is the idea of variables and showing how they can be passed through messages ($N$ in our message $hello(N)$). However, during the actual interaction, you do not want to say “Hello $N$”, you want to say “Hello Bob”. This means that $N$ has to be assigned to the value ‘Bob’. This is achieved by the last part of the line $\leftarrow$ **getMyName**($N$). We call this thing a constraint because we can only send the message $name(N)$ if we have a value for $N$ (Bob or whatever).

Let’s look at that line in more detail.

$$name(N) \ \Rightarrow \ \mathbf{a}(\text{greeter}, Person1) \leftarrow \mathbf{getMyName}(N) \tag{5.5}$$

This can be read: ‘Send $name(N)$ to greeter $Person1$ **only if** I can $getMyName(N)$’. We will discuss more about how $getMyName(N)$ actually works later, but for now, assume it sets the value of $N$ to be the responder’s name. In a bit, we will also discover what happens if the responder has amnesia and cannot remember their name!

Let’s expand the example a little further and introduce the idea of alternate options. For example, as a greeter we may only wish to ask someone their name if we don’t already know it. So first we should ensure we only ask for their name if we do not know who they are. You will hopefully have noticed that this is a constraint on the $askName$ message on the first line of the greeter role: ‘ask name to responder **only if** I do not recognise them’.

$$askName \ \Rightarrow \ \mathbf{a}(\text{repsonder}, Person2) \leftarrow \mathbf{doNotRecognise}(Person2) \tag{5.6}$$

In the case that **doNotRecognise**($Person2$) is true, the message asking for their name will be sent. What happens if the constraint is false and we *do* recognise that person? We want to simply say hello. So, to summarise, we ask their name if we don’t recognise them (then we say hello) *or* we just say hello.

The *or* operator allows us to say that directly in LCC. Let's see the whole greeter role with the alternative options:

$$
\begin{array}{l}
\mathbf{a}(\text{greeter}, Person1) :: \\
\quad ( \\
\qquad askName \ \Rightarrow \ \mathbf{a}(\text{responder}, Person2) \ then \\
\qquad name(N) \ \Leftarrow \ \mathbf{a}(\text{responder}, Person2) \ then \\
\qquad hello(N) \ \Rightarrow \ \mathbf{a}(\text{responder}, Person2) \\
\quad ) \\
\quad or \\
\qquad hello \ \Rightarrow \ \mathbf{a}(\text{responder}, Person2) \\
\quad then \\
\quad goodMorning \ \Leftarrow \ \mathbf{a}(\text{responder}, Person2)
\end{array}
\tag{5.7}
$$

Notice the use of parentheses for grouping together the operations on either side of the *or* operator. The indentation is merely to make it clear which statements are in the branches of the *or* operator; LCC is agnostic to white-space.

It is important when using the *or* operator with constraints to understand the idea of back-tracking. If a constraint fails (such as *doNotReconise* returning false) the interaction will 'back-track'. This just means that it retraces its steps until it gets to an alternate option (specified by the *or* ) and executes the other option. Of course, there are certain points in the interaction model that cannot be re-traced. For example, if the model sends a message and then a subsequent constraint fails, the interaction cannot back-track past the message operation; the pragmatic reason being that the message is now on the network and cannot be retracted. If this happens, or there are no alternate options available, the interaction will fail and will be shut-down.

Now we will have a look at the other role in this interaction:

$$
\begin{array}{l}
\mathbf{a}(\text{responder}, Person2) :: \\
\quad ( \\
\qquad askName \ \Leftarrow \ \mathbf{a}(\text{greeter}, Person1) \ then \\
\qquad name(N) \ \Rightarrow \ \mathbf{a}(\text{greeter}, Person1) \leftarrow \mathbf{getMyName}(N) \ then \\
\qquad hello(N) \ \Leftarrow \ \mathbf{a}(\text{greeter}, Person1) \\
\quad ) \\
\quad or \\
\qquad hello \ \Leftarrow \ \mathbf{a}(\text{greeter}, Person1)) \\
\quad then \\
\quad goodMorning \ \Rightarrow \ \mathbf{a}(\text{greeter}, Person1)
\end{array}
\tag{5.8}
$$

Again, this role mirrors the other role fairly closely. This role will start by waiting for the *askName* message to arrive; in the case that it doesn't the alternative branch of the *or* will be executed, where the *hello* message is expected. If some other message arrives, the interaction will continue to wait for one or other of these messages.

