

OpenKnowledge

FP6-027253

## Architecture Description of the OpenKnowledge Kernel

David Dupplaw<sup>4</sup>, Uladzimir Kharkevich<sup>5</sup>, Spyros Kotoulas<sup>1</sup>,  
Adrian Perreau de Pinninck<sup>2</sup>, Ronny Siebes<sup>1</sup>, and Chris Walton<sup>3</sup>

<sup>1</sup> Faculty of Sciences, Vrije Universiteit Amsterdam, The Netherlands

<sup>2</sup> Artificial Intelligence Research Institute, IIIA-CSIC, Spain

<sup>3</sup> School of Informatics, University of Edinburgh, UK

<sup>4</sup> School of Electronics and Computer Science, University of Southampton, UK

<sup>5</sup> Dept of Information and Communication Technology, University of Trento, Italy

Report Version: final

Report Preparation Date:

Classification: deliverable D2.1a

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

# Architecting Open Knowledge

David Dupplaw<sup>4</sup>, Uladzimir Kharkevich<sup>5</sup>, Spyros Kotoulas<sup>1</sup>, Adrián Perreau de Pinninck<sup>2</sup>, Ronny Siebes<sup>1</sup>, and Chris Walton<sup>3</sup>

<sup>1</sup> Faculty of Sciences, Free University Amsterdam, The Netherlands  
{kotoula|ronny}@few.vu.nl

<sup>2</sup> Artificial Intelligence Research Institute (IIIA-CSIC), Barcelona, Spain  
adrianp@iiia.csic.es

<sup>3</sup> The University of Edinburgh, Edinburgh, UK  
cdw@ed.ac.uk

<sup>4</sup> Electronics and Computer Science, University of Southampton, UK  
dpd@ecs.soton.ac.uk

<sup>5</sup> University of Trento, Povo, Trento, Italy  
kharkevi@dit.unitn.it

**Abstract.** OpenKnowledge (OK) is an European research project. Its aim is to implement a framework that allows sharing knowledge easily in an open environment. Open in the sense that anyone can join at any time, at low individual cost. The key idea is to share knowledge by focusing on semantics related to interaction. Making semantic commitments incrementally at run time, instead of a priori. This paper describes the OK system architecture. OK is a P2P framework on top of which all kinds of services may be built. The core framework will offer services such as discovery, publishing, interaction model interpretation, visualisation, and matching.

## 1 Introduction

The existing, open Worldwide Web has been successful on a global scale because the cost of participation at a basic level is low and the individual benefit of participation is immediate, rising rapidly as more participants take part. The same cannot currently be said about semantic based systems because the cost of being precise about semantics for sophisticated components is prohibitively high and the cost of ensuring an individual, absolute semantics for a component rises rapidly as more participants take part. OK aims to break out of this deadlock by focusing on semantics related to interaction (which are acquired at low cost during participation) and using this to avoid dependency on a priori semantic agreement; instead making semantic commitments incrementally at run time. The “Open” in OpenKnowledge thus is significant in two senses:

- It assumes an open system, which anyone may join at any time.
- It assumes an openness to being joined, achieved through participation at low individual cost

OpenKnowledge is a Peer-to-Peer network of knowledge providers. Each computer in the network is a peer which can offer services to other peers. Since we are designing a framework, it will only provide some core services which will be shared by all the peers. This framework allows all kinds of application services to be plugged on top. These plug-in applications are called OK Components (OKC).

Interaction between OKCs is a very important part of the architecture. Through a simple language, developers will be able to define the Interaction Models (IM) that specify the protocol that must be followed in order to offer or consume a service. OKCs are the ones in charge of playing the IM roles. Since there is no ‘a priori’ semantic agreement (other than the IM), the matching service will be used to automatically make semantic commitments between the interacting parts. Interaction with the final user is needed too. Therefore, visualisation of data and input from the user are a main part of the architecture. Furthermore, in a P2P network where no centralized server maintains the information respecting ‘who knows what’, a discovery service that allows OKCs and other resources to be published and found also needs to be provided.

This paper aims at the simplest architecture that might possibly work. In the spirit of agile software development, we are designing the architecture for the first iteration. It will be a proof of concept, or prototype, to be able to work on quickly. With early hands-on experience, we will be able to polish the architecture in subsequent iterations.

The paper is organized as follows. Section 2 describes the different core services of the OK architecture, their APIs, and the dependencies between them. Section 3 gives a step-by-step account of the events that relate to each core service, using as examples two main use cases. Section 4 explains the OK architecture’s contributions. Finally, section 5 deals with the improvements that are planned to appear in subsequent iterations of the architecture.

## 2 Describing the Parts

The OK architecture consists of the set of modules seen in Figure 1. The arrows define the dependency relations between modules. The different modules have been separated into four different layers: The user layer contains the visualisation module and the OK Components. The control layer is composed of the control and context management modules. The service layer is where the core services offered in OK are present. Finally, the network layer is the module used to join the P2P network and interact through it.

The following subsections offer a complete description of the functionality of each module, their API, and the dependencies on the functionality of other modules.

### 2.1 OK Graphical User Interface (OK GUI)

The OK system must have a GUI so that users can interact with the framework. The OK GUI is the entering point for users, it offers the basic functionality to

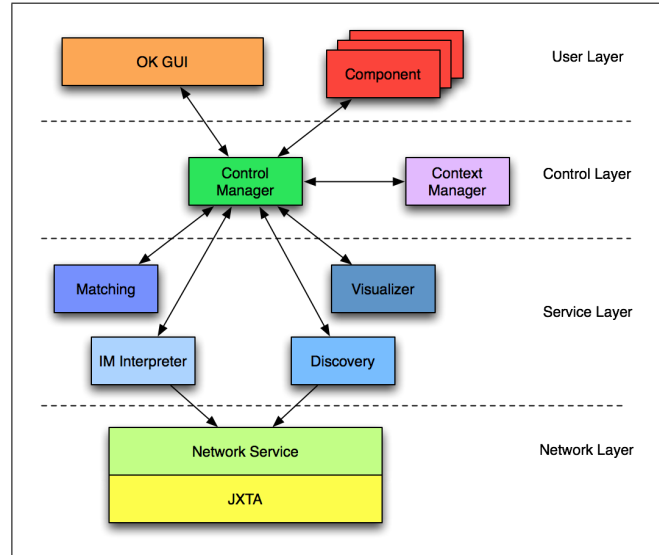


Fig. 1. OpenKnowledge Architecture

interact in OK. The OK GUI module does not have an API as the other modules do, because it only responds to user events. The actions that the user can execute through the OK GUI are described below:

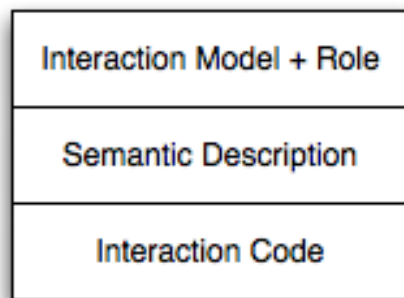
- *Search for Interaction Models* - Users can search for Interaction Models in the network by typing keywords and hitting the search button. All the Interaction Models found matching the required terms will be presented in a list.
- *Search for OK Components (OKC)* - Much in the same way as users can look for Interaction Models they can look for OKCs too. Another way to look for OKCs is to select an Interaction Model and ask the system to look for OKCs that can play any of its roles.
- *Download OKC* - Once a list of OKCs is shown to the user, the user can choose to download any of the OKCs in the list. Once the OKC has been downloaded the user may use the next two functionalities.
- *Execute OK Component* - A downloaded OKC is idle until it is executed. After that point the OKC can start playing the Interaction Model with other OKCs.
- *Stop OK Component* - In order for an OKC to remain idle after having been executed, this functionality has to be employed.

## 2.2 OK Component

OK Components (OKC) are small applications that can be plugged into the OK peers. An OKC can be replicated in many peers, and peers can contain

many OKCs. OKCs are made up of the following parts (see Figure 2): 1) The *Interaction Model* (IM) describes, in LCC, how OKCs can interact with each other. Each OKC will play one of the roles in the IM. 2) A *Semantic Description* which contains information (in natural language or a Semantic-Web language like OWL) like a description of what the component can do, who wrote it, some pointers to peers that can play each role, mappings between datatypes, a performance indication, etc... We assume for now that each OK-component, at least has a set of keywords describing the functionality, which is needed by the “default Component Discovery Service” (see section 2.6). 3) The *Interaction Code* is the actual application code that will play one of the roles in the IM.

OK Components are saved on a peer and can be either idle or instantiated. When an OKC is instantiated it is ready to play its role as defined in the IM. Instantiated OKCs will play the IM with other instantiated OKCs whose roles are compatible.



**Fig. 2.** OpenKnowledge Component

**Interaction Model** Our interaction models are effectively templates that OKCs follow in order to interact successfully. Each model defines a pattern of interaction that is designed to accomplish a specific goal. For example, we can readily construct a book purchasing interaction model, which defines the acceptable sequences of interactions that may be performed to purchase a book. Each interaction model is parameterised by roles, which identify the different participants in the interaction. For example, our book purchasing model would likely be parameterised with buyer and seller roles. Every OKC is assigned a role for an interaction model, which identifies the appropriate interactions that they should perform.

**Interaction Model Syntax** Our interaction models will be defined in the Lightweight Coordination Calculus (LCC) [?]. This is a language designed specifically for expressing Peer-to-Peer (P2P) interactions within multi-agent systems. The expressive power of LCC makes it ideal for our purposes here, as we are attempting to define interactions between independent components. However, the key feature of LCC is that it has a formal semantics, which is based on process calculus. This will enable us to prove properties about our interactions (e.g. related to security and trust), and to verify the correctness of our interaction models [?]. The formal basis is the primary reason that we have chosen to use LCC over more popular languages, such as WS-Coordination, BPEL4WS, and the OWL-S process model. Nonetheless, we intend to define translations from these languages into LCC as future work.

The abstract syntax of LCC is presented in Figure 3. In our implementation we will use a concrete XML-based syntax. The schema for this XML syntax has yet to be fixed, but it will likely follow the same design as the MAP variant of the LCC language [?]. We use the abstract syntax here as it is useful to explain the language without the extra syntactic baggage of XML.

---

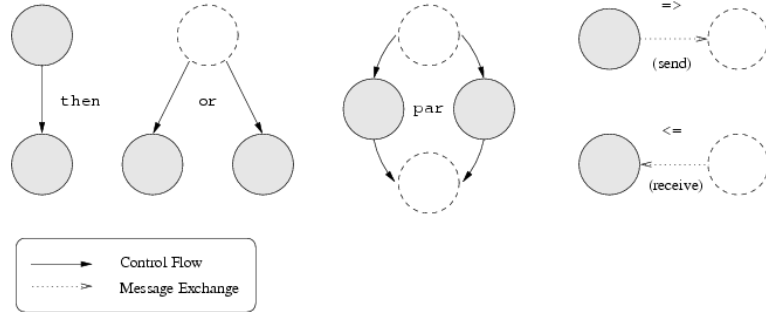

$$\begin{aligned}
Framework &:= \{Clause, \dots\} \\
Clause &:= Agent :: Dn \\
Agent &:= a(Type, Id) \\
Dn &:= Agent \mid Message \mid Dn \text{ then } Dn \mid Dn \text{ or } Dn \mid Dn \text{ par } Dn \mid null \leftarrow C \\
Message &:= M \Rightarrow Agent \mid M \Rightarrow Agent \leftarrow C \mid M \Leftarrow Agent \mid C \leftarrow M \Leftarrow Agent \\
C &:= Term \mid C \wedge C \mid C \vee C \\
Type &:= Term \\
M &:= Term
\end{aligned}$$

Where *null* denotes an event which does not involve message passing; *Term* is a structured term and *Id* is either a variable or a unique identifier for the agent.

**Fig. 3.** Abstract Syntax of LCC.

---

There are five key syntactic categories in the definition, namely: *Framework*, *Clause*, *Agent*, *Dn* (Definition), and *Message*. These categories have the following meanings. A *Framework*, which bounds an interaction in our definition, comprises a set of clauses. Each *Clause* corresponds to an agent, which is the name that we give to an interacting component. Each agent has a unique name *a* and a *Type* which defines the role of the agent. The interactions that the agent must perform are given by a definition *Dn*. These definitions may be composed as sequences (*then*), choices (*or*), or in parallel (*par*). The actual interactions between agents are given by *Message* definitions. Messages involve sending ( $\Rightarrow$ ) or receiving ( $\Leftarrow$ ) of terms *M* from another agent, and these exchanges may be constrained by *C*. These different kinds of operations are illustrated as graphs in Figure 4.



