OPENKNOWLEDGE MANUAL

TABLE OF CONTENTS

1. INTRODUCING OK

Overview

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. OpenKnowledge 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 system which allows peers on an arbitrarily large peer-to-peer network to interactive productively with one another without any global agreements or pre-run-time knowledge of who to interact with or how interactions will proceed. Existing services can be translated into OpenKnowledge peers and can then make use of the full functionality of the system [1]: this translation process already exists for WSDL services and can be created for most other kinds of services.

This is made possible through the use of shared interaction protocols, which can be written by any user of the system and then reused by any peer on the network. These protocols describe interactions between two or more peers, and detail the messages that will be sent as part of that interaction and the constraints on those messages, which give information about the semantics of the messages and under what circumstances they can be sent. A peer can play a role in any of these protocols in which it can satisfy all the relevant constraints.

This shift of emphasis to interaction means that semantic agreement can be reached locally. The lack of a global ontology means that peers will be using different terms which may not match, but interpretation of these terms need only be done within an interaction and only for those terms that directly affect the interaction. The problem of semantic heterogeneity is therefore reduced to the specific case of terms that are actually in use and the matching takes place within a particular context - the context of an interaction - which aids the interpretation process.

To learn how to use OpenKnowledge system, click here. [2]

Overview

Source URL: http://www.openk.org/introducingOK/overview

Links:
[1] http://www.openk.org/howOKworks/service
[2] http://www.openk.org/howOKworks/overview

Research Topics

The key aspects of the OpenKnowledge Project are:

1. **Interaction**

   How are these shared protocols written and coordinated?

   For more information, see Deliverables 1.1 - 1.3 [1]

2. **The Discovery Service**

   How do you discover who to interact with and what protocol you should use?

   For more information, see Deliverables 2.1 - 2.3 [1]

3. **Semantic Matching**

   Since there is no global ontology, the semantics surrounding specific interactions must be negotiated at run-time.

   For more information, see Deliverables 3.1 - 3.6 and 4.1 - 4.6 [1]

4. **Trust**

   How can one be sure that a peer qualified to play a role will behave in the agreed manner and will produce goods/information/services etc.
   of an appropriate standard?

   For more information, see Deliverables 4.2 - 4.8 [1]

5. **Visualisation**

   How can users interact with the system: to keep track of what their peers are doing, to make decisions and oversee interactions and to provide information and feedback where necessary?

   For more information, see Deliverables 5.1 - 5.4 [1]

6. **Multimedia**

   How can multimedia applications be supported via the peer to peer architecture and how can interaction models be harnessed for multimedia?
   For more information, see Deliverables 8.1 - 8.5 [1]

Research Topics

Source URL: http://www.openk.org/introducingOK/researchtopics

Links:
[1] http://www.openk.org/deliverables

Testbeds

We are currently evaluating the effectiveness of OpenKnowledge in two testbeds. For more details of these testbeds, as well as details of other projects which are making use of the OK system, see Case Studies [1].

- **Emergency Response**

  During an emergency situation - for example, a large flood, an earthquake or a terrorist attack - the ability of different agencies and individuals to interact quickly and effectively, understanding one another and building and executing mutual plans, is key and can potentially save many lives. There will generally be some kind of central command that has been put in place in anticipation of disaster and which directs other players along pre-arranged lines. However, the nature of an emergency means that nothing can be taken for granted: communication lines can fail; key players (and even the command centre) can become unavailable for any number of reasons; outside players who were not anticipated in pre-disaster planning can have valuable help to offer. OpenKnowledge provides the infrastructure for executing pre-arranged plans between expected players who know how to communicate with one another, but also, crucially, can allow productive communication and action to continue even when one of these major stumbling blocks are encountered.

  We have developed a simulation based on real data from large-scale flooding in the Trentino region to demonstrate the ability of OpenKnowledge in this domain. Tests are currently being made to evaluate the effectiveness of our approach.

  For more information, see Deliverables 6.5 - 6.7. [2]

- **Bioinformatics**

  Modern biological experimentation requires computational techniques of different kinds to enable large-scale and high-throughput studies. For example, structural genomics efforts aim to understand the function of proteins from their 3-D structures, which are either determined by experimental methods (e.g., X-ray crystallography and NMR spectroscopy) or predicted by computational methods. Proteomics efforts as another example, aim to understand the functional consequences of the collection of proteins that is present in a cell, or tissue, at a given time - particularly where differences are observed between healthy and disease states. In both examples the data, and the analytical methodology applied to them, are obviously central to accomplishing the aims of these scientific domains. In addition, however, a framework is required that allows researchers to access the data, interpret the data, and exchange knowledge with one another. In the OpenKnowledge peer-to-peer framework, any experimental protocol that is followed when one, or several, researchers are undertaking a bioinformatics experiment can be viewed as a series of interactions between the researcher(s), the databases from which the data are obtained, and the tools that are applied to derive secondary information from this data. Many bioinformatics protocols can be represented as consecutive interactions, or steps in a workflow. These two simple (though non-trivial) first bioinformatics analyses involving consistency checking amongst comparable data from different databases and different bioinformatics programs, respectively, were enacted (i), as a means of introducing the system to the bioinformatics community, and (ii), to illustrate the commonalities of the underlying interactions, and accordingly the transferability of the underlying protocols, in OpenKnowledge.

For more information, see Deliverables 6.1 - 6.3. [2]

Testbeds

Source URL: http://www.openk.org/introducingOK/testbeds

Links:
[1] http://www.openk.org/applications/cases
[2] http://www.openk.org/deliverables

2. HOW OK WORKS

Overview

This page gives general information on how to use the OpenKnowledge system. This information is appropriate for those who wish to understand the process of OpenKnowledge or for people who wish to become users of OpenKnowledge but do not wish to write their own components or interactions. For more detailed information - once you have read this page - go here. [1]

In order to become an OpenKnowledge user, you simply need to download the OpenKnowledge kernel from here [2] together with some additional components [3] that you might want to use. This basic infrastructure will allow you to create an OK peer. Existing WSDL services can also be made into OK peers. See here. [4]

overview

Source URL: http://www.openk.org/node/581

Links:
[1] http://www.openk.org/beingAnOKuser/intro
[2] http://www.openk.org/resources
[3] http://www.openk.org/okc
[4] http://www.openk.org/howOKworks/service

Lifecycle

This is the lifecycle of interaction from the point of view of user.

i) **Why is this interaction taking place?** The user inputs a need reflecting why he wants to participate in an interaction. For example, this might be *buy camera*, or it might be that he wants to provide a service in exchange for payment, or so on. This need is expressed in terms of keywords.

ii) **What will the interaction protocol be?** The user will then need a protocol to describe the interaction that he wishes to take part in. This is called an *Interaction Model (IM)* and is described here [1]. There are various roles (at least two) in an IM and the user must choose which his peer is to play. The user may have written his own IM for this interaction, but more usually he will wish to find one that has been published on the network. In this case, the discovery service is invoked and will return a list of potentially suitable IMs. When IMs are published, they are annotated with a set of keywords, and these keywords are matched to the user's need to discover the potentially appropriate IMs. These IMs are ranked according to their popularity - i.e. how many times they have been used.

The user must then choose which of the potential IMs he wishes to use. This can be done in various ways:
a) Choosing the highest ranked IM;
b) Re-ranking the IMs according to the scores returned by the automated matching operator, which matches the ability of the user's peer to the abilities required by the role it is to play;
c) The user looking through the IMs personally.

iii) **Subscribing to the interaction** Once the IM has been chosen, the user's peer should subscribe to the appropriate role. The following information is required for subscription:
a) The peer ID (this can be verified and it is not possible to lie about this).
b) The matching score that was produced by matching the peer's abilities to those required (it is possible for dishonest peers to lie about this).
c) Some keywords giving more details of the peers wants or intended service (this is optional).

iv) **Initiating the interaction** Once all roles in an IM have at least one subscriber, the interaction can begin. This may happen immediately if all the other roles were subscribed to when the user's peer subscribed, or may not happen for some time (or never) if some of the roles currently have no subscribers. Whether a user will wish to subscribe to a role in an IM that won't commence immediately, or whether he will wish to search for one that is ready to go will depend on the situation. For example, a seller may be prepared to subscribe to a role and then wait around until a buyer appears, whereas a buyer may perhaps want to purchase something immediately.

Once all the roles are filled, a peer on the network will be arbitrarily chosen to be the coordinator for this interaction. This may or may not be a peer that also subscribed to that IM, though in a system with a large number of peers, it is very unlikely that they will be. Any peer can opt out of being chosen to be a coordinator; the coordinator will be chosen from all the peers that are willing to do this.

v) **Choosing partners** The coordinator will send information of which peers have subscribed to which roles to all of the subscribers of the IM. Each peer must then choose which of the peers subscribed to other roles it is prepared to interact with. This can be done automatically with the help of the trust model [2] or can be done through intervention of the user. Each peer then sends back to the coordinator a set of potential partners.

vi) **Allocating roles** The coordinator then allocates peers to roles in such a way that no peer is forced to interact with a peer that is not in its list of chosen peers. The coordinator informs all subscribed peers of the outcome. The process now ends for peers that have not been chosen. They may resubscribe to the same role in the same IM if they wish, and wait for the process to begin again.

vii) **The interaction** Whenever a peer is due to send a message, the coordinator will prompt that peer to give it the details that should be sent in that message, and the coordinator then passes that message on to the appropriate peer. This message passing is not visible to all peers, only to the sending and receiving peer in each case.

viii) **The interaction terminates** If the interaction proceeds smoothly, this will happen after all the messages of the interaction have been sent. However, this may happen earlier if a peer fails to respond appropriately.

ix) **Interaction feedback** The coordinator informs all peers involved in the interaction as to whether the interaction terminated successfully (they may or may not be able to judge this for themselves, depending on their role in the interaction). The coordinator will also send back information about certain messages so that peers can determine how things went during the interaction (how and why this happens is discussed here [2]).

x) **Learning from interactions** Optionally, and depending on settings, the user can provide more targeted feedback about certain aspects of the interaction. This may only happen some time after the interaction: for example, if the user's peer acted as a buyer, some judgement can be formed about the success of the interaction through observation of message passing and successful termination of the interaction, but the full quality of the seller can only be ascertained once the goods have actually arrived and been examined by the user. These observations can be used to judge which peers to interact with in the future.

lifecycle

Source URL: http://www.openk.org/node/582

Links:
[1] http://www.openk.org/howOKworks/im
[2] http://www.openk.org/beingAnOKuser/trust

What is an Interaction Model?

The shared protocols describing interactions are called *Interaction Models* and are written in Lightweight Coordination Calculus (LCC) [1]. This, as the name suggests, is designed to be very lightweight and to convey an interaction simply without getting bogged down in complexity. However, since there may be details that are pertinent to an interaction that cannot be expressed in LCC, we allow annotation files to be written for IMs. We do not discuss this here in this simple description of LCC, but details [1] of this are available.

```
a(requester, A) ::
    ask(X1) => a(informer, p2) <-- query_from(X1, p2) then
    tell(X1) <= a(informer, p2) then
    ask(X2) => a(informer, p3) <-- query_from(X2, p3) then
    tell(X2) <= a(informer, p3)

a(informer, B) ::
    ask(X) <= a(requester, B) then
    tell(X) => a(requester, B) <-- know(X)
```

Figure 1. An example *informer/requester* interaction model.

Figure 1 shows a simple diagram for discovering information: a *requester* asks for information and an *informer* responds.