So, our final interaction for two actors greeting each other is shown below.

$$\begin{array}{l}
\mathbf{a}(\text{greeter}, Person1) :: \\
\quad (askName \ \Rightarrow \ \mathbf{a}(\text{responder}, Person2) \ then \\
\quad name(N) \ \Leftarrow \ \mathbf{a}(\text{responder}, Person2) \ then \\
\quad hello(N) \ \Rightarrow \ \mathbf{a}(\text{responder}, Person2) \\
\quad \ or \\
\quad hello \ \Rightarrow \ \mathbf{a}(\text{responder}, Person2)) \ then \\
\quad goodMorning \ \Leftarrow \ \mathbf{a}(\text{responder}, Person2) \\
\\
\mathbf{a}(\text{responder}, Person2) :: \\
\quad (askName \ \Leftarrow \ \mathbf{a}(\text{greeter}, Person1) \ then \\
\quad name(N) \ \Rightarrow \ \mathbf{a}(\text{greeter}, Person1) \leftarrow \mathbf{getMyName}(N) \ then \\
\quad hello(N) \ \Leftarrow \ \mathbf{a}(\text{greeter}, Person1)) \\
\quad \ or \\
\quad hello \ \Leftarrow \ \mathbf{a}(\text{greeter}, Person1) \ then \\
\quad goodMorning \ \Rightarrow \ \mathbf{a}(\text{greeter}, Person1)
\end{array} \tag{5.9}$$

For details of how to implement the OpenKnowledge components that will provide the functionality of the **getMyName**($N$) constraint, see section 5.3.1. Alternatively, this constraint could be satisfied visually, using constraint visualisation, described in section 5.2.

### LCC Visualisation

During the interactions between peers, it will sometimes be necessary to get some user input. Earlier in this chapter we introduced and defined in LCC a simple interaction that described how two actors would greet each other (see Model 26). One of these actors has the constraint **getMyName**($N$) which retrieves into the variable $N$ the name of the actor. This could be achieved through user input, allowing the user at the peer to type in their name. In this section will be show how this can be achieved using visualisation before discussing a little more generally about the visualisations that are provided with the OpenKnowledge core software.

Let's return to the simple interaction where two actors greet each other. The 'greeter' first asks the 'responder' their name, and replies with a personalised greeting.

The responder role contains this line within the model (for the full model see Model 26).

$$name(N) \ \Rightarrow \ \mathbf{a}(\text{greeter}, Person1) \leftarrow \mathbf{getMyName}(N) \ then \tag{5.10}$$

The constraint **getMyName**($N$) returns the name of the responder in $N$. We could retrieve this name by asking the user to type in their name. To do this we could implement an OpenKnowledge Component that performs some user interaction. However, this is not recommended, as your OKC may be used on any number of different types of peers on the OpenKnowledge network and you have no idea the best way to present that interaction; inputting words on a PC is quite different to using a mobile phone. So, an extension to the basic LCC provides a simple means for suggesting user interactions within an interaction model very simply by adding one line the model.

For our example model, the visualisation line would look like this:

$$\textbf{visual}(\textbf{getMyName}(N), \textbf{qask}(N)) \tag{5.11}$$

The two parameters to the **visual**(, ) operator define a mapping from the constraint to a 'visual term' that describes what type of user interaction is required. The example here shows that **getMyName**($N$) is mapped to the visual term **qask**($N$). **qask**() is a visual term that describes asking the user a question. The **qask**() visual term has been implemented such that it can also accept the question text. For example, this line maybe re-written:

$$\textbf{visual}(\textbf{getMyName}(N), \textbf{qask}(\text{“Please enter your name”}, N)) \tag{5.12}$$

The advantage of providing visualisations in this way is that a peer may implement the **qask**() visual term in which ever way it sees fit. It looks like Figure 5.2 for a standard Windows PC, but may be implemented in a different way on a mobile device, or may be disallowed on a rack-mounted server machine that has no screen.

### 5.2.1 Publishing Interaction Models

You can publish interaction models from the tool provided by the standard user interface. To get to the tool select “Publish IM” from the *Tools* menu.

Figure 5.2.1 shows the dialog box for publishing interaction models. This dialog box allows you to enter a new interaction model, or you can load one from your local disc using the “Load Interaction Model From File” button. Figure 5.2.1 shows the selection of an interaction model from the disc. Use the drop down box to change the type of interaction model you wish to search for (the default is LCC).