**Fig. 4.** LCC Operations.

---

The LCC language ensures coherence of interaction between agents by imposing constraints relating to the messages they send and receive in their chosen roles. The clauses are arranged so that, although the constraints on each role are independent of others, the ensemble of clauses operates to give the desired overall behaviour.

$$\begin{aligned}
 a(r1, A_1) :: offer(X) \Rightarrow a(r2, A_2) \leftarrow p(X) \text{ then } accept(X) \Leftarrow a(r2, A_2) \\
 a(r2, A_2) :: offer(X) \Leftarrow a(r1, A_1) \text{ then } accept(X) \Rightarrow a(r1, A_1) \leftarrow q(X)
 \end{aligned}$$

As an example, the above LCC fragment places two constraints on the variable  $X$ : the first ( $p(X)$ ) is a condition on the agent in role  $r1$  sending the message  $offer(X)$  and the second ( $q(X)$ ) is a condition on the agent in role  $r2$  sending message  $accept(X)$  in reply. By (separately) satisfying  $p(X)$  and  $q(X)$  the agents mutually constrain the variable  $X$ . Further details on the actual meaning of the LCC constructs is given in section 2.2.

**Interaction Model Semantics** The behaviour of an agent in a given role is determined by the appropriate LCC clause. Figure 5 gives a set of rewrite rules that are applied to give an unfolding of a LCC clause  $C_i$  in terms of protocol  $\mathcal{P}$  in response to the set of received messages,  $M_i$ , producing: a new LCC clause  $C_n$ ; an output message set  $O_n$  and remaining unprocessed messages  $M_n$  (a subset of  $M_i$ ). These are produced by applying the protocol rewrite rules above exhaustively to produce the sequence from  $i$  to  $n$ :

$$\langle C_i \xrightarrow{M_i, M_{i+1}, \mathcal{P}, O_i} C_{i+1}, \dots, C_{n-1} \xrightarrow{M_{n-1}, M_n, \mathcal{P}, O_n} C_n \rangle$$

We refer to the rewritten clause,  $C_n$ , as an expansion of the original clause,  $C_i$  and write  $expanded(C_i, M_i, \mathcal{P}, C_n, O_n)$  when this expansion is performed. In the next section we describe how this basic expansion method is used for multi-agent coordination.

The following ten rules define a single expansion of a clause. Full expansion of a clause is achieved through exhaustive application of these rules. Rewrite 1 (below) expands a protocol clause with head  $A$  and body  $B$  by expanding  $B$  to give a new body,  $E$ . The other nine rewrites concern the operators in the clause body. A choice operator is expanded by expanding either side, provided the other is not already closed (rewrites 2 and 3). A sequence operator is expanded by expanding the first term of the sequence or, if that is closed, expanding the next term (rewrites 4 and 5). A parallel operator expands on both sides (rewrite 6). A message matching an element of the current set of received messages,  $M_i$ , expands to a closed message if the constraint,  $C$ , attached to that message is satisfied (rewrite 7). A message sent out expands similarly (rewrite 8). A null event can be closed if the constraint associated with it can be satisfied (rewrite 9). An agent role can be expanded by finding a clause in the protocol with a head matching that role and body  $B$  - the role being expanded with that body (rewrite 10).

**Interaction Code** The Interaction Code is executed whenever the IM comes to a point where a constraint needs to be fulfilled, values are needed for variables, or a path taken when a choice point is reached. The code will be executed by the Control Manager whenever the Interpreter demands it. Code is optional, that way IMs can be shared between peers without the code that plays its roles. When the code is present, the *Interaction Model* will also contain information as to what role the code can play.

The interaction code must contain the following functions:

- *Constraint functions* - Each of the constraints in the Interaction Model must be satisfied by the OKC code. When one of the constraints is found which cannot be satisfied automatically, the corresponding function in the interaction code will be called to satisfy the constraint.
- *Function for choice points* - LCC allows for choices in the Interaction Model. When one of these choice points is reached it is the interaction code that will have to choose.
- *Function for choosing variable values* - At some points in the Interaction Model, variables are used that can contain any value (most times it will be restricted to a given set of values). It is the code in the OKC that must choose what value to use. When one of these variables is found, the function will be called that chooses the appropriate value.

## 2.3 Control Manager

The control manager is the module that manages the whole interaction process. It serves as a mediator between the services layer, the context management, and the user layer. Most of the time it will be receiving calls from other modules and redirecting them to the appropriate module, without much computation



---

$A :: B \xrightarrow{M_i, M_o, \mathcal{P}, O} A :: E$	$if B \xrightarrow{M_i, M_o, \mathcal{P}, O} E$
$A_1 or A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E$	$if \neg closed(A_2) \wedge$ $A_1 \xrightarrow{M_i, M_o, \mathcal{P}, O} E$
$A_1 or A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E$	$if \neg closed(A_1) \wedge$ $A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E$
$A_1 then A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E then A_2$	$if A_1 \xrightarrow{M_i, M_o, \mathcal{P}, O} E$
$A_1 then A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} A_1 then E$	$if closed(A_1) wedge$ $A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E$
$A_1 par A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O_1 \cup O_2} E_1 par E_2$	$if A_1 \xrightarrow{M_i, M_n, \mathcal{P}, O_1} E_1 \wedge$ $A_2 \xrightarrow{M_n, M_o, \mathcal{P}, O_2} E_2$
$C \leftarrow M \leftarrow A \xrightarrow{M_i, M_i - \{M \Leftarrow A\}, \mathcal{P}, \emptyset} c(M \Leftarrow A)$	$if (M \Leftarrow A) \in M_i \wedge$ $satisfy(C)$
$M \Rightarrow A \leftarrow C \xrightarrow{M_i, M_o, \mathcal{P}, \{M \Rightarrow A\}} c(M \Rightarrow A)$	$if satisfied(C)$
$null \leftarrow C \xrightarrow{M_i, M_o, \mathcal{P}, \emptyset} c(null)$	$if satisfied(C)$
$a(R, I) \leftarrow C \xrightarrow{M_i, M_o, \mathcal{P}, \emptyset} a(R, I) :: B$	$if clause(\mathcal{P}, a(R, I) :: B) \wedge$ $satisfied(C)$

A protocol term is decided to be closed, meaning that it has been covered by the preceding interaction, as follows:

$$\begin{aligned}
& closed(c(X)) \\
& closed(A or B) \leftarrow closed(A) \vee closed(B) \\
& closed(A then B) \leftarrow closed(A) \wedge closed(B) \\
& closed(A par B) \leftarrow closed(A) \wedge closed(B) \\
& closed(X :: D) \leftarrow closed(D)
\end{aligned}$$

$satisfied(C)$  is true if  $C$  can be solved from the agent's current state of knowledge.  
 $satisfy(C)$  is true if the agent's state of knowledge can be made such that  $C$  is satisfied.  
 $clause(\mathcal{P}, X)$  is true if clause  $X$  appears in the dialogue framework of protocol  $\mathcal{P}$ , as defined in Figure 3.