This simple IM is sufficient to illustrate the three key aspects to an IM: roles, messages and constraints. Firstly, we should note that in LCC, variables are identified by beginning (or consisting of) an uppercase letter, whereas constants begin with (or consist of) a lowercase letter.

Roles: IMs must consist of two or more roles, with each role describing the necessary actions for one of the peers that will take part in the interaction. Role identifiers are of the form *a(role,ID)*. For example, in Figure 1, the first role is *requester*, and this will be played by a peer with ID *A*. *A* is a variable that will become instantiated with the appropriate peer ID when this is known. It would be possible to make this ID a constant, and thus the role would only be playable by the particular peer identified by that constant, but this is contrary to ideal of reusing IMs so is not generally encouraged.

Messages: These are indicated by the double arrow. To the left of this double arrow is message that is to be passed and to the right is the role to which it is to be sent or received from. Double arrows pointing to the right (towards the role) indicate messages going to that role, and vice versa. Within an IM, most messages will contain one or more variables that are not determined by the IM: for example, in Figure 1, the variable X is passed around. The value of this variable is only determined during run-time: therefore, the IM is reusable in different situations.

Any message that is sent in an IM must also be received by the appropriate role within that IM, and thus all messages appear twice, once as a sent message within the sender's role and once as a received message within the receiver's role. When there are only two roles, this makes the message passing in those roles completely symmetric, as can be seen in Figure 1.

Constraints: These are put on messages, appearing to the right of them in an IM and pointing to the relevant message with a single arrow, and have two functions:

- to explain how the variables in the messages should be instantiated;

- to limit the circumstances under which a message can be sent: it can only be sent if the constraint can be satisfied by the peer playing the role.

For example, during run-time, we do not wish to pass around the variable *X*; instead, it should be instantiated to a suitable value. This is done by the constraint *query(X)*. So the peer that takes on the role of requester must be able to satisfy this constraint, and in doing so, it will instantiate the variable X to whatever it wishes to query about.

Any OK peer is capable of passing any message, provided the necessary constraints are satisfied, so the ability to play a particular role is equivalent to the ability to satisfy all the constraints on messages in that role.

what_is_an_im

Source URL: http://www.openk.org/node/583

Links:
[1] http://www.springerlink.com/content/ux5dy5ggryu0mxna/fulltext.pdf

What is an OKC?

OKCs (OpenKnowledge Components) are plug-in components that contain the methods used to solve constraints in the Interaction Models.

An OKC is a jar containing a facade class extending the *OKCFacadeImpl* class in the kernel and an xml file, called *okcinfo.xml* describing the component. The jar file can obviously contain other support classes and resources, but the facade class is the one that must expose all the methods that solve constraints.

A method must:

- be public
- return a *boolean*
- every argument must be of type *Argument*

More than a single OKC can be used to solve the constraints in an role: the *matching* process finds the methods corresponding to the constraints in the available OKCs in the peer and creates the adaptors between them.

For example, a role clause like the following:

```
a(role1, A)::
     msg1(X,Y) => a(role2,B) <- C1(X) and C2(Y)
     then
     msg2(Z) <= a(role2,B)
     then
     null <- C3(X,Y,Z)
```

can have the constraints C1,C2,C3 matched to the methods m1,m2 from OKC1 and m4 from OKC2:

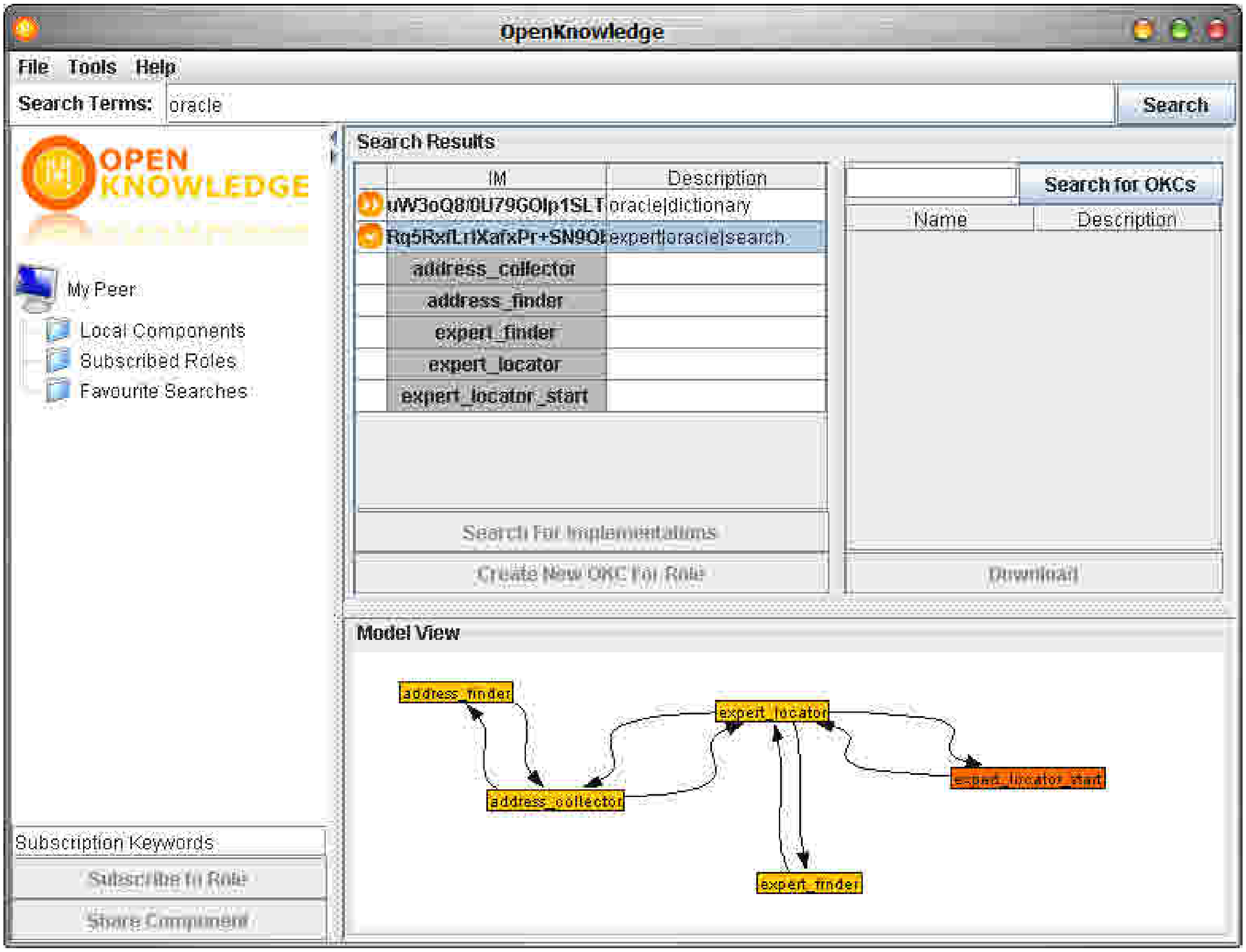
```
role1:         OKC1:
  C1(X)  <--->   m1(X)
  C2(Y)  <--->   m2(Y)
                 m3(J)
                 ...
                 OKC2
  C3(Z)  <--->   m4(Z)
                 m5(L)
                 ...
```

what_is_an_okc

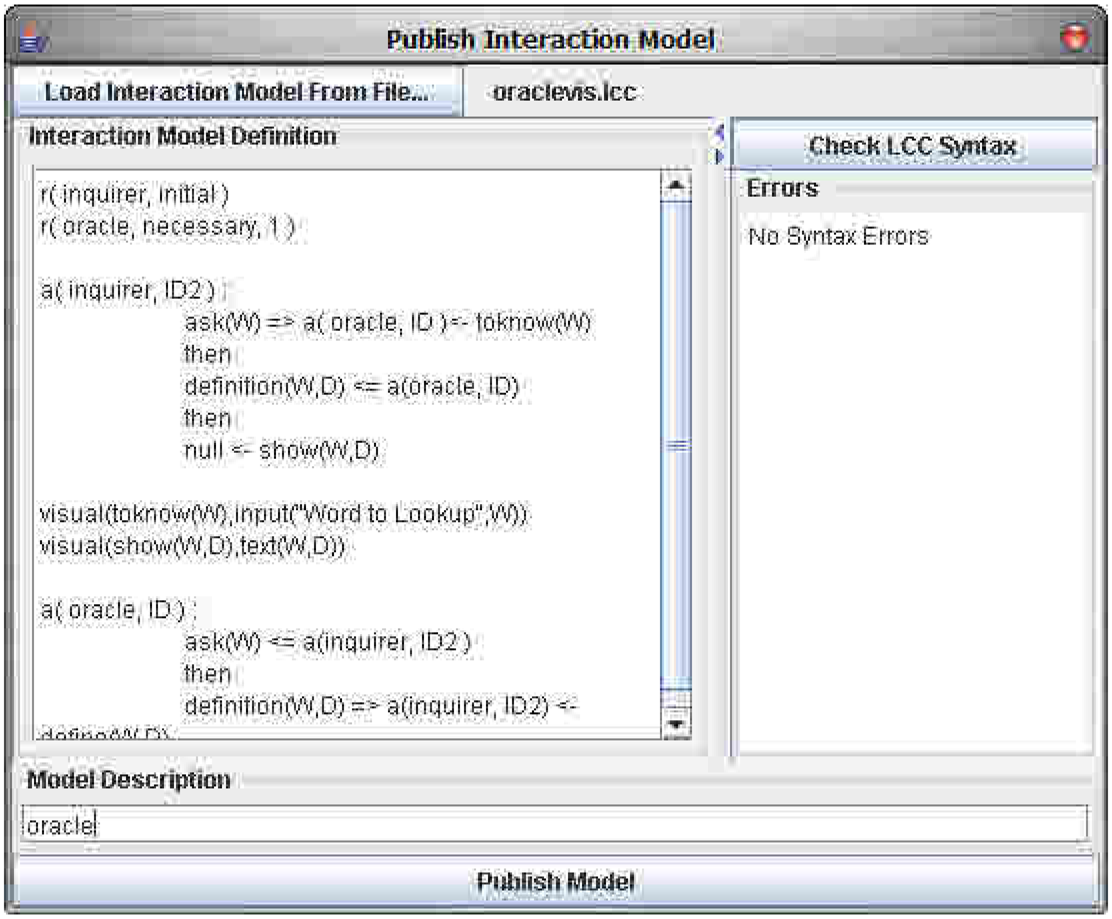
Source URL: http://www.openk.org/node/584

Interaction with the User

The OpenKnoweldge interface has a GUI so that the user can input their need and visualise the interactions that are going on. The OpenKnowledge kernel comes with a basic user interface that allows you to search the OpenKnowledge network for interaction models and OpenKnowledge components. Interaction model search results are shown graphically and you can execute the interaction models by subscribing in particular roles. A publishing tool is provided that allows the syntax of an interaction model to be checked prior to it being publishing. A tool is also provided for creating OpenKnowledge components from Java code and then sharing them onto the network. The interface also allows you to see the status of your peer and make changes to its configuration.