Once the interaction model that you wish to publish has been completely defined, you can check the syntax using the “Check Syntax” button on the right of the window shown in Figure 5.2.1. Then, enter a set of keywords for describing the interaction model in the box at the bottom. Bear in mind that these keywords will be matched against when a user searches the OpenKnowledge

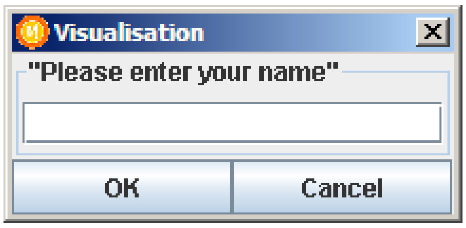


Figure 5.1: Built-in visualisation invocation to ask a user's name

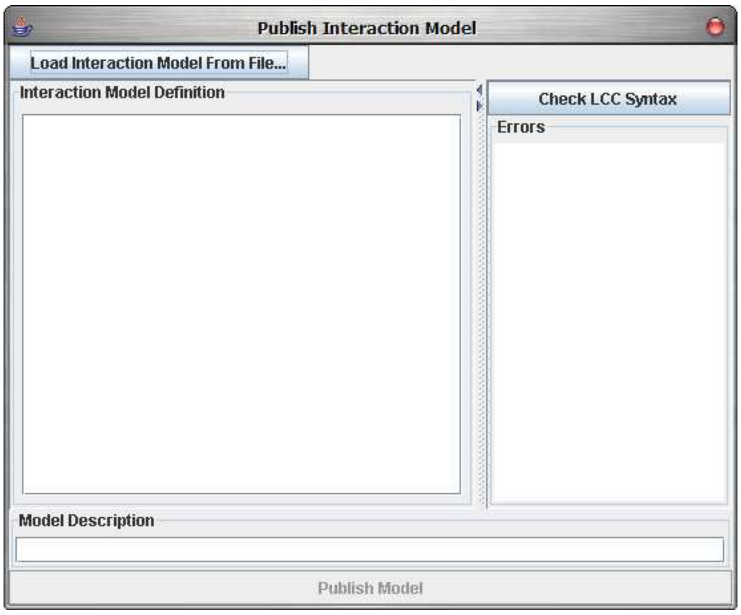
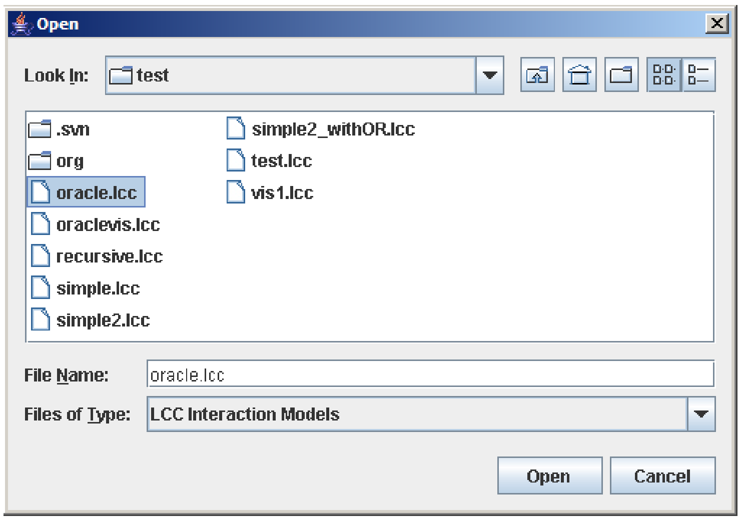

Publish Interaction Model
Load Interaction Model From File...
Interaction Model Definition
Check LCC Syntax
Errors
Model Description
Publish Model