**Fig. 5.** Semantic Interpretation of LCC.

---

involved. It serves as a dynamic interface for different modules that have to interact with each other. It also knows the structure of the complete peer interaction process, which it guides by delegating to other modules.

The control manager will manage the interaction process between the other modules in order to fulfill the OK functionality. The architecture is susceptible to change quite a bit during this phase of the project. The changes are necessary since we are exploring what the functionality of each module should be. In order to have greater flexibility, the control manager isolates each module by having only an interface to it. Most modules in the OK architecture have dependencies to other modules. Since these dependencies are mediated by the control manager,

the changes to any module's interface will only affect the control manager leaving other modules unaffected.

The functionality of this module is provided by the API in figure 6. The API offers operations for searching OKC on the net, or getting a list of the ones stored locally. It also has operations for downloading, starting, and stopping OKCs. All of this functionality will be used by the OK GUI module. There are also operations that will be used by the IM Interpreter to communicate with the OKCs; the `chooseBranch`, `satisfyConstraint`, `fillVariables`, and `newMessage` are operations that will be forwarded to the OKC involved in the interaction. The `getLocalOKCs` operation may also be used by the Discovery Service when other peers are looking for OKCs on the network.

---

```
public interface ControlManager
{
    // Delegated to Discovery
    public OKC\_Id[] searchOKCs(String query);
    public OKC\_Id[] getLocalOKCs(String query);
    public OKC downloadOKC(OKC\_Id component);

    // Delegated to IM Interpreter
    public void startOKC(OKC\_Id component);
    public void stopOKC(OKC\_Id component);

    // Delegated to OKCs
    public Branch chooseBranch(OKC component, Branch[] options);
    public Object satisfyConstraint(OKC component, Constraint constraint);
    public Binding[] fillVariables(OKC component, Variable[] variables);
    public void newMessage(OKC component, Message message, Type type);
}
```

**Fig. 6.** Control Manager Interface Definition.

---

## 2.4 Context Manager

When an OKC interacts with other OKCs, some information needs to be saved so that the peer can manage the interaction. Some of the information will be: the information related to the OKCs it is interacting with (peer id, reliability, etc...), the point/state the conversation is in, the common mappings used during the interaction, the values of the bindings in the LCC protocol, the path the conversation has followed, etc... Some of this information is perishable, and need not be retained for further interactions. Other parts of the context will be useful in subsequent interactions and will be saved for that purpose.

The functionality of this module is simple (see Figure ??). It contains functions to update or add context values, and functions to retrieve and search the context. There are two writing functions: One for saving transient objects that will be erased when the actual interaction is finished, and another one for saving objects permanently. In both cases the parameters to the save functions are the same; the OKC and the Interaction Id in which the object is being used, and the actual object to be saved. The second set of functions is for retrieving objects, they return a list of Objects that match the search parameters; when the OKC and InteractionId parameters are null, the search will look for the objects saved for any OKC and InteractionId, otherwise the objects returned must match the given OKC and InteractionId. The Query parameter is a string specifying search restrictions. The Dates parameter indicates the interval of dates between which objects will be searched. Finally, the Type parameter indicates the type of Object to be searched (mapping, context attribute, variable binding, ...)

---

```
public interface ContextManager
{
    // Storing data in context
    public void saveTransient(OKC component,
        Interaction interaction, Object data);
    public void savePermanent(OKC component,
        Interaction interaction, Object data);

    // Search data from context
    public Object[] search(String query, OKC_Id component,
        Interaction interaction, Type type);
    public Object[] search(String query, OKC_Id component,
        Interaction interaction, Type type);
}
```

**Fig. 7.** Context Manager Interface Definition.

---

## 2.5 Visualiser

Visualisation of data is very important in any system as without it no results can be provided to the user. However, it is also important in OpenKnowledge to allow interaction with the user, and visualisation of the execution of an interaction model is important for users to understand what part of the interaction model is being played.

**Interaction Visualiser** During the execution of an interaction model, various constraints within the model are satisfied by automatic means, either directly or through some ontological matching that allows the constraints to be satisfied. However, there are cases when the constraints are unable to be satisfied by automatic means and in this case there are two options available to the system:

- *User Interaction* - Ask a user what to do.
- *Give Up* - Roll back the interaction model and try a different path through the model.

Clearly if we are to provide user interaction, we need to consider how to achieve this, understanding that the interaction model execution may currently be on a remote peer. There is also the consideration of what we should ask the user, and even which user we should ask. Let us examine these more closely in the context of an example.

Let us take the example from the OK Manifesto paper [?] and imagine that a user was executing an interaction model to find experts. When executing the clause in equation 1 on some remote peer, this interaction model had a call to a database, encoded as the *get\_experts(X, D, S)* constraint within the interaction model.

$$\begin{aligned}
 a(\textit{expert\_finder}, E) :: \\
 \textit{request\_experts}(X, D) \Leftarrow a(\textit{expert\_locator}(X, D, L), C) \textit{ then} \\
 \textit{specialists}(X, D, S) \Rightarrow a(\textit{expert\_locator}(X, D, L), C) \Leftarrow \textit{get\_experts}(X, D, S)
 \end{aligned}
 \tag{1}$$

This would lead to some code associated with the OK Component (see section 2.2) being executed that would make this database call. However, during the execution of this code the database did not respond and so the constraint could not be satisfied automatically. Therefore, a user must be consulted.