	IM	Description
	uW3oQ8/0U79GOlp1SLT	oracle|dictionary
	Rq5RxfLrIXafxPr+SN9Q	expert|oracle|search
	address_collector	
	address_finder	
	expert_finder	
	expert_locator	
	expert_locator_start	


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

```
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) <-
```

No Syntax Errors

oracle

However, during interaction, additional interaction with the user may be necessary: sometimes constraints should not be satisfied through the peer's knowledge base but by asking the user directly; at some choice points the direct opinion of the user should be sort, and so on. We use the annotation file [1] to provide such information about constraints so that an IM can easily adapt the user experience as necessary.

interaction_with_user

Source URL: http://www.openk.org/node/585

Links:
[1] http://www.openk.org/beingAnOKuser/usingLCC/annotation

Services as Peers

Existing services, such as WSDL services, can easily be made into OpenKnoweldge peers. This depends on a translation process so that an OKC is created for each service. This will allow the first-order constraints to function as service inputs and outputs.

The first-order terms that we match do not distinguish between inputs and outputs in the same manner as, for example, WSDL. Instead, both inputs and outputs are arguments of the same predicate. In Prolog notation, this is indicated by using a + for an input and a - for an output. Thus the term:

purchase(-Price,+Vehicle,+Number)

indicates that *Vehicle* and *Number* are inputs and *Price* is an output. During run-time, we can distinguish between inputs and outputs because inputs must be instantiated and outputs must be uninstantiated. In order to use our tree matching techniques for web services, we therefore make use of an automated translation process we have created that will map between a first-order term such as the above and a standard WSDL representation of the same information. This approach can also be used for other kinds of services in addition to web services; all that is required is that a translation process is created to convert between the representation of the service and our first-order terms.

This translation process has been written for WSDL services. For services using different kinds of representations, translation processes would have to be written.

services_as_peers

Source URL: http://www.openk.org/node/586

3. BEING AN OK USER

Using LCC

Example Interactions

The diagram in Figure 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*.
- *p1* sends a message *ask(q(Z))* to *p3*.
- *p3* sends a message *ask(q(b))* to *p1*.

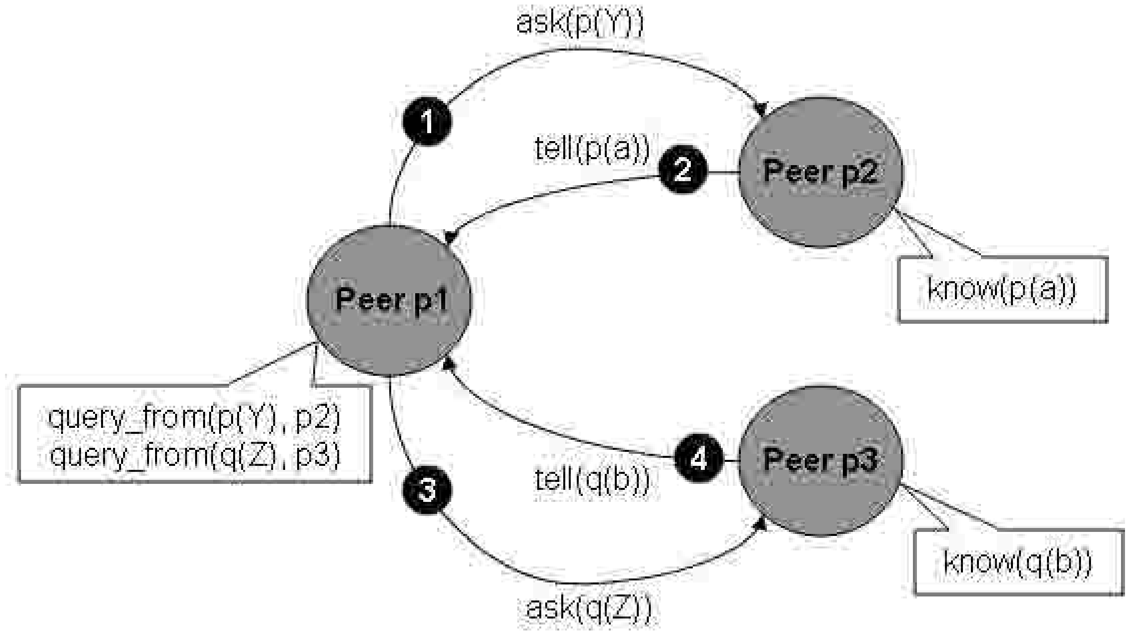


Figure 1. Basic Interaction Example

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*.

```
a(requester, A) ::
    ask(X1) => a(informer, p2) <-- query_from(X1, p2) then
    tell(X1) <= a(informer, p2) then
    ask(X2) => a(informer, p3) <-- query_from(X2, p3) then
    tell(X2) <= a(informer, p3)                                             (1.1)

a(informer, B) ::
    ask(X) <= a(requester, B) then
    tell(X) => a(requester, B) <-- know(X)
```

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), ...]*, 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 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.

```
a(requester(L), A) ::
    (ask(Query) => a(informer, Peer) <-- L = [q(Query, Peer)|Lr] then
    tell(Query) <= a(informer, Peer) then
    a(requester(Lr),A))
    or
    null <-- L = []

a(informer, B) ::
    ask(X) <= a(requester(), B) then
    tell(X) => a(requester(), B) <-- know(X)
                                                                                (1.2)
```

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 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.

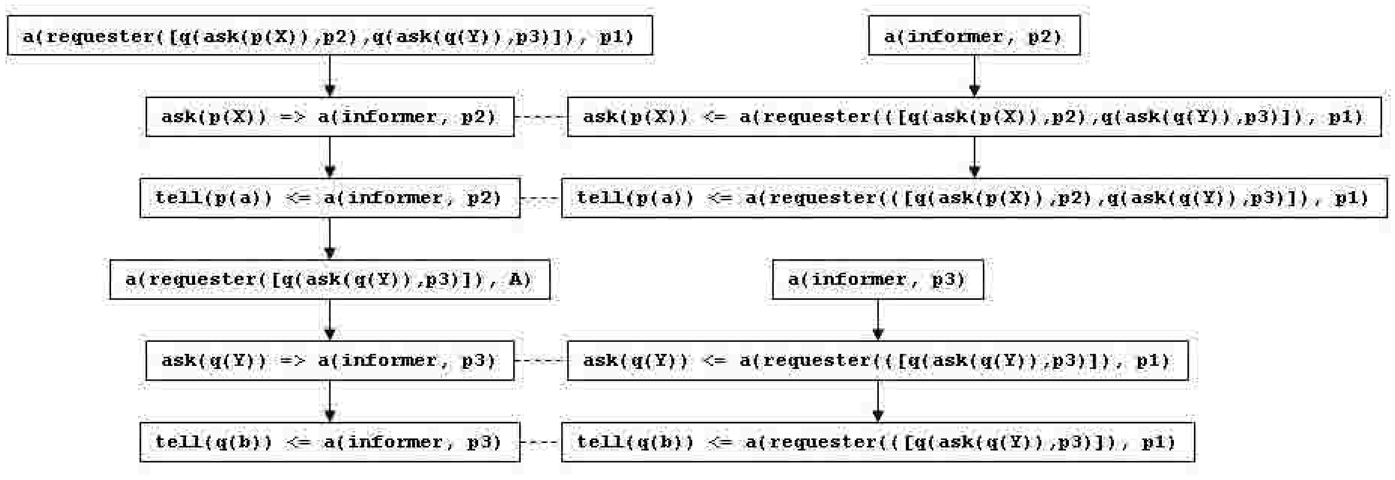

a(requester([q(ask(p(X)),p2),q(ask(q(Y)),p3)]), p1)
a(informer, p2)
ask(p(X)) => a(informer, p2)
ask(p(X)) <= a(requester(([q(ask(p(X)),p2),q(ask(q(Y)),p3)]), p1)
tell(p(a)) <= a(informer, p2)
tell(p(a)) <= a(requester(([q(ask(p(X)),p2),q(ask(q(Y)),p3)]), p1)
a(requester([q(ask(q(Y)),p3)]), A)
a(informer, 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 2. Event Sequences for the Example

In our earlier example (definition 1.1 above) we made some of the message passing events contingent on constraints. For example sending the message *ask(X1) => a (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 repositories of methods that they find useful. We call these OpenKnowledge Components (OKCs). A more detailed description of OKCs can be found here [1].

examples

Source URL: http://www.openk.org/node/618

Links:
[1] http://www.openk.org/howOKworks/okc

Syntax

Each LCC interaction model is defined by a set of clauses where each clause has the syntax shown in Figure 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.

```
  Clause := Role :: Def
    Role := a(Type, Id)
     Def := Role | Message | Def then Def | Def or Def
 Message := M  ⇒ Role | M  ⇒ Role ← C | M⇐ Role | C   ← M⇐ Role
       C := Constant | P(T m, ...) | ¬C | C   ∧ C | C ∨ C
    Type := Term
      Id := Constant | V ariable
       M := Term
    Term := Constant | Variable | P(Term, ...)
Constant := lower case character sequence or number
Variable := upper case character sequence or number
```

Figure 3. LCC Syntax

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

- **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 $F(A_1 ..., A)_n$ 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.

- **Messages**

There are two types of messages:
Incoming Messages : are of the form *Term* **a**(Role,*ID*), where *Term* is the content of the message. When using ASCII, the symbol is written using <=.
Outgoing Messages : are of the form *Term* ⇒ **a**(Role,*ID*), where *Term* is the content of the message. When using ASCII, the symbol ⇒ is written using =>.

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

- **Constraints**

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

```
Message  ← constraint(Arg1, ..., ArgN)                              (1.3)
```

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 ← 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.

```
visual(constraint(Arg1, ..., ArgN), visualTerm(vArg1, ..., vArgN))        (1.4)
```

The OpenKnowledge kernel has a number of built-in visual term implementations, listed below:
msg(M<,T>) Display a message *M* to the user, with the optional title *T*.
text(<T,>M) Display a large amount of text in *M* to the user, with the optional title *T*.
input(<Q,>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:

```
... ← L = [H|T]                                 (1.5)
```

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.

```
... ← L = [H|T]                                  (1.6)
```

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:

```
... ← L = []                                     (1.7)
```

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.

- **Comments**

  To comment your LCC you can use the C-like comments *//...* or */*...*/*. 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
*/
```

- **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.

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 out a(r2, Y) if 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 if M in a(r2, X)* to describe the message, *M*, being received, with *C2* giving a constraint that should hold as a consequence of receiving it.

```
a(r1,X) ::
     . . .
     M  ⇒ a(r2, Y )  ← C1
     . . .
                                                      (1.8)
a(r2, Y ) ::
     . . .
     C2  ← M ⇐ a(r2,X)
     . . .
```

  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)*.

```
a(sender, X) ::
     M  ⇒ a(recipient, Y)  ← message(M) and recipient(Y)
                                                                      (1.9)
a(recipient, Y) ::
     accept(M)  ←M ⇐  a(sender, X)
```

- **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*.

```
a(r,X) ::
     . . .
     E1 then                                              (1.10)
     E2
     . . .
```

  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*.

```
a(sender,X) ::
     M1 ⇒ a(recipient, Y ) ← message(M1)and recipient(Y ) then
     accept(M2) ← M2⇐ a(recipient, Y )
                                                                 (1.11)
a(recipient, Y ) ::
     accept(M1) ← M1⇐ a(sender,X) then
     M2 ⇒ a(sender,X) ← response(M1,M2)
```

- **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 if C1 or E2 if 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 if C2*) will not be attempted.

```
a(r,X) ::
     E1 ← C1
     or                                                  (1.12)
     E2 ← C2
     . . .
```

  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.

```
a(buyer, X) ::
     . . .
     accept(Offer) ⇒ a(seller, Y) ← acceptable(Offer)             (1.13)
     or
     reject(Offer) ⇒ a(seller, Y) ← unacceptable(Offer)
```

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:

```
a(buyer, X) ::
     . . .
     accept(Offer)  ⇒ a(seller, Y)  ← acceptable(Offer)             (1.14)
     or
     reject(Offer)  ⇒ a(seller, Y)
```

- **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.