Figure 5.2: The 'Publish Interaction Model' dialog box



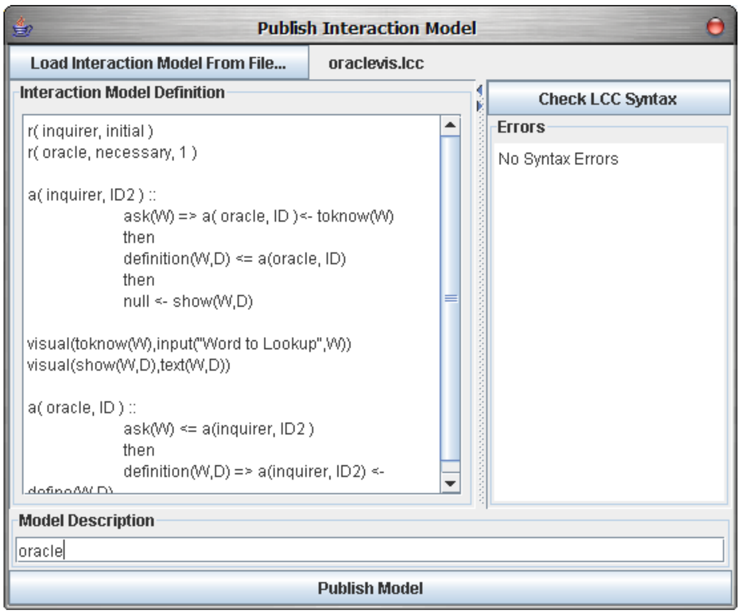
Look In: test

```
.svn             simple2_withOR.lcc
org              test.lcc
oracle.lcc       vis1.lcc
oraclevis.lcc
recursive.lcc
simple.lcc
simple2.lcc
```

File Name: oracle.lcc

Files of Type: LCC Interaction Models

Figure 5.3: Selecting an interaction model to publish


Publish Interaction Model
Load Interaction Model From File...
Check LCC Syntax
Publish Model

oraclevis.lcc

Interaction Model Definition

```
r( inquirer, initial )
r( oracle, necessary, 1 )

a( inquirer, ID2 ) ::
        ask(W) => a( oracle, ID )<- toknow(W)
        then
        definition(W,D) <= a(oracle, ID)
        then
        null <- show(W,D)

visual(toknow(W),input("Word to Lookup",W))
visual(show(W,D),text(W,D))

a( oracle, ID ) ::
        ask(W) <= a(inquirer, ID2 )
        then
        definition(W,D) => a(inquirer, ID2) <-
define(W,D)
```

Errors

No Syntax Errors

Model Description

oracle

Figure 5.4: Giving descriptive tags to the interaction model

network, so choose keywords that would be expected to return an interaction model like the one you are publishing.

Press the ‘Publish Interaction Model’ button to send the model to the discovery service. If the publishing succeeds a message will pop up (see Figure 5.2.1).

## 5.3 Creating Components

OpenKnowledge Components (OKCs) provide implementations of services for the OpenKnowledge network. The interaction models describe how these pieces of code can be used together to provide an application. The interaction models specify what functionalities the services must implement.

The functionality that an OpenKnowledge component must implement is specified in the interaction model in the form of constraints. Let us take a look at the example below:

$$\begin{array}{l} \mathbf{a}(\text{stockist}, ID) :: \\ \quad checkItem(X) \Leftarrow \mathbf{a}(\_, ID) \leftarrow validItem(X) \\ \quad itemStockCount(X, N) \Rightarrow \mathbf{a}(\_, ID) \leftarrow inStock(X, N) \end{array} \tag{5.13}$$

This simple example shows one role in a stock check operation. Other roles interact with this role by sending it the *checkItem*$(X)$ message, where $X$ is the item to check. If the item exists and the item is in stock, the model returns the number of items in stock in the message *itemStockCount*$(X, N)$.

The sections below use this example to describe how to build code to provide the functionality for these components (section 5.3.1), how to create component JAR files (section 5.3.2), and how to publish your components on the network so that others may use their implementations (section 5.3.4)

### 5.3.1 Programming New Components

We introduced the simple stock check interaction model in Model 30. This simple example requires two specified functions to be provided, and they are defined in the constraint description: *validItem*$(X)$ and *inStock*$(X, N)$.

The constraint simply define what function needs to take place at this point in the interaction; it does not provide any particular implementation of this function. OpenKnowledge components provide this implementation and there may be many implementations for any particular interaction model.

For the Java version of OpenKnowledge, implementations of constraints are provided in the form Java code wrapped into a Java Archive (JAR) file. These files are shared on the network and contain the code to run when a constraint needs to be satisfied by the model.

Preparing the code to do this has been made as simple a task as possible.