To summarise, the constraint that could not be satisfied was designed to populate a list of people’s names with names of experts for some given discipline and the user who executed the interaction model was attempting to find the names of those experts. When the constraint satisfaction fails the system will ask a user on the current peer. As the current peer is providing the knowledge for the constraint satisfaction, it may be pragmatic to assume that a user on that peer will have some knowledge of the roles their peer is able to provide. However, it could be that the peer providing the role of *expert\_finder* will not have a user interface having been setup as a headless server.

For the first prototype, we will assume the following:

- A user is always present at the peer that has to solve the constraint
- This user will be able to provide answers to unsatisfiable constraints

The control manager will identify when a constraint is not satisfiable. When it has detected such a case, it will invoke the visualisation routine to provide user interaction. Because we are considering only the simplest case, the API for the

interaction routines will be very simplistic. A discussion of how the visualisation routines may be extended to provide much more flexibility is provided in section 5.

To explain the API we shall consider the example above; *get\_experts* fails. The error handling function (from the LCC interpreter) is handled by calling the visualiser. The function for the visualiser is:

```
void visualise( Component p, Class c, Context x ) (2)
```

This will ask the user to introduce the data needed asking for an input of type *c* using the context *x*. The context may be used to provide a better visualisation, although in the early prototype this may simply contain the name of the constraint that was unsatisfiable so that the visualisation may at least label the input boxes.

During the time that the visualisation routine is being executed (or at least being waited upon), the peer that was unable to satisfy the constraint must wait until the user fills the visualisation result.

As the visualisation is completed (e.g. user presses “Submit”) the info will be sent back to the OKC who requested the visualisation. The interaction model will continue as if the data had been returned by the OKC.

Using a single method to provide the visualisation on any peer provides the simplest possible integration with other parts of the system. It still relies heavily on support in the controller and the IM interpreter.

## 2.6 Discovery Service

### Ronny and Spyros are in charge of this section

The Discovery Service takes care of storing OK-components and peer descriptions. In this way, given a query, the Discovery Service can return OK-Components that match the query. Also, it is meant to be able to return peers that are able to fulfill roles described in the Interaction Models which is part of an OK-Component of interest. To keep things modular, we separate these two different functionalities into a *Component Discovery Service* and a *Peer Discovery Service*. Both Discovery services get their knowledge about the components via *publish activities*, sent by peers. For example, if a user made designed a component, it can send it via the OK-system to a Component Discovery Service. We plan to make this Service completely distributed so that it stays scalable. To solve the bootstrap problem, we decided that in our default client<sup>6</sup>, we install some standard services, like the two Discovery services, on each peer. The user itself may decide if (s)he actually wants to host such a service that plays a role in an default interaction model.

**Component Discovery Service** The core functionality of the Component Discovery Service is to match queries with OKCs. Thus, the role of this service is to receive a query and return a set of pointers to OKCs.

<sup>6</sup> <http://www.openk.org/downloads>

---

```

public interface DiscoveryService
{
    // Finding and publishing OKC
    public OKC_Id[] getComponent(String description);
    public boolean storeComponent(OKC component);

    // Search data from context
    public PeerId[] getPeers(OKC_Id component, RoleId role);
    public PeerId[] search(String Description);
    public boolean registerPeer(OKC_Id component, RoleId role, PeerId peer);
    public boolean registerPeer(String description, PeerId peer);
}

```

**Fig. 8.** Discovery Service Interface Definition.

---

The programmers interface of this service can be seen in figure 8:

- **getComponent** Returns OKC IDs matching the description provided. For now, the description is a set of keywords, which semantically will be matched against the descriptions of the components stored at the service.
- **storeComponent** Stores a Component in the CDS. Returns **true** if it succeeds, **false** if not. Note that we do not say anything about the implementation. It could be that the service runs on one machine, or that it is completely distributed over many. Actually, we will do the latter.

**Peer Discovery Service** The core functionality of the Peer Discovery Service is to match roles of OKCs with services running on peers. Thus, the role of this service is to receive a role description together with the Interaction model and return a set of pointers to available OKCs. We plan to write two different ways how peers register themselves at a PDS:

- *by component- and role identifier*: the user of a peer or the peer itself selects the role(s) it wants to play in a specific component. Given that components and roles all have unique identifiers, the PDS will contain a set of triples containing the *peerID*, *componentID*, *roleID*. This is a very simple and robust solution, however it is less flexible than the following, more complex solution.
- *by peer description*: the user of a peer or the peer itself describes its capabilities (i.e. the functionalities of the services that it provides) by a semantic description, which in our case will be a simple set of keywords. In essence this will be identical to the process happening when components are published at a CDS.

The interface of this service is as follows:

- **getPeers** Returns peerIDs that subscribed to the RoleID from a component identified by the ComponentID.
- **getPeers** Returns peerIDs matching the Description provided. For now, the description is a set of keywords, which semantically will be matched against the descriptions of the components stored at the service.
- **registerPeer** Here, a peer, identified by its PeerID, registers itself to play a role in a certain component, respectively identified by the RoleID and ComponentID.
- **registerPeer** Here, a peer registers its 'expertise' via a Description (in our case a set of keywords). Returns **true** if it succeeds, **false** if not.

## 2.7 Interaction Model Interpreter

As previously noted, interaction models are an important part of our architecture. These models permit separately-defined Open Knowledge Components (OKCs) to interact together at runtime. Without an interaction model, we would be limited to applications consisting of a single OKC, or a hard-wired collection of OKCs. Using interaction models, we can dynamically compose OKCs at runtime. This gives us the ability to construct and execute applications on-the-fly, and the flexibility to alter these applications as necessary.

The interaction model interpreter is the key to enabling different OKCs to interact. In our architecture, each peer has its own interpreter. This interpreter is responsible for handling the interactions for all the OKCs in that peer. The interpreter takes an interaction model as input and performs the appropriate actions for the OKC as defined in the model. These actions are dependent on the role of the OKC and may involve sending and receiving messages, performing computations, and making decisions. If the interaction model is properly designed, and the OKCs are able to interact, then the interactions will lead to the desired outcome.

It should be clear from our discussion that the interaction model interpreter plays a crucial role in the construction of applications for the OK architecture. Consequently, we have put a lot of effort into the representation and interpretation of these interaction models. In particular, we have previously provided a formal syntax and semantics for the interaction models, so that they can be defined and interpreted without ambiguity. In the remainder of this section we describe the interface to the interaction model interpreter. This is the interface to the module that will parse the Interaction Models, (for the syntax and semantics see sections 2.2 and 2.2) apply the rewrite rules and demand the necessary operations to be executed by the OKC.