```
a(r(A),X) ::
     (. . . R(A,Ar) . . .
     a(r(Ar),X))                                          (1.15)
     or
     (. . . P(A) . . .)
```

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*.

```
a(r(A), X) ::
     (M  ⇒ a(r2, p2)  ← A = [M|Ar] then
     a(r(Ar), X) )                                      (1.16)
     or
     (null  ← A = [] )
```

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*.

```
a(r(N, M), X) ::
     (M  ⇒ a(r2, p2)  ← N > 0 and N1 is N - 1 then
     a(r(N1, M), X))                                            (1.17)
     or
     (null  ← N =< 0)
```

patterns

Source URL: http://www.cisa.informatics.ed.ac.uk/OK/drupal/node/620

Annotation

An LCC model only defines the abstract exchange of messages between peers, and the preconditions and postconditions on those message. The messages and the constraints are first order predicates, and the arguments are variables.

However, constraints need to be matched against methods in OKCs. The OKC methods are described by the Java Annotation *@MethodSemantic:*

```
@MethodSemantic(
language="tag";
args={"title"}
)

public boolean askTitle(Argument t){...}
```

The arguments can be complex objects:

```
@MethodSemantic(
language="tag";
args={"email(subject,priority,body)"}
)

public boolean writeEmail(Argument email){...}
```

In this case, the argument email will contain a structure composed of three elements:

```
 email
    |- subject
    |- priority
    |- body
```

Interaction Models can be annotated as well. The annotations mechanism is general and can be used for different purposes. In particular, the LCC annotations are used to semantically annotate the variables in roles and to annotate messages and constraints for enabling their logging during an interaction run.

annotation

Source URL: http://www.cisa.informatics.ed.ac.uk/OK/drupal/node/621

Visualisation

Visualisers are small user interface modules that are used to allow the user 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. Visualisers are implemented in the same way as OKCs.

Visualisations can be suggested through the use of annotations in the LCC. For example, the getMyName constraint might be annotated in the LCC as follows:

```
@annotation( @constraint( getMyName(N) ), visual( qask("Please enter your name", N) ) )
```

As with other annotations, the first parameter defines the constraint to which the annotation is associated. The second part of the annotation utilises the "visual" annotation label. The definition of the visual annotation refers to a specific visualisation, in this case "qask" which asks the user a question.

How *qask* is implemented is independent of both the OKC and (potentially) the user interface. It is, however, specific to the platform on which the visualisation is running. For example, an image viewer provided on a mobile phone will be different to that provided on a desktop PC. It could be that specific peers and user interfaces that implement specific applications on top of the OpenKnowledge system will have specific visualisers that provider greater user interface integration, however, these interfaces must be robust to changes in the network that might mean different interaction models are used and therefore different visualisations are requested.

visualisation

Source URL: http://www.cisa.informatics.ed.ac.uk/OK/drupal/node/622

Designing OK Peers

Overview

- **Peers and OKCs**

  OpenKnowledge peers must have at least one OKC [5] in order to play a role but may have arbitrarily many. Each OKC is a set of methods, and these methods provide information on how constraints should be satisfied. It may be the case that an OKC is designed for a particular role, so that methods to satisfy all the constraints for that role are contained within the OKC. This is useful because it means that this OKC can be shared across the network and any peer that downloads it has the necessary methods for performing that role. However, when playing a particular role, a peer may use methods from many different OKCs. For example, a peer may have a *util.okc* in which it keeps many common methods that are used to solve constraints that reappear in a large number of roles. This prevents the peer from having to keep many copies of the same method; however, it is not forbidden for the peer to have more than one copy of the same method: it may download an OKC for a particular role that contained methods it already had and it would not be obliged to remove all of these duplicates.

  Although the sharing of OKCs is central to the concept of OpenKnowledge, there will be many situations in which peers wish to use their own trusted OKCs rather than ones developed by unknown others because this allows them more control over their actions. This can lead to the situation where peers need to play a role using an OKC that only imperfectly matches that role: this situation is dealt with through mapping constraints to methods [2].

  Peers are more than the sum of their OKCs. At the very least, peers subscribe to interactions and interact with their users, whereas the OKCs they contain merely provide methods for solving constraints. But, crucially, we would expect most peers to have their own knowledge that will be used to assist in these methods, because most methods access the peer's internal state. Thus, different peers may download the same OKC and yet produce different results and outcome when performing the same role.

- **Mapping Constraints to Methods**

  One of the central tenets of OpenKnowledge is that IMs and OKCs can be shared between peers on the network without any prior agreement about semantic standards. When a peer takes on a role, it performs it in its own manner: this may be through using an existing OKC for the role, through making a choice of many existing OKCs or for reusing its own OKC that is generally appropriate for the role. For example, a peer playing a *buying* role may have an OKC for buying but may be searching for IMs that already have *selling* peers attached. Thus they may end up taking part in an interaction that their OKC was not specifically designed for. This flexibility is crucial to the vision and feasibility of the OpenKnowlege project but it leads to the problem of semantic mismatch. Therefore, we have developed a *matching component* that can match the first-order constraints in an IM to the first-order methods within an OKC.

  This matching process returns a value in [0 1] indicating how similar the constraint and the method are. This allows the peer to form its own notion of *good enough* and determine whether this match meets this standard. If the match between a method and a constraint is below a certain threshold (determined by the peer), it may decide that this method is not good enough to satisfy the constraint and either decide not to play that role or decide to search for an OKC that better matches the role.

  For more details of this, see Good Enough Answers. [6]

- **Peer Ontologies**

  OpenKnowledge does not place any restrictions on or have any expectations of peers. Anything that is able to run the OpenKnowledge kernel and to solve constraints can act as OpenKnowledge peer. Therefore, we cannot either have any expectations of what ontologies these disparate peers will have and how they will be represented: peers are fully autonomous and must make their own choices.

  However, there are many reasons why having a well-formed ontology will allow the peer to perform more successfully in interactions. Firstly, some of the machinery of OpenKnowledge - the matching service and the trust component - rely on peers having a taxonomy of terms. Both of these services will work without this - especially the matching service, where use can be made of facilities such as WordNet - but their success rate will be very low if the peers cannot provide any semantics. Additionally, peers may find it hard to satisfy constraints even if the matching process is successful if they do not have well ordered knowledge.

  Therefore, we would recommend that peers have at least a basic taxonomy to represent their knowledge.

- **Services as Peers**

  This page [7] tells you how services can be used as peers.

Accessing 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.

The scope of an OKC instance is a single interaction: it cannot share data with runs of other interactions. However, an OKC often needs to access persistent information. For example, an interaction for the purchase of some product may need to access the peer catalogue, and may change the state of the peer's warehouse. An issue to consider in defining the access to the peer from the OKC is that OKCs and peers may be developed by different entities, and there cannot be many assumptions. The mechanism for accessing the peer state, therefore, relies on adaptors.

An OKC declares the methods it needs to access in the peer using the Java5 annotation *@RequiredPeerAccess*, listing the method signatures in the peer:

```
@RequiredPeerAccess(
 methods={"doSomething(name)","doSomethingElse(text)"}
)
```

The peer will have a class implementing the dummy interface *PeerAccess*, where exposes the methods it provides to the OKCs. The methods must be annotated with @MethodSemantic, and the arguments are all of type *Argument*.

```
class MyPeerAccess implements PeerAccess{
  @MethodSemantic(
    language="tag",
    args={"name"}
  )
  public boolean doSomething(Argument n){...}

  @MethodSemantic(
    language="tag",
    args={"text"}
  )
  public boolean doSomethingElse(Argument t){...}
```

The peer, in the initialization, will instantiate its PeerAccess class, and pass it to the OKManager calling the method *setPeerAccess()*. The manager will compare the requests of OKCs requiring access to the peer with the exposed methods in the PeerClass. OKCs whose requirements do not match what the Peer provides cannot be added to the OKC storage. By comparing the OKC requirements with the PeerAccess class, the manager generates an adaptor, that is then passed to every instantiated OKC.

When an OKC needs to access the peer, it will call the method *invokePeer(methodName, Argument...args)*:

```
try {
  invokePeer("doSomething", new ArgumentImpl("name","testname"));
} catch (AdaptorException e) {
  e.printStackTrace();
}
```

Important Notice
One important element to notice is that the Arguments passed in the OKC methods are mapped to constraint parameters, while the arguments passed to the peer methods are mapped to the parameters in the peer's methods: therefore it is **not possible** to use the arguments received in the OKC method in peer's methods invocation.

The following example is wrong:

```
@MethodSemantic(language="tag", args={"word"})
public boolean okcMethod(Argument a){
  try {
    invokePeer("doSomething", a);
  } catch (AdaptorException e) {
  e.printStackTrace();
  }
}
```

If the argument is an *input* argument, the correct solution is:

```
public boolean okcMethod(Argument a){
  try {
    invokePeer("doSomething", new ArgumentImp("word",a.getValue()));
  } catch (AdaptorException e) {
  e.printStackTrace();
  }
}
```

If the argument is an *output* argument, the correct solution is:

```
public boolean okcMethod(Argument a){
  try {
    Argument lw = new ArgumentImp("word");
    invokePeer("doSomething", lw);
    a.setValue(lw.getValue());
  } catch (AdaptorException e) {
  e.printStackTrace();
  }
}
```

Specific case: starting a new IM from another IM
An interesting possibility for an OKC is to ask the peer to start a new IM. Here is a simple implementation that exploits the InteractionTask class.

In the Peer access class:

```
class MyPeerAccess implements PeerAccess {
 protected Peer peer;

 public MyPeerAccess(Peer p){ peer=p; }

 ...

 @MethodSemantic(language="tag", args={"role_name","interaction_description"})
 public boolean attemptIM(Argument rn, Argument imd){
   InteractionTask task = new InteractionTask(this.peer, rn.getValue(), AcceptPolicy.ACCEPT_ONE,
false);
   String[] q = (String[]) imd.getValue();
   task.attempt(q);
 }

 ...
}
```

To call the method in peer from an OKC:

```
@RequiredPeerAccess(
 methods={"attemptIM(role,description)"}
)
public class MyOKC extends OKCFacadeImpl {

  public void someConstraint(...){
    try{
      invoke("attemptIM", new ArgumentImpl("role", "buyer"), new ArgumentImpl("description","pur
    } catch (AdaptorException ae){
      ae.printStackTrace();
    }
  }
}
```

accessing peer state

Source URL: http://www.openk.org/beingAnOKuser/okpeers/state

The Trust Component

- **Overview**

  This differs from other elements we have discussed so far because this is not part of the kernel and its use is entirely optional for OpenKnowledge peers. Some peers may choose not to use any trust model, some may choose other trust models that may exist or may wish to write their own (see below for information about how to do this). We believe that our trust model is useful in many scenarios but freely admit that it is not the most appropriate model in all situations.

  On this page, we briefly introduce the general notions of our trust component. For further details and for the algorithms, please see Deliverable 4.5 [4] and Trust in P2P. [5]

- **What Does the Trust Component Provide?**

  The trust component keeps records of all previous interactions, which can be used to evaluate how well a potential peer is likely to play a role. In addition, these stored records can be shared with other peers through gossip, and gossip can be used in the trust calculation - in addition to personal experience, or in place of it if no personal experience exists - and can be weighted according to the provenance of the trust information.

  When considering how well a peer is likely to play a role, there are two factors to consider:
  1. is the peer capable of playing that role?
  2. is the peer willing to play that role?
  Since any peer can play any role in which it can satisfy the constraints, observation of previous constraints that have been satisfied can help us to answer the first question. However, there are two different levels of peer ability in playing a role:
  a. satisfying the constraints;
  b. providing good quality goods/information/etc. (depending on the type of role).
  Complying with the first condition is the minimum requirement of successful interaction, and observation of previously satisfied constraints can answer this

question (fully or partially). The second condition is more nebulous and can only be answered by considering user feedback in identical or similar situations.

The second question is more difficult, and depends on factors such as the good intentions of the peer. To answer this question, context is more important. A peer that has performed well in a similar situation is more likely to perform well than one that has only performed well in a very different situation.

The trust algorithm considers all these factors and uses a probabilistic measure to produce a score in [0 1] indicating the trust value for a peer playing a particular role in a particular IM.

- **What Does the Trust Component Require?**

There are two kinds of information about how well a peer performed its role in an interaction:

- information about whether an interaction terminated or not.
This can be generated automatically by the coordinator of an interaction. It cannot provide information to help assign blame for the failure of an interaction to terminate: all peers involved in the interaction are blamed equally. This means that in isolation, it does not provide a reliable judgement. However, over many interactions, it is more useful. An able, well-meaning peer may sometimes interact with failing peers and thus be involved in a failed interaction, but most of the time its interactions will terminate successfully.

- information about whether the service was performed well.
This may, depending on the situation, mean providing high quality goods, appropriate and correct information, and so on. This usually cannot be judged through automated observation of the interaction. This is because such considerations are outside the scope of an interaction. Expectations of quality would not normally be expressed as a constraint and cannot usually be judged until after the interaction terminates and the goods or information are actually received and/or used. Additionally, such a judgement is using subjective and depends on the judgement and perhaps taste of the user. Therefore, this information must be given directly by the user.

Once the interaction has successfully terminated, the trust component will cause a small window to pop up in the OpenKnowledge GUI to remind the user that feedback is pending on this interaction (if the interaction terminated unsuccessfully, the user will be informed and no further feedback will be necessary.) The user can ignore and dismiss this request or, at an appropriate point (for example, after the goods have been delivered if the interaction involved buying goods), can provide feedback about the quality of the service provided. If this feedback is provided, it will be used by the trust module as part of the stored information about the transaction and used to judge expected performance in future interactions.

- why and when would you prefer to use another trust model?

Our model is based on frequent interactions. In order for the values produced to be meaningful it is necessary to have several previous interactions to refer to. This does not all have to be personal experience as much useful information can be received through gossip. However, in a domain where interaction is sparse, there may only rarely be enough information to make the output from this trust component useful. Once example of such a domain is bioinformatics.

Additionally, in some particularly communities, trust is judged in very different ways and peers involved in such communities may wish to judge trust in their own way.

the trust component

Source URL: http://www.openk.org/beingAnOKuser/trust

Links:
[1] http://www.openk.org/beingAnOKuser/trust #overview
[2] http://www.openk.org/beingAnOKuser/trust #trustProvide
[3] http://www.openk.org/beingAnOKuser/trust #trustRequire
[4] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D4.5.pdf
[5] http://www.openk.org/research/tpp

Publishing IMs

Overview

The section on Using LCC [1] will tell you all you need to know to create an IM.

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.

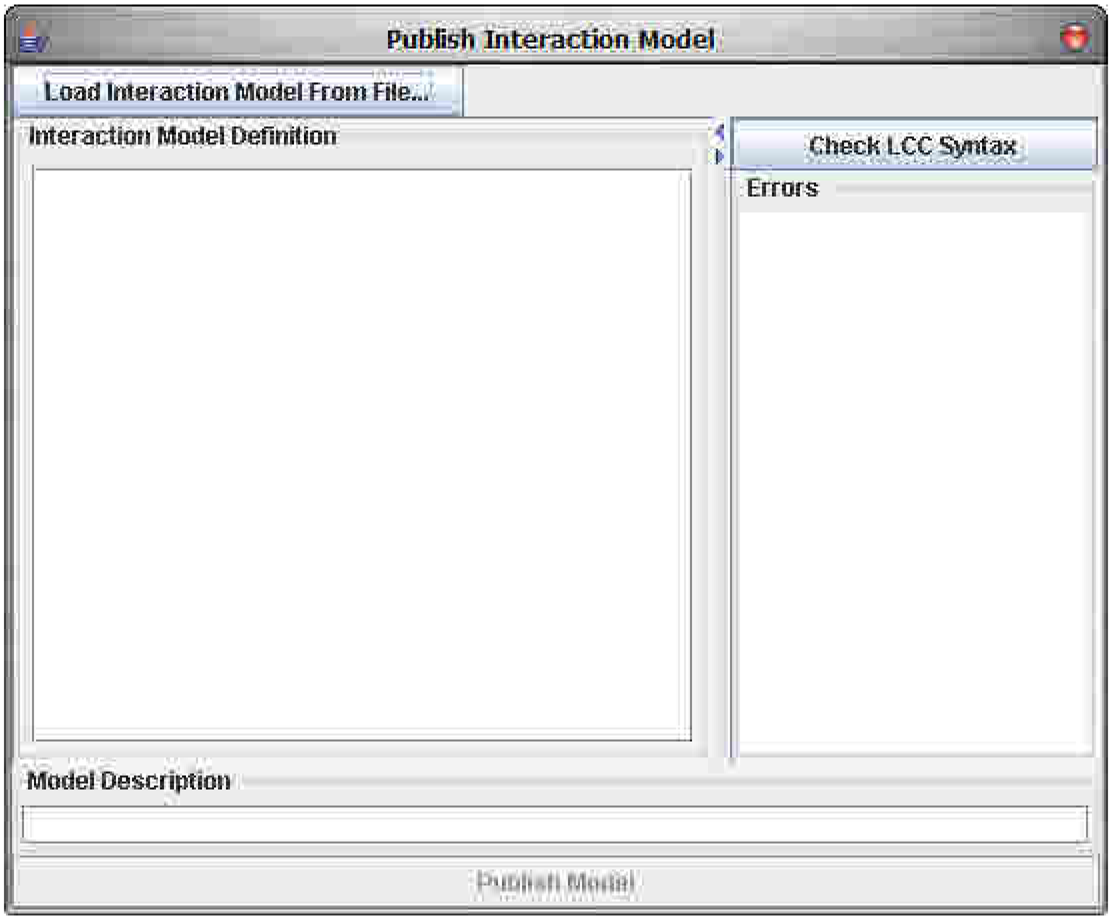

Publish Interaction Model
Load Interaction Model From File...
Interaction Model Definition
Check LCC Syntax
Errors
Model Description
Publish Model

Figure 1. The `Publish Interaction Model' dialog box

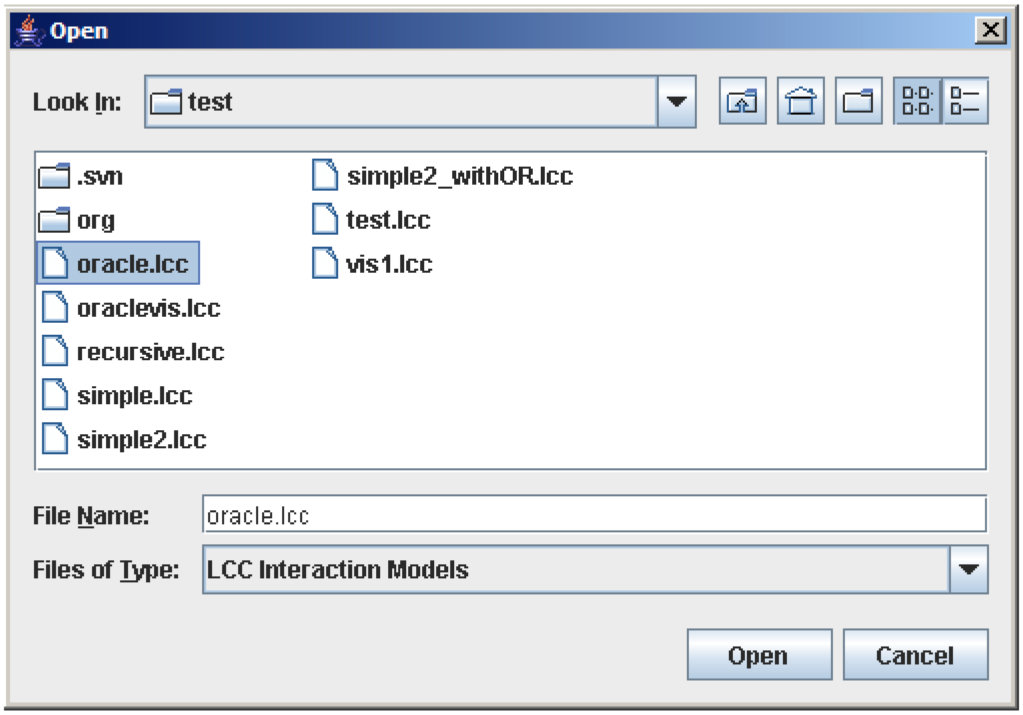
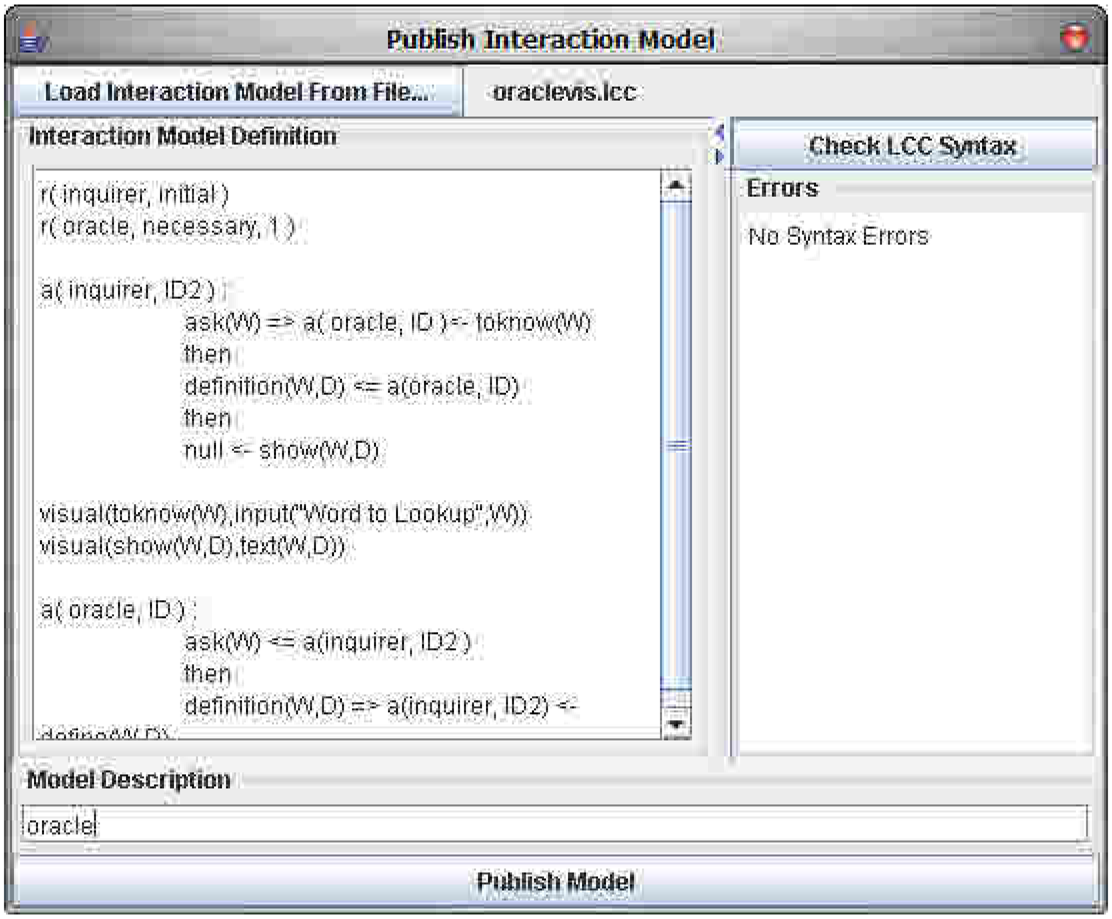


Figure 2. Selecting an interaction model to publish

Figure 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 2 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).


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

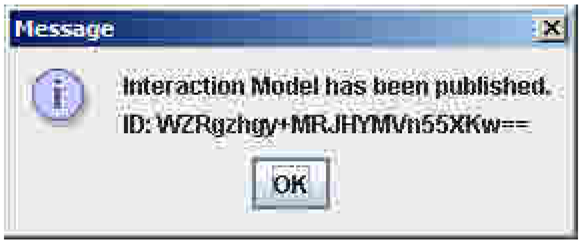
```
r( inquirer, initial )
r( oracle, necessary, 1 )