Components must implement a specific (empty) interface that is defined as part of the OpenKnowledge core software. However, they do not need to implement any specific methods other than those required for the constraint satisfaction. In Java, what this means is that the class you write to provide the implementation for the constraints must implement the interface `org.openk.core.OKC.OKCFacade`. This interface actually defines some methods, so to make programming of components as easy as possible, an implementation of these has been provided in the class `org.openk.core.OKC.impl.OKCFacadeImpl`. You should extend this class to create your OpenKnowledge component.

Code Listing 5.1 gives the skeleton for an OpenKnowledge Component.

```
package myokc;
import org.openk.core.OKC.OKCFacadeImpl;

public class MyOKC extends OKCFacadeImpl
{
}
```

Listing 5.1: Skeleton Class for OpenKnowledge Components

You can see that there is nothing more to creating an OpenKnowledge Component than to extend `OKCFacadeImpl`.

To make it even easier, the interaction model defines the method signature that you must implement. Because the library works by using reflection on you OpenKnowledge Component classes, you just need to implement the constraints as methods.

Your methods for the constraints should return boolean values; this represents whether the constraint was satisfied or not. You methods can throw exceptions and this will be considered as a constraint failure by the system.

Listing 5.2 shows an example of a full implementation of the stock checker OpenKnowledge component.

The arguments that are passed to your constraint methods match those that are defined in the interaction model. You can use `Argument.getValue()` and `Argument.setValue()` to change the interaction's state.

### 5.3.2 Creating Component JAR Files

In OpenKnowledge, components can be shared across the network, so that other users can download your component and run it on their machine; that is, they can assume a certain role in an interaction by using your code for that role. The components are shared using a Java Archive, which is similar to a zip file. The easiest way to create these components is by using the tool built into the default user interface.

First you need to publish the interaction model for which you have a component. Once published, search for it on the network. If it is already published on the network, then you can simply search for it.

Expand the role-list for the interaction model in the results table, and click on the role for which you have a component. The button “Create New OKC For Role” becomes available. When you click this button a window will appear that will let you create the component JAR.

### 5.3.3 Loading in Local Components

If you have created some OpenKnowledge components in JAR files that you have stored locally on a disc, you can load these into the local state of your peer. To do this, use access the File menu and select “Import OKC”. You will be presented with a dialog box from which you can select the OKC JAR file.

Note that restoring OKCs from disc into the state of your local peer will not subscribe your peer to any role; to do this, read section 5.3.4. b The component you have loaded will appear under the “Local Components” under ‘My Peer’.

### 5.3.4 Publishing Components

Publishing components is very easy from the user interface. Once a component has been loaded into the local state of your peer (see section 5.3.3) you can select that component from ‘My Peer’ (it will be listed under ‘Local Components’). Once selected click the ‘Share Component’ button; this will send a copy of the component to the network where it can be retrieved by other parties and used by them.

#### Programming a New Visualisation

Visualisations are small user interface modules that are used to satisfy constraints in an interaction model. They are entirely distinct from the user interface, but the user interface is responsible for providing a means for displaying them on the screen (see section 5.3.5 on providing alternate user interfaces).

Section 5.2 described how visualisations are incorporated into interaction models. They utilise the **visual**(, ) operation. The LCC below shows as example of the visualisation introduced earlier.

$$\mathbf{visual}(\mathbf{getMyName}(N), \mathbf{qask}(\text{“Please enter your name”}, N)) \tag{5.14}$$

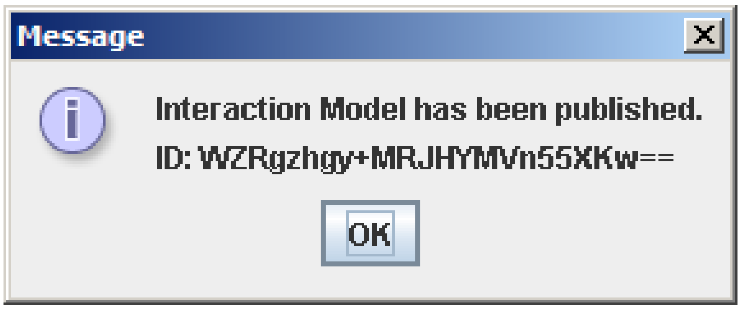
The first part of the visualisation definition is the interaction model constraint ($\mathbf{getMyName}(N)$), and the second part is the *visual term* ($\mathbf{qask}(\text{“Please enter your name”}, N)$).