**Interaction Model Interface** We have now defined how the interaction model interpreter will operate. To recap, this interpreter will accept an interaction model defined using an XML-variant of the LCC syntax that we have presented. The interpreter will then evaluate this model using the formal rewrite rules that we have described. These rules define precisely when messages should be sent and received between components, and the flow of control that will result in

these message exchanges. The interpretation process will also involve performing computation, which is specified by the constraints in our interaction models.

To complete the definition of the interpreter, it remains to define the interface to the interpreter, so that it may be connected with the other facilities described in this document. This interface is very straightforward, and is shown as a Java interface definition in Figure 9.

---

```
public interface Interpreter
{
    // Components managed by the peer
    public void setComponent(Component comp) throws ComponentError;
    public void removeComponent(String name) throws ComponentError;
    public Component getComponent(String name);
    public List<Component>[] getComponents();

    // Models associated with components
    public void setModel(Component comp, Model model) throws ModelError;
    public void removeModel() throws ModelError;
    public Model getModel();

    // Roles played by peer inside each component
    public void setRole(Component comp, Role role) throws RoleError;
    public void removeRole(Component comp, String name) throws RoleError;
    public List<Role>[] getRoles(Component comp);

    // Execution of components
    public void startInterpreter(Component comp) throws ExecError;
    public void stopInterpreter(Component comp) throws ExecError;
    public Status getStatus(Component Comp);
    public void setTimeout(Component comp, int timeout);
}
```

**Fig. 9.** Interaction Model Interpreter Interface Definition.

---

Our interface defines the methods that comprise the interpreter, and their types. There are four sets of operations in this interface, which have the following purpose:

1. As previously noted, a single peer can interact simultaneously with a number of different OKCs. Thus, the first set of methods are used to managed precisely the components that are associated with the peer. New components can be registered with `setComponent`, and old components can be removed with `removeComponent`. We can also retrieve all of the components by `getComponents`, or a single named component with `getComponent`.



2. The second set of operations allow us to manipulate the interaction models that will be interpreted. Each OKC will be associated with a single specific interaction model. We can assign an interaction model to a component with `setModel`, retrieve the model with `getModel`, and remove the model with `removeModel`.
3. Before we can interpret an interaction model, we must assign components to the various roles within the model. This is accomplished by the third set of methods. The role that the component will take within the interaction model is assigned with `setRole`, and cleared with `removeRole`. We can also obtain a list of possible roles with `getRoles`.
4. The final set of operations enable us to interpret the interaction model associated with a specific component. The interpreter can be started with `startInterpreter` and stopped with `stopInterpreter`. We can also query the interpreter at any time, to find the state of the interaction using `getStatus`. Finally, we can adjust the timeout value (seconds) for the receipt of messages with `setTimeout`. We note that the interface does not give much detail about the state of the interaction. This is given by the interaction visualiser that we have previously described.

## 2.8 Ontology Matching Service

The Ontology Matching Service solves the semantic heterogeneity problem among different knowledge representations. This service offers match routines which produce mappings between the nodes of the graph-like structures that correspond semantically to each other. This functionality is exploited by Control Manager in at least three different phases:

- *Role matching* deals with the semantic heterogeneity in a role description (e.g. the matching of roles: *expert\_finder* and *person\_finder*).
- *Term matching* deals with the structural heterogeneity in a role description (e.g. the matching of methods: *get\_address (Full\_Name)* and *get\_address (Name , Surname)*).
- *Query/Answer matching* deals with the semantic heterogeneity arising from the statement of a query and the values returned in its answers (e.g. the matching of needed for interaction module operation *get\_address ('Stephen Salter')* and the operation *get\_address ('Salter, Stephen')* that a particular peer can actually perform).

In order to solve these problems Ontology Matching Service offers the interface in figure 10.

We call the former routine *string match* routine and the later one *object match* routine. These routines provide a unified way to deal with different types of heterogeneity. The *string match* routine takes two strings, automatically recognizes implicitly described structures inside them, and produces the semantic relations between these structures in the form of *mapping elements* (`MapElement[]`). A *mapping element* (ME) is a 5-tuple  $\langle IDi, Ni, Nj, R, C \rangle$  where

---

```

public interface MatcherService
{
    // Matching methods
    public MapElement[] match (MapElement[] previous,
        String source, String target )
    public MapElement[] match (MapElement[] previous,
        Object source, Object target, String type)
}

```

**Fig. 10.** Ontology Matcher Interface Definition.

---

IDij is a unique identifier of the given mapping element; Ni is the i-th node of the source structure; Nj is the j-th node of the target structure; R specifies a semantic relation (e.g. more or less general, equivalent to, disjoint from) which may hold between the concepts at nodes Ni and Nj; and C is numerical coefficient between 0 and 1 showed *plausibility* of ME. The *object match* routine takes source and target structures as *Objects* and a type of these *Objects* as the *type* parameter. Then this routine produces an array of ME's between the concepts at nodes of source and target structures which are hidden by the *Objects*. In the case when the given structures have been already partially matched (i.e. there is a subarray of ME's), both routines may reuse this information (pME parameter in the routines definition) in order to produce rest of mappings faster.

In the case of *Role Matching* the LCC description of role might be represented as a plain text or as a java object. The plain text is matched by the *string match* routine. The *object match* routine is used to match java objects. A term in the *Term Matching* could be presented either as text and be processed by the *string match* routine or as a term with context (Ctx) in the form of object < Ctx, Term > and be processed by the *object match* routine. The *Query/Answer* matching only exploits the *string match* routine.

The Ontology Matching Service works as independent service and does not exploit any functionality from other services.

## 2.9 Network Service

This module is the abstraction layer to the communication channel. It has been decided that the communication channel will be implemented with JXTA. JXTA is a framework for P2P communications which provides the functionality needed to build P2P applications without having to worry much about the network structure.

The network functionality can be divided into the following blocks (see Figure 11): One with the functionality to manage connections between two peers (this is between two peer modules), with one operation to create a connection and

another one to close it. Another block with operations for sending and receiving messages over a connection. Finally, a block with operations to manage network groups; create, join, leave, and get a list of group members.