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) <-
```

No Syntax Errors

oracle

Figure 3. Giving descriptive tags to the interaction model

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



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

Figure 4. Successfully published interaction model

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 4).

Source URL: http://www.openk.org/beingAnOKuser/publishing/publishingIMs

Links:
[1] http://www.openk.org/beingAnOKuser/usingLCC/examples

Creating and Publishing OKCs

- Programming New Components

In the informer/requester IM [1], there is one constraint to be satisfied: *query(X1)*. The constraint simply defines 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 1.1. gives the skeleton for an OpenKnowledge Component.

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

public class MyOKC extends OKCFacadeImpl
{
}
```

Listing 1.1. 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 1.2. shows an example of a full implementation of the stock checker OpenKnowledge component.

```
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 1.2. 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.

- 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.

- 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 *Publishing Components* below. The component you have loaded will appear under the "Local Components" under 'My Peer'.

- 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 *Loading in Local Components* above) 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 on Providing Alternate User Interfaces [2]).

Visualisation [3] described how visualisations are incorporated into interaction models. They utilise the *visual(,)* operation. The LCC below shows as example of the visualisation introduced earlier.

```
visual(getMyName( N), qask("Please enter your name",  N))                (1.14)
```

The first part of the visualisation definition is the interaction model constraint (getMyName (*N*)), and the second part is the *visual term* (qask("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 Visual Constraints [4] for a list.

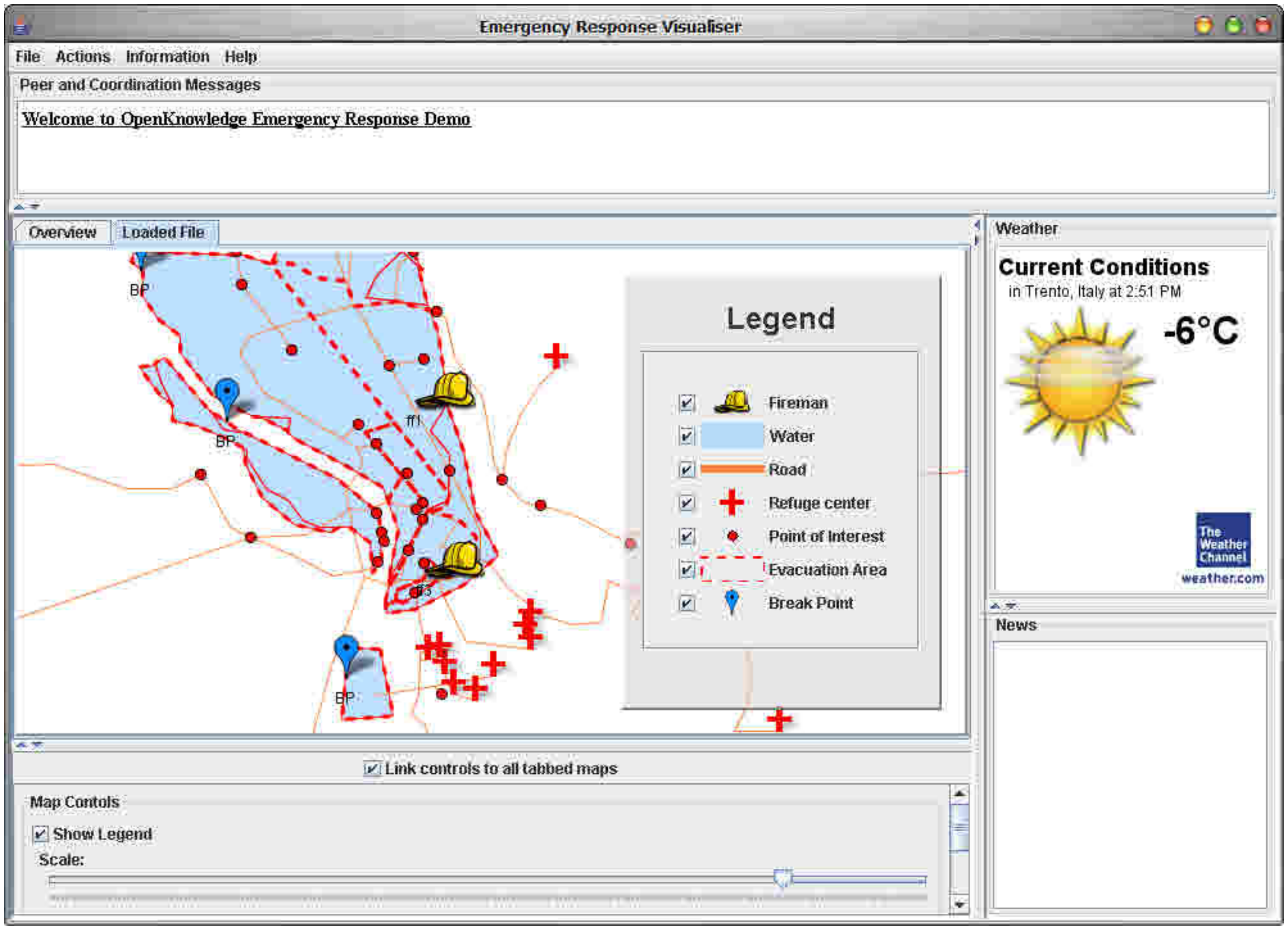
Source URL: http://www.openk.org/beingAnOKuser/publishing/publishingOKCs

Links:
[1] http://www.openk.org/howOKworks/im
[2] http://www.openk.org/beingAnOKuser/publishing/interfaces
[3] http://www.openk.org/beingAnOKuser/usingLCC/visualisation
[4] http://www.openk.org/beingAnOKuser/usingLCC/syntax/ #visCons

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 1 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.



Welcome to OpenKnowledge Emergency Response Demo

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

✓ Link controls to all tabbed maps

✓ Show Legend

Figure 1. Emergency Response User Interface

Source URL: http://www.openk.org/beingAnOKuser/publishing/interfaces

Writing Your Own Trust Component

The best way to design your own trust component will vary greatly depending on what sort of domain you are designing it for and what the reasons are that you wish to move away from the trust component already provided with OpenKnowledge. We therefore cannot give you much direction in this: you must consider what trust means in the domain in which you are interested.

However, any trust model must consider two questions:
- how do we determine how trustworthy another peer is?
- how do we use this information?

In the existing OpenKnowledge trust component, the former question is answered through looking at previous interactions and making judgements on the basic of past performance. However, for some domains this may be much less automated: for example, a peer may be trusted if it represents a known source of information that is trusted within the community. In this situation, users may be unwilling to depend on automatically generated trust scores and prefer to go with community norms. In other situations, more simple evaluation techniques such as page rank may be appropriate. In order to use a trust component, you must have some way of extracting this knowledge and providing it to peers who must choose who to interact with.

The answer to the latter question is that information must be used during the pre-interaction phase (this page [1] describes this) when determining which other peers to interact with. This is how any trust component links in to the OpenKnoweldge system: it is used to form part of the decision to reject or accept potential players in an interaction. For some domains, this could be as simple as providing the peers with a list of other peers that they can interact with: any other peer will be refused. In other domains, this would be a more complicated, situational calculation.

Therefore, what is required of any trust model is:
- to provide a mechanism for evaluating trust (which may be fully manual, fully automated or interactive) and to provide this information to one's own peer, or allow this information to be shared as appropriate;
- to provide a mechanism for peers to use this information appropriately when determining who they are prepared to interact with.

Source URL: http://www.openk.org/beingAnOKuser/publishing/writingtrustcomps

Links:
[1] http://www.openk.org/howOKworks/lifecycle

OK Design Support

Overview

The OpenKnowledge system makes knowledge more easily shared in open systems by always sharing knowledge in the context of a formal model of the interaction process that has stimulated the knowledge sharing. The language we have chosen to represent the interaction process is LCC so LCC specifications have to be produced from somewhere. In the long term we want to make that production process as fast and straightforward as possible because, for the OpenKnowledge approach to flourish, it is necessary for large numbers of potentially useful interactions to be described. This document describes three ways of facilitating this:

- Through the use of patterns and high level languages to raise the level of description of LCC (Structured Design). [1]
- Via automated analysis of LCC specifications, prior to deploying them (Interaction Model Analysis). [2]
- By relating LCC to other systems of design (and process representation) that have already developed design communities in their domains of application (Connecting to Other Systems of Design). [3]

The solutions we present in each of these areas are not definitive or exhaustive (applied methods seldom are) but they demonstrate what can be done. For each of the methods in each area we summarise the idea in general and connect detailed technical reports (plus source code where appropriate) of the specific way in which we developed it. This, provides a resource down to coding level for those who want to replicate or extend our efforts.

Source URL: http://www.openk.org/beingAnOKuser/design/overview

Links:
[1] http://www.openk.org/beingAnOKuser/design/structured
[2] http://www.openk.org/beingAnOKuser/design/analysis
[3] http://www.openk.org/beingAnOKuser/design/connect

Structured Design

- **1. Structure Editing for LCC**

  The most direct way to assist in constructing LCC is by providing an editor in which editing operations are based on structural patterns that are meaningful in terms of the engineering of the specification. For example a common pattern for specifying recursive roles is as follows:

  ```
  a(F(A1...An),X) ::
       ( then a(F(A1...An-1,AnN),X))
       or
       null <-
  ```

  where: F(A1...An) is a role definition with functor F and arguments A1...An; the new argument AnN is derived as a consequence of the earlier definition in the recursive part of the definition; and the base case is determined by some constraint, .

  Patterns like the one above provide skeletal definitions for LCC specifications that can then be elaborated using further editing operations. This view of structured design is similar to the idea of techniques editing for logic programs. A basic structure editor for LCC is described in detail, along with examples of the system in operation, in our technical report on techniques editing [1].

  Structure editors of the sort described above provide assistance in design but assume that those being assisted are interested in manipulating the target language (in our case LCC) directly; the engineer is always aware that he or she is working on a LCC specification. There are other forms of editing where this need not be the case and in the next section we consider one of those.

- **2. Finite State Based Editing for LCC**

  LCC is a process language in which the definition of a process orders message passing events. Consequently, there is no explicit representation in LCC syntax of the space of states that can be encountered in an interaction; this space can be inferred from a LCC specification, rather than being directly described by it. Some systems of design, however, start from a finite state model of interactions, in which the different states of the interaction are explicitly represented and events in the interaction appear as transitions between these states. One such style of specification in the multi-agent systems community is that of electronic institutions. Here the nodes represent the different states of the conversation and the directed arcs connecting the nodes are labeled with the actions that make the scene state evolve. The Electronic Institutions Development Environment (EIDE) is a tool for describing electronic institutions, with a translator that produces, automatically, LCC from the electronic institution specification. This means that designers who prefer a state-oriented rather than a process oriented view of interaction can still contribute to OpenKnowledge.

The mechanism for translation from electronic institutions in EIDE to LCC is described in Section 5 of the OpenKnowledge report on visualiser components and visual authoring tools [2]. The basic idea is that definitions for the sequencing in LCC definitions of roles correspond to traces through the finite state machine of the electronic institution. This translation does not preserve some of the distinctions made in an electronic institution model because electronic institutions have, as primitive, a concept of scene composition that is absent (for reasons of parsimony) in LCC. The relationship between electronic institutions and LCC is described in detail in the OpenKnowledge report on Ambient LCC [3].

- **3. Generating OpenKnowledge Components (Groovy)**

  In Sections 1 and 2 we described systems to support the construction of LCC specifications. In order for interactions to do useful work, however, it is necessary to supply OpenKnowledge components that can be used by peers to satisfy the constraints in an interaction. Although component design inevitably involves some application programming, we can make this programming simpler by supplying a higher level, Java-compatible language targeted at component design.

  Groovy [4] is an agile and dynamic language for the Java Virtual Machine, it builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and other scripting languages. Groovy also increases developer productivity by reducing scaffolding code. It integrates with all existing Java objects and libraries seamlessly, therefore Groovy can be used to create OKCs that run within the OK kernel. Moreover Java developers are able to use Groovy with almost-zero learning curve, and it's also relatively easy for novice programmers to learn and use, enlarging the group of programmers able to write OpenKnowledge components themselves.

  To demonstrate Groovy in use for component definition we applied it to one of our bioinformatics service coordination examples. The technical details of this (along with an illustrative example) are in our technical report on Groovy [5].

Source URL: http://www.openk.org/beingAnOKuser/design/structured

Links:
[1] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/structure-editor.pdf
[2] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/d54.pdf
[3] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/ambientlcc.pdf

[4] http://groovy.codehaus.org/
[5] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/groovy.pdf

Interaction Model Analysis

Like all sophisticated process languages, LCC specifications can be complex so it is not always easy for the designer of an interaction to be certain that the specification generates the interaction that he or she had in mind when writing it. To raise confidence that it does behave as intended it is useful to be able to explore the LCC behaviours prior to deploying the interaction on the OpenKnowledge system. There are numerous ways of doing this but here we explore three of these: the first is simulation via trace generation from the LCC specification; the second is temporal property checking of these traces; the third is the inclusion of a (virtual) real-time environment in the simulation.

- **1 Generating Behavioural Traces (Meta-Interpretation)**

  LCC is an executable specification language and the style of execution of LCC in the OpenKnowledge kernel is based on the idea of unfolding the clauses of each peer's role definition as a means of representing change in the state of the interaction. Although the kernel is Java based for portability, Prolog is a more elegant language in which to describe unfolding. For this reason the behavioural trace generator (and the other related LCC mechanisms in Section 3) are implemented as Prolog meta-interpreters. A detailed description of the process of unfolding to generate a trace is given in the LCC operational semantics definition [1]. The basic idea, however, is that the meta-interpreter "walks" through the role definitions, sending messages in sequence and simulating concurrency via non-deterministic choice. For example, the LCC interaction model:

  ```
  a(r1, X) ::
      ( m1 => a(r2, Y) or m2 => a(r2, Y) ) then
      M <= a(r2, Y).

  a(r2, Y) ::
      ( m1 <= a(r1, X) then m3 => a(r1, X) ) or
      ( m2 <= a(r1, X) then m4 => a(r1, X) ).
  ```

  would be capable of generating the following two traces for peer p1 in role r1 and peer p2 in role r2:

  ```
  [m(p1, m1 => a(r2, p2)), m(p2, m1 <= a(r1, p1)), m(p2, m3 => a(r1, X)), m(p1,
  m3 <= a(r2, p2))]
  [m(p1, m2 => a(r2, p2)), m(p2, m2 <= a(r1, p1)), m(p2, m4 => a(r1, X)),
  m(p1, m4 <= a(r2, p2))]
  ```

  where each element of the trace above is either a message, M, being sent from peer

S to peer R (m(S, M => R)) or is a message being received by peer R from peer S (m(R, M <= S)).

The source code for a generating traces from LCC specifications (along with some example specifications can be downloaded as a zipped folder [2].

This sort of simulator is useful both for exhaustively exploring the state space for interactions (a ropic of the following section) and for running multiple simulations of interactions with random selections of events at choice points in the trace generation. This latter method was used in developing our peer rank algorithm (used in our bioinformatics testbed) because it gave us a way of rapidly running thousands of interactions in concert with the peer rank reputation mechanism without having to set up a much more complex (and less easily controlled) test harness for the OpenKnowledge kernel. The peer rank algorithm eventually was provided as a service for the kernel but only after this initial testing phase. A description of peer ranking and some of the simulation results appears in our technical report on peer rank simulation [3]. The full source code for the peer rank simulator, plus test interactions, downloaded as a zipped folder [4].

- **2 Checking Temporal Properties of Interactions (Tabled Resolution)**

  In Section 1 our concern was to be able to generate individual traces corresponding to a permitted behaviour in an interaction specification. Sometimes, however, we are interested in knowing whether some temporal property can occur across the space of all interaction behaviours (for example if a particular message is always eventually followed by some other particular message; or if a given sequence can never occur).

  A basic temporal property checker that utilises a trace generator, as described in Section 1, applied to business process examples is described in our technical report on constraint verification [5]. This provided a way of checking the following properties of traces (each of these is given formally in Section 5 of our technical report):

  - Safety : If B represents some undesirable condition, a model is said to satisfy the safety property with respect to B, if in all the runs of the model, the condition B is never satisfied.
  - Liveness : If A represents some desirable condition, a model is said to possess the liveness property in any run, if at some point in the run the condition A is met.
  - Deadlock : indicates a situation in which two or more processes wait for each other and are unable to proceed on their tasks as each is waiting for the other. For LCC, each process waiting for a message
    from the other process before it can proceed with its part of the interaction gives a deadlock.
  - Correctness: aims at meeting all the functional requirements expected of the given model, where the conjunction of all constraints is implied by the conjunction of all the property specifications and
    the conjunction of all constraints holds in the integrated behavioural model.
  - Termination : requires the completion of all roles in an interaction. In LCC this

means that all the roles associated with the messages of an interaction should be closed.

Although our basic temporal property checker can analyse important properties of LCC interaction specifications, it explores the search space of possible interaction traces using a standard Prolog search strategy. This limits the efficiency of search space exploration because it involves a high proportion of redundant search. In OpenKnowledge we addressed this problem by using a tabled resolution based Prolog system: XSB [6]. This allowed us to perform more complex forms of property checking on larger LCC specifications. More surprisingly, it allowed us to perform limited but useful forms of property checking in only a few seconds of real time which makes it possible to check LCC interaction models not only in advance of their deployment (the traditional approach) but also, in some circumstances, during their deployment. This provides a novel form of trust-related verification. The XSB-based property checker is described in detail in our technical report on runtime verification of trust models [7] and the code used to implement it can be downloaded as source code [8].

- **3 Virtual Environments (Unreal Tournament)**

All of the analytical methods described so far in this section assume that the environment on which interactions is run need not be modelled as part of the analysis, other than in terms of constraints satisfied by the LCC interaction model (or, as in Section 2, by a combination of LCC and service specifications). In some cases, however, we are interested in detailed simulation of environments. This is especially the case when we want to assess performance of LCC as a coordination medium in real-time systems, where response times in a rapidly changing environment are of utmost importance. To perform these sorts of analyses one needs a simulator for the dynamic environment. A popular source of this sort of simulator comes from the computer gaming world where commercial success has depended on providing semi-realistic environments in which to play.

One of the standard gaming simulators is Unreal Tournament [9] which is a popular gaming environment in its own right but also provides an accessible game engine that can be used by developers to introduce automated game playing agents ("bots" in UT jargon) into the game. It also provides a rich source of complex virtual environment topologies, courtesy of the environment design community that has built up around the game. We have built a means of linking a LCC interpreter to UT-bots so that LCC can be used to coordinate message passing between them. This allows us to use LCC to define collaborative strategies for game playing which we then test by playing teams of coordinated "LCC-enabled" bots against teams of individually superior but uncoordinated conventional bots. We have been able to produce remarkably fluid and effective team play by this means.

A description our most recent work with the Unreal Tournament environment is available as a Quicktime movie [10] with a set of accompanying notes [11]. The second half of this video shows the environment in action (note that the movie is a comparatively large file, 75MB, so may take several minutes to download). The bots that you see in this video are highly autonomous,using machine learning algorithms fed by data from the environment to develop their individual behaviours as the game proceeds. The LCC being used to coordinate them is simple and reactive (in fact the system uses a reactive subset of LCC for speed of response) as can be seen from the accompanying notes. In the long term, we hope that this sort of architecture could develop into a framework for behaviour-based software system development analogous to the subsumption architectures used to combine behavioural modules in robotic systems.

Source URL: http://www.openk.org/beingAnOKuser/design/analysis

Links:
[1] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/simulator/unfolding.pdf
[2] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/simulator/basic-simulator.zip
[3] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/simulator/peer-rank-simulations.pdf
[4] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/simulator/okrank.zip
[5] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/rudradevaru.pdf
[6] http://xsb.sourceforge.net/
[7] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/model-checking/model-checking.pdf
[8] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/model-checking/code.zip
[9] http://www.unrealtournament3.com/
[10] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/UT-movie.mov
[11] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/UT-movie-notes.pdf

Connecting to Other Systems of Design

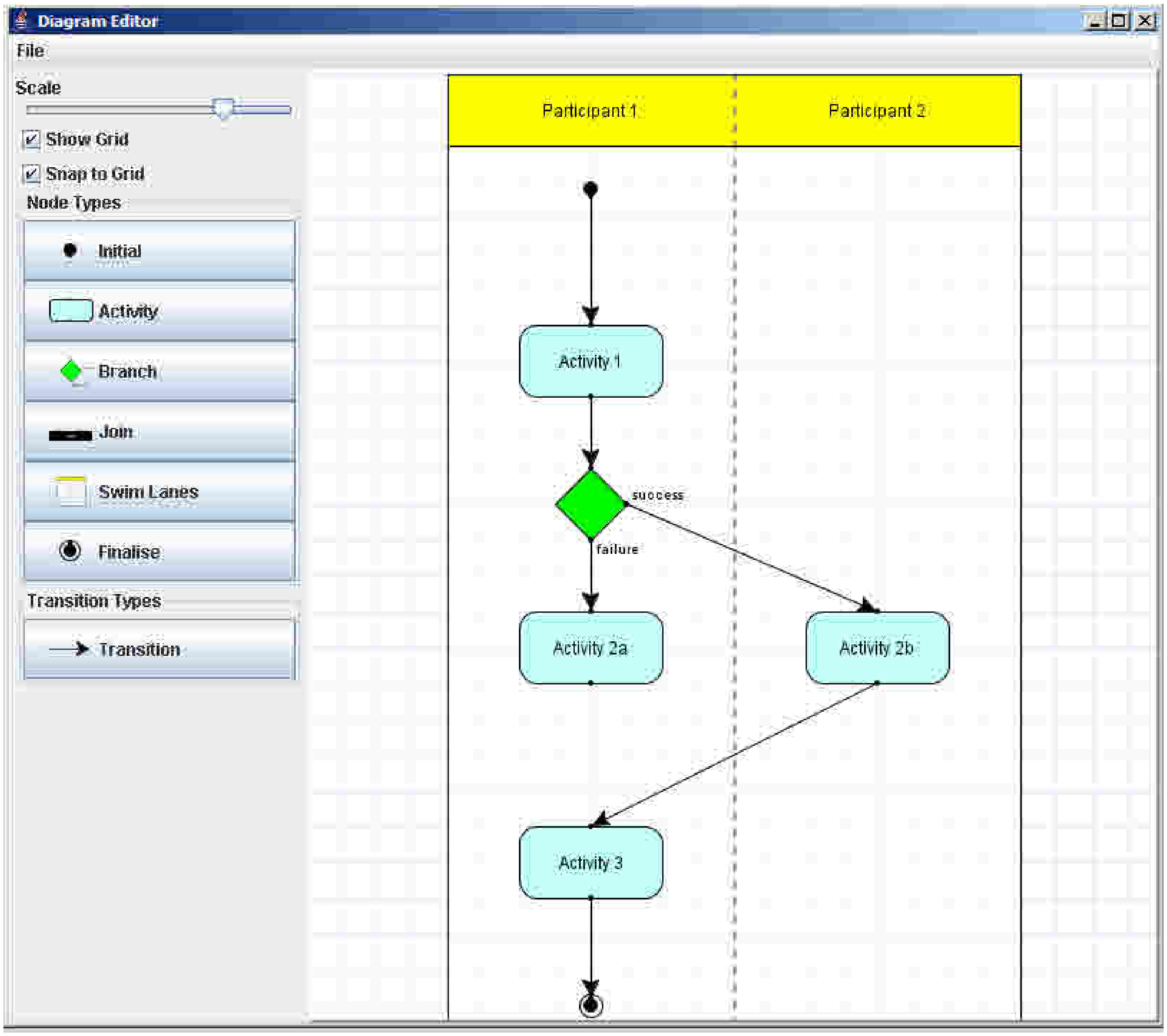
LCC is as lightweight and parsimonious as we can make it. This does not mean, however, that it is straightforward for every engineer to use. Application domains develop engineering cultures that often take as a focus particular styles of design, supported by task-specific design notations. Thousands of these have developed and many continue to be invented so it is impossible to catalogue exhaustively the relationship of all these to LCC. Instead, we give examples below of ways to connect OpenKnowledge to established systems of design. First, in Section 1, we demonstrate the most direct route, via translation from a more traditional language to LCC. Then, in Section 2, we consider the case where the traditional language to which we wish to connect is providing a different functionality from that of LCC so extension of LCC is required to embrace it. In Section 3 we explain the more radical alternative of writing, in LCC, an interpreter for an established language. Finally, in Section 4, we discuss the most radical step of all - to replace LCC in the OpenKnowledge kernel with an alternative process language.

- **1. Translation to LCC From Established Languages (UML, SCUFL)**
  - **1.1 Translation to LCC from UML Activity Diagrams**

    The LCC coordination calculus lends itself well to display in a graphical form. Its concepts of participants, messages and constraints, are similar to that of the UML activity diagram [1] that has partitions, flows and activities. As UML is a common tool for the design of software that is well used in industry, by providing a conversion between UML diagrams and LCC, we are able to minimise the intellectual cost of joining the OpenKnowledge network for many industry software houses and developers.

    UML editors are relatively common, yet they tend to focus less on data flow than data design. There are very few open source UML editors and those we found did not provide activity diagram support. Commercial products would be very hard to augment for LCC output without considerable cost and/or reverse engineering. So, as one of the support tools for the OpenKnowledge project we have implemented a basic UML activity diagram editor that has the option of outputting LCC code. The editor itself has been designed in a modular, extensible way so that other open-source developers could extend it such that it would handle other diagram types. However, we have, for now, only implemented the basic UML activity diagrams.

A screenshot of the editor appears below. Various node types are displayed on the left of the editor and they are added to the diagram by clicking their appropriate button. Once in the editing area, they can be dragged around and their properties edited by double clicking.



The conversion is achieved by searching for the initial node (the black dot) and

Transitions are considered LCC sequence ("then") statements unless they cross the partition swimlanes, in which case they are rendered as message sending and receiving. Activities are translated as constraint satisfactions. The branch nodes are rendered as "or" branches in the LCC.

The LCC is rendered by traversing the graph and outputting the appropriate LCC for the node encountered. The LCC is buffered so that when "or" nodes are encountered, the LCC can be re-written. Back-tracing of the graph is done at various points to ensure that activities that occur after branches in other participants are rendered correctly.

The LCC below shows the LCC export for the diagram shown in the UML diagram above:

```
// ----------------------------------------------------------------------------
// LCC File generated by UML-to-LCC exporter.
// test.lcc
// 24/10/2008
// ----------------------------------------------------------------------------
r( participant1, initial )
r( participant2, necessary )

// ============================================================
a( participant1, ID ) ::
(
        null <- activity1() then
        msg() => a( participant2, Participant2ID ) then
        msg() <= a( participant2, Participant2ID ) then
        null <- activity3()
)
or
(
        null <- activity2a()
)

// ============================================================
a( participant2, ID ) ::
        msg() <= a( participant1, Participant1ID ) then
        null <- activity2b() then
        msg() => a( participant1, Participant1ID )
```

The implementation of the basic UML editor and translator to LCC can be downloaded as source code [2].

- **1.2 Attaching to Existing Design Systems via Translation (Taverna)**

One of the main testbeds for the OpenKnowledge system is in bioinformatics. Although this domain of application is comparatively new, there already exist accessible design tools for bioinformatics workflow, especially in Grid systems. One of the best known systems is Taverna [3] (although there are others, such as Kepler and Triana). The Taverna system provides a visual editor for describing workflows to be enacted on a Grid system and produces a specification of the workflow in the SCUFL process language. For the sector of the bioinformatics community interested in using Taverna, the easiest way to connect to the OpenKnowledge system would be to continue to use their familiar design tools but, instead of using a Grid system for workflow enactment, to use the OpenKnowledge system.

To make a switch to OpenKnowledge as straightforward as possible for this community we built an automatic translator from SCUFL to LCC. This does not disrupt in any way the current methods of use of the Taverna system (which we assume are well honed to the Taverna community) but simply provides an additional step beyond the traditional endpoint of Taverna specification (which is a workflow specification in SCUFL) to LCC. The SCUFL to LCC translator is described in detail in our technical report [4] and the Java code used to implement it can be downloaded as source code [5].

- **2. Connecting LCC to Compatible Languages With Different Functionality (OWL-S)**

  In the previous section our approach to integrating with an existing design system was automatic translation, with the aim of altering established design practice as little as possible. Some systems of design, however, are oriented to a different problem than the one being addressed by OpenKnowledge and then the issue becomes one of complementarity: establishing connections such that the two systems can be used together to mutual advantage. An example of this is the OWL-S [6] language for semantic web service specification.

  OWL-S is essentially a typed language for specifying input/output interfaces to Web services. It's use of OWL [7] as a type language gives it a strong connection to semantic web efforts. OWL-S, however, deliberately avoids prescribing any language in which the processes to choreograph services might be specified (so as to remain neutral to choices in choreography language). Conversely, LCC deliberately avoids commitment to a service specification language (so as to remain neutral to service specification infrastructure). LCC and OWL-S therefore tackle service choreography from different perspectives.

  As an experiment in combining LCC and OWL-S, we built a prototype discovery system for services when enacting LCC interaction models. This involved adding type annotations to constraints in LCC interaction models and, from these annotations, automatically extracting service descriptions that could be matched to OWL-S service specifications using a Description Logic reasoner. The relationship between OWL-S and LCC is described in detail in our technical report [8] and the Prolog code used to implement his OWL-S based discovery system for LCC (and the DL reasoner) can be downloaded as source code [9].

- **3. Writing an Interpreter for an Established Language in LCC (BPEL4WS)**

  In Section 1 we gave an example of translation as a means of linking LCC to a task-specific design system. In Section 2 we gave an example of connecting to a complementary task-specific language via annotations in LCC. We now introduce a third way to bring a task-specific language into the sphere of OpenKnowledge: by writing an interpreter for the language in LCC.

Although it is unconventional to use one protocol language as an interpreter for another, a similar idea - that one can use a declarative language to write a meta-interpreter for another language - is quite conventional in declarative programming. As an example, we chose the Business Process Execution Language for Web Services (BPEL4WS [10]) which is an industrially used language for specifying business interaction protocols. Instead of writing a translator from BPEL4WS to LCC (the route we chose with UML and SCUFL in Section 1) we wrote a protocol in which the principal role is to act as a BPEL4WS interpreter. A BPEL4WS specification is given as a parameter to this role and enacting the role interprets the BPEL4WS specification to produce appropriate message passing and invocation of services. The LCC specification needed for this is, of course, complex but for BPEL4WS users this complexity is no more apparent than in any other system for enacting BPEL4WS. Conversely, from an LCC point of view, the BPEL4WS interpreter is just a normal (though complex) interaction model.

The LCC interpreter for BPEL4WS is described in detail in Chapter 5 of our technical report on enacting decentralised workflow [11] and the same source also demonstrates, in Chapter 4, the more conventional route of translation from BPEL4WS to LCC.

- **4. Replacing LCC with an Established Language (WS-BPEL)**

OpenKnowledge chose LCC as its core language because it provided a parsimonious yet powerful language with strong links to more abstract, generic process specification and declarative languages. Nevertheless, we have always recognised that other process languages could have been substituted for LCC and we have made the OpenKnowledge kernel system as independent as we could from specific choice of core language. This raises the possibility, for those who prefer a core language in some style other than LCC, to replace LCC with a process language of choice. We have demonstrated this with the Web Services Business Process Execution Language (WS-BPEL [12]).

This approach to integrating OpenKnowledge with other design systems requires much deeper understanding of the OpenKnowledge system, since it requires the LCC interpreter in the OpenKnowledge kernel to be replaced with an interpreter for a different language. Once done, however, it allows the more traditional process language (in our case WS-BPEL) to take advantage of the peer to peer discovery methods and other OpenKnowledge infrastructure. For those deeply committed to a non-LCC process specification language, this could be a more attractive long-term option than translation or meta-interpretation. Our experience of replacing LCC with WS-BPEL is described in our technical report [13].

Source URL: http://www.openk.org/beingAnOKuser/design/connect

Links:
[1] http://www.agilemodeling.com/artifacts/activityDiagram.htm
[2] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/UML-editor.zip
[3] http://taverna.sourceforge.net/
[4] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/taverna/taverna.pdf
[5] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/taverna/code.zip
[6] http://www.w3.org/Submission/OWL-S/
[7] http://www.w3.org/TR/owl-features/
[8] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/owls/lcc-owls.pdf
[9] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/owls/code.zip
[10] http://www.ibm.com/developerworks/library/specification/ws-bpel/
[11] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/bpel4ws-interpreter.pdf
[12] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
[13] http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D7.5/bpel-for-lcc.pdf