The visual term does not specify how the visualisation will be realised, it only provides a hook for providing implementations. This means that a peer may have many implementations for a particular visual term, while also allowing different devices to have different implementations. For example, an image viewer on a mobile phone will be different to that on a desktop PC. There are a number of visual term implementations built-in to the kernel; see Section 3.2.3 for a list.

### 5.3.5 Providing Alternate User Interfaces

The OpenKnowledge kernel has been specifically developed to be easy to extend. All of the components that interface to the kernel have an application programmers' interface (API) defined for them. The user interface is no exception to this, meaning that you can create new applications that use the OpenKnowledge network, but look distinct from the default user interface that has been supplied.

As an example, Figure 5.3.5 shows the user interface that has been developed for coordination of emergency services in one of the OpenKnowledge demonstration systems. In this application each emergency service vehicle (ambulance, fire engine, etc.) is a peer on the OpenKnowledge network. They communicate through the network to coordinate themselves to aid in an emergency. For this scenario, the default OpenKnowledge user interface is too limited. The application is specific and requires a specific user interface that provides a map of the emergency area showing where the individual emergency vehicles are.

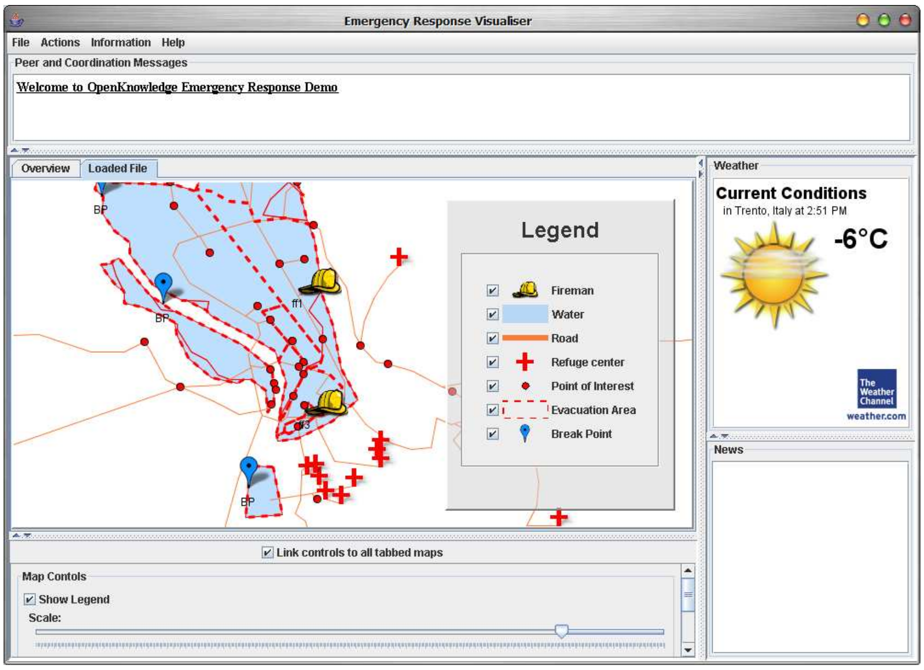


Message

Interaction Model has been published.
ID: WZRgzhgy+MRJHYMVn55XKw==

OK

Figure 5.5: Successfully published interaction model



Welcome to OpenKnowledge Emergency Response Demo

Current Conditions
in Trento, Italy at 2:51 PM
-6°C

Figure 5.6: Emergency Response User Interface

```
package myokc;
import org.openk.core.OKC.OKCFacadeImpl;

public class StockCheckerOKC extends OKCFacadeImpl
{
        /**
         *   Solve the validItem(X) constraint
         *   Succeeds if X is a valid item number
         *
         *   @param X The item identifier
         *   @return TRUE if X is a valid identifier,
         *           FALSE otherwise
         */
        public boolean validItem( Argument X )
        {
                String itemID = X.getValue();
                if( StockChecker.validItem( itemID ) )
                        return true;
                return false;
        }

        /**
         *   Solve the inStock(X) constraint. Returns
         *   the number of items of X in stock in N.
         *
         *   @param X The item identifier
         *   @param N The number of items in stock
         *   @return Always returns TRUE
         */
        public boolean inStock( Argument X, Argument N )
        {
                String itemID = X.getValue();
                N.setValue( StockChecker.checkStock(
                  itemID ) );

                return true;
        }
}
```

Listing 5.2: Stock Checker OpenKnowledge Component