---

```
public interface NetworkService
{
    // Managing connections
    public OKConnection connect(Module module, PeerId peer);
    public void close(OKConnection connection);

    // Message passing
    public void send(Message message, OKConnection connection);
    public void receive(Message message, OKConnection connection);

    // Group management
    public GroupId createGroup(String description);
    public void joinGroup(GroupId group, Module module, PeerId peer);
    public void leaveGroup(GroupId group, Module module, PeerId peer);
    public MemberId[] getMembers(GroupId group);
}
```

**Fig. 11.** Network Service Interface Definition.

---

### 3 Tying it together

In order to understand how the modules of the architecture fit together, two use cases are defined in this section. For each one of them, the sequence of events in the use case are seen. The first events are those started by the end user, they propagate to events from one module to another (usually in the form of function calls).

The use cases show how a OK user may find interesting videos through Open Knowledge. This motivation gives place to two subsections: 1) the user will need to find OKCs in the net that suit his needs and will install the one he likes best, 2) he will then use the new OKC in his computer to find the videos he wants.

#### 3.1 Find and Download OK Components

In this first use case, the user wants to find videos through the OK network. So he looks for OKCs that allow him to search and download videos. Once the user finds an OKC that he likes, he downloads it into its local OKC repository.

- The user starts the OK Graphical User Interface (OK GUI).
- The OK GUI shows the search panel.
- The user types in the search preferences. This can consist of a text entry with "Video Download".
- The user clicks the "search" button.
- The OK GUI executes the searchOKCs operation in the Control Manager Module.
- The Control Manager (CMgr) redirects the search preferences to the Discovery Service by executing its GetComponent operation.
- The Discovery Service (Dvry) looks for local resources that match the search preferences. It runs its distributed algorithm for search through the network of peers.
- At this point remote peers are involved in the search process through their own Discovery Service (RDvry).
  - The RDvry receives a query from the Network Service. The query has information about the kind of OKCs to return.
  - The RDvry asks the Remote Control Manager (RCMgr) to search for local OKCs that match the query. This is done through the RCMgr getLocalOKCs operation.
  - For each OKC stored locally at that peer its description is passed to the Remote Matching Service (RMat) to see if it matches the query. The match operation is used for this purpose.
  - All the OKCs whose Semantic Description matches the query terms are returned to the LDvry.
  - The LDvry sends the list of local OKCs to the originating peer through the Network Service.
- The LDvry gathers a list of all the resources matching the original query by using the previous procedure with as many remote peers as needed.
- Dvry returns the list of OKCs to the CMgr who returns it to the OK GUI.
- The OK GUI shows a list of all the returned resources to the user.
- The user selects the OKC in the list that it likes best.
- The user clicks on the "download" option.
- The OK GUI executes the downloadOKC method of the CMgr.
- The CMgr redirects the call to the Dvry.
- The Dvry finds a peer that holds the resource and downloads it.
- The OKC is returned to the CMgr which saves it. This is done by calling the storeComponent operation on Dvry.
- The Dvry publishes to the network the fact that it now holds this resource too.

### 3.2 OK Component Interaction

This use case shows how the user can use an OKC. This use case happens once the user has downloaded the OKC he likes for downloading multimedia. He then uses the OKC to find multimedia through the OK network.

- The user has the OK GUI opened in front of him.
- The user clicks to see the local resources (could have used the search panel too).
- The OK GUI executes the getLocalOKCs operation in the CMgr.
- The CMgr returns a list of all the OKCs stored locally.
- The OK GUI shows the list of local resources to the user.
- The user selects the OKC it wants to use (Multimedia Download OKC - MMOKC).
- User selects to start executing the OKC.
- The OK GUI calls the startOKC operation in CMgr.
- The CMgr loads the OKC.
  - The CMgr gets the MMOKC from the storage.
  - The CMgr loads the MMOKC code.
  - The MMOKC launches its user interface. We assume the user interface contains a search panel and a list of results.
  - The CMgr gets the MMOKC's Interaction Model and loads it using the setModel operation in the IM Interpreter (Int).
  - The MMOKC can now start interacting with other OKCs in the network.
- The Int can do the following with the Interaction Model given to it. What the Int does depends on the LCC code in the IM. Int accomplishes all the following functionality by calling the appropriate method of CMgr. The CMgr will forward the call to the OKC involved in the interaction.
  - Ask the OKC to choose what LCC branch to follow. CMgr's chooseBranch.
  - Ask the OKC to check a constraint. CMgr's satisfyConstraint.
  - Ask the OKC to give values for unbound variables. CMgr's fillVariables.
  - Send or Receive messages from the network. In each case the CMgr is informed of the message being sent or received, and the CMgr informs the OKC.
- The user types the kind of multimedia it wants to search (Type: Video, Search text: Madonna).
- The MMOKC fills in the gaps and constraints of the message typed by the user.
- The Int asks CMgr for a list of peers it can play the IM with as searcher
- The CMgr redirects the petition to Dvry, by executing the getPeers operation.
- The Dvry searches for the peers that have a running copy of OKCs that can play the role of searchee.
- The Dvry returns the list to the CMgr.
- The CMgr hands the list to the Int.
- The Int sends the message to all of the other peers.
- The remote peers are now involved:
  - Remote Interpreter (RInt) receives the message.
  - RInt checks if the message is a correct part of the IM.
  - RInt redirects the message to the according Remote OKC (through the RCMgr).

- ROKC does not understand some part of the message and asks its RCMgr to find translations.
- RCMgr redirects translation query to RMat.
- RMat finds mappings to the terms.
- The protocol continues as long as the IM definition allows.
- Eventually the user may want to stop the MMOKC from running. It can choose to do that through the OK GUI.
- The OK GUI will execute the stopOKC operation in CMgr.
- The CMgr will call the removeModel operation on Int and will unload the MMOKC code.

## 4 Conclusions

The architecture in this paper is for a first prototype of the Open Knowledge P2P platform for knowledge sharing. The platform uses a mix of technologies in the research area, such as: Semantic Routing, Ontology Matching, Interaction Modeling, and Visualisation technologies. These technologies are quite new. Through this prototype we will be able to test these technologies in order to gain a better understanding of how they can better fit together. With better understanding we will be able to modify the architecture to better suit our needs and to add other technologies (see section 5) that will make it more user-friendly.

We have designed a system simple enough so that it will be built in a reasonable amount of time. The platform built will provide the basic functionality to make knowledge sharing possible. Each computer that acts as an OK peer, will be able to provide services to other peers and use services from others in the form of OK Components. OKCs will use the core services provided in the architecture to interact with each other and accomplish the tasks the end user desires.

## 5 Future Work

This section describes functionality that we want to have implemented in the final version of the OK Architecture, but will not be present in the first prototype version. The idea is to give pointers to the things that will be present in the next versions of the architecture.

### 5.1 OK Component Library

The OK platform will be downloaded with the essential services needed to operate. In future versions it will be bundled with some useful OK Components. Nonetheless it is a good idea to maintain a server with an OKC library so that OK users know where to find new OKCs and download them.

The Library will be hosted on a dedicated server (which may also act as a OK peer) , but eventually the OKCs will be distributed among all the OK peers and the dedicated server will not be needed.

## 5.2 Ambient LCC

For our initial prototype using LCC as the interaction language will do. But, as applications get more complex, we will need to extend LCC. Ambient LCC is an extension to LCC. This extension provides means to specify virtual spaces where processes can take place (i.e. an ambient). Ambients may hold information such as attribute/value pairs, and may also have other ambients inside. It will allow to model a richer set of Interaction Models.

In future versions of the OK platform, the IM language will be Ambient LCC, and the IM Interpreter will be modified accordingly.

## 5.3 Interaction Model Translation

LCC is not the only language available to define Interaction Models. We have chosen LCC because of its simplicity and well established semantics. But we do not want to close the door to other languages. Future versions of the OK Architecture will allow the IM to be defined in other well-known languages such as; WS-Coordination, BPEL4WS, OWL-S, or Electronic Institutions. In order to allow this, we propose having a translator module that will translate Interaction Models defined in these languages, into LCC.

It is also foreseen that a visual tool will be provided to aid in the specification of Interaction Models. The visual composer will help users define IMs through state diagrams, such as those in Electronic Institutions. The modeler will help users manipulate IMs downloaded from OK and compose new ones.

## 5.4 Security, Trust and Reputation

An important part of any software platform is security. This has been omitted in the first OK prototype, but it is intended for further versions. Security has been studied in centralized systems, and to a great degree it has been accomplished there. But security in distributed and open systems is a much harder task. Since OK is both open and distributed, we need to look for a mechanism that will work in this environment.

Trust and reputation will be used in OK as a distributed and social mechanism to ensure security. Further OK platform versions will be bundled with a Trust and Reputation service that will inform its OKCs about other OKC and peer reputation so as to avoid harmful interactions.

## 5.5 Automatisation

Since our aim for the first prototype is for the simplest thing that can possibly work, the users will have to take part in a lot of decisions. In order for OK to gain acceptance, the required input from the user must be kept to a minimum.

Some of the processes that will be automated are; peer selection, IM selection, term matching, and filling values for messages templates.

## 5.6 Interaction Model Verification

LCC allows for a great degree of freedom when defining an Interaction Model. This freedom does not prevent coding IMs that are not consistent across the different roles, or that may have deadlocks. One solution for this is to verify that the IMs fulfill the properties needed.

We plan to implement an IM verifier. The verifier will insure that interaction models are correct. It will not be used during the interactions, but will be useful when creating a new interaction models or when choosing one to be downloaded from the network.

## 5.7 Approximate and Partial Ontology Matching

P2P network is a highly dynamic, real-time environment where peers are free to do whatever they wish. Dynamic nature of peers imply that mappings, which have been established between concepts at nodes of source and target ontologies, often might become only an approximation of real mappings, in the case when concepts in matched ontologies were slightly changed, or even become partially incorrect, in the case when these changes were more dramatic. Consequently peers might have to live with approximate and incomplete mappings before such links are recovered. Matching process is very costly and thus it is impractical to establish complete and accurate mappings which will almost inevitable become incomplete and approximate after a time. To deal with this situation we have to find some balance between the matching performance and quality of matching results.

The freedom of peers implies that different sources might choose to represent the same thing in different ways. The Ontology Matching solves this semantic heterogeneity problem but it is not always a precise process and sometimes the inaccuracy of ontology matching cannot be entirely prevented: e.g., *type\_of\_fire(Source, Risk\_Level)* is only an approximate match to the constraint *type\_of\_fire(Type)*. Source might be matched to Type with some level of approximation and Risk level might be ignored. So we are always interested in cost-effective, approximate and/or partial matching which produce good enough mappings at the present moment for the particular task.

## 5.8 Reusing Mappings

The process of matching is a costly one. Therefore the mappings that are returned from this process are very valuable. It is interesting for OK to reuse knowledge that has already been computed. Mappings used in previous interactions are pieces of knowledge that can be stored for reuse. In future versions we are planning to reuse those mappings that have been successfully used in previous interactions. This means that not only must each peer be able to save and search for previous mappings, but it must also be able to search for them throughout the network. The Discovery Service will be in charge of discovering useful mappings that will be given to the Matching Service so that it may reuse them in order to avoid the costly process of calculating them.



## 5.9 Services as OKCs

In the first architecture prototype we are defining some of the basic services to be implemented into the framework as hard-coded modules. Our intention for future versions of the architecture is to go for a pure component-based approach. We plan to have all the services be OKCs that will interact with each other through Interaction Models as regular OKCs. This approach will give greater flexibility, and will also allow OK users to create OKCs that can offer the same functionality as the actual services, or even improve it.

If every service is an OKC, users may be able to search the OK network for OKCs that can act as Ontology Matchers. They can then choose the one they like best and use it. With a component-base approach each user will be able to tune their peer to suit him best.

## 5.10 The Diagnostic Visualiser

The diagnostic visualiser is for visualising the interaction model progress as it is being executed. This allows for a user to find where there are problems in the system, as well as providing a nice way for them to understand how the system is working while they wait for it to finish.

In the final version of OpenKnowledge, the visualisation may be best presented in the same way that the visual composer is presented, as it will be familiar to the user and will avoid them having to learn multiple user interfaces.

The diagnostic visualiser will work by flagging in the initial context of the instantiated interaction model that diagnostic visualisation is required. This will force each component that receives the interaction model to send a message back to the originating peer to inform it of the progress.

This functionality will require support in the controller and the contextual manager for setting of the flag, and also support in the message queue processor for intercepting the message. The controller will then call the diagnostic visualiser to fire the appropriate message back to the originating peer.

The operation to fire the message will require a component  $p$  on some peer (the originating peer in this case) to be sent the state of the interaction model, encoded in some structure  $m$ . The originating peer will retrieve the message and update its display.